Runtime Monitoring for Open Real-Time Systems

Dissertation

zur Erlangung des akademischen Grades Doktoringenieur (Dr.-Ing.)

vorgelegt an der
Technischen Universität Dresden
Fakultät Informatik

eingereicht von

Dipl.-Inf. Martin Pohlack

geboren am 15. Juni 1978 in Löbau

Betreuender Hochschullehrer:	Prof. Dr. rer. nat. Hermann Härtig Technische Universität Dresden
Gutachter:	Prof. Dipl.-Ing. Dr. Gerhard Fohler Technische Universität Kaiserslautern
Tag der Verteidigung:	1. Juli 2010

Dresden im August 2010

Acknowledgements

First, I would like to thank my supervisor, Prof. Hermann Härtig, for his support and for providing a work environment at the Operating Systems Chair of the TU Dresden that made this work possible.

Many members of the operating-systems group have helped to make this work a reality. I would especially like to thank Björn Döbel and Michael Roitzsch, with whom it was a joy to work with. Björn Döbel and Torvarld Riegel contributed to this thesis by working on runtime-monitoring matters for their master theses.

I also want to thank the members of the Magpie research project at Microsoft Research Cambridge / UK for providing me with an extremely interesting research-internship experience.

Several people proofread versions of this thesis and gave very valuable feedback for which I am thankful: Prof. Gerhard Fohler, Prof. Hermann Härtig, Dr.-Ing. Michael Hohmuth, Juliane Pohlack, Sebastian Pohlack, and Michael Roitzsch.

My parents always supported me and believed in me for which I am grateful.

Last but not least, I want to thank my wonderful wife Juliane and my children for supporting me and for their patience.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The trend to higher integration in the computer industry has been unchanged in the last years and software systems kept up with the general growth leading to systems with an ever-increasing complexity, at least under the hoods. Monitoring of runtime properties in such complex systems becomes increasingly important.

We see not only a quantitative growth in that more and more functionality, storage space, and computing power is integrated into devices, but also a qualitative growth where functionality, which was formerly only available for dedicated real-time systems or embedded devices, is now requested for standard desktop systems or even mobile devices.

For example, a state-of-the-art smart phone usually comprises not only phone capabilities, contact management, and a personal organizer, but also video and audio playback support, navigation-system facilities, and a range of games. Here, typical desktop applications have found their way into a mobile device, and everyone expects that today's desktop system are capable of handling these tasks too, including the real-time aspects.

Another phenomenon is the trend towards openness for device classes with real-time purposes (e. g., appliances and mobile phones). Customers expect to install third-party applications (e. g., games, VoIP software, navigation software, extensions) on their devices today. These grown requirements demand increased capabilities of the underlying platform, for example, in the form of memory-protection units or full-grown file systems both of which where previously only found in larger device classes, such as notebooks and desktops. In fact, current mobile platforms run variants of full-fledged operating systems: Linux is part of Google's Android platform and a variant of Mac OS X is used in Apple's iPhones.

Obviously, this increased complexity in form of more complex hardware, additional software, larger software packages, and additional system layers will lead to new bugs, sources of instability, and unpredictability, making the need for a *common* diagnostic infrastructure more pressing than before.

Not only is a single common diagnostic approach desirable for the whole system, but a whole new range of diagnosis targets opens up with the integration of real-time and non–real-time tasks. In addition to typical debugging and profiling needs, now runtime monitoring of real-time properties, for example, whether deadlines for disk request were met or whether single video frames were computed and displayed in time, is desirable.

I derive, from above trends, a set of abstract requirements for such a monitoring facility. These requirements define the frame for my thesis and for my prototypical implementation named Ferret:

- The monitoring facility shall be usable in all software layers uniformly. It therefore must only have very few functional requirements itself. Furthermore, because of its pervasive deployment, it must not introduce new problems itself, for example in the form of instability or new information channels in the system.
- To be usable in *real-time environments*, a low intrusiveness on the observed system is required, in order not to comprise real-time properties. In case of resource shortage, original system execution shall be prioritized above monitoring.
- To be usable in the typically more complex *non–real-time environments*, also low intrusiveness is important but in a different form. Here, overall system performance (e. g., throughput) may be deemed more important than deadlines of specific tasks.
- The concrete monitoring targets will never be fully static but will evolve over time. Therefore, the monitoring facility will have to be flexible to follow this evolution over time.

I will provide a more detailed analysis and breakdown of these abstract requirements in Section 3.1.2.

1.1 Contributions

This work contains three main contributions and two auxiliary contributions.

Main contributions

Runtime-monitoring architecture for open real-time systems

I developed an architecture for runtime monitoring of open real-time systems, with the following properties:

- A single monitoring approach can be used for systems comprising real-time and non–real-time parts.
- Open systems with software from independent vendors are supported. There exist protection domains and several independent event producers and consumers can coexist in the system.
- The approach can be used for all system layers: kernel, libraries, system servers, and applications as no restrictions are imposed on threading models.
- The evaluation of properties works across component boundaries.
- The approach is noninvasive through a set of design aspects:
  - Monitoring information is only transported over shared memory, no other API of target systems is used, especially no potentially blocking kernel primitives.

  - There is no synchronous notification for event creation.
  - Shared memory between monitored processes and monitors forms one-way channels as monitors only have read access. This prevents information flow back to monitored processes.
  - Monitoring itself has a low overhead in terms of processor cycles.
- The architecture allows reusing different instrumentation techniques that are established for a target environment, as instrumentation and event transportation are separated.

Adaptation of fast, low-overhead user-space atomic sections for monitoring in a microkernel environment

I adapted a technique for user-space atomic sections for monitoring in a microkernel real-time environment.

- This technique provides both *very low average-case* execution time for throughput-driven non–real-time load and *low worst-case–execution-time* guarantees for deadline-driven real-time load.
- Due to the restriction to the monitoring use case, I could significantly simplify the implementation compared to generic atomic sections for user-space code (cf. to Section 3.2.4.3).
- I devised a new, low-overhead method for ensuring the return-of-control to the system from within atomic sections that I call *static cloning*.
- I adapted a calling convention for reusing compiler-generated code within atomic sections that simplifies sensor-access-code development significantly compared to writing assembly code by hand.
- The separation of the mechanism in form of atomic sections for user-space code and its usage in form of concrete sensor-access code allows to put only the small mechanism into the microkernel.
  - I demonstrated the usage of this mechanism in rollback and rollforward mode by means of different sensor implementations in user space.
  - The mechanism may be useful to other problem domains as well.
- In contrast to lock-free approaches, the usage of atomic sections allows efficient online evaluation as monitors' view on published events is always current (cf. to Section 3.2.4.2 on page 54).

Verified and evaluated design on the basis of an experimental implementation

I evaluated my design by means of a prototypical implementation named Ferret.

- I implemented different sensor types to verify broad suitability of the basic mechanisms.

- FERRET's *suitability for several problem domains* was evaluated on the basis of various scenarios as described in Sections 5.1.
- I provide quantitative evaluations for macro- and microbenchmarks, a non–real-time workload, and a real-time workload in Section 5.3 to verify my design approach.
- Section 5.2 contains an evaluation of several *qualitative properties* for different sensor approaches, which allows gauging the suitability of the sensors approaches for specific problem domains (real-time and non–real-time). I conclude that FERRET's rollforward sensors provide a good solution for mixed real-time and non–real-time systems.
- I discuss the portability of the monitoring design to other systems in Section 4.3.

Auxiliary contributions

Singularity, instrumentation of generated communication code

A predecessor of FERRET was implemented for the Singularity research operating system, was integrated into the build process for automatic instrumentation of generated communication code, and was used for collecting communication patterns in Singularity (cf. to Section 5.1.3.2).

Magpie, magpy, and Magpyvis

I developed further *magpy*, the Python variant of the Magpie software used for post-processing event traces, to work with Event Tracing for Singularity and FERRET. I also adapted *Magpyvis*, a visualization tool for manually inspecting captured event traces, which I used extensively for visualizing experimental results for this work (cf. to Figures 5.5, 5.10, and 5.11).

1.2 The scope of my work and organization

In this work, I describe the design, implementation, and evaluation of a runtime monitoring system for open real-time systems.

After giving a brief motivation here, I start out this work by providing a detailed survey of related work in the next Chapter 2. I discuss there an overview about instrumentation techniques, techniques for transporting monitoring data, different approaches for evaluating monitoring data, and a collection of smaller but still interesting topics. I close that chapter by comparing the terminology used in the literature and by defining which terms I will use throughout this work.

After introducing the foundations by providing an overview over the state of the art and identifying shortcomings and possibilities for reuse for my envisioned target systems, I describe in Chapter 3 these target systems in more detail and refine the requirements analysis. The chapter continues with a detailed discussion of design decisions and the

architecture of FERRET, wherein I touch the topics of roles in the system, online vs. offline evaluation and external schemata, sensor variants and timestamps, and sensor synchronization approaches. Chapter 3 closes with the description of an idea for *atomic sections in user-space code* that I adapted for sensor synchronization.

The succeeding Chapter 4 contains detailed explanations of implementation aspects, most prominently how these atomic sections are implemented with rollback and rollforward guarantees in a microkernel-based real-time system, and how these mechanisms are put to use for specific sensor variants. In this chapter, I additionally discuss the placement of instrumentation points and portability of FERRET.

I provide a detailed evaluation of my work in Chapter 5. The chapter starts out with a description of ten different use cases for which I applied or ported FERRET or its predecessors. In the second part of the chapter, I provide a qualitative evaluation of different sensor types for specific problem domains. The chapter concludes with a quantitative evaluation comprised of low-level microbenchmarks and macrobenchmarks for gauging FERRET's suitability for real-time and non–real-time environments.

This thesis concludes with a summary and suggestions for future work in Chapter 6.

1.3 List of Publications

[1] Michael Roitzsch and Martin Pohlack. Video quality and system resources: Scheduling two opponents. *Journal of Visual Communication and Image Representation*, 19(8):473–488, 2008.

[2] Hermann Härtig, Steffen Zschaler, Martin Pohlack, Ronald Aigner, Steffen Göbel, Christoph Pohl, and Simone Röttger. Enforceable Component-Based Realtime Contracts — Supporting Realtime Properties from Software Development to Execution. *Springer Real-Time Systems Journal*, 35(1), January 2007.

[3] Hermann Härtig, Steffen Zschaler, Martin Pohlack, Ronald Aigner, Steffen Göbel, Christoph Pohl, and Simone Röttger. Enforceable component-based realtime contracts: Supporting realtime properties from software development to execution. *Real-Time Systems*, 35(1):1–31, January 2007.

[4] Michael Roitzsch and Martin Pohlack. Principles for the Prediction of Video Decoding Times applied to MPEG-1/2 and MPEG-4 Part 2 Video. In *Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS 06)*, Rio de Janeiro, Brazil, December 2006. IEEE.

[5] Martin Pohlack, Björn Döbel, and Adam Lackorzynski. Towards Runtime Monitoring in Real-Time Systems. In *Proceedings of the Eigth Real-Time Linux Workshop*, Lanzhou, China, 2006.

[6] Ronald Aigner, Christoph Pohl, Martin Pohlack, and Steffen Zschaler. Tailor-Made Containers: Modeling Non-functional Middleware Service. In *Workshop on Models for Non-Functional Aspects of Component-Based Software (NFC'04) colocated with UML 2004*, Lissabon, Portugal, October 2004.

[7] Steffen Göbel, Christoph Pohl, Ronald Aigner, Martin Pohlack, Simone Röttger, and Steffen Zschaler. The COMQUAD Component Container Architecture. In *WICSA '04: Proceedings of the Fourth Working IEEE/IFIP Conference on Software Architecture*, page 315, Washington, DC, USA, 2004. IEEE Computer Society.

[8] Steffen Göbel, Christoph Pohl, Martin Pohlack, and Steffen Zschaler. The COMQUAD Component Container Architecture and Contract Negotiation. Technical Report TUD-FI04-04-April-2004, TU Dresden, 2004. Available from URL: `http://os.inf.tu-dresden.de/drops/doc.html#tr-comqarch`.

[9] Martin Pohlack, Ronald Aigner, and Hermann Härtig. Connecting Real-Time and Non-Real-Time Components. Technical Report TUD-FI04-01-Februar-2004, TU Dresden, 2004. Available from URL: `http://os.inf.tu-dresden.de/papers_ps/tr-rtnonrtcomp.pdf`.

[10] Ronald Aigner, Martin Pohlack, Simone Röttger, and Steffen Zschaler. Towards Pervasive Treatment of Non-Functional Properties at Design and Run-Time. In *Proceedings of 16th International Conference "Software & Systems Engineering and their Applications (ICSSEA 2003)"*, Paris, France, December 2003.

[11] Lars Reuther and Martin Pohlack. Rotational-Position-Aware Real-Time Disk Scheduling Using a Dynamic Active Subset (DAS). In *24th IEEE Real-Time Systems Symposium (RTSS)*, pages 374–385, Cancun, Mexico, December 2003.

[12] H. Härtig, J. Löser, F. Mehnert, L. Reuther, M. Pohlack, and A. Warg. An I/O Architecture for Microkernel-Based Operating Systems. Technical Report TUD-FI03-08-Juli-2003, Dresden University of Technology, Dresden, Germany, July 2003.

[13] Lars Reuther and Martin Pohlack. Work-in-Progress Report: Using SATF Scheduling in Real-Time Systems. *Work-in-Progress Report: 2nd USENIX Conference on File and Storage Technologies (FAST-2003), San Francisco, CA, USA*, March 2003.

Chapter 2

Foundations and related work

There exists a plethora of related work in the area of tracing and monitoring that covers different aspects of the problem space. Some projects focus primarily on a single aspect (e. g., event transportation [ZYW+03] or instrumentation [Kri05,Moo01]) while others cover an entire workflow (e. g., [CSL04,BDS07b,OPr08,DD06]). I distinguish three main aspects of the topic in this chapter and will discuss related work in context of each of those.

In addition to providing an introduction into the area of runtime monitoring, the overview given here also demonstrates that there is no solution in the literature that combines all relevant properties for runtime monitoring for open real-time systems (cf. to Section 3.1.2).

This chapter starts with a section about *instrumentation* and is followed by a discussion of trace data *transport* approaches in Section 2.2. Section 2.3 attends to the *evaluation*, that is, the actual use of trace data. The following Section 2.4 deals with aspects not fitting any of the three larger aspects. Section 2.5 introduces the *terminology* used throughout this work and relates it to the diversified terminology used in the literature whenever there are conflicts. I conclude this chapter with a summary on related approaches and derive requirements from that in Section 2.6.

2.1 Instrumentation techniques

Instrumentation techniques are a necessary means for this work to transport the monitoring functionality into the target system. In the following, I will discuss different approaches and their effect on monitoring in real-time systems.

In a broader sense there are two approaches to instrumentation, the static and the dynamic one. The static approach comprises all techniques at build time (compiling and linking). Dynamic techniques work on binaries on disk, on suspended programs, or even on fully running systems. The distinction between both approaches is blurred sometimes with dynamic recompilation and combined approaches (e. g., Linux Kernel Markers [Des08]). Several papers [BDS07b,CSL04,LCM+05] discuss static and dynamic instrumentation in their respective scope and may provide a good introduction.

Static instrumentation can be done simply by modifying source code and requires no runtime infrastructure. Due to its simplicity, nearly every programmer has used it in the form of the infamous `printf` debug statements. Static source-code instrumentation enjoys the full compiler optimization potential (it is treated as regular source code) and is applicable in all, otherwise problematic, environments (such as interrupt-handler

routines). Additionally, static source code instrumentation typically has access to all local variables and intermediate states that might be inaccessible or even nonexistent in optimized binaries. Maintaining instrumentation code is simplified with static instrumentation as the instrumentation code is directly interweaved with regular source code. Also instrumentation overhead is reduced with static instrumentation as no additional runtime penalty (such as expensive trapping instructions) has to be paid.

Of course, this simplicity also has its drawbacks, most prominently in the form of inflexibility. Changed requirements in the form of new or modified events (contents or location) require a full build-and-run cycle, whereas dynamic instrumentation could just adapt a linked binary or a running system. Static-instrumentation probe-handler cost has to be paid always whereas dynamic instrumentation can be activated selectively, that is, at a certain time or for certain system parts only. Dynamic compilation, however, might not have full access to all local variables and states. On the other hand it also does not prevent compilers from completely optimizing them away.

2.1.1 Static instrumentation

Thane

In his PhD thesis *Monitoring, Testing and Debugging of Distributed Real-Time Systems* [Tha00] Thane discusses the problem of intrusiveness for systems with embedded software sensors. He argues to generally leave them in the system even for the final deployment. After all testing has been done with the sensors in the system, removing them for the final deployment would be a too big change. Also, by making them part of the system already early in the design process, there is no *additional* overhead involved in having runtime monitoring, so the intrusiveness is (defined to) zero.

While this approach might work and even be required for embedded systems, applying it to a typical desktop machine is problematic. The software environment is much richer and more dynamic. It is very hard to predict all potentially required sensors beforehand. While single sensors have a negligible overhead, statically instrumenting all potentially interesting places in, for example, a Linux kernel would slow down a system considerably. Thane's approach is applicable for simple, hard–real-time parts of a system, where functionality and monitoring requirements are known beforehand and are static. There is a realistic chance that, for example, for instrumenting a microkernel, the type, number, and functionality of sensors can be determined statically with only rare revisions.

K42

K42 [AAD+02] is a microkernel-based operating system, designed from the ground up with correctness debugging, performance debugging, and performance monitoring in mind — online and offline. Tracing code is statically anchored in components on all layers (kernel, libraries, servers, and applications) and adding, removing, or changing events requires a full build cycle. Event logging can be controlled only in larger groups (major event identifier) and there is only one central event buffer per processor in the system, used by both kernel code and user-space code.

LTT and LTTng

The *Linux Trace Toolkit* (LTT) [YD00] includes a patch to the Linux kernel with statically defined tracepoints. Its successor, the *Linux Trace Toolkit Next Generation* (LTTng) [DD06], also uses a static instrumentation approach as Desnoyers and Dagenais consider dynamic approaches to be too intrusive for their designated target systems

(high-performance and embedded real-time) "as it implies using a costly breakpoint interrupt". This statement probably reflects CISC reality, in particular x86, with its variable-length instruction set architecture (ISA). For RISC architectures, there are better suited alternatives as discussed in Section 2.1.3 below.

Google-sized ...

In [BDS07b] Bligh et al. describe the merging of LTTng and Google's Ktrace and propose to use static markers for most cases. The authors justify this as follows: static trace points are significantly faster than dynamic probes with Kprobes (slow `INT3` mechanisms), static probe points can be used anywhere while dynamic probes are limited in scope, easy access to local variables or registers with static probes, ease of maintenance for trace points, and static trace points have a practically negligible overhead if deactivated. Single trace points can be activated dynamically, which shifts this approach into the hybrid direction.

Tool-assisted instrumentation

While the previous approaches work directly on source-code level by manually inserting tracing code, tools provide more sophisticated support.

Aspect-oriented programming (AOP) tools allow placing instrumentation support in source code [MSSP02], usually allowing fine-grained control over source code manipulation. There are production-quality AOP tools for C++ and Java, but not for C.

Compilers, such as gcc, support tracing at the function level. Normal functions are wrapped by inserting calls to user-defined entry and exit functions. Instrumentation can be activated per compile unit. While with older version of gcc (e. g., 3.4.6), `-finstrument-functions` prevented function inlining [GC304], current versions do not have this limitation [GC405]. However, there might still be a negative affect on code size, as an addressable copy of the inline function must be kept in the binary, which could have been optimized away otherwise.

KFT - Kernel Function Trace

The *Kernel Function Trace* project [KFT08] uses this gcc capability to instrument the Linux kernel. Once the kernel is instrumented, tracing can be controlled with triggers and filters. The tracing configuration can also be altered in the running Linux system.

2.1.2 Hybrid approaches

Linux Kernel Markers

Linux Kernel Markers [Des08] are best described by citing Mathieu Desnoyers' email to the Linux kernel mailing list (from 2007-15-02, 17:30:37 -0500, subject: "Re: [PATCH 00/05] Linux Kernel Markers - kernel 2.6.20"):

> I think the final agreement was the need for some kind of code marking system, which I tried to implement as best as I could. It gives very good performances while tracing (advantage of static tracing), has a very very minimal performance and binary size impact when disabled (advantage of dynamic tracing) and it can be activated dynamically (advantage of dynamic tracing).

Linux Kernel Markers are compiled in statically, but their only built-in functionality is to be configurable dynamically.

Feather-Trace

The *Feather-Trace* event tracing toolkit [BA07] targets a very similar area. Its markers can be activated and deactivated atomically and are multi-processor safe. They contain a call to a probe function and a relative jump over this call in the deactivated state.

The marker is activated by changing a single byte in the jump instruction to not skip the call. Feather-Trace supports Linux and FreeBSD and is relatively portable as only little kernel infrastructure is used. A somewhat similar marker technique is described in [Stö05].

2.1.3 Dynamic instrumentation

Paradyn

The Paradyn project [Uni08] hosts several dynamic instrumentation approaches, three of which shall be detailed in the following.

Dyninst

Dyninst [BH00] allows the instrumentation of running user-space programs. Dyninst's application programming interface (API) has two primary abstractions — points and snippets. Points define *where* to instrument (similar to join points in AOP) and snippets define *what* code to insert (the concrete aspects in AOP). The intended application of Dyninst's dynamic technique is debugging, performance monitoring, and composition of applications out of packages. Snippet code is placed into trampoline pages and wrapped in glue code for preserving machine state. Jumps to these trampolines are patched into the actual programs. Dyninst supports x86, SPARC, Alpha, and PowerPC platforms.

KernInst

With *KernInst* [TM99] Tamches et al. present a fine-grained instrumentation framework for Sun's Solaris. KernInst runs on an unmodified kernel. The key concept of KernInst is safe instruction splicing in hot code: Probing code can be inserted into the instruction stream by overwriting the target instruction with a branching or trapping instruction. Springboard heaps are used as trampoline locations if spliced instructions have limited jump range. [TM99] proposes to use single-byte trapping instructions for safe splicing on variable-length ISAs, such as x86.

CrossWalk

CrossWalk [MM03] essentially combines Dyninst and KernInst to allow kernel-boundary–crossing instrumentation. CrossWalk dynamically follows hotspots starting within user-space code and drills down into the kernel to find the bottlenecks. Mirgorodskiy et al. argue that trap-based approaches have to much overhead for performance monitoring. They demonstrate CrossWalk by means of a case study of the Squid proxy server on Solaris / SPARC[1].

GILK

GILK [PKFH02] is an attempt to transfer KernInst techniques to x86 with its variable-length ISA efficiently. GILK allows runtime code splicing in a running Linux kernel. To determine safe splicing locations GILK generates a control flow graph of the whole kernel. However, it remains unclear how indirect jumps, dynamic jumps, and self-modifying code is handled. Pearce et al. describe a interesting technique called local bouncing for cases were the normal 5-byte-long jump instructions cannot be used. 2-byte-long short-range jumps are used in that case to redirect control flow to locations where longer jumps can be used safely. GILK even splices in dummy instrumentation to make room for such trampoline points (this technique is similar to the springboards described in [TM99]).

DTrace

DTrace [CSL04] is a tracing framework originally developed for Sun's Solaris. Meanwhile it was also ported to FreeBSD and MacOS X. DTrace's kernel instrumentation is dynamic for the most part and does not incur any probe effect if unused in that case.

[1] SPARC is a Reduced instruction set computer architecture and therefore has fixed instruction size. Dyninst and KernInst work without traps on these architectures.

Also some static probes are used with a practically unmeasurable overhead. User-space programs as well as (most of) the kernel can be traced, thereto, probe points must be declared by providers. Dynamic instrumentation uses trapping instructions for the x86 architecture and jump instructions on SPARC. User-space programs are always instrumented with traps and probe code is then executed in the kernel.

However, DTrace is not especially suited for real-time systems, as probe code is executed synchronously with interrupts turned off in the kernel. Furthermore, more than one probe function may be active per probe point.

Atom

ATOM is a binary rewriting tool based on OM [SW94,SW92]. It originates from the MIPS architecture but was ported to Alpha early on (parts were later also transferred to the x86 architecture). ATOM provides an infrastructure and plain API that makes writing tools simple. Typical application scenarios include basic block counting, profiling, dynamic memory recording, instruction and data cache simulation, pipeline simulation, evaluating branch prediction, and instruction scheduling [SE94]. One of ATOM's principles is to modify application address spaces as little as possible on instrumentation.

Vulcan

Vulcan [SEV01] is ATOM's successor, improved in several areas. Vulcan supports three architectures: x86, IA-64, and CIL (Common Intermediate Language, also known as Microsoft Intermediate Language, MSIL) but stores programs internally in an abstract representation. Tools may work on this abstract representation but also the mapping to the original byte representation is kept. Finally, binaries are exported to a native format again (format conversion is possible).

In addition to the static mode described above, Vulcan also supports to dynamically instrument running programs. Also remote instrumentation is possible with the help of a proxy library.

DProbes

Dynamic Probes (DProbes) [Moo01] were derived from *Dynamic Trace* on OS/2 and allow dynamic instrumentation of arbitrary locations within applications and the Linux kernel. DProbes aim at minimal interference with kernel infrastructure and use *INT3* as a trapping instruction to mark probe points. That way, no overhead is incurred if probe points are inactive, otherwise trapping costs have to be paid. Whenever a probe point is hit, the framework executes user-defined probe handlers. They have to be written in a specialized language and are executed by an interpreter.

kProbes etc.

Kernel Probes (kProbes) [Kri05] have similarities to DProbes but are targeted solely at dynamic instrumentation of the Linux kernel. They are dynamically inserted into a running kernel by loading a kernel module. Any location inside the kernel can be instrumented by replacing instructions with a trapping instruction. On x86 architectures, `INT3` is used for this purpose. When such an instruction is hit, the kProbes framework performs all necessary tasks for event generation. It executes handlers, single-steps the original instruction, and finally returns control to the code that raised the exception. In addition to arbitrary kProbes, function-entry probes (jProbes) and function-return probes (kRetProbes) are available.

Direct Jump Probe

Direct Jump Probes (Djprobe) [HO07] do not use an expensive trapping instruction as kProbes do but directly insert jump instructions, which is possible in many situations. Djprobe verifies this on instrumentation and uses kProbes as a fallback solution.

Systemtap

Systemtap [EPC+05] uses kProbes for placing event handlers in certain kernel locations. In contrast to DProbes interpreted probe handlers, here, a script language (resem-

bling DTrace's D) is used to express handler functions. These handler functions are translated to C and compiled as kernel modules that are then loaded into the kernel and hooked onto events. [EPC+05] discusses how different security problems are tackled with varying approaches. However, Systemtap is only meant to be used by root, not by ordinary users.

Pin

Intel's *Pin* [LCM+05] takes a different route compared to the previously discussed approaches. Probe handler code is not spliced into running programs by overwriting instruction and creating trampoline code. Instead, basic blocks are completely recreated with dynamic recompilation. In the course of this, probe code can often be completely inlined into the original code. Dynamic recompilation can work very efficiently, also for variable-length ISAs, such as Intel's x86 or AMD's x86-64. Standard compiler optimization techniques (e. g., constant propagation, constant folding, register liveliness analysis, ...) can be applied, which may result in more efficient instrumentation than with static approaches. Pin works for Linux user-space programs and attaches itself as debugger via the ptrace interface.

JIT instrumentation

JIT Instrumentation [OMCB07] is in many aspects modeled after Pin, but works in the Linux kernel. In this environment available memory is more constrained and some optimizations done by Pin cannot be applied (e. g., function cloning).

Valgrind

Valgrind [NS07] is a framework for heavy-weight dynamic instrumentation of Linux and AIX user-space programs. The paper distinguishes two approaches: disassemble-and-resynthesize (D&R) and copy-and-annotate (C&A). Valgrind employs full dynamic recompilation (D&R) and thereto parses the original binary representation for conversion into an intermediate representation (IR). Valgrind applies several optimization steps onto this IR and also passes it to plugins that may augment it with their instrumentation. The IR is finally resynthesized to native representation for execution.

Valgrind is much more than an instrumentation engine. It tracks a shadow value for each memory location and each register, which allows plugins to perform various tasks. [NS07] establishes nine requirements for dynamic binary instrumentation frameworks and evaluates related approaches with respect to these requirements. With its heavy-weight approach Valgrind represents the opposite end on the overhead spectrum of what I target.

2.2 Transport techniques

In the previous section I gave an overview about related work that instruments target systems. Once this instrumentation is in place, event data accrues. In the following, I will describe how this data is managed and transported in related work. Solutions range from instant evaluation and processing of events (in the extreme case no data needs to be transported or stored) over online aggregation and transformation to full logging of all events on all processors.

The fundamental problem is similar to memory management for several independent clients. Some clients allocate new memory all the time (event producers) and some always have to free it (event consumers). These roles have to be synchronized. Furthermore, in real-time systems event producers must not block or depend on consumers.

Depending on the memory management approach, fragmentation is also a relevant criterion. Due to these specific requirements traditional memory management libraries but also real-time specific solutions, such as [MRCR04], are typically not sufficient. Often custom solutions are employed.

[AAD+02] provides a good motivation and a detailed discussion of *K42's* tracing infrastructure and design decisions. [WR03] provides more details on the implementation. K42 uses per-processor tracebuffers to be scalable. But there is only one such per-processor buffer for the whole system. The kernel, libraries, servers, and regular applications all have full access to this buffer if tracing is enabled for them. Therefore, K42 cannot guarantee separation between traced parties (this problem is also discussed in [DD06]). Buffer-access code uses lock-free atomic operations, which leads to good scalability. It may, however, be problematic for hard real-time systems as retry loops have no upper bound (cf. to [BA07]). Furthermore, even if all traced parties fully cooperate, data integrity in traces cannot be guaranteed. Event-producing programs that are temporarily suspended may resume writing to buffers much later when these buffer have in-fact been reused for other events due to buffer wrap-around. This may corrupt event traces. [AAD+02,WR03] argue that this problem only occurs in "*[. . .]* the most extreme conditions *[. . .]*" and that analysis tools should be written such as to be able to deal with incomplete or corrupted logs. K42

Relay [ZYW+03] (formerly known as relayfs) is the project that probably deals most explicitly with the transportation problem. Relay is a subsystem for the Linux kernel for highly efficient data transportation from kernel to user space. Relay knows different operation modes (locked vs. lock-less, per-processor buffer vs. global buffers, block or packet delivery). Buffers (called channels) are modeled after K42's tracebuffer, but relay supports several channels at the same time. In fact, channels are meant to be local to components or subsystems. That way, K42's separation problem is approached. Channels can be accessed independently in user space allowing finer control about who sees which events. Relay still suffers from the same corruption problems as K42 in lock-less mode with temporarily suspended writers (see above, incomplete or corrupted buffers may be delivered to consumers). relayfs

Relay's channels are usually split into sub-buffers, which is the typical unit of delivery to consumers. Relay allows online buffer resizing to adapt to various event rates. It also supports variably sized events. Relay users can register for five types of callback events (buffer start, buffer end, event delivery, buffers full, buffer resize). These callbacks allow, for instance, to define a policy for buffer-full conditions (overwrite, drop, or suspend). This also implies two other things: First, there is information flow from consumers to producers at least in the form of the consumers' buffer position. Second, only one consumer is supported per channel.

As single events do not span across sub-buffer borders, channel memory can be fragmented internally. To that end, sub-buffers are filled up with place-holder events. Relay supports a slow path (sub-buffer boundary crossing) and a fast path (no boundary crossing) for event creation. Consumers, however, have a continuous view on the event stream. This implies active copying and realigning of event payload by relay on delivery to consumers.

Relay's interface for creating events is very lean, only a channel identifier and a pointer to event data with length information is used.

Timestamping is also handled by relay on demand, that is, not every event needs to be timestamped. Relay supports both the efficient processor timestamp counter (TSC on x86) and the slow but globally consistent `gettimeofday()` timestamp source.

Today, relay enjoys increasing usage in Linux-kernel–related projects [BDS07b,DD06, DTI08].

LTTng

LTTng [DD06] uses relay for in-kernel tracing. User-space programs can either use a slow path (kernel entry for each event) or be traced with a user-space–only shared-memory approach (fast path in `liblttng-usertrace-fast`). Therefore, for each thread a companion process is created that dumps events to disk.

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

In [Stö05] Stöß describes how to use event tracing for efficient communication with *user-level multiprocessor schedulers*. Instrumentation points can be activated dynamically and per-processor shared-memory buffers are used for event transportation from both the kernel and user-space applications.

The scheduling problem targeted in this work is very specific hence the tracing approach is tailored to it. Only a short history of events is needed for scheduling decision, therefore small, cyclic memory buffers are used. Lock-free synchronization is used on these memory buffers.

Stöß defines the concept of accounting domains. This allows to reduce the number of events created, as often only inter–accounting-domain scheduling events are required and not intra–accounting-domain events. Furthermore, accounting domains are assigned independent memory buffers for easy event routing and robustness (memory buffer corruptions are restricted to single accounting domains).

Buffers used within the kernel are exported writable to user-level schedulers. Consequently, corruption of these buffers may represent a safety threat to the whole system. This problem is addressed by additional check code in the kernel.

ETW

In *Event Tracing for Windows* (ETW) [Micb,Mica,Mye05,PB07], communication with clients uses a set of processor-local buffers that are handed to consumers once they are full. This mechanism reduces communication overhead as not each single event is delivered but entire batches. However, there is no upper latency bound (e. g., if buffers are filled slowly). In [BDIM04] Barham and colleagues mention an approximate processing delay of one second for the online version of Magpie (cf. to Section 2.3.2 on page 19). It remains unclear how this bound is determined and if it is sufficient in all cases.

Additionally, ETW synchronously notifies event consumers with call-back functions that introduce additional causal relationships between the observed and the observing entity.

Different sources state different numbers about ETW's overhead: [Mye05] estimates 1 500–2 000 cycles and [Nar05] about 1 000 cycles. For posting ETW events from user-space, at least one kernel entry is involved.

ETW has the concept of sessions to allow several consumers at a time. Sessions can be configured independently, but events required for more than one session are copied to several buffers. Events can be filtered based on severity and the sub-component from which they originate.

DTrace [CSL04] uses a double buffering technique for the communication with event consumers who are responsible for timely extraction of data. DTrace guarantees data integrity (unlike K42's tracing and LTTng), that is, all received data is guaranteed to be correct and potential data loss can be detected. DTrace

In [Mcd77] Mcdaniel describes *METRIC: a kernel instrumentation system for distributed environments*. The different roles in METRIC (accountant, analyst, probes in object systems) reside on distinct physical machines to be independent from each other and influence each other as little as possible (only the actual probe code must be debugged very well to preserve system stability). Events are transported over Ethernet and are not acknowledged, there is no information flow back from analysts or accountants to probes. Single events are assumed to be of little value, hence transport over a lossy medium without protection is acceptable. However, event duplication and loss can be detected. Mcdaniel targets mostly statistical evaluation, therefore this restriction is acceptable for him. METRIC

Although this work is quite old, its basic principles and rationales are still valid and desirable for event transportation today: separation of the roles and true one-way communication. Both help minimizing the probe effect, the separation by reducing the amount of code that is executed *within* the context of the observed part and the one-way communication by preventing influence from parts executed *outside* of the observed party.

Active Harmony [HK99] uses an extensible metric interface to transport performance information system wide. Applications publish their resource requirements and the system publishes its capacity and utilization information. The adaptation controller monitors this information online to adapt the system and applications via their tuning controls. Applications register tunable parameters with the system. They also periodically update a notion of how good they can work with given resources. The adaptation controller can use this feedback information for refining resource allocations. Active Harmony

Feather-Trace [BA07] uses a single wait-free memory buffer for all events in a system. The buffer is partitioned into equal-sized slots and supports arbitrarily many writers at once. Each slot can carry one event and is either empty, full, or busy (currently being filled). Events can be lost if the buffer runs full (no empty slots). In that case an error counter is incremented. Feather-Trace

Feather-Trace only targets offline evaluation of traces, therefore only a single reader (event consumer) is supported that is meant to dump events to a trace file. The buffer contains a read index that is actively incremented by the reader, that is, the reader needs write access to the buffer. The reader also marks read entries as empty again. This construction is somewhat similar to the list sensor in Ferret's predecessor RT_Mon [Poh08] and suffers from the same problems.

Although, the algorithm is wait-free, in that no party needs to block, this property is achieved by taking the error path in the code. A slow writer, a long-interrupted writer, or a terminated writer can block a single slot for long durations or permanently by leaving it in the busy state. The reader will only inspect that slot to which the reader index points. It will never skip this slot if it remains busy. That way, the whole buffer will quickly run full and all further events will be lost.

DProbes [Moo01] uses per-processor log buffers for staging trace records. The special DProbes

probe handler language has primitives for pushing data from registers or system memory into these log buffers. Complete trace records are handed to external infrastructure (the paper mentions LTT as an example).

2.3 Evaluation approaches

In the previous two sections I discussed instrumentation and event transport. This section will target how evaluation of acquired traces is done. There are many projects in this area that are typically multifaceted. I will start by introducing the concept of internal and external request specification, followed by a discussion of work on specification languages. After that I discuss various single projects.

2.3.1 Internal vs. external request specification

A core abstraction for monitoring are requests. Requests are user-defined units of work, specific to a scenario. Typical requests are: everything that belongs to a single client's `HTTP` request (potentially crossing machine boundaries on the server side for accessing a database), or everything related to displaying a single video frame (e. g., data retrieval from a filesystem, decoding, display).

In general there are two ways of defining requests. They can be determined internally in the instrumented program or they can be created externally, decoupled from the running system, just by using the event trace. I will discuss both approaches in the following:

- *Internal request definition* is popular in the literature [CAK+04,PB07,GEGW02, FPK+07] (Activity ID, Request ID, ...). Typically, problem specific instrumentation is used and application interfaces are modified to carry along a request identifier if there is no natural identifier to be used. This modification may be very intrusive as it changes the application structure. Instrumentation, request definition, and evaluation are tightly coupled. [FPK+07] advocates transporting request IDs in-band for large distributed systems spanning several administration domains, because IDs may be passed through systems that don't have to be adapted. Typical use cases are IP options, TCP options, and HTTP headers.

  Also, the restriction to one concrete problem usually simplifies the whole instrumentation process.

  This coupling is disadvantageous, however, if any modification or reuse is attempted. Additionally, instrumentation and request definition have to be created at the same time (and probably by the same group of persons). This restricts the approach's applicability.

- *External request definitions* are proposed in [BDIM04,BIMN03,CSL04,FPK+07]. Instrumentation and request definition are decoupled. There is no notion of requests in the monitored system, instead requests are constructed purely from the event trace. To be more precise, external request definition is a superset of internal definition. Every technique from the internal approach can also be

used with external requests but external processing can be applied *additionally*. Program-internal request identifier are not enforced but can be used if they are available.

The advantage of external requests lies in the decoupling of instrumentation and request definition. Instrumentation may be done by the component developers, typically at times when concrete request requirements are not known yet. Request definition can be done at any later time, potentially by somebody else. Instrumentation may be reused for different request definitions in completely independent projects. Also, the estimated impact of instrumentation is smaller as existing interfaces and implementations usually need not be adapted to carry request identifiers across components.

External request definition can cope with situations where not the first event in a new request decides whether it is a new request at all. Imagine, for example, that a new network packet arrives. The first event is created at the interrupt, but request assignment can only be done much later, in a higher layer in the application stack after interpreting the packet's payload.

In practice, both approaches tend to blur together. Often, desired request definitions roughly follow system structure and may use natural identifiers[2]. Small adaptations in applications tend to ease request definition. External requests' advantages come into play when stitching together request parts that cross component or system boundaries. Also, sometimes events simply do not carry enough information for defining external request. Than a feedback loop is required for refining instrumentation based an concrete request requirements.

2.3.2 Specification languages

In [PW93] Perl and Weihl introduce the *PSpec* language, intended for automated checking of performance assertions of complex systems. PSpec is not meant for proving performance properties, but for regression testing, performance debugging, and clarification of expectations. A PSpec specification typically contains several PSpec assertions that are checked against log files. The specification writer is typically a human assisted by a solver tool, which helps in finding constants by processing concrete measurement results. The evaluator tools allow interactive viewing of log files and assists in formulating expectations. The checker tool applies specifications on concrete log files and determines whether they are fulfilled offline. PSpec supports four concepts: intervals, metrics, events, and assertions. Intervals are the typical unit of interest and are the abstraction of several related events (bounded by start and stop events).[3] Metrics are properties of intervals (e. g., elapsed time, throughput, and utilization). Single events also have attributes, such as, timestamp, processor ID, and thread ID. Assertions typically express expectations about aggregated interval attributes (e. g., the average duration of read requests).

PSpec

[2] *Natural* here means: already in use in the system before instrumentation was added.
[3] Intervals resemble requests as discussed in Section 2.3.1.

PSpec was designed to be very compact and readable and therefore its expressiveness is limited. Intervals have to be continuous with PSpec, for example, computing the processor time demand for a disk read request that typically has a very long idle period between submitting the request and delivering data is not easily possible with PSpec.

Pip

Pip [RKW+06] is an infrastructure for monitoring distributed systems. Actual behavior is compared to assumed behavior. A behavior specification language is a core component of Pip. It allows efficient expressing of sequential execution, parallelism, alternative execution location (called *thread patterns*), and asynchronous execution (called *futures*). The language also contains dedicated primitives for matching path fragments and performance characteristics. In contrast to a generic language, Pip's expectation language is confined to its domain. No other post-processing is possible and expressing complicated expectations is laborious to impossible (variables are not supported).

The article contains several interesting (as they reflect experience) restrictions to the common monitoring case. Pip defines the following metrics: real time, processor time, number of context switches, message size, and latency. Events are grouped relatively strictly in Pip. It knows *tasks* (like a profiled procedure call, with beginning and end), *messages* (any communication between hosts or threads), and *notices* (opaque string). Notices are typically only used by manual instrumentation. Automatic methods use the other event types.

Pip contains tools for verifying specified expectations against trace files, for visualization of expectations, and automatic generation of initial expectations from collected traces.

The article contains an interesting discussion on the topic of model checking, which does not only apply to Pip but many monitoring systems. Model checking is an exhaustive approach, all possibilities are covered. Monitoring can only cover execution runs actually seen in a system. On the other hand, instrumenting a system for monitoring is a much simpler task than transforming it to a to-be-checked model. Monitoring can be used to check actual *implementations* in contrast to some form of *specification* that is model checked. The transformation step between the model and the implementation is eliminated. I describe the application of this approach with Ferret in Section 5.1.2.2.

Instrumentation for Pip is done by source code annotations. Explicit request IDs (called *Path IDs*) are used. Pip collects trace files locally and merges them. Evaluation is done offline.

Run-time Monitoring of Real-Time Systems

In [Jah95] Jahanian uses real-time logic (RTL) to specify system properties. He therefore distinguishes two types of events: task events (describe changes inside a task) and run-time system events (scheduling decisions, (de-)blocking of tasks, etc.). Instead of creating new event instances in tracebuffers, here each event has a history comprised of time-value pairs. Monitors must be able to access this history fast and with bounded blocking. A bound on event history length can be derived from to-be monitored RTL formulae.

Jahanian furthermore distinguishes between synchronous and asynchronous monitoring. Synchronous monitoring is embedded into the executing task and can directly manipulate event histories. Asynchronous monitors are external and receive event history with messages. External monitors are scheduled as normal real-time tasks in the

system and can be included into the admission process. However, unbounded priority inversion on access to event histories must be prevented.

Timestamps must be obtained atomically with the occurrence of events, otherwise comparing timestamps for different events is not meaningful. Clock synchronization, resolution, and reading overhead is also discussed.

Event calculus

In *Towards Security Monitoring Patterns* [SKA07] Spanoudakis et al. present patterns for expressing the three security properties *confidentiality*, *integrity*, and *availability* in event calculus [Sha99]. The authors also describe a framework gathering the required events on the basis of a case study. These and similar patterns could also be checked online in a monitor consuming Ferret events.

Magpie and ETW

Magpie [BDIM04,BIMN03] is a toolchain that can extract requests, based on observed events of one or several connected systems. Requests are typically the unit of interest for human interpretation or performance evaluation. What comprises a request is highly problem specific and can be defined in external schemata in Magpie. This property is a key difference to work discussed above, where request definition is often strongly interweaved with the instrumentation code itself. Also other event post-processing, such as resource-usage determination is specified in schemata in Magpie.

Originally, Magpie was contrived only as a client for the ETW framework. In the context of this thesis, I adapted Magpie to work with both Event Tracing for Singularity (ETS) and Ferret for finding, building, and post-processing requests. Schemata are implemented within a Python framework.

Magpie variants can process traces online and offline, but do not give real-time guarantees. However, online usage was shown to be feasible in a typical business scenario [BDIM04].

Post-processed requests are used to generate behavior models and canonical request using machine learning techniques [CO94].

2.3.3 Other projects

Automated Known Problem Diagnosis with Event Traces

In [YLW+06] Yuan et al. describe a system for *Automated Known Problem Diagnosis with Event Traces*. The high-level idea is to have a database of symptoms of known problems (together with a solution, if available) and good situations. Symptoms of a current bad situation are compared to the database to infer cause and solution.

Symptoms are post-processed event traces, which should not necessarily reflect actual execution sequences but be useful as a stable key into the database.

Yuan et al. discuss the problem of finding the right abstraction for events. High levels are to be preferred as they usually convey more semantic information. Yuan et al. chose to trace system-call–level events, also to allow easy transferring of their approach to other systems.

Usually, raw traces are very redundant and contain irrelevant details (temporary file names, machine names, handle numbers, actual ordering of thread execution, etc.). Therefore, the authors describe how traces are filtered and canonicalized, events are segmented, reordered, and sorted to form stable symptoms.

Performance debugging

Aguilera et al. discuss *Performance Debugging for Distributed Systems of Black Boxes* in [AMW+03]. All components are treated as black boxes and only messages between

components are observed (also message content is not inspected). All communication is modeled as a graph with nodes where nodes may also be independent machines. Aguilera et al. want to support RPC style and message based communication. The goal is to identify causes of latency in distributed systems. Causal paths are a key abstraction for identifying high-impact call sequences and nodes that add unusual delay to requests. Traces are collected in the online phase of the approach, whereas causality is inferred statistically by observing the timing of messages in the system in an offline phase. Aguilera et al. discuss different tracing approaches for different target systems. At network level they use either dedicated trace capture hardware or packet sniffing at each host. In J2EE systems middleware instrumentation is used. The authors also briefly discuss kernel and application level instrumentation, and conclude with an extensive evaluation of accuracy and overhead of their approach.

The article demonstrates that event traces at different levels may be required, depending on the granularity of the analysis to be done. Furthermore, even without much semantic information (black box approach), useful results can be obtained.

Path-based

In [CAK+04] paths also are a central abstraction. Chen et al. target large distributed systems and treat single components as black boxes. Again, paths track user requests through a system. Therefore, paths are explicitly tagged with a request ID that travels through the system. In contrast to, for example, Magpie many software packages need to be modified one way or the other to generate or transport this ID. Main goals of the work are failure management and evolution.

Chen et al. describe how performance and correctness deviations can be found via statistical significance tests. Structure deviations as well as latency deviations may show partial failure or overload. Also regression testing against newer versions of the system is discussed.

Whodunit

Transactions play a central role in *Whodunit: Transactional Profiling for Multi-Tier Applications* [CCZ07]. They closely resemble requests as discussed in Section 2.3.1. Chanda et al. discuss how transactions can be followed through different layers in distributed systems for different communication types in order to enable performance debugging. The communication types are: shared memory, events, stages, and interprocess communication (IPC). *Event* and *stages* are handled by instrumenting the respective handler libraries. The profiler is implemented as a library that is linked to the to-be observed programs. It uses statistical profiling from the csprof package. *IPC* is handled by wrapping send and receive operations for messages. The current transaction state's synopsis (4 byte) is piggybacked on application data in the send wrapper and extracted on the receiving side. For *shared memory* communication, the authors make the following restricting assumptions. Whodunit recognizes critical sections only if they use locks or mutexes for protection. Instructions inside critical sections are emulated using the processor model from the QEMU project [Bel05] (without MMU). Also, a producer–consumer relationship is assumed.

Chanda et al. define transaction crosstalk by observing blocking times in exclusive locks held by other transactions. They also briefly compare their work to Magpie [BDIM04] and conclude that their approach does not require expert knowledge on application structure and for schema writing. This simplification, however, stems at least in part from their restrictive assumptions on application types.

In [GKM82] Graham et al. describe *Gprof: A call graph execution profiler*. The paper defines three different types of call graphs: complete call graphs, static, and dynamic ones, where the later two are subsets of the first one. The dynamic call graphs, targeted by Gprof, can either be obtained by sampling or by instrumenting function entries and exits to emit events. Both approaches are used by Gprof. The paper furthermore describes how time samples are merged to graphs and discusses result display problems. In Section 2.1.1, I described how gcc's function-instrumentation feature can be used to obtain similar effects. Combined with an event-transportation mechanism that is available across different system layers, I can generate dynamic call graphs in all those layers. I describe such an experiment for a device driver in Section 5.1.3.1. gprof

Feedback scheduling as described in [SGG+99] is also a typical user of a monitoring system. Steere et al. propose *symbiotic interfaces*, which expose a metric of progress and the roles of participating resource users to the kernel. In producer–consumer problems, for example, the shared buffer could implement such an interface. Its fill level determines whether the producer or the consumer need more processor time. I already discussed the Active Harmony project [HK99,cCH02] in Section 2.2 as a typical representative of this class. Feedback scheduling

A kernel profiler written with *KernInst* [TM99] does not expose single system events but knows the following basic sensor types: basic counters, cycle timers, accumulators, average-over-time accumulators, and virtual timers (virtual timers can be used to determine processor-time demand for tasks in multitasking environments). KernInst

In contrast to approaches as [BDS07b] and similar to KernInst discussed above, events in *DTrace* are filtered first, before being logged. Therefore, probes are written in a dedicated language called D and are executed in a specialized virtual machine in the kernel, guaranteeing safe execution. That way, every (authorized) user on a system can trace the system without affecting system stability. Cantrill et al. did not solve the halting problem for user supplied trace code, but restricted D not to be Turing complete (no backward branches and no loops are supported). D provides powerful aggregation and predicates to filter out as much tracing data as possible early on. DTrace

I introduced *METRIC's* [Mcd77] transport layer in Section 2.2. Evaluation of data is a two step process in METRIC. *Accountants* typically reside on a distinct physical machine from the object systems. They record, filter, or drop events delivered to them individually or in large packets. Apart from filtering, they only store traces to log files. *Analysts* may again be located on separate physical machines and process these log files. Analysts themselves are layered again. The lowest kernel level traverses log files in sequential order and tracks sequencing problems. Above that, a utility layer provides language and operating specific functionality. The analyst's highest layer is the application layer, which is completely problem specific. For METRIC, single event are assumed to be of little value, and can therefore be lost without big problems. This stems from the fact that mostly statistical evaluation is targeted. METRIC

In [Cag06,Cag07] Cagney introduces *Frysk*, an user-space, real-time (online), distributed debugger. Frysk is designed to be always enabled and is interesting because it combines information and events from various sources (ptrace, utrace, proc file system, ...). Frysk

Knüpfe et al. describe *The Vampir Performance Analysis Tool-Set* in [KBD+08], a Vampir

tracing solution aimed at high-performance computing (HPC). The two main parts are the open-source VampirTrace for actually obtaining the traces and the visualization and evaluation part that is covered by the commercial Vampir. The main focus of the tool set is to help with the optimization of performance and throughput of highly parallel applications. Therefore, their dynamic runtime behavior is investigated. Instrumentation is realized at source-code level using four complementing approaches: A compiler wrapper is used for function level instrumentation (also cf. to Section 2.1.1 on page 9 and Section 5.1.3.1), source-to-source instrumentation with a preprocessor targets OpenMP parts, pre-instrumented libraries cover MPI messages, and manual instrumentation directly using the VampirTrace API covers the rest (cf. to Section 2.3.1).

Timestamps are either assumed to be globally synchronized or a timestamp exchange is done at the start and end of the tracing period with linear interpolation in-between. There is additional support for tracing hardware performance counters, memory usage, and Posix IO.

The single events are first captured in memory and are later flushed to disk, independently for each process. In Section 5.3.3.1, I briefly compare the overhead for tracing single events with Ferret and VampirTrace.

The Vampir tool set aims at a very different problem space than Ferret — highly parallel high-performance applications. Consequently, acquired traces may be huge and their evaluation needs to scale well. The graphical user interface (GUI) of Vampir itself is built as a client-server application. Most evaluation computation is done where the data lies: in the High-performance computing cluster itself. This allows interactive evaluation with large traces, including different timeline views and statistical evaluations.

2.4 Miscellanea

This section deals with various smaller aspects not fitting any of the previous three larger categories. I start with discussing sampling approaches, followed by problems specific to distributed systems (Subsection 2.4.2). A discussion of stable application binary interfaces (ABI) of events follows in Subsection 2.4.3, ensued by considerations about event and timestamp sizes (Subsection 2.4.4). This section concludes with a literature survey on probe effect, intrusiveness, and monitoring overhead (Subsection 2.4.5).

2.4.1 Sampling techniques

Sampling techniques are related to runtime monitoring and are typically used for system-wide profiling, that is, identification of hotspots and optimization candidates. Correct resource accounting or tracing of causality flows is however *not* easily possible as only statistical results are obtained. I describe two projects in the following.

OProfile

The projects discussed in the previous sections often rely on instrumenting relevant system parts, such that important events can be traced. If, however, only statistical guarantees are required sampling approaches can show their strength. *OProfile* [OPr08] for Linux or its predecessor described in [ABD+97] do not require that system parts are instrumented, instead the system is inspected at certain intervals. Intervals can be regular or random, or even based on resources other than time by programming

processor performance counters to trigger interrupts (e. g., after a certain number of cache misses). By adjusting the inspection interval, one can trade statistical accuracy against overall overhead.

Why slowdown?

Basu et al. try to answer the question *Why Did My PC Suddenly Slow Down?* in [BDS07a]. They therefore periodically sample Windows performance counters for each process and build models based on historical data. Only sampling the performance counters instead of recording full event traces greatly reduces the amount of data to be handled additionally by the system. The paper contains a case study of Windows client machines. Users are given a simple feedback channel. They can flag *points of frustration* with a single keystroke. In case of a frustration event current processes are shown ranked by their anomaly (their current behavior compared to their model). This often allows to identify the anomaly-causing process(es).

2.4.2 Distributed systems

Distributed systems pose special problems to tracing approaches. Following traces across machine and maybe administration domains is one aspect, not having a globally synchronized time base is another. Also, the sheer amount of data to be evaluated may be problematic. The following short literature survey points out solutions to these problems.

In [PT04] Price et al. describe how relocating an originally distributed setup onto a set of virtual machines hosted on one physical machine solved the time-base problem. While moving an installation into a virtual machine surely can be seen as intrusive, the described approach still elegantly solves the time-base problem for some application classes.

Fonseca et al. discuss distributed requests in [FPK+07]. The authors target highly distributed applications across several administration domains and layered requests (tasks). All task IDs are transported inline inside application data and at protocol crossing points *next* and *down* operations must be implemented for connecting sub-requests.

In *WebMon: A Performance Profiler for Web Transactions* [GEGW02] Gschwind et al. do not start instrumentation in client browsers and try to dig into servers. Instead they deliver instrumented web-pages to clients, which themselves contact dedicated sensor web servers. This information is correlated with normal instrumentation on the server side to allow an over-all view.

I already discussed the black box approach from [CAK+04] in Section 2.3.3 above.

Interpreting the Data: Parallel Analysis with Sawzall [PDGQ05] deals with distributed evaluation of collected data. For the Sawzall language to be able to analyze data massively parallel, query operations need to be commutative and associative.

2.4.3 Stable event ABIs

The stability of application binary interfaces (ABIs) of events is relevant for widely deployed systems much for the same reasons as stable ABIs are desired for operating-system interfaces or libraries. If they change, binary compatibility with the installed software base is broken.

To hide implementation details of traced components (e. g., memory layout of structures in the kernel) *DTrace* uses the concept of translators. Translators can provide implementation neutrally structured information and can be used to provide a stable ABI across internal changes in traced components.

In *ETW* this problem is solved by providing a versioned ABI, that is, each event contains not only its type identifier but also a version number. If updated components change the information that events provide or the internal layout, event consumers can detect this and also use updated layout descriptions for unpacking event content.

For Ferret, I follow the ETW approach as it incurs minimal runtime costs. Of course additionally, arbitrary translations can be applied offline to stored trace files.

2.4.4 Event size and timestamp size

Different projects make different trade-off decisions about event sizes, what should be stored in single events, and with what precision.

Relay supports variably sized events and does not require for each event to be timestamped. Only the beginning of each sub-buffer must be timestamped.

LTTng, which now uses relay for event transport, uses its own timestamps based on processor timestamp counters (TSC). It timestamps each event and has support for wraparound protection on architectures that only support 32 bit timestamps.

In *ETW* each event header contains a *16-byte-long* globally unique identifier (GUID) that denotes its class. Each header additionally contains the event type, the version, and a timestamp.

In [BDS07b] Bligh et al. aim at a typical event size of only 8 bytes, where the first 4 bytes are used for type *and* timestamp information. The remainder is application-specific payload. The timestamp is still based on a 64 bit processor TSC, but its 10 lowest and its 27 highest bits are discarded. This leaves 5 bits for type information and 27 bits for timestamps. To deal with rollovers and frequency changes, regularly, events with full timestamps and current frequency information are logged.

2.4.5 Probe effect, intrusiveness, and monitoring overhead

Defining Probe Effect, Heisenberg

A central problem for monitoring is that the target system is changed by attempting to measure its properties. In the literature this effect is called *probe effect* [Gai85,Sch91, Sch93] or *Heisenberg uncertainty* principle applied to software [MH89,LDSP85].

How to handle

Schütz names four ways to approach this problem in [Sch91]: *(a)* ignoring, *(b)* minimizing by "sufficiently" efficient monitoring operations, *(c)* avoiding (*(c1)* by hiding behind logical time [Lam78] or *(c2)* by leaving the monitoring infrastructure in production systems — by making it part of the system [Tha00]), or *(d)* dedicated hardware. He concludes that the first two approaches *(a)* and *(b)* are not suitable for real-time problems. Avoiding by using logical time *(c1)* only works for a restricted subset of problems where only the order of events matters but not their real-time distances. Dedicated hardware *(d)* is expensive and inflexible. I will therefore not pursue this approach for this work. The remaining approach is integrating the monitoring infrastructure with the real-time system also in productive operation *(c2)*. For this work, I will additionally pay

special attention to the minimizing approach as parts of my envisioned target system are non–real-time systems with performance requirements.

In [Sch91] Schütz furthermore distinguishes between *intrusive* [MLCS89] and *interfering* [TFCB90] monitoring.

Intrusiveness

[MLCS89] defines five conditions for *intrusiveness*: incorrect results, creation (or masking) of deadlocks by event order changes, deadline misses for real-time programs, drastic execution time increment, and the impossibility of debugging parallel programs. Intrusiveness is a gradual property whose acceptance depends on the application actually monitored and desired monitoring results and is present if any of the above conditions is intolerable. Marinescu et al. propose to decrease intrusion level by augmenting hardware support. They present the event-action paradigm upon which they formally define normal active processes and *re*active processes (monitors). Based on this definition, systems can be partitioned into a target system $\mathcal{T}$ and a monitoring system $\mathcal{M}$. Intrusiveness is based on the actions the monitoring system does and information flow. Three different communication types between the target system and the monitoring system are identified: unidirectional, unidirectional with acknowledgments (where the monitoring system may delay the target system), and truly bidirectional communication. Interestingly, Marinescu et al. state that "If the information flow between $\mathcal{T}$ and $\mathcal{M}$ is truly bidirectional, intrusive actions can occur." Obviously, also pure delaying, as seen with unidirectional communication with acknowledgments, may change a target system's behavior. The authors also present layered architecture models, one for an intrusive setup using software and another nonintrusive hardware setup.

Interference

In [TFCB90] Tsai et al. describe *A Noninterference Monitoring and Replay Mechanism for Real-Time Software Testing and Debugging*. The authors follow an extremely strict interpretation of *noninterference*, that is, the execution of the target system shall not be modified at all (not a single processor cycle). They therefore describe a hardware solution (based on an MC68000 processor) to capture the target system's execution trace. This trace is then evaluated offline.

Avoidance

In [WSG96] Wu et al. discuss *On-line Avoidance of the Intrusive Effects of Monitoring on Runtime Scheduling Decisions*. The key technique is modifying scheduler decisions to compensate for delays introduced by monitoring. Time as a resource is virtualized with a set of modules that wrap access to local time, the scheduler, and message handling. Other resource types (input-output devices, caches, etc.) or real-time are not handled.

Minimize

Gupta et al. target the special problem of message pool ordering in *Dynamic Techniques for Minimizing the Intrusive Effect of Monitoring Actions* [GS95]. The authors assume a distributed system with message communication. Monitoring intrusion manifests itself then by altering the message order in receive queues compared to undisturbed systems. Gupta et al. use a globally synchronized clock and delay specific processes in the system to stabilize execution order.

2.5 Terminology

The diverse literature reviewed in this chapter uses an inconsistent terminology. In the following, I will introduce some key terms used throughout the remainder of this thesis

and will relate these terms to concepts from the literature with a focus on [Bir08,Poh08, Rie05,MLCS89,Micb,CSL04,Mcd77,TIoEEE].

In a typical monitoring setup there are three roles in the system: the monitored role, the monitoring role, and a controlling role, which steers many monitoring aspects.

- The entity to be observed fulfills the monitored role. I will call this entity *monitored process*. It is typically instrumented to produce events and emit them through sensors.

  In the *POSIX Trace standard 1003.1q* (PTS) [TIoEEE] the term *traced process* is used, in [MLCS89] the entity is called *target system*, and in the Event Tracing for Windows context it is called *event provider*.

- The monitoring role is held by an observing entity that typically consumes abovementioned events. In the context of this work it is called *monitor* or *event consumer*.

  In PTS it is called *analyser process*, in ETW *consumer*. [Mcd77] actually splits this role into the *accountant* and the *analyst*.

- The controller role is the most diverse role in that its exact task varies much in the literature. For Ferret, the *sensor directory* fulfills this role. Its purpose is providing a namespace for sensors and determining sensors' lifetimes.

  ETW uses the name *controller*. In PTS, this role is called *trace controller process*. It sets up streams (which roughly correspond to Ferret's sensors), manages their live cycle, and installs filters.

Events are used almost universally as an indication for a state transition in the literature. They are typically assumed to occur instantaneously and have properties associated with them. Events need timestamps in one form or another, either directly by making a timestamp part of the event or implicitly by deriving a (logical) timestamp from the event order in a buffer (e. g., relay only requires timestamps for the first events in a subbuffer). Events often have further properties, such as identifiers for determining their type and origin (e. g., provider, type, and version number in ETW) and application specific data. Therefore, event size requirements are often not constant.

Events need to be stored and transported to a consumer. Ferret uses shared memory regions called *sensors* for that purpose. Relay calls them *channels*. Fiasco (the mircokernel upon which I implemented Ferret) has a built-in *tracebuffer* [Wei03] that can be accessed as a special sensor with Ferret. The term *tracebuffer* is also used for Linux [Bir08]. ETW knows the *session* concept. It describes the currently activated event sources for a consumer. A persistent event stream, for example, stored on disk, is often called *trace file*, *trace log*, or *event log*.

Events are often created by monitoring code inserted at *instrumentation points* in the monitored processes. SystemTap calls these points *probe points*, [BDS07b,AAD+02] call them *trace points*. The Linux Kernel Marker project creates *static markers* meant as instrumentation location. [BH00] calls them *points* and in AOP they are referred to as *join points*.

2.6 Summary

After surveying the extensive related work, the individual highlights and shortcomings, there is no solution that solves all relevant problems for *runtime monitoring for open real-time systems*, that is, systems that meld varying requirements (including real-time) in a single system.

Of course, the approach must support the *real-time* case. In particular this requirement implies that instrumentation, event transport, and evaluation must be real-time capable. A guiding principle should therefore be to prioritize the unhindered execution of the monitored system over monitoring itself.

- For the *instrumentation* to be real-time capable two things must hold: *First*, the instrumentation mechanism itself must be real-time capable (upper bounds for execution must be determinable). Static instrumentation and static markers lend itself for this task due to their simplicity. Also the trapping approach may be applicable if the trapping path through the handling kernel is predictable. Static or dynamic recompilation are problematic. Dynamic recompilation introduces an unknown dynamic overhead. Both approaches modify the binary in opaque ways that may or may not retain execution-time characteristics. Further research is required in this area. *Second*, to avoid the probe effect for hard–real-time systems, monitoring must be made part of the system and must be included also in the production version [Tha00]. Static instrumentation lends itself for these system parts. Other parts of the system can use arbitrary other, more appropriate instrumentation techniques (e. g., kProbes for the Linux kernel, cf. to Section 5.1.2.4). The actual probe code itself must, of course, also be real-time capable. For DTrace, for example, the language enforces eventual termination of probe functions (e. g., no loops). However, several probe functions are executed with interrupts disabled inside a dedicated virtual machine in the kernel for a probe point, which may increase the system's interrupt latency.

- *Event transportation* also must offer predictable execution and prevent blocking to be real-time capable. Additionally, events must become visible for consumers in bounded time (e. g., in contrast to ETW's block-delivery approach, where slow producers may delay event delivery arbitrarily). Transportation also should not use system mechanisms that influence scheduling (e. g., IPC). Relay, for example, has a slow path and a fast path for event creation and potentially invokes callback function, making it unpredictable. A good example is LTTng's user-space–only event-transportation mechanism, which uses shared memory without any blocking synchronization.

- Real-time *evaluation* basically requires bounded access time from event delivery and some form of real-time scheduling, depending on the application-specific evaluation. Offline evaluation basically only requires "enough" memory to temporarily store events (until they are dumped to disk, over network, etc.) or a sufficiently fast transport channel to persistent storage.

For today's desktop real-time systems a split system architecture is popular [HBB+98, YB97,MDP00,J. 00]. The time sharing part runs side-by-side with the real-time part on top of a microkernel or similar mechanisms and is also a monitoring target. For these system parts monitoring *overhead* and therefore an efficient monitoring solution is important. In contrast to approaches such as DTrace or ETW, where a majority of instrumentation points reside in-kernel, here the overbearing majority is located outside of the (micro-)kernel. DTrace and ETW, for example, support user-space code with traps or explicit kernel entries only. This design is too expensive for the scenario described above. K42, allows direct memory access to the system-wide sensor without kernel entries. Also LTTng supports tracing in user space without kernel entries.

I believe that data *integrity* for monitoring data is of utmost importance, that is, if not all events can be captured this must be detectable. Also data corruption must at least be constrained to domains using single sensors. K42 compromises both properties for performance. DTrace, on the other hand, provides mechanisms for detecting lost data due to buffer overflow. It also guarantees the correctness of all received data.

Safety (i. e., system stability) for shared-memory–only systems is addressed by [Stö05]. Basically, all writable data must not be trusted and code must check bounds.

Security (i. e., confidentiality) requires that no additional information flow channels are established by monitoring than are explicitly required for monitoring (i. e., strictly from monitored processes to monitors). In K42, all tracing-enabled processes have full access to one global sensor. This basically connects everybody and is the opposite of what is required. Relay primarily addresses various producers in the Linux kernel, which are not separated from each other with address spaces anyway. However, for the consumer side, each channel is represented by a single file such that access for user-space consumers can be controlled with fine granularity. [Stö05] has the concept of accounting domains which use independent memory regions.

Restricting information flow not only enhances security but also the *probe effect* is reduced per design as monitored processes cannot be influenced by monitors. It is, however, immediately clear that a software-only solution (which is preferable for cost and flexibility reasons) will always have a nonzero probe effect (unless monitoring is *defined* to be system part). The best one can achieve is reducing the effect. In ETW, for example, consumers are notified synchronously when event buffers are full, which directly couples event production and potential scheduling of consumers. Such a coupling is to be avoided.

I did not design Ferret with only few use cases in mind, I wanted it to be *flexible*. More specifically, several monitors at the same time, potentially attached to the same sensors, shall be supported (cf. to relay and METRIC). Ferret shall be event-layout–agnostic, that is, its requirement on the layout shall be minimal. Relay's design points into a similar direction, as event creation in relay enforces only minimal structure. The architecture in [Stö05], on the other hand, is designed with a very specific type of consumer in mind, a user-space scheduler. For high–event-count setups, Ferret shall also support specialized sensors that aggregate data online (cf. to DTrace). Dedicated specialized hardware for monitoring is inflexible and expensive, I want Ferret to be a software-only approach.

The environment at which I target FERRET is diverse and comprises *different software layers* (microkernel, basic resource servers, convenience environment, para-virtualized time-sharing kernel, time-sharing user-space programs). FERRET shall be usable in all those layers similarly, therefore, minimal API requirements and simplicity are a must. LTTng, for example, provides two distinct approaches for kernel and user-space tracing. Recent work [DD08] shows that LTTng also aims into this direction (Xen hypervisor, markers for user space).

Minimality in the kernel is also a guiding principle for microkernel construction, which I try to follow with FERRET's design.

In the next chapter I will describe how I approach the requirements and problems discussed above.

Chapter 3

Design

Everything should be made as simple as possible, but not simpler.

(A. Einstein)

In this chapter, I introduce the systems that I target with my monitoring approach and derive requirements that a monitoring system shall fulfill. I will broaden the discussion of requirements when necessary in the next sections. In Section 3.2, I present the architecture of my monitoring system and discuss design decisions.

3.1 Target systems and requirements

A description of the target system types for my work starts this section. Following that, in Section 3.1.2, I explain properties that FERRET shall possess, and requirements to underlying layers (operating system, hardware).

3.1.1 Target systems

In the following, I describe the intended target system starting with high-level system properties and followed by a more detailed discussion.

There has been a trend in the last years to merge classical desktop operating systems and embedded operating systems. Desktop operating systems on the one hand are increasingly used in classical embedded and real-time domains (for example, factory automation and in hand-held devices or mobile phones), and are enhanced to cope with corresponding requirements. On the other hand, embedded systems, for example mobile devices, provide more and more desktop-like functionality [GS06,CRSBPC06,AAP06, XlRsT+06]. Consequently, many contradictory requirements have to be conjunct in one system.

In the following I describe a two-split system architecture approach that, I think, is common for such systems. The description details concentrate on the Dresden Real-Time Operating System (DROPS) that I used for my research, but other systems (RT-Linux, RTAI, J-Consortium Real-Time Core Extension (RTCore)) are similarly structured.

DROPS is split into two parts [HBB+98,HZP+07a,APPZ04,GPPZ04b]:

- One legacy container that basically serves reuse purposes for software parts without special requirements (i. e., real-time or security requirements), and

- One small and specially constructed part, which runs important software components. Depending on the scenario, these might have specific real-time requirements or particular security or safety requirements.

Target systems summary I will now describe a typical computer system that I target FERRET at: I assume a desktop-class system that has memory protection and that may be used in real-time contexts. FERRET's approach is not limited to such a system, but my implementation currently is. As described previously, real-time components are run in a dedicated system part but FERRET shall also be usable in pure real-time systems, that is, systems with no legacy parts. I furthermore presume an open system, that is, a system that is not fully under tight control of one software vendor and that is running software from independent (potentially user-defined) sources. Consequently, software components must be isolated from each other and monitoring must not interfere with the isolation. Monitoring in closed systems is more or less trivial as every part in the system can trust every other part. Currently, FERRET's research prototype implementation is limited to x86 systems running DROPS or Linux.

3.1.2 Properties and requirements

In the following, I will discuss properties that I want FERRET to possess. These properties are derived from three sources: First, the target-system discussion in the previous section, second, general requirements identified in the literature [AAD+02,EPC+05, BDS07b,ALR01] (also cf. to Chapter 2), and third, the requirements derived from my own practical experiments, some of which I discuss in the evaluation chapter in Section 5.1.

The target systems described in Section 3.1.1 typically consist of many heterogeneous software components, including virtual machines and normal time-sharing software inside them. Also, I want to support scenarios where real-time and non–real-time components interact (e. g., as in DROPS [HBB+98]). Therefore, monitoring all types of components and their interactions must be supported.

With FERRET, I aim at a software-only solution, that is, I have no additional hardware devices that snoop memory busses, processor cycles, or unusual high-precision timers, because I want FERRET to be easily usable on standard machines.

3.1.2.1 High-level requirements

I start with a high-level discussion and will refine the properties afterwards. Following that, I derive requirements to the hardware and the operating system.

- FERRET instrumentation and monitoring shall be *minimally invasive*, that is, it must not interfere with real-time properties for real-time parts, it must not have a severe influence on performance for non–real-time software, and system correctness must be preserved. More specifically:
  - I want FERRET to provide monitoring of real-time parts while maintaining the *real-time properties*, that is, FERRET must never block on unbounded external conditions and its execution time must be bounded.

  - FERRET shall also be usable for the non–real-time parts, potentially in high-throughput locations. Here, good *best-effort performance* is required.
  - FERRET's monitoring code must be robust in order not to introduce new bugs and monitoring itself must not interfere with observed causality in the system (*correctness*).

- The *open-system nature* requires several properties:
  - First, *isolation* between monitoring components. *Monitors* shall not influence *monitored* system parts in undefined ways and monitoring infrastructure must not introduce new covert channels. If information flow is strictly limited to originate in *monitored* code and to target *monitors*, influencing monitored parts by monitoring parts is precluded by design.
  - I cannot make assumptions about the structure of code to be instrumented in an open system, for example, I cannot exclude the usage of user-level threading, signals, and other asynchronous interruptions. Consequently, sensor code must be *unboundedly reentrant*.
  - Naturally, several *independent online evaluations* must be supported by FERRET, as independent system parts may want to use monitoring.

3.1.2.2 Low-level requirements

After outlining the property categories above and sketching a high-level idea for each point, I provide a refined discussion in the following. I use the same hierarchy as in the foregoing text:

- *Minimally invasive:*
  - *Real-time properties:*
    - Event creation must be possible in real-time, there must be an upper bound for it. This property is required for maintaining potential real-time properties of monitored system parts.
    - I regard real-time properties of event consumers as secondary, compared to those of producers, as consumers do not belong to the observed system itself. Maintaining real-time properties is more important than being able to observe the system in real-time in critical situations.

      Of course, depending on the actual scenario, additional and more stringent requirements may arise.
  - *Best-effort performance:*
    - Event creation, transport, and delivery should work very efficiently, in terms of processor time and memory requirements. Monitoring code will potentially be used in many hot code paths, consequently is a high-performance implementation considered important.

  - This does not only apply to the monitoring code, but also to the event representation in memory — a compact binary representation is desirable.
  - Furthermore, particular experiments may require customized sensors (e. g., histograms or counters) to further reduce the overhead.

- *Correctness:*
  - Using monitoring in a system should not create new causal relationships in a system, as would the usage of system messages and would synchronous notification of monitors in case events became available.

    It is clear that a pure software approach cannot be perfect in avoiding all causal relations between monitored and monitoring system parts as common resources such as system memory, bus bandwidth, and processor cycles are used. My goal here is to reduce the causal interference to a level of an independent and small background service — basically background noise.

    I use two construction methods to achieve this: *First*, I try to *minimize non-explicit communication* in the form of resource sharing, by only using very low-level system resources, such as processor cycles and memory, but not typical higher-level operating-system resources, such as file systems, semaphores, or device-driver services in Ferret.

    Second, I try to *reduce explicit communication* between monitored and monitoring system parts. For example, events are not *consumed* but are just *evaluated* by monitors, there is no channel for a monitor to signal event consumption to a monitored component. Event memory is finally reused by the *monitored* parts on buffer wraparound. Monitors detect new events by polling some time after they become available, that is, time triggered. I define these types of communication relations to only transport *weak causality*. In contrast to that, communication relations visible in the operating system, for example, direct IPC messages or threads blocking on a shared lock, are regarded as *strong causal relations* in the context of this work.
  - A correct and robust implementation of the monitoring code is important not only for ensuring the stability of the monitored system, but also for ensuring the confidence in obtained results. Furthermore, it simplifies post-processing as plausibility checks can be kept simple. (In contrast, for K42, guaranteeing correct traces was not deemed very important, cf. to Section 2.2 on page 13.)

- The infrastructure used for monitoring should be as compact as possible, to allow monitoring all system parts and layers (kernel, libraries, system servers, and applications) as simple and consistent as possible. In particular, this means that no API can be used of system parts that shall be observed.

  - Having few and small dependencies does not only allow widespread use but also *early* use after system start, when not all system services are yet available.

- *Open-system nature:*
  - *Isolation:*
    * Event consumers must not be able to influence event producers in an uncontrolled way. This can be achieved by allowing only read access to sensors for event consumers.
    * Event consumers must not trust data from event producers and must be protected against corrupt metadata in sensors.
    * Event producers must not be able to influence other event producers in an uncontrolled way. This can be achieved by using independent sensors.
    * Event consumers must also be protected from other event consumers. Again, only allowing read access to sensors achieves this goal.
    * Consequently, multiple independent sensors must be supported by Ferret (while maintaining global event order). In addition to above-mentioned safety and security reasons, this feature also eases the usage of Ferret by independent software components.
  - *Unboundedly reentrant event creation paths* and *independent online evaluations*, that is, several independent event producers and event consumers must be supported by Ferret:
    * On the producer side, tasks can be multithreaded or use other means of concurrency.
    * Sensors may also be used by several independent components in the system for event production.
    * On the consumer side, several independent evaluations may access a sensor concurrently.

- Finally, Ferret must also provide *useful* results:
  - Event order must be determinable for events from independent sensors. For cross-processor ordering of events, Ferret relies on timestamp synchronization between processors (cf. to Section 3.2.3.2 for details on timestamp sources).
  - Timestamp accuracy and resolution must allow measuring durations.
  - Ferret must be able to transport a sensible amount of payload to support various use cases.

  Purely using timestamps, events must be sortable for a single time source domain.

3.1.2.3 Requirements to hardware and operating system

In the following, I will briefly summarize the properties that an operating system and the underlying hardware has to possess to support Ferret:

- I require the hardware and the operating system to support shared memory between processes to establish sensors. I discuss the role of shared memory in more detail in Section 3.2.1.1 on page 39.
- Either the operating system or the hardware must support a form of efficient atomic sections to allow timely publishing of timestamped events in sensors. I provide a detailed discussion about the problem in Section 3.2.4.2.
- Either the operating system or the hardware needs to provide low-overhead, accurate timestamps. A discussion of alternatives is given in Section 3.2.3.2.
- Storage space for offline evaluation of traces must be available, either in memory, on disk, or on a remote machine using a network.
- For online evaluation, monitors must be scheduled on system resources. If monitoring shall possess real-time capabilities, monitors' execution must be planned to be part of the system schedule and the system must provide real-time support to monitors.

3.2 Design decisions and architecture

After having laid the grounds for the design of Ferret in the preceeding sections, I introduce its design and architecture in the following. To this end, this section contains three parts. I first describe the different roles of a monitored system and their interaction in the succeeding section. Following that, in Section 3.2.2, I discuss online and offline evaluation of monitoring data. I conclude this section with a detailed discussion of sensors, their types, timestamps, and different synchronization approaches in Sections 3.2.3 and 3.2.4.

While designing Ferret I tried to bring in-line general requirements for runtime monitoring systems mentioned in Section 3.1.2 as well as specific requirements derived from the real-time nature of the target system. If goals conflict, I prioritize the real-time requirements of the system to be monitored.

3.2.1 Roles and their interaction: Ferret's architecture

Based on the related work discussed in the previous chapter and the requirements discussed in Section 3.1.2, I identified three roles for a monitoring system: *monitored processes*, the *sensor directory*, and *monitors*.

I characterized their relation and naming to similar roles in the literature in Section 2.5. In the following, I define each role's tasks and behavior in more detail. A general overview of the interaction of the roles is shown in Figure 3.1.

Monitored processes With Ferret, the creation of sensors is triggered either by a monitored process or by a third party on behalf of the monitored process. In each case, the sensor itself is set up and configured in a shared-memory region by the sensor directory that then maps the region to the monitored process.

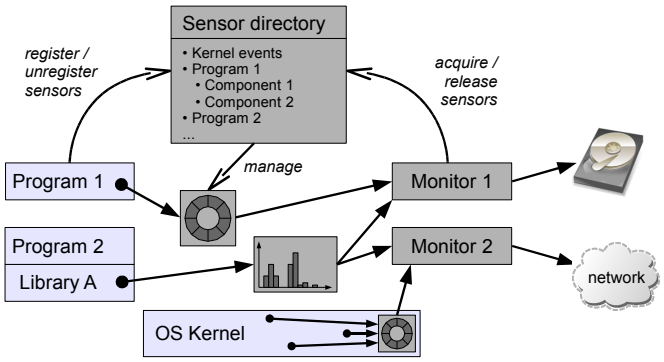


Figure 3.1: Architecture overview of FERRET: the sensor directory manages single sensors. Shown in the figure is a ringbuffer and a histogram sensor. Embedded FERRET code in programs and libraries registers sensors and monitors acquire access to sensors and process events or condensed information from sensors and store, transfer, or display their computation results.

The actual tracing of events happens by memory accesses into the sensor area from within the monitored process. Therefore, the monitored process uses small event-creation functions provided by FERRET.

FERRET comes with a set of common, predefined event types. Customized types can easily be defined by the user as well.

Sensor directory The FERRET sensor directory is responsible for creating and initializing new sensors, granting read-write access to secondary producers and read-only access to consumers, managing the sensor namespace, and the handling of instances. FERRET supports several instances of a monitored component running side by side independently, so, beside the sensor identifier, I use an additional instance identifier to address sensors. Closing of sensors and starting or stopping the whole framework is also handled by the sensor directory.

After setting up a sensor and granting access to producers and consumers, the sensor directory is not involved in the event flow at all. Filtering of relevant events can be done either in the producer, by just not posting certain events, or in the consumer by ignoring certain events. At a more coarse-grained level, filtering can also be realized by using specific sensors. FERRET allows for very fine-grained sensor usage, also across processes, such that events might be grouped thematically, rather than by source.

The sensor directory is the natural place for implementing an access-control policy at sensor granularity. The current research prototype demonstrates this with very simple checks that allow only creators of sensors to finally close them.

Monitors Monitors use the sensor directory as a directory service to find required sensors either via symbolic names (e. g., `/ferret/l4linux-kernel/syscalls`) or with well known numeric constants. After looking up sensors, monitors register themselves as consumers. The sensor directory gives them read-only access to the cor-

responding shared-memory region. These two mechanisms (the read-only access and the indirection in form of the sensor directory) prevent information flow back to monitored processes from the monitors.

After acquiring access to all required sensors, monitors can periodically check them for new events. Evaluation can happen either online or offline by storing event streams for later use. I further discuss this aspect in Section 3.2.2.

Monitored processes only communicate with the sensor directory to create or destroy sensors, never to post events. They never directly communicate with monitors (except by posting events to sensors, of course).

Monitors only interact with the sensor directory for subscribing to sensors or for releasing access to them. They also never directly communicate with monitored processes.

3.2.1.1 Discussion of specific design alternatives

In the following, I review several common design decisions seen in the literature (kernel implementation, synchronous notification, and when and where to execute sensor code) and justify my divergence from beaten paths. To conclude, I summarize my approach.

Kernel implementation Many systems [AAD+02,WR03,CSL04,Wei03] implement monitoring for user-space components with the help of system calls or kernel traps that care for sensor buffer management, taking timestamps, ordering, and atomic execution of monitoring code.

I have not chosen this approach for the following reasons:

1. It requires the additional overhead of a system call (kernel entry) per event access.
2. Specific components in the target systems cannot and must not directly interact with the microkernel using system calls (e. g., user-space programs inside a virtual machine).
3. Using kernel primitives for monitoring may simply be too intrusive as it could change the system's behavior that I currently want to observe (e. g., scheduling).
4. One fundamental principle for constructing microkernels is the minimality of the kernel — whatever can be implemented efficiently and securely in user space should be done so [Lie95,Lie96b].
5. Using a kernel implementation might be to unflexible, in case a new special-purpose sensor has to be added to Ferret. This addition can be done much more easily in user space (this is also indicated by driver-development experience in user space [McK04,Kan07]).

Synchronous notification Alternative approaches notify parts of the monitoring infrastructure synchronously on the occurrence of events [CSL04,Micb,Wei03].

I refrain from synchronously notifying monitors of the availability of new data or single events. This would be costly in the event-creation path and will create new causally strong communication relations that do not exist in the unmonitored system (cf. to Section 3.1.2.2 on page 34 for a discussion of causal relations). Also, the execution of the monitored program becomes dependent on the acceptance of those notifications by monitors, which is a problematic approach for real-time components.

A guiding principle for my design is, that if there are not enough resources to do both *monitoring* and *executing the actual system*, then the monitoring task should be restrained instead of compromising the system's original functioning. In case a non–real-time system is to be monitored and monitoring is of paramount importance there, the monitoring infrastructure itself can be run as a real-time task, ensuring its proper execution.

Sensor code For executing user-supplied probe code in a real-time scenario, not only the halting problem has to be solved or circumvented (the code will terminate *eventually*, cf. to Section 2.3.3 on page 21) but an upper bound for the execution time has to be determined[1]. Therefore, writing probe code is split into two steps in Ferret. The actual probes are implemented by a system developer. Real-time properties of this code can be determined offline. Administrators and users can only use supplied probes. However, monitors can run arbitrarily complex code written in any language. They are not restricted to a specific language (e. g., D for DTrace) and no special support is required for them in the kernel (i. e., no virtual machine for monitoring code is required in the kernel, as it is for DTrace [CSL04]).

My approach The core of my approach is storing event data in shared-memory buffers. Shared memory is such a common and low-level feature that it is available in all environments that I target. This design choice mostly eliminates the problem of different environments. Only in the monitoring setup phase, environment-specific code is run to establish sensor buffers. Instrumented real-time and non–real-time applications in all environments are able to post events by only using memory accesses.

This approach, however, opens up a set of interesting problems that I address in the context of this work.

- Monitoring must be *real-time capable*. In more details, this requirement means that involved memory pages must be pre-mapped and pinned to avoid page faults. Also, sensor access must not use any form of blocking locks for synchronization (otherwise arbitrary new blocking dependencies may arise in a system with shared sensors). Furthermore, there must be an upper time bound for accessing sensors, that is, executing sensor access code.

[1] The halting problem combined with a deadline can be solved in pseudo-polynomial time if the timing of the primitive operations is known. The supplied program snippet can be checked for all input-data combinations and all potential indeterminism possibilities in parallel. One can determine whether all, some, or none of the variants terminate before the given deadline.

- Monitoring should exert as *little influence* as possible on the monitored system. More concretely, this means that processor overhead for monitoring should be *as small as possible* for non–real-time parts of the system and *predictable* for real-time parts. The least amount of work required for monitoring should be done inside the system, consequently, the amount of code run inside the monitored system should be minimized. If something can be done outside the monitored system, as a post-processing step or in monitors, it should preferably be done there.
- For security reasons, *information flow* due to the monitoring system should only be directed from monitored programs to monitors. Monitors should not need to send any information back to monitored programs.
- Also for *safety and robustness* reasons, monitors should not be able to influence monitored programs in other ways than arbitrary background processes can. Furthermore, several monitored programs should not gain additional influence on each other through the monitoring system. This can be achieved by using several independent sensors. Still, access to sensors must be possible safely for multi-threaded programs.

3.2.2 Online evaluation, offline evaluation, and external schemata

FERRET is designed to support both online and offline evaluation. I will first describe several peculiarities of online evaluation and how FERRET handles them, followed by a discussion of offline evaluation and external event schemata. After that, I briefly discuss the controlling of the monitoring process.

3.2.2.1 Online evaluation

For online evaluation, monitors run concurrently to the to-be-observed system. While this may increase the load on a system, it also offers advantages: *First*, only online evaluation allows fast reaction to certain system states, such as error conditions. The system may be put into a special debug state to do potentially expensive inquiries for additional system-state information. *Second*, online evaluation may actually be less intrusive on the monitored system as data can be filtered early on and only relevant data need to be processed now or stored for later. *Third*, online processing may be the only option for long-running experiments, which generate large amounts of data that cannot be stored fast enough.

Monitor processes can be run as low-priority background tasks (if there are enough spare resources available) or can be integrated into the system as regular real-time load. However, monitors will never be notified synchronously by event producers or FERRET. Instead, they have to poll sensors. That way, no new strong causal relationships are introduced between the monitored system and the monitoring system (cf. to Section 3.1.2.2 on page 34 for a discussion of causal relations).

For real-time problems where event creation and event consumption can be modeled as jitter-constrained stream [Ham97], the required sensor buffer sizes can be calculated such that no events are lost [Rie05].

In general, online evaluation needs to use resources sparingly and work fast. Ferret supports this by allowing arbitrary monitoring code and by guaranteeing event delivery in order. Ferret also contains sensor types which already aggregate online such that buffer memory requirements are bounded (cf. to Section 3.2.3.1). Section 5.1 contains several example scenarios that use online evaluation.

3.2.2.2 Offline evaluation and schemata

In contrast to online evaluation, offline evaluation can use all the resources that a dedicated machine may offer. Additionally, offline processing does not need to run in real time. It particularly allows interactive trace inspection by humans. Furthermore, for offline evaluation the concrete evaluation procedures need not be known at instrumentation time or runtime of the system. They can be defined, refined, and adapted later as traces are in persistent storage and can be reevaluated.

A core abstraction for monitoring are requests. Requests are user-defined units of work, specific to a scenario. I discussed requests and their internal respectively external specifications in Section 2.3.1 in detail.

I briefly describe the Magpie toolchain [BDIM04,BIMN03] in the following as it has capabilities for finding, building, and post-processing requests. Originally, Magpie was built as a consumer for the Event Tracing for Windows framework only (cf. to Section 2.3.2 on page 19). I generalized Magpie to work with the monitoring system that I built for Singularity (Event Tracing for Singularity) and Ferret for Drops.

Magpie's means of synthesizing requests from event traces are schemata. Schemata are descriptions of how events relate to each other, what request boundaries are, which type of communication in a system belongs to a request, how certain parts of requests are stitched together, and so on. Once requests are available, Magpie can determine their resource usage, detect anomalous requests, and in general show communication patterns. I discuss several application examples in Sections 5.1.1.2, 5.1.2.3, and 5.1.2.4.

3.2.2.3 Controlling the monitoring

As Ferret is designed to cover a broad range of application scenarios, controlling it at runtime must be flexible. Situations range from interactive scenarios where sensor data is to be used by humans, for example, for visual inspections (cf. to Section 5.1.1.4), over experimental setups where a system needs to be traced for a short time (cf. to Section 5.1.1.1), to fully automatic scenarios where Ferret runs in the background (cf. to Section 5.1.2.2).

Ferret's sensor directory does not need any controlling at runtime. It provides the namespaces and acts on sensor register–unregister and acquire–release requests. The namespace can be traversed online, but the sensor directory is not involved in transporting events once consumer-producer relations are set up.

Controlling event production, for example, by dynamically instrumenting binaries, is not covered by Ferret itself, but is an orthogonal problem. In Section 5.1.2.4, I describe how Ferret can be used in combination with kProbes [Kri05] in the Linux or

L4Linux kernel. Also other dynamic instrumentation frameworks, such as Feathertrace [BA07] can easily be combined with Ferret.

What actually needs to be controlled are the monitors. I implemented several monitors for Ferret ranging from general purpose monitors with a GUI control (`merge_mon`, typically used for interactive debugging and recording traces) to very specialized monitors that only solve one particular problem and need little to no controlling, for example, `idle_switch_mon` (cf. to Section 5.1.2.1), `l4lx_verify_tamed_mon` (cf. to Section 5.1.2.2), and `l4lx_histo_mon` (a dedicated monitor for capturing Linux system-call histograms from within L4Linux).

I use the following three methods for controlling monitors in my use cases:

1. *Time triggered:* Monitors are simply preconfigured for a certain experiment and run for a limited amount of time and provide their results after that.

2. *Content triggered:* Monitors consume events from sensors and special content (e. g., violation of assertions) triggers actions (e. g., dumping of the current event history for offline analysis).

3. *Triggered interactively:* Monitors provide an interface for human operators to configure acquired sensors, start and stop consuming events, and dumping trace files to network or serial console. This interface must be appropriate for the situation, for example `merge_mon` provides a graphical interface for GUI setups, but can also be controlled via special control events through any sensor. That way, it can be controlled in headless setups or from inside virtual machines as well.

These three methods combined provide sufficient means of control for all the scenarios described in Section 5.1.

3.2.3 Sensors

In the following I motivate why different sensor types are required and introduce three forms. Additionally, I discuss timestamps, their purpose in the context of monitoring, variants, application areas, and sources.

3.2.3.1 A variety of sensor types

Although a one-size-fits-all sensor would be an attractive design goal, it is also an unrealistic one. The requirements regarding the type and amount of data to be monitored, the amount of post-processing required, and the degree of acceptable intrusiveness are simply to varied.

More concretely, from my experimental experience, I determined the following three main reasons for specialized sensors: *Simplicity*, *reduced overhead*, and *robustness*. Dedicated sensors may store their payload in a form close to the desired consumption format. This *simplifies* consumption, especially if online-evaluation is required. A *smaller overhead* may be achieved by direct data aggregation, compared to raw event traces, if the

aggregated form still contains enough information. More specifically, memory requirements for monitoring can be reduced drastically by direct data aggregation. Direct data aggregation may also eliminate the possibility of data loss due to buffer overflows. If only aggregated data is required, direct aggregation into a histogram is the most *robust* solution with regard to loss of monitoring data.

I discuss the usage of a two-dimensional histogram to reduce the load on the memory subsystem (which is a critical part of the to-be-observed system in the experiment) in Section 5.1.1.1. The experiment described in Section 5.1.1.4 illustrates various different sensor types and how they help in simplifying the usage of runtime monitoring. Furthermore, also in related work, powerful aggregation functions are supported (DTrace, cf. to Section 2.3.3 on page 21).

The selection of sensors discussed in the following is therefore a compromise between different requirements encountered in several sub-projects of DROPS over the last years and the urge for simplification and generalization: simple scalar counters, histograms, and general event sensors.

Scalar counters Scalars are almost the simplest form of a sensor (except for single bits). They basically only contain a single integer number as their data representation. FERRET allows directly writing into the scalar, adding to it, subtracting from it, and incrementing and decrementing it. The consumer side only uses a simple retrieve method. The scalar sensor is prepared to carry upper and lower limits and has room for storing the observed minimum and maximum, the write access count, and the accumulated sum (this allows online computation of the average). Scalars are, for example, used to count the number of deadline misses in an adaptation period in Verner (see Section 5.1.1.4).

Histograms Histograms are an extension to scalars with regard to dimensions, but they still aggregate data.

Histograms are useful in different flavors. The most simple form only contains a minimal amount of metadata (low and high bound, number of bins) and the bins themselves in the form of a linear array. For fast access to specific bins a precomputed multiplier is used. Also restrictions on the multiplier, the number, and the size of bins can be used to speed up the access to the histogram. Histograms can also be treated as an array of scalars by directly addressing the bins in the array.

Counting the underflows and overflows for the histogram allows adaption of histogram sizes for specific setups. FERRET histograms also allow tracking the count and the sum of inserted values (for fast computation of average values, also if underflows and overflows where observed). Additionally, storing the observed minimum and maximum helps in adapting the dimensioning of the sensor and to provide a first rough impression of the value distribution.

Amongst other things, histograms are used for acquiring the resource usage for various routines in the system, for example, the video and audio decoding steps in the video player Verner [Rie03]. The obtained numbers are directly used for processor-time reservation in Verner. The array mode of histograms has been used for determining

calling frequencies of system calls in L^{4}Linux in [Döb06]. I also used multi-layered and multi-dimensional histograms for specific problems (cf. to Section 5.1.1.1).

Event sensors Event sensors are the most general form of sensors. Each event is stored and timestamped individually. Thus, system behavior over time is traceable. Using event sensors is useful when early aggregation is not possible, for example, if the way to aggregate is yet unknown or if the time relations of events are important.

Requirements for event sensors stem from different areas: It should be possible to establish temporal order over events, also over events from different sensors. I will therefore discuss timestamps in more detail in Section 3.2.3.2. The event-buffer size of sensors should be configurable. It determines how many and how large events can be buffered before they have to be dropped. The buffer size is one determining factor for monitors' poll frequency. There are different event types that bring different amounts of payload and have different payload-layout requirements. Therefore, variable event sizes should be supported and minimal requirements on payload layout should be enforced. One sensor may be used from within different components at the same time, that is, contained events may originate from different sources. More generally, event sensors may be used by several producers and consumers at the same time.

I have defined a common event header denoting the origin and type of events, followed by custom data. I describe the data layout for the custom parts of all events in the Magpie toolchain (cf. to Figure 3.2).

Producers typically write single events, not too close in succession. Producers strictly use an append-only access pattern. They never look at or modify old events. Monitors may have arbitrary access patterns in the general case. However, I only used sequential access in my monitors. Both my state machines discussed in Section 5.1 and storing traces for offline evaluation only require sequential access. Therefore, event sensors can be treated as FIFO (first in first out) queues. There may be several writers on the producer side. On the consumer side, each participant has to track its own stream position. Also, consuming does not mean removing, as that would require monitor coordination in case of several monitors. It would also require write access for monitors. Instead, consuming basically means to get a copy of an event and to increment the own stream position. Events become unavailable when they are overwritten with new events. Monitors therefore also need a way to detect event loss.

3.2.3.2 Timestamps

Timestamps serve several purposes in the context of runtime monitoring. *First*, Ferret itself uses timestamps to deliver events in creation order. Order between events of a single sensor could be derived from the order that the events are put into the sensor, but if more than one sensor is involved (or if a sensor does not preserve insertion order, i. e., if it is implemented as a hash table and not as a list) timestamps are required. *Second*, timestamps can be used to deduce causal relations in traces. *Third*, for many monitoring purposes, real-time distances between events are required. This includes, for example, the verification of real-time timeouts and the computation of resource usage.

```
0 Kernel
#type context_switch 10
{
    context,     ItemULong
    eip,         ItemULong
    pmc1,        ItemULong
    pmc2,        ItemULong
    _pad0,       ItemChar
    _pad1,       ItemChar
    _pad2,       ItemChar
    _pad3,       ItemChar
    dest,        ItemULong
    dest_orig,   ItemULong
    kernel_ip,   ItemULong
    space,       ItemULong
    sched_cont,  ItemULong
    from_prio,   ItemULong
}
```

(a) Context-switch event

```
1 L4LXK
#type atomic_begin 2000
{
    l4tid, ItemULongLong
}

#type atomic_end1  2001
{
    l4tid, ItemULongLong
}

#type atomic_end2  2002
{
    l4tid, ItemULongLong
}
```

(b) Atomic blocks

Figure 3.2: Shown is the exemplary data layout description for *(a)* microkernel context-switch events with detailed information and *(b)* simple begin–end events for atomic sections in the L^{4}Linux kernel, containing just a thread ID. The first lines define the mapping from major event IDs to symbolic names. The `#type` lines establish mappings for minor event IDs.

I will discuss possible timestamp sources in the following, which can be grouped into *physical* and *logical* clocks.

In the remainder of this work I will only use physical clocks. The discussion of logical clocks and derived approaches in the following is meant for follow-on projects that want to transfer results of this work over to other domains with different requirements and conditions.

Physical clocks The main property of physical clocks is that they are related to the real world and consequently to real time. Their purpose is to deliver timestamps associated with events from the real world such that real-world durations can be tracked.

Important properties are the resolution of timestamps, the overhead to obtain timestamps, and the accuracy (how well synchronized to other clocks, drift). Furthermore important for programming time sources is how timestamps can be obtained.

Current desktop computers contain several hardware devices suitable for this purpose, which, however, have diverse qualities. [Mic02] gives a good overview about which time sources are suitable for current operating systems and about time-source limitations:

- The 8254 Programmable Interval Timer (PIT) has a resolution of 1 ms and is expensive to access.

- According to [Mic02], the Real-Time Clock (RTC) also has a resolution of 1 ms and is also expensive to access. A current RTC datasheet [DS] mentions possible frequencies from 2 Hz to 8192 Hz in periodic mode and alarm interrupts with a one-second granularity.
- The local APIC Timer has a poor resolution, and will be paused in certain power conditions. Furthermore, buggy hardware implementations are in use and it is not prevalent.
- The Power Management Clock (PM Clock or ACPI Timer) was designed to provide fast timestamps and nothing else.
- The HPET (High Precision Event Timer) [Int04d] has an excellent resolution and low access times, however, it still needs to be programmed by the kernel.

Additionally to the external circuitry mentioned above, modern processors contain timekeeping registers. Intel-compatible processors starting with the Pentium contain a timestamp counter (TSC) register. The TSC can be read with the `RDTSC` instruction. Using the TSC has two tremendous advantages over the external circuitry discussed previously: It can be read extremely fast (down to 12 cycles on some architectures [Adv05]) and it has a very high resolution (currently in the range of the processors frequency). Reading is usually fast, because the time source is in the processor (or close) and the `RDTSC` instruction can be executed from user code (unless the feature bit is deactivated in the `CR4` register), which makes kernel entries superfluous.

There are several problems associated with using the TSC, notwithstanding:

- The `RDTSC` instruction is not serializing, that is, the processor can reorder it in the instruction stream. This problem can be fixed by wrapping it in serializing instructions (e. g., `CPUID`). Current AMD processors provide the `RDTSCP` instruction, which, besides being serializing, also atomically delivers the value of TSC_AUX together with the contents of the TSC. TSC_AUX can be set by the operating system to allow user code to distinguish on which processors it executes `RDTSCP`.
- Using the TSC as the only time source sometimes proved unreliable for operating systems on older hardware. Different power states and dynamic frequency scaling may lead to drift between the different TSCs in multi-processor systems and also between single-processor TSCs and wall-clock time (e. g., Linux kernel code contains reliability checks if TSCs can be used as time source) [Pro,Bru05].

  Current processors from both Intel and AMD have frequency-stable TSCs. A quote from "IA-32 Intel Architecture Software Developer's Manual — Volume 3: System Programming Guide" [Int06], Chapter 18: Debugging and Performance Monitoring, Section 18.8: Timestamp Counter attest that (my emphasis):

  > "For Pentium 4 processors, Intel Xeon processors (family [0FH], models [03H and higher]): the timestamp counter increments at a constant rate. *[...]* Constant TSC behavior ensures that the duration of each clock tick is uniform and supports the use of the TSC as a *wall clock timer*

even if the processor core changes frequency. This is the *architectural behavior moving forward*."

AMD's documentation [Adv07] contains similar assurances.

- Using the TSC may incur a significant runtime overhead because it may be virtualized (trap into the kernel or hypervisor) or it may be too expensive to read for a given application scenario (cf. to [Adv07], *Table 14. System Instruction Latencies* and [Int07], *Table C-12a. General Purpose Instructions*).

  Furthermore, a TSC may not be usable or available at all for the following reasons:

  – Other architectures may not have an equivalent register (e. g., ARM),

  – User-space access to the TSC may be disabled in the kernel (`CR4` register),

  – Using future hardware support for memory transaction may prevent reading the TSC from within transactions [Adv09], and

  – The use of accurate physical timestamps may be undesirable in order to rate limit potential timing covert channels.

In the context of this work, I will use the TSC as a time source. Above-mentioned restrictions do not represent a serious limitation for my practical research. I will, however, provide a discussion of how to adapt Ferret to other clock types in the following so as to allow others to deploy Ferret in other environments with different constraints.

Virtual physical clocks As a special case, sometimes thread-local wall-clock time is required, that is, a real-time clock that ticks for each thread independently and only if a thread is active. In theory, this information can be computed from global-clock timestamps if all scheduling events with their timestamps are available. In practice however, this approach may be too expensive (as on each context switch events have to be created). Instead the operating-system kernel must account for the time spent in each thread.

I implemented this feature for Singularity. The current value can be requested via a regular system call.

For Fiasco, this accounting is already available (the *Finegrained CPU accounting* option). For obtaining the value without an actual kernel entry I modified Fiasco to publish the current accumulated thread time ($T_{AccThread}$) into the kernel info page (KIP) together with a current TSC value ($T_{TscSched}$) on each context switch. If a thread wants to obtain a timestamp for its virtual physical clock it has to read both values from the KIP and get a current value from the TSC (T_{TscCur}). The sum of $T_{AccThread}$ and the difference between both timestamps ($T_{TscCur} - T_{TscSched}$) is the real thread time:

$$T_{Thread} = T_{AccThread} + (T_{TscCur} - T_{TscSched}) \tag{3.1}$$

To prevent a race condition, one has to reread the TSC value from the KIP again after obtaining T_{TscCur}. If both $T_{TscSched}$ values are equal there was no interruption by a scheduler, otherwise the procedure has to be restarted.

Logical clocks In contrast to physical clocks, logical clocks offer no notion of durations. Timestamps can only be used for ordering events. Lamport's clocks provide a happens-before relation [Lam78], which basically guarantees that if event A causes event B then the timestamp of event A will be smaller than that of event B. The converse cannot be concluded in general. Vector clocks, independently developed by Mattern [Mat89] and Fidge [Fid88,Fid91], extend Lamport's logical clocks by defining causality relations. Vector clocks establish a partial order over events such that all event pairs can either be compared (implying a causal relation between them) or not (implying that both events are independent), that is, vector clocks provide information that physical clocks cannot provide.

After selecting the TSC as a timing source for Ferret in the previous section, there still remain advantages of logical clocks for specific application scenarios. Besides the provided causality relationship from vector clocks, logical clocks may also be cheaper to use in several situations, as discussed in Section 3.2.3.2.

However, vector timestamps scale with the number of nodes in a domain, so they may need to be huge and may even change size if nodes leave or join the system. Directly supporting vector time within each event in Ferret would therefore be to costly and complex. Instead, each Ferret event has room for a single timestamp (currently 64 bit). This can be used either for TSC timestamps as mentioned above or for logical-clock timestamps. Lamport's clocks are directly supported this way. On message arrival (according to the model of logical clocks), the local clock would have to be adjusted to the maximum of the local and the remote clock.

Furthermore, vector-clock timestamps can be derived from event traces in case causal dependencies are required, provided that the corresponding send and receive events for messages can be identified and if there is node-local ordering available. In other words, a global observer can construct vector-clock timestamps if monotonically increasing local timestamps are used and if corresponding send and receive parts of remote messages can be matched[2]. In fact, creating this kind of structure is an essential part of schemata creation described in Section 3.2.2.2.

Furthermore, the custom area of each event can contain any type and number of timestamps (if space permits). This allows arbitrary custom solutions on top of Ferret, such as bringing events from several independent clocks in (partial) order if there are events stamped with several timestamps.

Implementation problems I want to discuss several implementation specific problems in the following.

In its simplest form, a logical clock can be implemented in shared memory using atomic increment operations on a counter variable. The x86 architecture can increment 32-bit values atomically in memory using `ADD`. Using `XADD`, one can also atomically get the old value from memory. `XADD` does not work with 64-bit values, which are desirable to practically prevent overflows[3]. While `CMPXCHG8B` allows atomic 64-bit writes, its up-to-

[2] For shared memory communication, matching or even identifying corresponding send and receive parts of messages is not easily possible as every single memory access may constitute such an event.

[3] One may consider this as a temporary limitation until 64-bit architectures with support for atomic 64-bit adds are common, for example, x86-64.

date check may fail if other components have updated the counter concurrently. It has to be used inside a loop and gives no local progress guarantee without further procedures.

Valois discusses a related problem in [Val96]. He proposes to use a *SafeRead* operation, which atomically dereferences a pointer and increments a reference counter on the corresponding object. Valois shows a lock-free implementation of such an operation for cooperating threads in shared memory regions. However, also within cooperating threads indefinite looping might occur, which is undesirable for real-time systems.

The described software implementations have the drawback that all participating components have to be trusted to modify the counter in memory only monotonically — they have to cooperate.

This is a much harder requirement than that components that share a shared-memory sensor have to trust each other with regard to memory corruption. The latter requirement can be solved by simply using several sensors. The former cannot be fulfilled with simple means, as events from different domains may still need to be compared.

I imagine two implementation versions for solving these problems: *First*, a kernel implementation for logical clocks that provides logical timestamps via explicit system calls.

Second, one could imagine hardware support for logical clocks in two forms. First, hardware could support increment-on-read regions in memory[4] (also cf. to [Wik]). Second, there could be an explicit `RDLC` (**r**ea**d** **l**ogical **c**lock) machine instruction. In contrast to `RDTSC` logical counters would only have to be incremented *on usage*. Furthermore, more than one counter could be supported, which would enable to separate system parts from each other.

Combinations of physical and logical clocks: hybrid clocks There are situations and platforms[5] where a good (referring to the properties in Section 3.2.3.2) physical clock is not available but logical clocks are not good enough either (e. g., measuring real-time durations is required, but not with a high precision).

To be more precise, for all purposes, except establishing event order only coarse-grain timestamps are required, establishing event order, however, requires a much better resolution as events may occur in tight bursts.

In summary, *hybrid clock* timestamps are composed of a low-resolution real-time timestamp in the most-significant bits and a logical time stamp in the least-significant bits.

The inefficient access to the real-time source is handled by the kernel or external drivers (e. g., via regular timer interrupts). The current real-time timestamp is then published to user code via shared memory, such that it can be queried efficiently, that is, without kernel entry or context switches.

The logical part of timestamps is taken from a second clock, which could be as simple as an integer in shared memory that is incremented on access by participating com-

[4] A single memory page would be split up into several logical clock counters, each 64 bit wide.

[5] The ARM "RealView Emulation Baseboard HBI-0140 Rev C" has "Oscillator test registers, SYS_-TEST_OSCx" that provide a high precision timer [ARM07]. This timer, however, is only available on the developer's board and can furthermore not be easily made available to user programs without exposing other platform state to the user code too.

ponents. I discussed several variants of logical clocks and their implications in Section 3.2.3.2 above. The logical clock has to be reset on each increment of the real-time clock.

The exact partitioning of timestamps' bits depends, of course, on the target platform, time source properties, and the software to be run. For example, on a platform that has a time source similar to PC's RTC (1 ms resolution) one may pick the most simple partitioning scheme of 32 bit for both the real-time clock and the logical-clock part. This would allow a plentiful 4 294 967 296 events/ms (i. e., roughly 4 295 events/cycle on a 1 GHz processor). The real-time part covers roughly 49.7 days — enough for many purposes.

3.2.4 Sensor synchronization approaches

In the following sections I discuss synchronization approaches for sensor data structures. In the general case, several readers and writers can operate on each sensor, whereas both sides may have real-time requirements.

The next section briefly discusses common approaches in the literature and documents why they are unsuitable for my requirements. The succeeding Section 3.2.4.2 details this discussion for one of Ferret's predecessors and shows the unsuitability of a class of algorithms in more detail. Finally, based upon these results, Section 3.2.4.3 explains the approach that I follow with Ferret.

3.2.4.1 Common approaches

The common core problem of the different approaches is the synchronization of access to sensor data structures. I briefly summarize common approaches in the following and point out their shortcomings with regard to my requirements:

Kernel A standard approach to provide atomicity of code sequences is to run them in the *kernel* with sufficient protection (e. g., with closed interrupts). Drawbacks of this approach are manifold. First, one has to pay the extra overhead for kernel entries. Second, this approach is inflexible, as for all changes to Ferret, the kernel has to be modified. Third, this approach is opposed to the minimality approach of microkernels (cf. to Section 3.2.1).

Locking Any type of locking that may lead to blocking in event producers (mutexes, semaphores, ...) is unsuitable as no temporal guarantees can be ensured.

Double buffering Approaches based on *double-buffering* techniques, where the sensor is split into sub regions, such as used in Event Tracing for Windows (ETW) [Mica] and K42 [AAD+02] do not work if event creation is not atomic, as slow event producers can block sub buffers arbitrarily long. In ETW, the problem is circumvented by creating events atomically in the kernel. In K42, event streams are *not* guaranteed to be correct. Slow event producers can essentially corrupt arbitrary parts in the stream.

Lock-free algorithms In [ARJ97] Anderson et al. consider using *lock-free synchronization in real-time systems*. By restricting the scheduling algorithms in the system

to a set of real-time algorithms, they are able to compute upper bounds on the loops of lock-free algorithms. This allows them to compute bounds for the execution time of those algorithms. These results are interesting but too restrictive for my approach. I want FERRET to work in non–real-time parts of a hybrid real-time—non–real-time system too.

Delayed preemption as described in [L4K06] is a general low-overhead mechanism for executing atomic sequences in user-space code. User-space code informs the kernel about entry into an atomic section by switching on a bit in shared memory. The kernel, in turn, tries not to interrupt user-code execution inside this section by delaying preemptions. The underlying assumption is, that user-space code typically runs through its critical section and turns off the signaling bit without actually being interrupted by the kernel. In the common case the signaling bit is not evaluated. Only upon kernel entry however, for example, caused by an interrupt, it is acted upon by the kernel.

There is a set of open issues with delayed preemption, especially in the context of real-time systems: How are maximum durations for user-code critical sections determined and specified?[6] What happens upon overrun of the latter? How does delayed preemption interact with time-slice donation and real-time time slices? How are page faults inside critical sections handled, should they be supported? If not, how should they be prevented?

Delayed preemption, as described in [L4K06], additionally has the concept of a sensitive priority, which defines a threshold above which tasks can interrupt critical sections. This approach may solve some of the problems discussed previously but furthermore complicates an implementation. On the other hand, it is not powerful enough to support two completely independent critical sections. They would still interfere with each other.

Simple Delayed Preemption In [OS], we tried to define a simplified version of delayed preemption, as a starting point for an implementation. *Simple Delayed Preemption* does not have a sensitive priority. It is targeted at small critical sections, where the full execution has the same order of magnitude as several context switches. We also tried to define a global maximum for the execution time of 10 µs to be able to incorporate this in a global real-time schedule. Still, the problem of page faults from within critical sections remains. Also, to be safely usable, the kernel has to setup a watchdog timer, each time it detects the interruption of a delayed preemption usage. Furthermore, it remains completely unclear what mechanism to provide to the user in case of deadline overrun inside a critical section. Ideally, a user handler function would have to be called synchronously to cleanup state, which again would have to be bounded in its execution time. Therefore, the kernel again needs a watchdog timer. But what happens if this function does overrun again?

[6] Current desktop platforms may not be suitable for hard real-time problems, cf. to measurements in Section 5.3.2, which show regular System-Management–Mode interruptions of more than 1.68 ms.

It seems that delayed preemption tries to provide too broad semantics to be implementable safely in a real-time system.

Atomic rollforward sections Atomic rollforward sections as described by Mosberger et al. in [MDP96] target a similar area as delayed preemption with simplified requirements. I will cover this approach in Section 3.2.4.3 in detail. In contrast to Anderson et al.'s approach, rollforward sections do not rely on specific scheduling algorithms. They are therefore more general regarding that aspect. On the other hand, they are more restrictive (because they are simpler) as not all types of algorithms are supported on top (not all algorithms can easily be rolled forward or back).

I therefore chose the atomic–rollforward-sections approach for FERRET and discuss it in more detail in Section 3.2.4.3.

3.2.4.2 An evaluation of the event sensors from: A Generalized Approach to Runtime Monitoring for Real-Time Systems

In this section, I discuss the event sensors from *A Generalized Approach to Runtime Monitoring for Real-Time Systems* (GRTMon) [Rie05] in more detail. I start by describing how I model checked the implementation for a certain property. Following that, I present a problem description in the context of real-time systems. I generalize this in the following and show that a class of algorithms cannot be used in real-time systems.

Verification of mission-critical code GRTMon uses a lock-free approach for sensor synchronization. Lock-free algorithms are often intricate and hard implement correctly. As sensor code will potentially be used all over the system, its correctness is crucial. Therefore, I applied the model checker Spin [Hol06] on the sensor access code to verify that memory of a single event is never written to by two threads simultaneously.

Spin cannot directly verify C/C++ source language, but instead uses its own language — Promela. Therefore, I translated GRTMon's sensor code into Promela. I furthermore had to simplify the code to make model checking attainable. To this end, I reduced the problem space to two threads and six entries in the event sensor. I also tried to use minimal data types for all the structure members. With all these tricks applied, the state space was still larger than 4 GiB. One key problem with model checking here is that it requires a small state space to be tractable — every single bit counts as the number of verification steps grows exponentially in the worst case with the state space size. Lock-free data structures, on the other hand, typically use large monotonic counters to circumvent the ABA problem (cf. to [Tre86]). Also the event timestamps are usually very large. As both approaches conflict, the problem-minimization compromise is required to even check a limited number of algorithm steps.

With the help of Spin, I checked the following condition in the model: There must never be two threads that can write to the same event buffer location at the same time. Up to a depth of about 1 300 steps this did not occur (then, the model checker ran out of memory), therefore, it can be assumed that the concurrent sensor of GRTMon is safe with regard to *this* property.

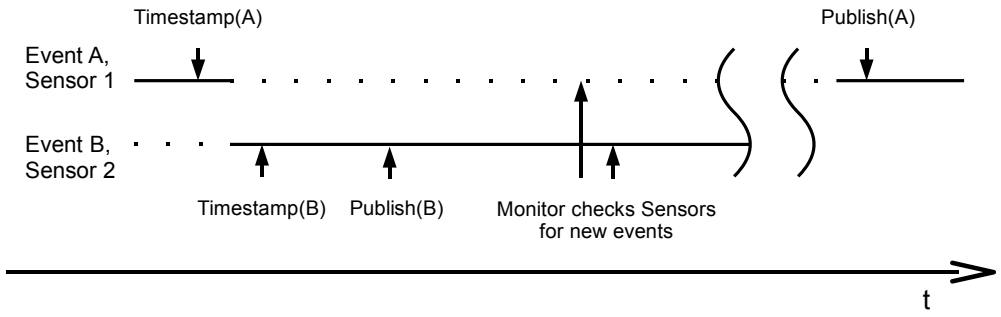


Figure 3.3: Depicted is a situation where event-timestamp order and visibility order do not match. A monitor checking sensors 1 and 2 cannot safely decide for how long to wait for a potential event A to become visible.

Timely visibility of events Although I did prove a safety property of GRTMon's event sensors as described previously, there is a problem with event ordering when using more than one sensor of this type in a system simultaneously. I will discuss the problem in this section and solutions in Section 3.2.4.3.

GRTMon's *Concurrently Invocable Sensors* guarantee the ordering of events within each instance of a sensor. A monitor that listens to more than one sensor cannot establish the ordering of events from those sensors safely in all cases in real-time. The core problem is that an event producer can be interrupted between obtaining a timestamp and publishing the corresponding event in a sensor. A critical situation occurs if two events, A and B, are created and timestamped in that order in independent sensors (depicted in Figure 3.3). However, the production of A is interrupted between timestamping and publishing it in the sensor, that is, B becomes visible in its sensor before A in its. If a monitor now checks both sensors in-between the publishing of events B and A, it cannot tell whether in the future an event A will become visible in the first sensor. Furthermore, it can also not determine a safe upper bound for waiting on this sensor for a potential A to become visible as there are no real-time reservations for event creation in the general case.

In conclusion, a monitor cannot decide how long to wait for new events to become visible if more than one sensor is used. No efficient online monitoring is therefore possible as events shall typically be processed in order (ETW shows similar symptoms for related reasons, cf. to Section 2.2 on page 14).

I will now briefly discuss obvious solution attempts and why they are unsuitable:

- Each element is timestamped not only with one timestamp but with two, one denoting the beginning of the event production and one the finalization. Besides the increased complexity due to the semantic changes (an event cannot be considered instantaneous anymore but has a *duration*, other work, such as Event Calculus [Sha99], relies on the concept of *instantaneous* events) the problem of atomically obtaining the second timestamp and publishing the event is still there. However, at least for offline traces a partial order of events can be established and truly parallel events can be detected.

- One could mark whole sensors as *active* (e. g., by having a global writer counter or a global timestamp denoting when a writer starts event creation). Now monitors can at least decide whether waiting on a sensor would be pointless, because no event is currently in production.

  There is still no way, however, to determine an upper bound of how long to wait on such an *active* sensor as there is no knowledge of when the event producer will resume.

Proof that such an ordering cannot be guaranteed without atomic sections To motivate atomic sections that I will introduce later, I will now show that given a common type of timestamp source, defined in the following, there cannot be a safe and bounded solution for posting events in absence of atomic sections.

Definition 1 (swift time source): *I call a time source* swift *if timestamps can be obtained and stored to arbitrary memory locations in* one atomic operation*.*

For example, the timestamp counter (TSC) of current x86 processors is *not* swift as timestamps obtained using `RDTSC` are delivered in the processor registers `EAX:EDX` only. Storing to memory has to be done in a second step.

High Precision Event Timers (HPET, cf. to [Int04d]) may (with some critical restrictions, discussed in the following) be used as a swift time source if timestamps can be obtained using memory-to-memory copy operations in memory-mapped input-output regions.

Definition 2 (steps): *An algorithm that publishes events to sensors contains at least the following three steps:*

1. *Obtaining a timestamp from a time source,*
2. *Writing of the timestamp to the event, and*
3. *Making the event visible in a sensor.*

For swift time sources, steps 1 and 2 are executed atomically.

These three steps form a critical section. If an event producer is interrupted in this section, a monitor cannot efficiently decide how long to wait for the producer to finish.

For GRTMon's Concurrently Invocable Sensors these steps are: (1) `RDTSC` (which stores timestamps in the registers `EAX:EDX`), (2) copying the timestamps from `EAX:EDX` to event memory, and (3) committing events by writing references in the index area.

The following definition captures time-source properties of practical relevance for monitoring.

Definition 3 (usable time source): *I call a time source* usable *if (a) it can safely be used from user code (the mechanism to use it should not unnecessarily export system-internal state) without overhead in the order of magnitude of a kernel entry, and (b) if the clock delivers timestamps of sufficient resolution to fully order any application-generated events on a processor.*

HPETs' timer can be accessed via memory-mapped input-output, however, whether this can be securely made available to user code is debatable (a): The memory page with the main counter also contains internal information about the HPET, such as interrupt routing, further timers, and timeouts, which should not be available to arbitrary user code.

Access to this memory region can be restricted below page granularity only using segments. Segments are x86 specific and are practically not supported within the x86-64 architecture. Furthermore, Fiasco does not currently support segmentation. It provides a flat space model (segment register would have to be saved on context switch, which does not come for free, cf. to [Hof02]).

Furthermore, 64-bit HPET timers cannot be read atomically. One would have to loop potentially (cf. to Section 2.4.7 Issues related to 64-bit Timers with 32-bit CPUs in [Int04d]).

Memory-to-memory copy operations are only possible with `REP MOVS` on the x86 architecture and are not executed atomically. The ARM platform does not support them at all (strictly load-store architecture).

Requirement (b) excludes periodic time sources driven by the kernel (e. g., Real-time clock (RTC) and 8254 Programmable Interval Timer (PIT)) as their resolution is limited by a sensible timer interrupt frequency (e. g., 1 ms for both mentioned clocks).

The following definition merges the Definitions 1 and 3.

Definition 4 (swiftly usable time source): *I call a time source* swiftly usable *if it is (a) swift according to Definition 1 and (b) a usable time source according to Definition 3.*

Conjecture 1: *Due to the discussion above, I will assume in the following that swiftly usable time sources are practically not available, especially not as a cross-platform concept.*

Assumption 1: *I assume that thread execution can be interrupted and suspended transparently by the underlying operating system as I do not want to restrict in which environment monitoring is to be used.*

I will now prove by contradiction that there cannot be a safe and bounded solution for posting events without some form of kernel support.

Proof 1: We require three properties from a solution: (a) it must not wait or loop unboundedly, (b) there must be no information flow from monitors to event producers, and (c) it must preserve event order. We now assume there is such an algorithm.

This algorithm has to contain at least steps 1 and 2 from Definition 2 above. There are no swiftly usable time sources available (Conjecture 1), but one requires a usable time source for event ordering (Definition 3 and property (c)). Therefore, the time source will not be swift and, hence, cannot execute both steps atomically. Due to Assumption 1, this algorithm can be interrupted between the two steps for an arbitrary duration.

Assume a monitor is scheduled to run in-between steps 1 and 2 of an event producer. There are two cases to distinguish now:

- The monitor cannot decide whether an event producer is in a critical section. It can therefore not decide whether such a timestamp will potentially become visible in the future in one of its sensors or not. Consequently it will have to wait for arbitrary durations and, hence, cannot work efficiently online.
- Assuming there was a protocol to let the monitor know of the existence of such in-flight timestamps it still cannot decide *when* they would become visible as there is no guarantee when execution in the event-producing thread will proceed.

In both cases property (a) is violated.

Also the event producer cannot react to critical situations, for example, by recreating timestamps, as it does not know of such situations due to property (b). ■

For completeness, I will now informally discuss further design problems assuming that swiftly usable time sources were actually available.

With swiftly usable time sources the steps 1 and 2 of Definition 2 would be executed atomically. However, for the whole algorithm to work efficiently, also step 3 must be executed atomically. In the absence of a working double compare-and-swap (DCAS) implementation for x86 and other widely used architectures, this can only happen by constructing the sensor memory layout such that step 2 makes events visible, that is, steps 2 and 3 are merged.

GRTMon's implementation of Concurrently Invocable Sensors circumvents memory fragmentation in sensor memory by indirection (sequential access to events is typically required for producers and consumers). Therefore, events are dequeued and committed. Dequeued events can be manipulated safely and arbitrarily. Even leaking single dequeued events by crashed threads does not restrict the sensor as a whole. Event consumption order by monitors and recycle order in the sensor by later producers is the commit order. The indirection area in the sensor provides the required sequential view. Therefore, however, dequeuing and committing happen independently from timestamping events.

If this indirection would be abandoned, sensor memory could be arbitrarily fragmented. Events in the middle might still be in modification while *later* events may already be completed. This prevents efficient sequential access. Monitors, and worse (cf. to Section 3.1.2), event producers, would have to search the whole sensor area to find newest and oldest events.

Consequently, for nontrivial (more than one sensor involved), safe (an upper bound exists for the time a monitor has to wait for sensors), and efficient (only shared memory is used, usually no kernel entry) monitoring some form of kernel support for atomic execution is required. This mechanism should be more efficient than actual kernel entries. I will describe such a design in Section 3.2.4.3.

Superficially, there are several ways to circumvent the problems explained previously. I will now briefly describe these approaches and discuss why they *cannot* fully solve the problems.

- Inside the event producing code, one could check whether an interruption occurred. This could either be realized by measuring the progression of time, or the kernel could expose interrupt and scheduling information to user space (e. g., by shared memory). If exposing detailed scheduling information is a security problem in a system, because other parts can now be observed, the kernel could just export *whether* scheduling occurred between certain points in time[7]. In case of interruption the whole event production would have to be restarted. For safely applying this pattern a DCAS instruction or a similar primitive is required.

- Alternatively, one could work without kernel support with the help of a globally visible memory area in which every timestamp obtained in the system has to be published. Publishing is done by writing the most recent timestamp to memory. After publishing the own timestamp event creation is prepared. On event publishing the freshness of the own timestamp is verified in the global memory area. Also for this approach a DCAS primitive or equivalent is required. If the freshness check fails, the whole process has to be restarted.

  A further problem of this approach is that all parties in a system whose timestamps have to be ordered must cooperate for the protocol to work correctly. The shared memory region also opens up an undesireable global communication link between *all* entities in the system.

Both approaches require rolling back and restarting the event production in case of conflict. There is no upper bound for the number of rollbacks required and hence for the time for event production. In consequence, both approaches are not suitable for real-time systems.

After showing that safe and efficient sensors cannot be constructed without some form of kernel support, the question arises which form this support could have. I will answer this question in the succeeding section.

3.2.4.3 Atomic sections in user-space code for sensor synchronization

In [MDP96] Mosberger, Druschel, and Peterson give a good overview about the design space for user-space atomic sequences. In the following, I will discuss important items of this design space in the context of my envisioned target system (cf. to Section 3.1.1). I will summarize thereafter:

[7] Instead of using a global monotonic scheduling counter, which would allow any program to infer some knowledge about other parts of the system, the kernel could set a (sufficiently large) new random number in a publicly readable *epoch field* in shared memory each time an interrupt occurred or scheduling was done. The user program can now know for any two points in time whether interruptions happened between them by sampling the epoch field. If both samples are equal, there was no interruption, if not, there was at least one interruption.

Preemption granularity Mosberger et al. assume a unix-like system where signals are available in user space and are presumed to be much rarer than hardware interrupts. Generally, the highest abstraction level for the preemption point is desirable. Preemption point candidates are signal delivery or user-space scheduling decisions for simple user-space applications. For systems with shared memory across address spaces, kernel-level scheduling decisions are relevant preemption points. If interrupt handlers shall use atomic sections, interrupts are the lowest relevant preemption source. Choosing interrupts results in the most general solution at the highest costs.

For FERRET, interrupts must be chosen as preemption granularity, as interrupt-handler routines in drivers running in user space are typical targets for FERRET instrumentation.

Registering atomic sequences Mosberger et al. mention four possibilities for registering atomic sequences in [MDP96]: designated sequences (marker code around atomic sequences), static registration (executed at program start), dynamic registration (executed each time before an atomic sequence), and hybrid registration (a combination of designated sequences and dynamic registration). All of these approaches have disadvantages that make them unsuitable.

Atomic-sequence support for FERRET needs not be general, but only a set of event-posting routines must be supported. Ideally, this set does not change at runtime. Therefore, for FERRET I found an even simpler solution, which I call *designated area*. In every address space I reserve a designated area that only contains atomic sequences. Those sequences need thus not be registered online or at program start and surrounding code doesn't have to be inspected for marker code to determine whether a sequence is atomic. Instead, they are identified by their location in the address space. To determine whether a program currently executes an atomic sequence only the user-space instruction pointer must be compared to the reserved region. To prevent conflicts with applications requiring specific address-space layouts, I put the code in the kernel's address range[8] but allow user code to execute it.

Return of control Rolling forward user-code sections, that is, executing them atomically, virtually without interruptions raises the question of how control is returned safely back to the system. Mosberger et al. mention four approaches: code rewriting, cloning, computed jumps, and controlled faults. For *code rewriting* callback code to the system is temporarily inserted directly after the atomic sequence. *Cloning* creates a copy of the original atomic section that ends with a system callback. *Computed jumps* require that each atomic sequence ends with an indirect jump which can be rerouted by the system. *Controlled faults* refers to a technique where each atomic sequence ends with an instruction that normally behaves as a `NOP` (no operation) but the kernel can modify system state to generate a fault on this instruction.

[8] The idea to identify atomic sequences based on their location is taken from the ARM port of Fiasco.

For FERRET I use an approach based on cloning and controlled faults, which I call *static cloning*. As the address of the atomic sequences is fixed it is simple to provide a second copy that is pre-instrumented. The memory layout of both versions is similar except for a small detail: The uninstrumented version contains a `NOP` opcode, directly after the final return statement. For the instrumented version the final return statement is moved to the `NOP` and is preceded by a trapping instruction. Switching from one version to the other is as simple as changing the page-number part of the user-space instruction pointer to point to the instrumented version. I describe the implementation in detail in Section 4.1.3.2.

Limiting the duration of a rollforward section is not a problem in the FERRET scenario, as the code executed with rollforward privilege is controlled by the kernel. It can be constructed to be bounded. Determining execution time, also worst-case execution time, for this code can be done offline.

In contrast to rollforward code, there is, in principle, no need to control code executed within a rollback section. Rollback code is basically restarted if interrupted. Any user-space–provided code section, even an endless loop, could be run with rollback semantics from a whole-system–stability point-of-view. The code will simply not make progress. However, the remainder of the system is not delayed or blocked, as interrupts are not suppressed in the rollback case.

Not all code can be run with rollback semantics. It must be designed to support arbitrary interruptions and restarts.

Page faults during the execution of atomic sequences will interrupt the atomicity. There are three sources for page faults with FERRET: code page faults in the sensor code, data page faults on the destination sensor, and data page faults on the source area of data to be written to sensors. Code faults are prevented per construction as sensor code is provided by the kernel in pinned memory. Also sensor memory is allocated from pinned memory pools by FERRET. Source data may not be pinned in all cases (cf. to Section 4.1.4).

Mosberger et al. propose to, instead of preventing page faults altogether, only do so at "inopportune places". Proposed solutions are:

- A combined page for data and code. This does not work for FERRET for several reasons: *First*, the code page must not be writable, otherwise, the system loses control over the code actually executed (which is a prerequisite for the optimizations discussed above). *Second*, that way, atomic sequences could not be used independently on several processors at the same time. *Third*, modified code pages may be executed inefficiently on current processors, as trace caches have to be invalidated [Int04a].
- A single data page, which is paged in on the first fault. In contrast to the system Mosberger et al. envision, in my target architecture paging may also be done by user-space processes (pagers) that may also be instrumented with FERRET. Event creation in pagers would therefore interrupt event creation in paged processes. It may, however, be possible to reserve a pinned memory

area for transferring event data to event-posting atomic sequences in environments that support this (DROPS, L^{4}Linux kernel).

- Prologue code before the actual rollforward section touches all required data pages and triggers page faults if pages were not present. The prologue code itself can be implemented as a rollback section, which can only be completed if all page faults are resolved. This approach is the most general solution (usable in L^{4}Linux user code).

I will discuss several implementation versions that guarantee page-fault freedom in Section 4.1.6

To conclude, in comparison to the general case of atomic sequences in user-space code, the following simplifications hold for the FERRET case:

- Only specific sensor access code must be supported.
- This code may be constructed to have bounded execution time, therefore, no watchdog timers etc. are required to ensure termination of the atomic sequence.
- Sensor access code doesn't have to be modified at runtime and can thus be mapped read-only to user programs.
- As the code is fully controlled by the system, it can be trusted to have and keep the required properties from above.
- Atomic-sequence code does not need to be instrumented dynamically. Instead, a statically instrumented copy can be provided.
- There is no need to register the code sequence statically or dynamically to be executed with rollbackward or rollforward semantics. Registering happens implicitly by positioning the sequences in address spaces.
- Page faults in atomic sequences are precluded per construction.
- Support for (potentially scheduling) system calls from within atomic sections is not required for FERRET.

3.3 Conclusion

I started this chapter with a discussion of the envisioned target system and a requirements analysis in Section 3.1. Starting out with high-level requirements I refined the discussion to low-level requirements and requirements to the hardware and to operating systems.

Based on this, I discussed design decisions for FERRET in Section 3.2 and gave an architecture overview. I discussed identified roles, design alternatives, online and offline evaluations, as well as external schemata.

Section 3.2.3 treated different sensor types and timestamps.

In the last part of this chapter, Section 3.2.4, I discussed different approaches for synchronizing access to sensors. A significant part of this section is spent on introducing the concept of *atomic sections in user-space code*, a concept that I adapted for use in real-time and microkernel environments. The implementation of this concept for FERRET ontop of the Fiasco microkernel and the implementation of specific sensor access code using this concept is discussed in the first part of the next chapter in Section 4.1.

Chapter 4

Implementation

In theory, there is no difference
between theory and practice.
But, in practice, there is.

(attributed to many)

In this chapter, I will explain several implementation specifics of the design presented in the previous chapter.

The chapter starts with an overview on atomic sections in user space and discusses the implementation in Fiasco. Also the implementation of the actual user-space sensors is illuminated. This builds upon the motivation for supporting atomic sequences for user-space code that I detailed in Section 3.2.4.2 and the discussion of the design space and identified potential simplifications with respect to the general case in Section 3.2.4.3. Section 4.2 continues with a discussion of generic instrumentation approaches for different system architectures. This chapter concludes with several observations about portability and tracing in different environments in Section 4.3.

4.1 Atomic sections in user-space code

In the following, I will describe my current prototypical implementation for the microkernel Fiasco and the x86 architecture, for which I tried to adhere to the following three constraints:

1. The implementation should work efficiently in the Fiasco microkernel.
2. It should only require minimal changes to the system architecture. Fundamental concepts, such as synchronous IPC should not be modified.
3. The implementation must not conflict with good IPC performance, that is, it must not slow down critical paths.

The remainder of this section is structured as follows: At first, common concepts for atomic sections are introduced. Following which, single approaches are presented in more detail — the rollback, the rollforward, and a combined approach in Sections 4.1.2, 4.1.3, and 4.1.4, respectively. I conclude with a short summary of the implementation for Fiasco in Section 4.1.5 and notes on using atomic sequences in specific sensor types in Section 4.1.6.

4.1.1 Common concepts

The two different concepts for atomic sections in user space (rollback and rollforward) described in Section 3.2.4.3 have the following common implementation requirements:

- In all relevant kernel-entry paths, the kernel must check whether an atomic section is currently being executed. For Fiasco, the following paths are relevant:
  - timer interrupt: `entry_int_timer` and `entry_int_timer_slow`,
  - other hardware interrupts: `all_irqs`,
  - traps and page faults: `slow_traps` and `entry_vec0e_page_fault`.

  The IPC paths need not be modified as there is no need to support system calls from within atomic sections for FERRET. If a system call is done from within an atomic sequence the atomicity guarantee is lost.

- The actual code of the atomic sections must be executable (but not writable) from within all address spaces. For this purpose, the kernel maps memory pages with the code into each address space at a well-defined address inside the kernel's address range. Placing the code inside the kernel's address range prevents conflicts with arbitrary libraries linked to monitored programs — this approach works with *all* programs.

  As a consequence, the code implementing the actual atomic sequence cannot be inlined into instrumented programs but must be called.

- The target address range of the atomic sequence is chosen such that checking for inclusion of the users-space instruction pointer can be done with few cheap machine instructions.

  Rolling back to the previous rollback point can be done by `AND`ing out the least-significant bits of the instruction pointer (therefore, rollback points have to be aligned accordingly). For switching from the normal to the instrumented version of the atomic sequence for rollforward code, only a single bit needs to be switched on in the instruction pointer, if the instrumented version is placed on the subsequent page and the whole block is aligned to multiples of two pages.

- Usually, both rollforward code and rollback code is simply executed, similarly to normal functions. Only in the, hopefully rare, case of actual interruptions by hardware interrupts or page faults special handling by the kernel is needed. This lazy approach optimizes for the average case.

Atomic sections in user-space code do not guarantee atomicity across processors. In the literature, sensors are often per-processor data structures [Micb,AAD+02]. The same approach can be applied to FERRET. On current versions of the x86 architecture, the `RDTSCP` instruction can be used to atomically acquire a timestamp and a processor identifier (cf. to Section 3.2.3.2). This identifier can be used to index into processor-local data structures. Thread migration in the middle of event creation is prevented by the rollback and rollforward approaches.

4.1.2 The rollback approach

Rollback sections must be written in a way that they can always be reset to their most recent rollback point. Therefore, each rollback-capable function is partitioned by rollback points. In each partition, results are first prepared and precomputed. They are committed in one final, noninterruptible machine instruction.

In practice, my sensor functions contain only two parts where actually only the first one contains relevant code and the second one only contains a return instruction (cf. to Figure 4.2).

In the common case, rollback sections are simply executed without any interruptions, similarly to normal functions. In case of interruption by an external interrupt or trap, they are reset to their most recent rollback point as described above.

Traps might occur if a rollback section is called with pointers to paged-out memory. In that case, the page fault is handled by the operating system as usual and the atomic section is restarted after the page fault has been handled.

Note that this way even page faults on the last, committing instruction can be handled. The whole section is simply restarted. However, FERRET uses pinned memory for sensors, and the committing instruction writes to that memory, so no page fault will occur on this instruction.

4.1.2.1 Limitations of the rollback approach

Although the kernel support for rollback sections is relatively simple, writing user-space code for atomic sections is not. Despite the usual problem of writing correct code, the code has to be written in assembler to have control over the concrete machine instructions used. Also, code alignment to rollback points and custom calling conventions can be realized only that way.

Solving this problem is the job of the implementer of the monitoring framework. A framework user likely does not care (as this code is already written and can be used) unless he wants to extend the framework.

Besides getting sensor code functionally right (including the rollback semantics), there is another problem with this approach in the context of real-time systems. There is no *hard* guarantee about the amount of rollbacks per function call, hence, there is no hard upper bound for the execution time. Rollback sections differ fundamentally from normal code in that there is no guarantee of progress if the scheduler quantizes time too fine or if there are to many other interruptions. I will provide a qualitative comparison of different sensor synchronization approaches in Section 5.2.

In the following, I provide a statistical argument, which might alleviate the problem for many practical considerations and some soft–real-time applications.

A short computation by rule of thumb shows that the inter–timer-interrupt distance, or more appropriately, the smallest possible timeslice available in Fiasco (1 ms) is four orders of magnitude larger than the processor-time demand for calling the rollback code once (1 ms vs. 100 ns, a factor of 10 000; cf. to row *AList (rollback)* in Table 5.2). So, with a very high probability, there should be maximally one timer-interrupt–caused rollback per event.

Again, using 100 ns as an estimate for event production cost, interrupts would have to hit the processor at a rate higher than 10 MHz to indefinitely delay event production. Such a setup is extremely unlikely with current desktop machines.

Still, one cannot easily determine a safe upper bound for the number of rollbacks. To also be on the safe side from a theoretical perspective, one has to use rollforward code, which I discuss in the next section.

4.1.3 The rollforward approach

Similarly to the rollback approach discussed above, rollforward code sections are simply executed in the common case.

In the case of a hardware interrupt, the kernel only does minimal interrupt handling (enqueuing). The kernel resumes execution on the *instrumented* version of the rollforward code directly afterwards. Additionally, at that point interrupts are disabled system wide by the kernel.

The instrumented version guarantees that control flow returns to the kernel eventually. As control for the code executed with rollforward privileges is in the hand of the kernel (cf. to Section 3.2.4.3) tight timing constraints can be guaranteed by constructing the code appropriately. For example, the AList (atomic sequence list, introduced in Section 4.1.6.1) sensor code in FERRET contains only one small bounded loop for copying the actual event content into the sensor (currently, up to 64 bytes).

Disabling interrupts effectively limits the number of disruptions seen within each rollforward code sequence to *one*. This results in *fundamentally* different progress guarantees than with the rollback approach discussed above, where progress can only be guaranteed statistically. Rollforward sections will terminate and will incur maximally one interruption. I provide detailed throughput and worst-case measurements for sensors that use the rollforward approach in Section 5.3.3.

The implemented method uses a lazy approach for disabling interrupts, which, in the common case, does not require kernel activity at all for atomic sections. The general assumption here is, that atomic sections are fast and the probability of them being disrupted by hardware interrupts being extremely low. This assumption is supported by the rule-of-thump contemplation in the previous section.

Rollforward code is usually simpler than rollback code as there is no need to establish an initial state again. It may also use more complex data structures as it is not limited to a single final atomic instruction. Naturally, the rollforward approach cannot cope with page faults from within the atomic sequence. Page faults cannot be handled without losing atomicity.

4.1.3.1 Alternative implementation approaches

In alternative implementations, interrupts could also be disabled and enabled eagerly, that is, before each invocation of rollforward code. Different implementation variants for this approach have different drawbacks:

- The instrumented code could be given the privilege to control interrupts in user space. Consequently, it would have to be trusted ultimately. Globally disabling–

enabling interrupts is usually a kernel privilege as the stability and timely guarantees of the whole system depend on it. Granting this privilege to arbitrary user code is not an acceptable option for open systems.

- The kernel could provide system calls for controlling interrupts and only provide them for trusted applications. Also, this approach has severe limitations. First, paying the overhead of two kernel entries for each event to be logged would be excessive. Second, not only trusted applications need to be monitored. In fact, quite often the opposite is true: Buggy and misbehaving applications shall be monitored and be debugged. Consequently, the whole system would have to be trusted again, which again is not acceptable for open systems.

- Besides the stability and trust arguments from above, disabling and reenabling interrupts is also not a cheap operation on current processors (e. g., cf. to Table C-8 in [Int04a]). Thus reducing the frequency of these operations is beneficial and is achieved with the lazy approach.

4.1.3.2 Implementation details

The instrumentation for transferring control back to the kernel is currently implemented as a trapping instruction directly before the final return statement of the rollforward code in the instrumented version of the code (cf. to Section 3.2.4.3 for a discussion of the two code versions). The code path triggered by this trap handles the previously queued interrupt and reenables interrupts globally. I call this trapping instruction the *interrupt-restoration trap* (IRT). The IRT is one implementation method. One could also implement a dedicated system call for the end of a rollforward section. Trapping instructions, however, can be short. For the x86 implementation, I use the `HLT` instruction with an opcode size of one byte. In the noninstrumented version, I use a single-byte `NOP` as a placeholder *after the corresponding* `RET`, such that similar code blocks are aligned (`RET` is not necessarily the spatially last instruction of a function).

As the IRT is located directly *before* the final return instruction in the rollforward code, scheduling might occur *technically in* the rollforward routine (the user-space instruction pointer is still in the rollforward page). It is important that the implementation distinguishes this case in all kernel-entry paths, as there is no safe IRT after the first IRT. In an early and erroneous implementation I did encounter this problem, which could result in a freeze of the whole machine. What the kernel did, was to disable interrupts again also in the case that the rollforward code had already executed the trap and was interrupted on the final return statement. Of course, after that return, there was no additional IRT that would be hit again. Execution returned to arbitrary user code *with interrupts disabled*. Furthermore, the return statement in the noninstrumented version needs to be aligned with the IRT in the instrumented version to catch interrupts that occur directly on the return.

An alternative to the preceding scheme works as follows: No `NOP` is inserted before the return statement. Instead, the return statement in the instrumented version is *replaced* by `HLT`. One additional memory page is required, which only contains a single return statement. The user-space instruction pointer is pointed to this return statement (i. e.,

```
rollback_point:
  <atomic rollback code sequence>
rollforward_point:
  <atomic rollforward code sequence>
```

Figure 4.1: Schema for atomic sequences following the combined approach.

outside of the actual rollforward area) directly before returning from the trap handler in the kernel. That way, the final return statement cannot be misinterpreted as belonging to the rollforward section.

The price to pay for uninterrupted execution of user-space code is increased interrupt latency by the time it takes to produce events. However, as long as this duration is not larger than the maximal duration that the kernel works with closed interrupts, the maximal interrupt latency of the system is *not increased*. [MHH02,MHSH01] mentions 85 µs respectively 58 µs for an older versions of Fiasco. I measured a worst-case execution time of 9.1 µs for event creation in AList sensors (cf. to Figure 5.14 in Section 5.3.3).

4.1.4 The combined approach

In the following, I describe a combination of the previously detailed rollback and rollforward approaches ([MDP96] discusses using a rollback section for prefaulting memory pages for the succeeding rollforward section). The motivation here is two-fold: *First*, the combined approach abstracts both former approaches into one general view as shown in Figure 4.1. From this general picture, restrictions for specific requirements can be derived.

Second, the combined approach allows sensor constructions that combine the advantages of both the rollback and the rollforward architecture.

In contrast to the freestanding rollback approach from Section 4.1.2, I distinguish two sources of interruptions that are handled differently here: asynchronous interrupts and page faults. Asynchronous interrupts are handled as in the freestanding rollforward section described previously, that is, the atomic section is not reset but resumed with disabled interrupts. This holds for *both parts* of the combined approach — the rollback section and the rollforward section. Only page faults in the rollback section cause a true rollback (interrupts are enabled again).

Now, the typical two classes of applications — real-time tasks and other programs that use *pinned memory* (e. g., the L^{4}Linux kernel) and other best-effort tasks that usually *do not pin* all relevant memory pages (e. g., L^{4}Linux user-space applications) — can use the same sensor architecture.

The rollback section is used to ensure the absence of page faults in the rollforward section. If execution reaches the rollforward point, all memory is guaranteed to be paged in and the rollforward section can proceed safely.

This approach guarantees bounded execution time for real-time tasks as no page faults will occur in the rollback section and at most one kernel entry is observed due to asynchronous interrupts.

I will now describe the three sources for memory accesses that can occur on event production to better understand page fault sources.

First, *function call parameters* are transferred on the stack if the normal C calling convention is used. Simple rollforward sensors can use a register calling convention, such that no stack access is required in the rollforward code for function parameter access. Rollback sensors need to transfer important parameters on the stack as they must potentially recover their initial state. The x86 architecture does not provide enough registers to both store the initial state for a potential recovery *and* execute nontrivial event production code. The stack should not be grown inside event production code in order not to trigger page faults.

Second, the actual *data to be written* to sensors (the event payload) may reside on the stack or the heap. It is the responsibility of the caller to provide this data in pinned memory, ensuring that no page faults occur on accessing it. Real-time clients or the L^{4}Linux kernel have no problem with this approach as their stacks are pinned and the wrapper functions (e. g., for ALists) put data on the stack. Only normal Linux user-space programs and other paged clients have to be careful. Page faults could be prevented if wrapper function always explicitly used pinned memory, such as provided by the Posix system call `mlock()`.

Third, the *sensor memory area* itself is the target of write operations. As Ferret guarantees that this memory is always pinned no page faults can occur here.

To summarize, page faults may occur in clients that directly try to submit data from paged memory into sensors (stack and heap) or that use complex rollforward sensors or any rollback sensors (parameters on stack). This may be prevented by both turning all clients into semi–real-time clients by using *pinned memory* (`mlock()`, `mlockall()`) or by *prefaulting* all necessary pages (in a rollback section as described previously or by skillful placement in the stack as described in Section 4.1.6.2).

4.1.5 The implementation for Fiasco

Both the rollback and the rollforward implementations contain: support for code pages in kernel space that include the user-executable event production code, debug code, macros for detecting whether user code executes atomic sections, initialization code, and small adaptations to the build system.

The rollback implementation hooks into five kernel-entry paths. It contains about 140 lines of code (including comments, configuration options, and one assembler version of sensor access code).

The rollforward implementation is an incremental patch to the rollback implementation and additionally contains code for handling delayed interrupts (especially the timer interrupt), disabling interrupts for user code, and an instrumentation trap handler. It hooks into three kernel-entry paths. The patch adds about 330 lines of code, this time including two versions of assembler sensor access code (the vanilla version and the statically instrumented version).

In summary, the implementation for rollback and rollforward sections is already compact and still has potential for optimization and shortening. I interpret this compactness also as a sign of easy portability of the mechanisms to other kernels.

Floating point usage (this includes Intel's Multi Media Extension (MMX) and Streaming SIMD Extensions (SSE), see [Int04b]) from within atomic sections is currently not possible, amongst other things because of the way Fiasco implements floating-point–unit (FPU) multiplexing. The FPU is disabled with each thread switch and the FPU context is restored lazily on the first fault. Using the large MMX respectively SSE registers for event construction may seem beneficially at first sight. It would, however, produce three problems: *First*, it may trigger faults inside atomic sections if event production code is the first FPU user after a context switch. This problem could be solved by either implementing support for handling FPU faults inside atomic instructions or by prefaulting the FPU similarly to memory pages (cf. to Section 4.1.4). As also other systems (e. g., Linux) handle the FPU lazily, relying on FPU-fault support for atomic sequences would restrict FERRET's portability.

Second, FPU usage may prolong event production considerably on resolving faults.

Third, the FPU context would have to be saved more often by the kernel, that is, context switches would take longer if events were created. Thus, FERRET's probe effect would be increased.

These problems represent a considerable influence on the system such that ruling out FPU usage from within atomic sections seems appropriate.

4.1.6 Using atomic sequences in sensors

FERRET provides implementations for a range of list sensors and some aggregating data structures. I experimented with various synchronization methods both for evaluating the methods' feasibility but also to get baseline performance numbers with methods otherwise clearly not suitable for real-time systems (e. g., blocking semaphores).

I discuss two sensors in more detail in the following. AList (atomic sequence list) sensors support only fixed event sizes per sensor instance. There exists an AList variant for both rollforward and rollback sequences whose differences are highlighted in the next section.

After that, in Section 4.1.6.2, I discuss the VList sensor, which supports variable event sizes. On the basis of this sensor, I demonstrate how a C-language implementation can be used despite the lack of control for stack access. I also describe variations for different execution environments.

A detailed comparison of all list sensor types is given in the Section 5.3.

4.1.6.1 ALists

I implemented event production code using both the rollback approach and the rollforward approach in assembler. I will describe both implementations now in that order.

The rollback implementation as shown in Figure 4.2 uses a standard C calling convention with three parameters on the stack (pointer to sensor, pointer to event payload, and size of payload). It uses only one functional rollback part, the second part only contains a return statement. The first part starts by fetching the parameters from the stack into registers. It then sets up the memory copy operation, followed by writing a timestamp into the event. The first part finishes with advancing the global head pointer

```
; Parameters: list pointer ..................... (esp + 0x4) -> %eax
;             pointer to parameters to copy ... (esp + 0x8) -> %esi
;             word count to copy ............... (esp + 0xc) -> %ecx
.p2align(12)                        ; rollback point
kern_lib_start:                     ; start at page boundary
mov    0x4(%esp),%eax               ; get par. from stack: list ptr.
mov    0x8(%esp),%esi               ;   source area
mov    0xc(%esp),%ecx               ;   size of source area

mov    0x1c(%eax),%ebx              ; prepare memcpy: l->count_mask
mov    0x10(%eax),%edx              ;   l->head
and    %edx,%ebx                    ;   &
shl    $0x6,%ebx                    ;   *64
add    %eax,%ebx                    ;   l+offset
lea    0x48(%ebx),%edi              ;   dest: &l.data[offset].data

rep movsl %ds:(%esi),%es:(%edi)     ; do memcpy

incl   %edx                         ; ++edx
mov    %edx,%ecx                    ; save new head
lea    0x10(%eax),%esi              ; &l->head

rdtsc                               ; get timestamp
mov    %eax,0x40(%ebx)              ; store in event: l.data[offset]
mov    %edx,0x44(%ebx)
jmp    1f                           ; mind the gap, compiler will pad here
1:
mov    %ecx,(%esi)                  ; save new l->head
.p2align(6)                         ; rollback point
ret
.p2align(12)                        ; prevent exporting other kernel memory
```

Figure 4.2: Assembler rollback implementation for AList event production code.

in the sensor to the new element that was created before by means of a single, atomic memory access.

In contrast to the rollback implementation, the rollforward implementation doesn't copy parameters from the stack. It uses a register calling convention similar in concept to gcc's regparm(3) calling convention [GC304,GC405].

However, I have to use different registers, as I want to transfer parameters directly in those registers that are used in the atomic section. For instance, on the x86 architecture, the registers `ESI` and `EDI` must be used as pointers for string memory copy, but regparm uses `EAX`, `EDX`, and `ECX`. Figure 4.3 shows the noninstrumented version. For the instrumentation, the `RET` is replaced with a trapping opcode (`HLT`) and is relocated to the `NOP`. Please note that for simplicity reasons I show here only code for 32-bit head pointers (i. e., without additional tricks in the consumer code, only 2^{32} events can be supported). Support for 64-bit head pointers in rollforward code is straightforward. The

```
; Parameter: list pointer ..................... %eax
;            pointer to parameters to copy ... %esi
;            word count to copy .............. %ecx
.p2align(12)
rollforward_start:
mov    0x1c(%eax),%ebx          ; prepare memcpy: l->count_mask
mov    0x10(%eax),%edx          ;   l->head
and    %edx,%ebx                ;   &
shl    $0x6,%ebx                ;   *64
add    %eax,%ebx                ;   l+offset
lea    0x48(%ebx),%edi          ;   dest: &l.data[offset].data

rep movsl %ds:(%esi),%es:(%edi) ; do memcpy

incl   %edx                     ; ++edx
mov    %edx,%ecx                ; save new head
lea    0x10(%eax),%esi          ; &l->head

rdtsc                           ; get timestamp
mov    %eax,0x40(%ebx)          ; store in event: l.data[offset]
mov    %edx,0x44(%ebx)

mov    %ecx,(%esi)              ; save new l->head
ret                             ; "hlt" in the instrumented version
nop                             ; "ret" in the instrumented version
.p2align(12)                    ; prevent exporting other kernel memory
```

Figure 4.3: Assembler rollforward implementation for AList event production code.

new head pointer is written with two 32-bit `MOV`s. Rollback code would have to use the `CMPXCHG8B` instruction to execute the final write atomically [Int04c].

I provide wrapper functions for diverse variations of event payload that use normal C calling conventions. These wrappers themselves can and probably will be inlined in instrumented programs.

Optimization experiments I noticed that, not surprisingly, the stack layout of the arguments for the wrapper function (e. g., major and minor type, instance ID, application specific data, see Figure 4.4) closely resembles the event body's layout. The exception here is that two 2-byte parameters are 4-byte aligned on the stack (they are padded), as per the C calling convention [JR81].

Now, the optimization idea is to save realigning the parameters in a local data structure and instead directly use the function call parameters to the wrapper function on the stack.

I conducted a short experiment in which the wrapper function's signature was changed to receive the two 2-byte parameters merged into one 4-byte parameter to achieve the required data layout.

```
void ferret_alist_post_2w(ferret_alist_t * l, uint16_t maj, uint16_t min,
                          uint16_t instance, uint32_t d0, uint32_t d1)
{
    ferret_list_entry_common_t e;

    e.major     = maj;
    e.minor     = min;
    e.instance  = instance;
    e.data32[0] = d0;
    e.data32[1] = d1;

    asm volatile (
        "cld                              \n\t"
        "push %%ecx                       \n\t"
        "push %%esi                       \n\t"
        "push %%eax                       \n\t"
        "call __ferret_alist_post         \n\t"
        "add  $0xc, %%esp                 \n\t"
        :
        : "S"((&(e.timestamp)) + 1), "a"(l), "c"(4)
        : "memory", "edi", "ebx", "edx", "cc"
        );
}
```

Figure 4.4: AList post-routine wrapper with two 32-bit words payload.

Surprisingly, almost no cycles are saved (with small variations depending on the actual test machine, cf. to Table 5.2). The explanation is, that the compiler anticipates this optimization itself. The normal wrapper functions are usually inlined into instrumented code and the event structure is constructed only once, directly inside the instrumented function.

4.1.6.2 VLists

In contrast to the AList sensor, the VList sensor supports variable event sizes. Sensor access code is therefore more complex and high-level language support (e. g., C) is desirable.

I describe here a rollforward implementation for the VList sensor (a slightly simplified version is shown in Figure 4.5). In contrast to rollback points, here, no alignment restrictions apply. However, no calls to other functions must be made and page faults must not be triggered from within sensor access code. The first problem can be solved by enforcing the inlining of all used functions (e. g., `memcpy`). With C-language code, stack growth can only be controlled in a limited way, that is, in addition to the page-fault sources for the assembler implementation of AList sensors, the stack is a new source for page faults. Despite explicit stack usage with local variables in C, standard calling conventions define several callee-saved registers (cf. to [MHJM09,San97]). In addition to that, the compiler does use the stack as spill area in case of register pressure.

```
void ferret_vlist_postf(ferret_vlist_t * l, size_t len, ferret_list_entry_t * ev)
{
    size_t * dest;
    size_t remaining;

restart:
    // increment tail until we have room for current event
    while (((((l->tail.low & l->size_mask) + l->size - 1) - l->head) &
            l->size_mask) < len + sizeof(size_t))
    {
        size_t temp_size;
        temp_size = *((size_t *)((l->tail.low & l->size_mask) + &(l->data[0])));
        l->tail.value += temp_size + sizeof(size_t);  // forward one event
    }

    remaining = l->size - l->head;
    dest = (size_t *)(l->head + &(l->data[0]));  // address for next event
    // care for wrap-around: insert filler event
    if (remaining < len + sizeof(size_t))
    {
        *dest = remaining - sizeof(size_t);  // size prefix
        if (remaining >= sizeof(size_t) + sizeof(ferret_utime_t))
        {
            // a timestamp of 0 implies a fill event
            *((ferret_utime_t *)(dest + 1)) = 0ULL;
        }

        l->head = 0;
        goto restart;  // retry
    }

    *dest++ = len;                          // write size prefix
    *((ferret_utime_t *)dest) = rdtsc();    // timestamp
    memcpy(dest + 2, &ev->data, len - sizeof(ferret_utime_t));  // payload

    // finally update head
    l->head = (l->head + sizeof(size_t) + len) & l->size_mask;
}
```

Figure 4.5: Simplified C code for VList sensor code.

Although maximal stack usage cannot be determined for general functions, sensor access code tends to be compact and manageable in that regard. By inspecting the gcc-generated assembler code, I could easily determine the maximal stack usage for VList sensor code (24 byte together with the call-pushed return address, cf. to Figure 4.6). I adapted the calling convention slightly to prefault potentially new stack pages directly before the actual call as shown in Figure 4.7.

```
ferret_vlist_write:
pushl   %ebp        ; callee saved register 1
pushl   %edi        ; callee saved register 2
pushl   %esi        ; callee saved register 3
pushl   %ebx        ; callee saved register 4
subl    $4, %esp    ; room for local variables and register spill
...
```

Figure 4.6: Initial part of generated assembler code for C implementation of VList post routine.

```
asm volatile (
    "pushl %2                       \n\t"  // standard parameters
    "pushl %1                       \n\t"  // ...
    "pushl %0                       \n\t"  // ...
    "or    $0,-24(%%esp)            \n\t"  // prefault stack
    "call  __ferret_vlist_postf     \n\t"
    "add   $0xc, %%esp              \n\t"  // fixup stack
    : "=a"(dummy), "=c"(dummy), "=d"(dummy)
    : "r"(l), "r"(offsetof(ferret_list_entry_common_t, data32[2])), "r"(&e)
    : "memory"
    );
```

Figure 4.7: Inline-assembler fragment with a stack-prefaulting calling convention for VList sensor code.

This approach has two *limitations*: *First*, maximal stack usage for generated code must be determinable, which represents, however, no real limitation in this context, as more complex functions would not be suitable as sensor accessors anyway. *Second*, the calling convention used for invoking such sensor code must be handwritten, as the prefaulting must be done close to the actual call. Otherwise, the stack pointer might be modified by compiler-generated code leading to wrong prefault addresses. Furthermore, for functions with large stack requirements (i. e., more than one memory page) several prefault instructions are required. As an alternative to a handwritten calling convention, also binary instrumentation could be used to insert the prefaulting instructions directly before relevant calls.

The *advantages* of this approach are twofold: By writing sensor code in a high-level language such as C much time and effort can be saved. Furthermore, this way compiler optimizations can be fully exploited, other than with handwritten assembler code.

Calling convention for clients without pinned stacks The previously described approach targets real-time programs with pinned but not prefaulted stacks (stacks stay paged-in after the first fault). Normal applications in legacy containers may not be able to pin their stack pages. Memory may still be paged out between the prefaulting and the actual call.

```
asm volatile (
    "mov %%esp,%%ebx                    \n\t"  // save old stack pointer
    "and $0xfffff000,%%esp              \n\t"  // page align stack
    "pushl %%ebx                        \n\t"  // save old stack pointer pt. 2
    "pushl %2                           \n\t"  // standard parameters
    "pushl %1                           \n\t"  // ...
    "pushl %0                           \n\t"  // ...
    "call  __ferret_vlist_postf         \n\t"
    "movl  0xc(%%esp),%%esp             \n\t"  // restore old stack pointer
    : "=a"(dummy), "=c"(dummy), "=d"(dummy)
    : "r"(l), "r"(offsetof(ferret_list_entry_common_t, data32[0])), "r"(&e)
    : "memory", "%ebx"
    );
```

Figure 4.8: Inline-assembler fragment with a page-aligned–stack calling convention for VList sensor code.

By aligning the stack pointer to a new page boundary before pushing function parameters and before the actual call, every stack access in the target function will only touch a single memory page (provided that the function's own stack usage, including the function parameters and the call-pushed return address, stays below the memory-page size). The actual call is the last machine instruction executed before the rollforward section. It definitely prefaults the target stack page as it pushes the return address on the stack. The code in Figure 4.8 demonstrates this approach.

If the client also has no way of providing the actual event payload in pinned memory, this data too can be moved onto the single stack page by carefully choosing the address of the local event structure. This approach is shown in Figure 4.9.

To conclude, also complex sensor access code can be used (by writing C code and reusing the compiler generated assembler code) in rollforward sections efficiently. This works even for non–real-time clients that have no control over page faults whatsoever.

4.2 General-purpose instrumentation placement

In this section, I will discuss where general instrumentation should be placed, that is, instrumentation not aimed at a specific monitoring project or task but that shall be reusable for various purposes, for example, using external schemata.

There are several general considerations to keep in mind for finding such instrumentation locations. The abstraction level should be chosen as high as possible to maximally use semantic information. Also, choosing a high abstraction level results in lower instrumentation overhead as event counts are lower. Event payload, on the other hand, is likely to be more complex and diverse on higher levels. High-level instrumentation may also lead to over-simplification, in that relevant details are not captured. Also the number of instrumentation points may be a relevant criterion. System-call–level instrumentation is a common compromise for monolithic operating systems (e. g., used in [YLW+06]),

```
L4_INLINE void
ferret_vlist_post_2wX(ferret_vlist_t * l, uint16_t maj, uint16_t min,
                      uint16_t instance, uint32_t d0, uint32_t d1)
{
    // allocate on top of next stack page, 128 B reserve for local allocations
    ferret_list_entry_common_t * e = (ferret_list_entry_common_t*)
        (((((unsigned long)__builtin_frame_address(0)) - 128U) & L4_PAGEMASK) -
         sizeof(e));
    e->major     = maj;         // initialize e on new stack page
    e->minor     = min;
    e->instance  = instance;
    e->data32[0] = d0;
    e->data32[1] = d1;

    asm volatile (
        "movl  %%esp,%%ebx              \n\t"  // save old stack pointer
        "movl  %2,%%esp                 \n\t"  // align to new stackframe
        "pushl %%ebx                    \n\t"
        "pushl %2                       \n\t"  // standard parameters
        "pushl %1                       \n\t"  // ...
        "pushl %0                       \n\t"  // ...
        "call  __ferret_vlist_postf     \n\t"
        "movl  12(%%esp),%%esp          \n\t"  // restore old stack pointer
        : "=a"(dummy), "=c"(dummy), "=d"(dummy)
        : "0"(l), "1"(offsetof(ferret_list_entry_common_t, data32[2])), "2"(e)
        : "memory", "ebx"
        );
}
```

Figure 4.9: C wrapper that moves all data into a single stack page before calling the actual VList sensor code. This simplified implementation assumes that the local stack below the current stack pointer cannot be corrupted, for example, by asynchronous signal handlers. Otherwise, the protective setting of the stack pointer to the address of `e` and the initialization of `e` would have to be done in assembler in that order.

as some semantic is available at this level and existing instrumentation solutions can be reused (e. g., the ptrace interface).

This section is partitioned into two subsections in which I detail my experience with two specific *microkernel* systems — DROPS on L4 and Singularity.

4.2.1 Instrumentation of Drops on L4

L4 allows rich and untyped communication between processes, in fact, IPC is one of the core building blocks of L4. Hardware interrupts, exceptions, and other error conditions (e. g., scheduler time-slice overruns [Ste04]) are also delivered in the form of IPC. In addition to that, also shared memory can be established and used for communication between processes. IPC payload is mostly opaque to the kernel (except for kernel-generated IPCs, of course), that is, the semantics of the message content depends on

the receiver's interpretations[1]. Fiasco already allows tracing of relevant IPC metadata in the kernel's tracebuffer [Wei03].

Interfaces for many components on DROPS are specified with IDL (Interface Definition Language) and interface code is generated by the IDL compiler DICE [Aig07]. These interfaces provide much more semantics for instrumentation. Döbel implemented a DICE plugin for augmenting DICE-generated stub code with FERRET instrumentation [Döb06].

However, not all components use IDL for describing their communication interface. Furthermore, for not-rigidly componentized programs or the para-virtualized Linux kernel L^{4}Linux and its programs, I had to use application-specific instrumentation locations.

There is no single location for DROPS that allows easy, general-purpose instrumentation.

In practice, I instrumented DROPS in several places or used sensors from different levels for a complete system picture. Fiasco's tracebuffer forms the lowest level. IDL-embedded instrumentation and manual instrumentation of several L4Env services comprise the next level. The L^{4}Linux kernel is instrumented itself as a regular L4Env program. It also acts as a proxy for its user-space programs in that it provides them with access to a dedicated FERRET sensor.

A few problem-specific, high-level instrumentation points often have to be embedded into user-space programs.

4.2.2 Implementing Event Tracing for Singularity

On first sight, Singularity looks like *a system made for instrumentation.* Shared memory is never used between applications. Almost all application-crossing communication happens through strongly typed and stateful interfaces. Already the interface and message names carry high-level semantics (e. g., `R_CreateUdpSession` in the `NetStack` interface).

All interface code is directly compiled into CIL bytecode from interface description. There is no intermediate representation to instrument at source-code level, as there is with DICE for DROPS.

Instead of modifying the Sing# compiler, for a first prototypical implementation I chose the approach of directly instrumenting at the CIL-bytecode level. Therefore, assemblies have to be disassembled first to a textual form using `ildasm.exe`. A perl script then inserts a CIL–source-code template shown in Figure 4.10. I then reassembled the instrumented textual representation to binary form again using `ilasm.exe`. As of now the Sing# compiler can directly generate instrumented code.

For synthesizing requests for Singularitys web-server Cassini, I unfortunately also needed further instrumentation in the kernel[2]. I also used a single manual instrumentation point in the web-server to determine the start of new requests. For convenient orientation in trace files, I also captured interactive shell command lines.

[1] This has changed slightly with the introduction of message tags for IPC in newer experimental Fiasco versions. However, I do not examine this in the context of this work.

[2] Interrupt delivery; channel creation, transportation, and connection; low-level synchronization primitives: AutoResetEvent, ManualResetEvent, Mutex, WaitHandles; Thread creation and blocking with timeouts.

```
IL_${name}: ldc.i4    ${curcontract}
IL_XYZ1:    ldc.i4    ${msgid}
IL_XYZ2:    ldc.i4    0
IL_XYZ3:    ldarg.0
IL_XYZ4:    call      instance int32 [Singularity.V1]Microsoft.   \
                        Singularity.V1.Services.EndpointCore::    \
                        get_ChannelID()
IL_XYZ5:    ldc.i4    0
IL_XYZ6:    ldc.i4    0
IL_XYZ7:    ldc.i4    0
IL_XYZ8:    ldc.i4    0
IL_XYZ9:    call      void [${kernel}]Microsoft.Singularity       \
                        .Monitoring::Log(unsigned int16,          \
                          unsigned int16, unsigned int16,         \
                          unsigned int32, unsigned int32,         \
                          unsigned int32, unsigned int32,         \
                          unsigned int32)
IL_XYZa:    ret
```

Figure 4.10: One template used for instrumenting compiled Singularity code at CIL-bytecode level.

4.3 Portability and tracing special environments

In this section, I will briefly describe FERRET's architecture taking portability into account. Good portability allows comparative studies of different systems as FERRET can provide easily comparable traces. I will also briefly discuss particularities of FERRET versions for plain L4, L4Env, the L^{4}Linux kernel and its user-space programs, the Linux kernel, and Singularity in Section 4.3.2.

4.3.1 Portability and architecture

FERRET's *core functionality* is separated into several libraries. Usual sensor access code uses only shared memory and therefore is completely API independent and portable[3]. There are three roles for sensor access: setting up and destroying sensors, producing events, and consuming events. These roles are executed by three distinct parties: the sensor directory, monitored programs, and monitors. Therefore, functionality is provided in three separated libraries.

Setting up access to sensors and releasing access is specific for monitored programs and monitors and their environment. Therefore, distinct *communication libraries* are used for this purpose. These libraries have to be adapted to the target environment of the respective entity.

Auxiliary functionality, like parsing event–format-string descriptions, format conversions, and compression is encapsulated in a third library class.

[3] Experimental sensors with blocking semaphores are an exception to this rule (cf. to Section 4.1.6).

4.3.2 Portability and versions

FERRET's *L4Env version* is the most technically matured version. All sensor types are available and the sensor directory is fully available. Sensor memory management is integrated with L4Env's region mapper and pinned memory is provided in the form of dm_phys dataspaces.

In FERRET's version for *plain L4* programs (e. g., the name server `names`), specific sensor-memory placement has to be done. In addition to that, libraries for dataspace mapping have to be linked.

The L^{4}Linux kernel is a regular L4Env program as far as FERRET is concerned, consequently, no special handling is required. L^{4}Linux user-space programs have no access to the sensor directory for creating sensors. For that reason, I use the L^{4}Linux kernel as a proxy. The kernel currently provides a sensor for its user-space programs simply by mapping it into their address spaces into a dedicated region. Of course, a more elaborate scheme can be implemented that provides the complete sensor-directory namespace to L^{4}Linux user-space programs (similarly to relay [ZYW$^+$03]), and that allows full access to single sensors using the `mmap` system call.

FERRET has also been ported to the *Linux kernel*. Monitors obtain access to sensors via `mmap` on special files. Unmodified event production and consumption code can be used.

For the implementation of the *ETS (Event Tracing for Singularity) prototype*, I chose to initially use only a single system-wide list sensor. As all user code in Singularity is managed code, access to sensor memory can be restricted without using system calls. Instead, I could provide safe access code without system-call overheads system wide (also cf. to Section 4.2.2).

For FERRET, I use a trace file format that is derived from ETW's trace files, such that the adapted Magpie version can directly import it. The format is basically a simple concatenation of events in their binary representation, prefixed with their respective length. The file format also supports meta-events preceding larger chunks of simple events. These meta-events carry information about the trace and the target machine.

The binary layout of events is described in trace-message–format (TMF) files, which follow ETW's TMF event description in syntax (cf. to Figure 3.2 for an example).

Chapter 5

Evaluation

A man should look for what is,
and not for what he thinks should be.

(A. Einstein)

I start this chapter with a short recapitulation of previous' chapters contributions, followed by an overview of this chapter's content.

Chapter 2 contains a discussion of related work and shows that there is no solution in the literature that combines all relevant properties for runtime monitoring for open real-time systems (cf. to Section 3.1.2). Chapter 3 explains how these properties can be achieved from an architectural standpoint. The succeeding Chapter 4 explains the implementation of the previously presented design and highlights specific techniques.

The current chapter answers three main questions:

1. It answers the question whether by using Ferret a broad set of problems can be addressed using a single monitoring approach. I therefore describe several groups of use cases (model building, behavior checking, and others) for which Ferret (or its predecessors) were used in the following section.

2. In the second part, in Section 5.2, I discuss qualitative properties of different sensor approaches and compare the approaches with regard to those properties to determine which approaches are suitable for which environments.

3. To compare the costs caused by using the different sensor approaches, the third part in Section 5.3 contains a detailed quantitative evaluation of Ferret running in different scenarios, using microbenchmarks as well as macrobenchmarks.

5.1 Use cases

In this section, I will describe various use cases for which Ferret (or predecessors) have been used. I partition these cases into three groups: In Section 5.1.1, I describe cases for which establishing behavioral models using measurements taken with Ferret is a main objective. Section 5.1.2 explains cases for which the adherence or nonadherence to assumed system properties has to be assessed. A description of how to collect dynamic call-graph data from device drivers and a discussion of Event Tracing for Singularity comprises Section 5.1.3. I briefly summarize important peculiarities for each case.

```
for (j = dy + 1; j--; ) {
    d = (u32 *)dst; s = (u32 *)src;
    for (i = dx + 1; i--; ) *(d++) = *(s++);    /* copy line */
    src += scr_width;
    dst += scr_linelength;
}
```

Figure 5.1: DOpE's inner copy routine.

5.1.1 Model building

I use the term *model building* here for gathering information about real existing software and hardware, and for using this information to craft behavioral models that allow predicting the timing behavior of these software-hardware combinations.

I discuss four specific use cases in this section. It starts with a description of modeling the resource demand of the inner copy routine of DROPS' *Desktop Operating Environment* (DOpE) [FH03], that is, resource requirements of a *small part* of a real-time component are analyzed. Following that in Section 5.1.1.2, I explain how *larger requests*, spanning several components, can be analyzed. Section 5.1.1.3 discusses model building for a *real-time–capable disk scheduler*. A description of how runtime monitoring was applied in a *real-time video player* scenario follows in Section 5.1.1.4.

5.1.1.1 DOpE resource demand modeling

With DOpE, DROPS has a real-time display component, which can guarantee refresh rates for a requested rectangular area. DOpE keeps track of average and worst-case times for copying pixels from a shared-memory representation to graphics memory. The time estimation for the copy routine is based on area size to be copied (pixel count). In the following, I will describe how I used FERRET for verifying this estimation.

The foundation of this estimation is DOpE's inner copying routine (shown in Figure 5.1), which I instrumented to measure the time in processor cycles for copying rectangular areas.

I compute the time per pixel in place and store the information in a two-dimensional histogram indexed by the width and height of the rectangle. The histogram has two layers as well, whereas the first layer contains the accumulated copy time, and the second layer counts the number of occurrences for this width–height combination. Further processing of this data is done offline or, for online visualization, asynchronously in another task (monitor). The instrumentation code only contains cheap integer operations and no division or floating point operations.

I directly aggregate the information online to minimize the memory load in this experiment as I am taking a huge number of measurements. Redraw requests are created randomly with uniformly distributed width and height between 1 and 400 pixels using a small benchmark program. The benchmark runs until each point is measured at least 100 times (this usually takes several hours on the mentioned test machines). Then the average copy times are computed.

I took measurements on the following two machines:

Machine A has an AMD Duron processor with 1 200 MHz. First and second-level caches have 64-B cache lines. The machine has a 64-KiB level-1 instruction cache (2-way associative), a 64-KiB level-1 data cache (2-way associative), and a 64-KiB level-2 unified cache (8-way associative).

Machine B has an older Intel Pentium-Pro processor with 200 MHz. First and second-level caches have 32-B cache lines. The machine has an 8-KiB level-1 instruction cache (4-way associative), an 8-KiB level-1 data cache (2-way associative), and a 256-KiB level-2 unified cache (4-way associative).

In the experiments depicted in Figure 5.2 one can see that the fixed per-pixel-cost assumption is a viable approximation for a large range of rectangular sizes. However, also diversions in several places can be seen:

1. The mountainous area on the left side results from copy operations on rectangles with a small width. The huge increase in per-pixel copy times for very short pixel rows probably stems from a combination of the overhead for computing the pixel row addresses for source and destination buffer (loop overhead in Figure 5.1) and memory-system access latencies.

2. There are small trenches parallel to the height axis, corresponding to the cache-line size. Copying whole-numbered multiples of cache-lines sizes reduces overheads *per pixel*.

3. There is a valley in the front corresponding to the total processor cache size. The measurements depicted in the Figures 5.2a, 5.2b, and 5.2c were taken with a horizontal screen resolution of 1 024 pixels, resulting in aliasing effects with the cache colors[1]. The addresses of a pixel column contain only few index-bit combinations and therefore only few different cache lines can be used by a pixel row. This leads to thrashing the own cache set when copying areas with more than a certain height, *independently* of the width of the rectangle (therefore the valley has roughly rectangular shape).

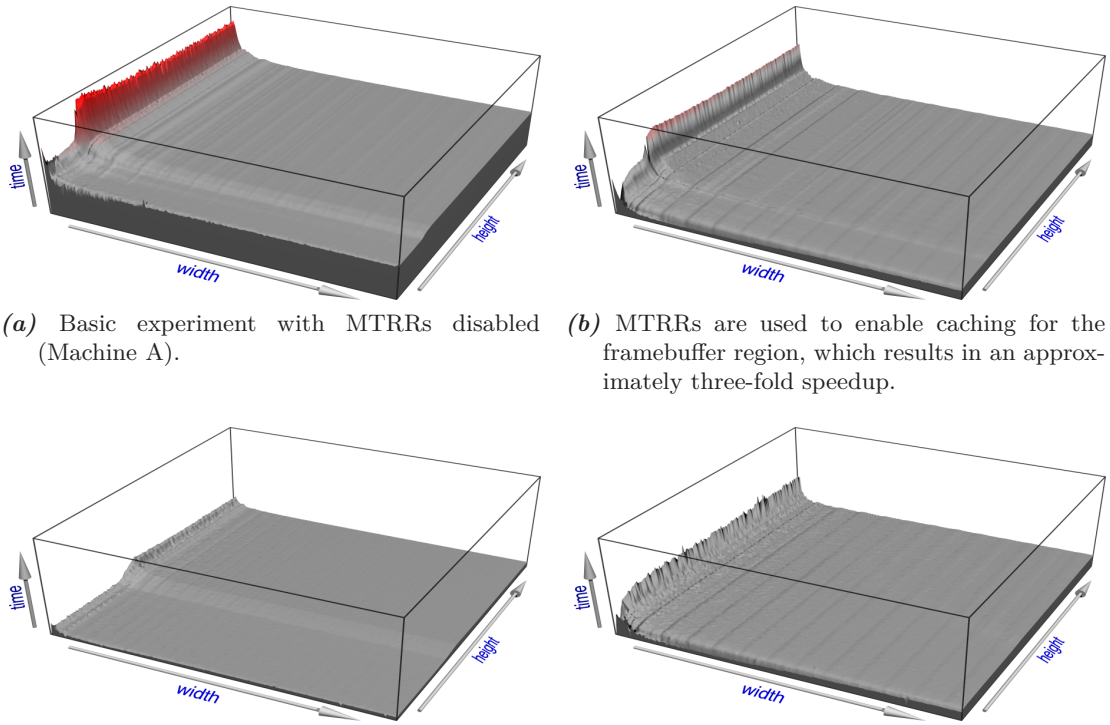
   Machine A has a data cache of 128 KiB (L1 and L2 added as the cache hierarchy is exclusive). Running with a graphics mode of 1 024 pixel/line, with 2 B/pixel results in a cache-trashing height of 64 lines, computed as follows:

   $$128\,\text{KiB}/(2\,\text{B/pixel} * 1\,024\,\text{pixel/line}) = 64\,\text{lines} \tag{5.1}$$

   This computed cache-trashing height can be seen in Figures 5.2a and 5.2b.

   The most prominent difference between Figures 5.2b and 5.2d is that data for Figure 5.2d was taken with an `800x600` resolution, whereas data for Figure 5.2b was measured with an `1024x768` resolution. In Figure 5.2b, the cache-trashing line

[1] Addresses belong to the same cache color if they compete for the same index in the cache, that is, if they have the same index bits, cf. to [Dil,Lie96a].



(a) Basic experiment with MTRRs disabled (Machine A).

(b) MTRRs are used to enable caching for the framebuffer region, which results in an approximately three-fold speedup.

(c) Also with MTRRs enabled but on *Machine B*, the memory access profile looks quite different.

(d) The same setup as in Figure 5.2b but the resolution is reducded to 800x600, eliminating the cache aliasing effect.

Figure 5.2: Depicted is the time for copying a single pixel to graphic-card memory, depending on the width and height of the rectangular area copied. The width and height axes range from 1 to 400 pixels each, the time axis shows relative times for each machine, so you can compare 5.2a, 5.2b, and 5.2d quantitatively (*Machine A*).

can be clearly seen (parallel to the width axis), whereas in Figure 5.2d it is gone, as there is no aliasing between cache colors and pixel columns.

Machine B has an effective data cache size of 256 KiB (maximum of L1 and L2 as the cache hierarchy is inclusive). Using equation 5.1 with otherwise equal settings I compute a cache-trashing height of 128 lines that can also be seen in Figure 5.2c.

For comparison I also run the measurements with *memory type range registers* (MTRR) disabled, as this was the initial situation when DOpE was created in the year 2002. One can see that enabling MTRRs for the framebuffer area results in an approximately three-fold speedup (compare Figures 5.2a and 5.2b). However, the *relative* differences in the histogram have *grown* with enabled MTRRs as well (compare the mountainous area on the left with the height of the flat area in the middle and right side), making the fixed–per-pixel-cost assumption questionable.

Also, in the measurements taken on Machine B, the differences in the copy-times per pixel contrast even stronger as depicted in Figure 5.2c.

From the experiments one can see that the typical optimization for throughput (e. g., using MTRRs) hurts predictability and thus might create problems for real-time applications. However, an execution-time model based not only on the area size but also on the length of both rectangle sides seems feasible from the measurements shown. Model calibration to a target machine could happen at component deploy time, or, if only few measurements are required, at startup time. Also, refining the model online seems practical for soft–real-time problems.

Summary of experiment properties The scenario described previously has very short sections to be timed accurately. I decided to use only online aggregation here as other alternatives would be to intrusive or expensive. Storing events in memory for later evaluation would require to much memory[2] and drastically saturate the memory subsystem (which is a central part of the to-be-observed system here). Online processing in an external monitor process would also have required to touch the same amount of memory over the experiment and would require additional scheduling in the system, thereby also biasing the results. I therefore implemented an online-aggregating sensor in form of a two-layered histogram.

The measurement and instrumentation is straightforward in this example. The critical block of code is wrapped within timestamping macros. After the second timestamp is taken, the offset into the histogram is computed and the measured time duration is inserted into the histogram.

Consequently, influence on the code to be measured is minimized in several ways: nothing is modified inside the code section to be observed, timestamps are taken with minimal overhead (using the processor timestamp counter), durations are aggregated online in order to minimize sensor size (cache pollution), and the small computation for indexing into the histogram is done after the observed section, almost directly before DOpE's event handler finishes, after which typically a context switch takes place.

5.1.1.2 DOpE requests

This project started with the following three questions:

> What exactly happens with the system when I move the mouse cursor in DOpE? Which parts in the system do interact and how? How expensive is this?

By means of the cross-component activities in a DOpE desktop setup I want to demonstrate the request concept (cf. to Section 2.3.1).

To recapitulate, requests are used to track chunks of work in a system, where typically more than one component is involved and chunks are not trivially small. Furthermore, requests are typically user defined and problem specific. Also, as they usually span more than one component, their development is not coupled tightly with components' development. Concrete request definitions can, for example, arise from debugging needs or from questions such as the ones posed above.

[2] $(400\,\text{pixel})^2 * 100\,\text{measurements/pixel}^2 * 2\,\text{events/measurement} * 64\,\text{B/event} \approx 1.907\,\text{GiB}$

The environment for this experiment consists of basic L4Env services and DOpE as a graphical user interface. One could easily make this setup more complex by adding L^{4}Linux with an X-Server to the setup, as Ferret also supports monitoring L^{4}Linux user-space programs. However, this would only complicate the following explanation without providing added value, so I refrain from doing so.

The setup, as it is, already contains four types of causality flows: normal synchronous IPC, interrupt IPC, time-driven activation in DOpE threads (polling), and events in shared memory buffers.

The first step to answering the questions posed above is determining which components are involved in the concrete interaction at all. This can usually be answered by a combination of domain-expert knowledge (I started with some initial idea which components will be involved) and basic communication monitoring.

The setup in its simplest form comprises DOpE itself, the libraries `input` and `omega0`, the `l4io` server and the Fiasco microkernel. This assumes that system setup is finished, so there will be, for instance, no page-fault interaction with memory managers visible in the requests.

Writing the schemata for later request extraction with Magpie is laborious and typically involves going through an instrument–run–extract–evaluate cycle several times (cf. to Section 3.2.2.2). Ideally, components are delivered with pre-marked instrumentation points at strategically relevant points, combined with documentation that allows schema writers to derive relevant semantics. Several commercial operating systems have actually been following this path for some time now [CSL04,Micb], and also for Linux there are approaches in development that would allow doing so [Des08,DD06]. For this concrete scenario with the research system Drops, I instrumented most parts manually. The Fiasco kernel has semantically relevant events for this scenario (generic context switches, IPC start and delivery).

Once all instrumentation is in place, the typical workflow involves the following steps: starting of the test setup, activation of the instrumentation, actually running the test setup, stopping the instrumentation or recording of events, transferring the gathered trace for offline evaluation, and finally doing the offline evaluation using the schema within Magpie.

Section 2.3.1 motivated why external request definition is to be preferred over internal definition. I now complement this abstract discussion with a description of what would have to be changed for internal request definitions on the basis of the current example: First, the starting of a new request must be done within the first event of this request. From there on, the new request identifier must be propagated recursively through the system and all involved APIs. In this scenario however, I do not know whether an event belongs to an old request, a new request, or none at all until several events later. Second, even if I could somehow manage to guess new request starts, the system would have to be modified *extensively* to propagate the corresponding request identifiers. I would have to change internal function call signatures, IDL interface definitions, the kernel's system-call ABI, and, for shared-memory–buffer—assisted causality flow, the internal layout of structures. This approach could hardly be called minimally invasive.

Instead, I simply post events in relevant places that may contain local identifiers. I use schemata to glue together those small snippets of events sequences externally.

In the following, I describe details about the instrumentation used here and several particularities of the schema. I use manual instrumentation for this project and modified the components identified previously as follows:

DOpE I added sensor initialization code and inserted several events around drawing routines (esp. mouse updates) and inside input event handling functions.

input I wrapped function calls to `omega0_request_timeout` and `__omega0_wait` within events.

omega0 I wrapped function calls to `signal_user_threads` and `handle_user_request` within events.

l4io I only added sensor startup code to be used by linked libraries (`input`, `omega0`). There is no instrumentation in l4io itself.

Fiasco I activated context-switch events and IPC events in the Fiasco kernel debugger [GML06] (`-jdb_cmd=JHOO+O2+I*IR+`).

Please note that this set of instrumentation points is not a minimal set required for request extraction. Instead, it is the set of events I used to understand the causality flow in this setup for constructing the schema. Now that the request schema definition is finished, the instrumentation could be reduced to the set of actually used events in the schema.

The schema, as shown in parts in Figures 5.3 and 5.4, basically contains two types of callback functions: prehandlers and handlers.

Prehandlers are called in *stream order*, as Magpie extracts requests from the trace file. They are used for demultiplexing the overloaded IPC-event semantics (`kernel_IPC_prehandler`), for splitting up compound events into single instances that better fit Magpie's model (`kernel_context_switch_prehandler`), and for parsing and unpacking binary structures (`kernel_IPC_res_prehandler`). Prehandlers can, in particular, be used for injecting additional events into Magpie's event stream as, for instance, done within `kernel_context_switch_prehandler`. In essence, prehandlers function as a kind of *preprocessor*.

Handlers are the actual core of schemata. Magpie calls handlers in *timestamp order*. To this end, Magpie uses a reorder buffer, which can be configured in size. In the current schema, I use handlers for several purposes: *First*, for simply making request visible for later manual inspection (`BIND_NONE`), *second*, for seeding requests (for each request, Magpie's flood fill algorithm has to start with one root event), and *third*, I use the special bind types `BREAK_BEFORE` and `BREAK_AFTER` for bounding requests, such that the flood filling algorithm terminates eventually. *Forth*, events can be bound to more than one timeline, thereby building a bridge for the request finding algorithm. For example, in `kernel_ACK_IRQ`, events are bound to their thread timeline and the interrupt timeline.

The schema shown in Figure 5.4 works mostly stateless, except that it tries to keep track of the first motion event within DOpE's event loop (`dope_B_event_loop`). If there is such an event it is used as request root (handler `dope_Event_motion`) and the flood-fill

```
def kernel_IPC_prehandler(ev):
    if ev.getAttrValue("snd_desc") == 0 and \      # check for IRQ event
           ev.getAttrValue("rcv_desc") == 0 and \
           ev.getAttrValue("dest") > 0 and ev.getAttrValue("dest") < 256:
        irqevent = event.Event(ev.guid, ev.version, 100, ev.ts)
        irqevent.name = "IRQ"
        irqevent.provider = "kernel"
        irqevent.setAttrValue("irq", ev.getAttrValue("dest") - 1)
        return [irqevent]
    elif ev.getAttrValue("snd_desc") == 0 and \    # check for ACK_IRQ event
           ev.getAttrValue("rcv_desc") == 0xFFFFFFFF and \
           ev.getAttrValue("dest") > 0 and ev.getAttrValue("dest") < 256:
        irqevent = event.Event(ev.guid, ev.version, 101, ev.ts)
        irqevent.name = "ACK_IRQ"
        irqevent.provider = "kernel"
        irqevent.setAttrValue("irq", ev.getAttrValue("dest") - 1)
        context = long(ev.getAttrValue("context"))
        irqevent.tid = l4util.tidstr_from_context(context)
        return [irqevent]
    else:                                          # normal IPC event
        context = long(ev.getAttrValue("context"))
        ev.tid = l4util.tidstr_from_context(context)
        dest = long(ev.getAttrValue("dest"))
        ev.setAttrValue("dest_tid",
            "%s" % (l4util.tidstr_from_l4_threadid_t(dest)))
        return [ev]

def kernel_IPC_res_prehandler(ev):      # taskid and threadid from context
    ev.tid = l4util.tidstr_from_context(long(ev.getAttrValue("context")))
    ev.setAttrValue("rcv_tid",
        l4util.tidstr_from_l4_threadid_t(long(ev.getAttrValue("rcv_src"))))
    return [ev]

def kernel_context_switch_prehandler(ev):
    ev.tid = l4util.tidstr_from_context(ev.getAttrValue("context"))
    ev.setAttrValue("dest", l4util.tidstr_from_context(
                              ev.getAttrValue("dest")))
    ev.setAttrValue("dest_orig", l4util.tidstr_from_context(
                                   ev.getAttrValue("dest_orig")))
    ev.name = "switchto"
    ev2 = ev.copy()
    ev2.name = "switchfrom"
    ev2.ts -= 1
    return [ev, ev2]
```

Figure 5.3: Excerpt from DOpE request schema: prehandlers.

```
def kernel_IPC(ev):
    key = ("tid", ev.tid)
    BIND(ev, key, ev.ts, schema.BIND_NORMAL)

    # if the snd descriptor of this event is 0xFFFFFFFF, then it is
    # an open_wait() and we must not rely on the dest_tid.
    snd_desc = int(ev.getAttrValue("snd_desc"))
    if (snd_desc == 0xFFFFFFFF):
        key = ("tid", "WAIT_ANY")
        BIND(ev, key, ev.ts, schema.BIND_NONE)
    elif (ev.getAttrValue("dest") > 0 and ev.getAttrValue("dest") < 256):
        key = ("tid", ev.getAttrValue("dest_tid"))
        BIND(ev, key, ev.ts, schema.BIND_NONE)
    else:
        key = ("tid", ev.getAttrValue("dest_tid"))
        BIND(ev, key, ev.ts, schema.BIND_BREAK_BEFORE)
    return [ev]

def kernel_ACK_IRQ(ev):
    key = ("tid", ev.tid)
    BIND(ev, key, ev.ts, schema.BIND_NORMAL)
    key = ("irq", ev.getAttrValue("irq"))
    BIND(ev, key, ev.ts, schema.BIND_NORMAL)
    return [ev]

def dope_B_event_loop(ev):
    global first_in_event_loop
    key = ("tid", ev.tid)
    BIND(ev, key, ev.ts, schema.BIND_BREAK_BEFORE)
    key = ("irq", 12)      # bind this to the mouse irq
    BIND(ev, key, ev.ts, schema.BIND_NORMAL)
    first_in_event_loop = True
    return [ev]

def dope_Event_motion(ev):
    global first_in_event_loop
    key = ("tid", ev.tid)
    BIND(ev, key, ev.ts, schema.BIND_NORMAL)
    key = ("irq", 12)      # bind this to the mouse irq
    BIND(ev, key, ev.ts, schema.BIND_NORMAL)
    if first_in_event_loop:
        ev.isrequest = True
        first_in_event_loop = False
    return [ev]
```

Figure 5.4: Excerpt from DOpE request schema: handlers.

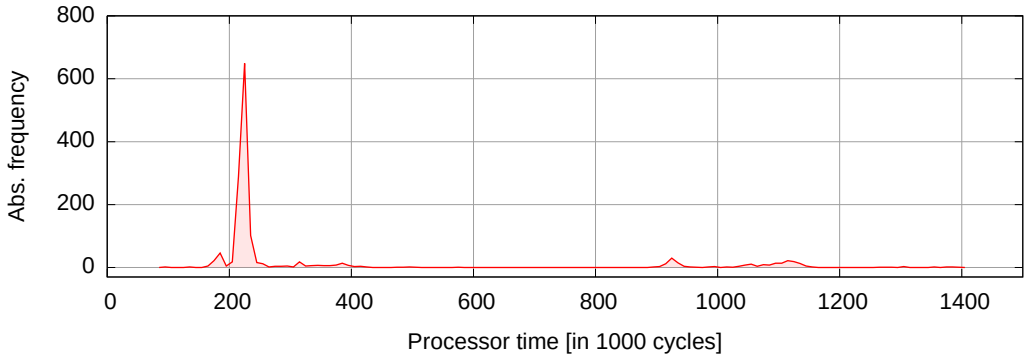


Figure 5.5: Histogram over the processor-time demand for single DOpE request.

algorithm starts from there. This also illustrates why request definition cannot always be done at the beginning of a request (cf. to Section 2.3.1).

Figure 5.5 shows a histogram of the processor-time demand for DOpE requests. I adapted resource-demand computation in Magpie to L4 as follows: Thread timelines are treated in a special way, as they correspond directly to processor time if scheduled. Magpie basically adds up all segments of thread timelines which belong to a given request and which are bound to a processor (are scheduled). The premise here is, that the schema correctly identifies request boundaries and scheduling. Given that, accurate cross-component resource accounting is possible. Figure 5.6 shows the Magpyvis visualization of a single DOpE request as identified by the schema.

In Figure 5.5 we see that most DOpE requests take approximately 200 000 cycles. However, there are also two small clusters at 900 000 and 1 100 000 cycles. Given this accounting facility, one can now generate resource-demand models for requests with both average-case and worst-case information. Also identifying requests with the largest execution time and inspecting them for anomalies and optimization options becomes possible.

Summary of experiment properties For the current scenario only small events are required (typical payload of 1–2 integers and one thread ID). Accurate timestamps are required for a subset of the events, namely those that denote request boundaries, as resource accounting has to be done. For the other events, at least global ordering is important so as to infer causality.

A low intrusiveness instrumentation is required (as performance evaluation is desired). Several system parts have to be instrumented: the microkernel, libraries, and programs. Several sensors are used in the system and event producers can run concurrently. In the current, general instrumentation situation, approximately 90 events are created per request. This can be reduced by dropping instrumentation points not relevant for this specific problem. An online monitor retrieves, orders, and stores these events for offline processing with a Magpie schema.

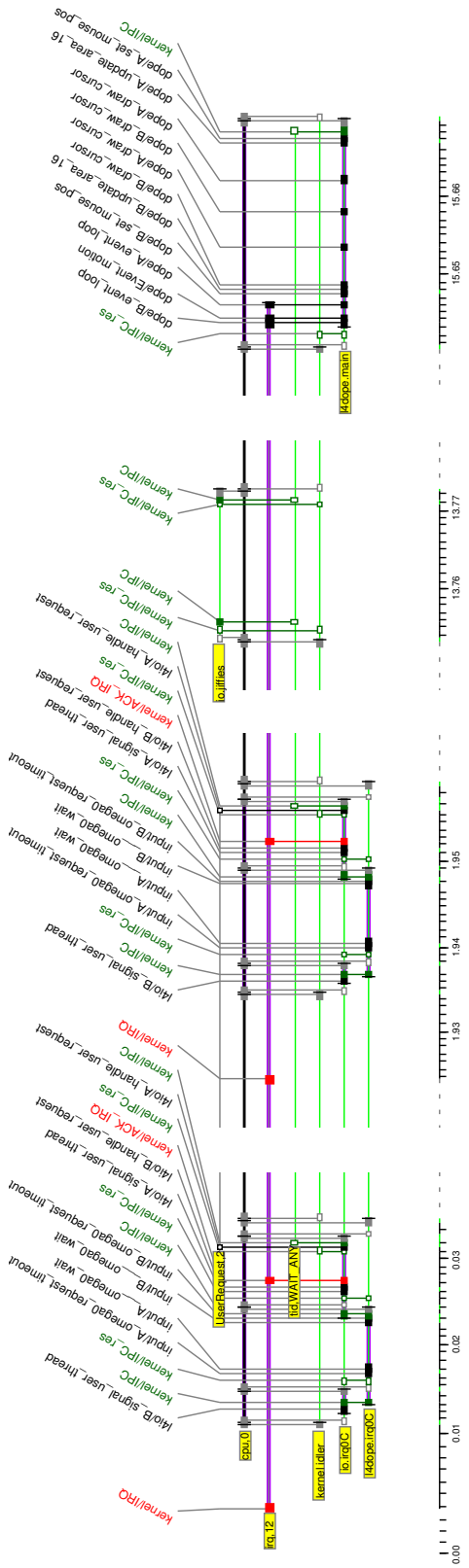


Figure 5.6: Shown is a single DOpE request containing two mouse interrupts and the resulting interactions in the system (around 0.00–0.03 ms and 1.93–1.96 ms) and a single, delayed redraw operation in DOpE (at 15.65–15.66 ms). The jiffies thread receives an alarm at around 13.76 ms. The time axis shows relative time in ms and is stretched and compressed in parts to enhance readability. Horizontal lines are timelines and vertical lines represent events. The dots show where events are bound to timelines.

5.1.1.3 Hard-disk–request modeling

The goal of this project is to build a real-time–capable hard-disk–request scheduler, which should not only be able to guarantee deadlines for single disk requests, but also be able to fully utilize remaining disk bandwidth for background and best-effort load [RP03a].

I will describe the runtime-monitoring aspects thereof in two parts. The first part explains the construction of an execution-time model for disks and the tools that were used to obtain necessary data. In the second part, I describe how this model was applied in the context of a real-time scheduler and how runtime monitoring supplemented it.

Model building In [Poh03,Poh02] I described the construction and usage of an execution-time model for disk request. The model's purpose is to predict how much time a certain disk request will *probably* take (average case) and how much time *maximally* (worst-case). Both predictions are required for online disk-request scheduling, for both real-time and best-effort requests.

The model requires the following information about target disks:

- Rotational speed,
- Number of platter surfaces used,
- Disk zones (regions of similar track length in sectors),
- Angular length for the sectors of each zone,
- Angular offset for each track's start to the first track's start,
- Seek time as a function of traveled cylinders,
- Mapping type (how logical sectors are physically positioned on disk), and
- Statistical information about seeking accuracy.

For obtaining this data, I use Linux with an instrumented kernel and record important events in the disk drivers (e. g., request sent to disk, result ready). These events primarily contain two pieces of information: A location identifier and a timestamp (*what* happens *when*).

A user-space model-building application sends probe requests to the disk to measure its temporal behavior. It obtains the in-kernel events through a dedicated character device, shortly after sending out the probe requests. Both the monitoring infrastructure and the model builder are tightly coupled in a loop, that is, the event consumer does not directly require real-time responses to its requests, but it often has to wait for event arrival before it can send out the next probe requests. Building a full disk model — including the whole physical layout — takes several hours. Even small delays in event delivery add up pretty severely in this scenario.

The monitoring infrastructure itself is relatively simple as it only has to serve one dedicated purpose. Only one event consumer has to be supported at a time and event memory is claimed back in the kernel directly after event delivery to user space.

To access the disk hardware as directly as possible, I disable the drives' internal write caches and circumvent Linux' buffer cache using `raw` devices (similar to the `O_DIRECT` flag for file access). As hardware read caches cannot be disabled reliably for all disks, I mostly use write requests for modeling.

As Linux is not a real-time operating system, I have to apply statistical filtering on the data obtained to eliminate erroneous measurements. For instance, for extracting the real rotational speed of a disk, the modeling program repeatedly issues pairs of write requests to the same disk sector, measuring the distance between the pairs' completion events. Some request pairs are not executed end-to-end by Linux, but rather the model-building program is interrupted between sending both requests. Consequently, the distances between such requests are much larger than what is to be expected for a typical disk drive. The real rotational speed is determined by eliminating these outliers and computing the median of the remaining data.

Timestamp requirements are high, contrary to what one might assume on first glimpse (disks are relatively slow peripheral devices). Timestamps must be accurate enough to measure the length of a single sector for a 15 000 rpm (rotations per minute) disk, with about 1 000 sectors/track [Poh03,Poh02] (probably even higher with today's disks, as recording density increases). A 15 000 rpm disk revolves with 250 rotations per second, that is, a single sector is under the disk head for only a $\frac{1}{250*1\,000}$ s.

Assuming an (extremely optimistic) model-building requirement of only 10 % accuracy error for the length of a single sector, timestamps have to have an accuracy (and therefore resolution) of $\frac{1}{250*1\,000*2*10}$ s, that is, 0.2 µs *minimum*. I therefore chose the processor's timestamp counter as time source. On a 1 GHz processor 0.2 µs correspond to 200 cycles.

Summary of experiment properties For model building only one event consumer needs to be supported, whereas event creation can happen in different places in the Linux kernel (different drivers, different levels in interrupt handlers). Therefore, synchronization in the kernel is done with a central reader-writer lock. Synchronization against the consumer happens also in the kernel similarly, as the consumer acquires events via system calls.

Memory requirements are relatively low and buffer management is simple, as typically only few events need to be stored before they are consumed (about 2–10 per disk request, depending on the amount of instrumentation). However, I later used the same kernel module for collecting filesystem access patterns. Thereto, the buffer size had to be increased significantly. Events need to be quickly available for the consumer so as to minimize the turn-around time for model building (an interactive usage pattern).

Finally, accurate timestamps are required for model generation.

Model usage We use the model extracted in the previous section for a real-time disk scheduler [Poh03,RP03a] for DROPS.

In summary, we timestamp each disk-request completion event. This timestamp is used for two purposes: *First*, it is used for accounting disk-usage times according to Quality-Assuring Scheduling [HLR+01]. *Second*, it is used for determining the current angular position of the disk platters. A current timestamp and the sector number of

the previous request combined with the model of the physical disk layout allows us to compute this angular position.

With this position, with the list of outstanding requests (for which, too, the physical position on disk can be determined using the layout model), and the execution-time–prediction model, the disk scheduler can choose subsequent disk requests to be executed.

In addition to minimizing the seek time — as traditional disk scheduling algorithms do — here, also the rotational delay is minimized.

Summary of experiment properties Compared to the model-building requirements described in Section 5.1.1.3, here we have more relaxed requirements. Only one time-stamped event is used at a time. There is only one producer (interrupt service routine) and one consumer (disk scheduler), so there is no need for event buffer management.

However, timestamps need to be highly accurate here as well and the event must be available effectively immediately. Any jitter in time measurement will lead to wrongly computed angular positions and will therefore result in wrongly predicted executing times. This, in turn, will lead to suboptimal scheduling decisions that reduce disk throughput and increase latency.

5.1.1.4 Verner

In [HZP+07a] we describe an extension to component-based software engineering that supports real-time properties through the whole lifecycle of software development. We demonstrate this by means of one example throughout the whole paper — a video player application (Verner, cf. to [Rie03]).

We use RT_Mon, a predecessor of Ferret, for both online and offline monitoring and measuring in this context:

- The offline part basically comprises measuring the distributions of execution times of the different components involved, such that scheduling can be planned at component deploy time.
- We also use RT_Mon's different sensor types within the running system for several purposes: to verify scheduling behavior (timeliness of component scheduling, event sensor), to monitor underflow and overflow of component-buffer fill levels (event sensor), to observe how processor-time demand changes with quality-level adaptations in the video player (histogram sensor), and to monitor processor-usage changes depending on the position in the video stream (both distribution and worst-case times, histogram sensor). We additionally monitor time-slice overruns in real-time mode with simple scalar counters and we use a two-dimensional histogram for monitoring execution-time predictions.

To do this, we instrument Verner's components with sensor code, basically around the work loops and in the case of the video decoder component (by far the most resource demanding and complex part) also inside the loop, before the optional work part. All sensors are registered in a sensor directory, using a hierarchical namespace. Online browsing of this namespace and online attaching to its sensors for visualization is possible. Figure 5.7 shows a typical session with visualization for different sensors.

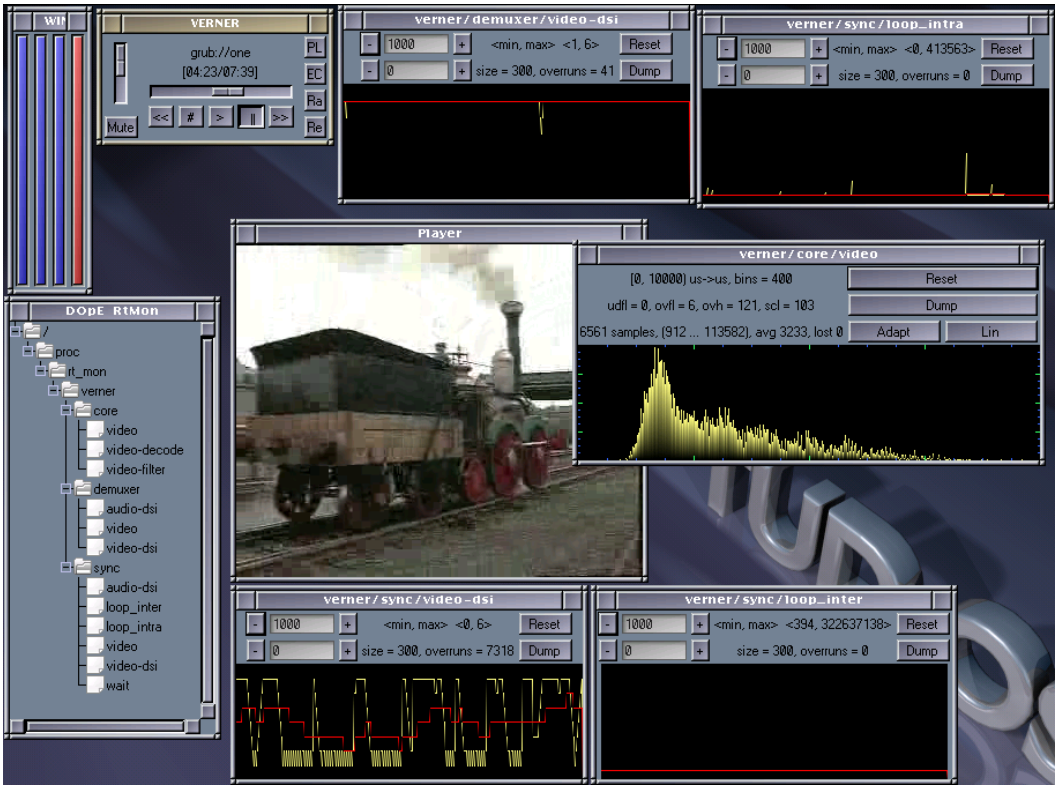


Figure 5.7: Screenshot of a typical session with active runtime monitoring. The hierarchical sensor namespace is depicted in the lower left window. Five other windows visualize sensor content.

Summary of experiment properties For this scenario, highly accurate timestamps are required to obtain accurate processor-time–demand measurements for components. Furthermore, thread-relative timestamps or thread-time accounting is needed, in case the to-be-timed work loops in components are interrupted.

Monitoring must induce only little overhead in terms of processor time, but it must also not affect scheduling of the real-time components in the system, as scheduling is potentially to be monitored and debugged.

The monitoring system must support several independent sensors (as there are several independent components) and tolerate multiple instances of components. It must be able to distinguish events from those instances. In addition to that, several different sensor types proved useful: event sensors, counters, histogram (one and two dimensional).

Interactive browsing of all available sensors and online visualization of important ones should also be supported without influencing the to-be-observed components significantly.

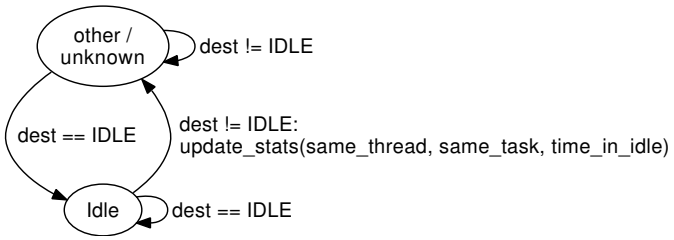


Figure 5.8: State machine for idle_switch_mon.

5.1.2 Behavior checking

The following section contains four use cases for which actual behavior is observed and compared to constraints. For the first use case, I describe how assumptions for kernel optimization can be validated with FERRET. Following that in Section 5.1.2.2, I explain how violations of scheduling constraints can be established. Section 5.1.2.3 details the application of FERRET to user-space driver development. The forth use case in Section 5.1.2.4 describes how the behavior of a para-virtualized version of Linux — L^{4}Linux— can be compared to its native counterpart.

5.1.2.1 Idle-switch optimization

Kernel developers in our group had the following questions regarding kernel optimization:

> How often is the thread that was blocked last before an idle period on a processor the one that is scheduled directly after the idle period? Does it make sense to support this special case separately?

The advantages of handling this case specifically are:

- Context switches could be saved on idling, as the idle thread would be scheduled within the address space of the current task, instead of in its own.
- The wakeup-latency could be reduced as no context switch would be necessary for this special case.[3]

In the following, I will describe how these questions can be answered with FERRET.

The scenario is relatively simple as only one sensor is involved — Fiasco's tracebuffer. I configured the tracebuffer to only log context-switch events and wrote a simple monitor to consume those events. The monitor consists of a small state machine (see Figure 5.8) that, besides counting the relevant cases, also measures how long the system idles. This idle-time information could be used similarly to Intel's powertop utility [Int08] for Linux.

From running the monitor parallel to a typical desktop setup with L^{4}Linux, I obtained the following results: In 2–3 % of all cases where the idle thread was scheduled, the same thread was scheduled directly before and after it. In 10–13 % of all cases where the idle

[3] Summary of a private email on the l4-intern mailing list.

thread was scheduled, the same address space was active directly before and after the idle thread. Short-term fluctuaions, for example, when starting up L^{4}Linux, ranged up to 20 % for this number.

Summary of experiment properties Only a single sensor with a single event type is required for the idle-switch–optimization setup. However, the kernel is the source of events in this case. For the pure frequency analysis any order-preserving logical clock would be sufficient as timestamp source. In case the idle duration shall be evaluated, clock resolution used for events must match accuracy requirements of the evaluation.

5.1.2.2 Taming L^{4}Linux

In this section, I will describe one specific problem I encountered with L^{4}Linux, its cause, and a way how to identify the problem now and in the future using Ferret. This shall demonstrate the utility of Ferret for finding timing bugs in such complex components as operating-system kernels.

Native Linux uses `CLI` (Clear Interrupt Flag) and `STI` (Set Interrupt Flag) instructions to protect critical sections by disabling and enabling interrupts, respectively. L^{4}Linux uses a replacement for `CLI`–`STI` as it runs without kernel privileges in user space where these instructions are not available. `CLI`–`STI` is replaced with a mutex that is acquired using atomic instructions in the noncontention case. In the contention case, a blocking IPC is sent to one serializing *tamer* thread. The tamer has the highest thread priority inside the L^{4}Linux kernel. Thereby, it is able to execute its code atomically with respect to all other L^{4}Linux kernel threads. When a thread leaves a critical section and there was contention, the leaving thread also notifies the tamer that, in turn, wakes up one contended thread according to its queuing policy. Here again, priority is used to guarantee atomic code execution with respect to all other L^{4}Linux kernel threads.

L4 kernels have a feature called donation, which optimizes a common communication case, where a client sends a short request to a server that immediately processes it and returns the result. However, the server in this scenario runs with the time slice *and the priority* of the client while processing the request.

Some L^{4}Linux driver stubs communicate with external servers via IPC, for example, for using external network-hardware drivers. The external server's worker threads may run on the same priority or even on a higher priority than the tamer thread inside L^{4}Linux as both are in separate subsystems.

This situation was assumed to be a reason for L^{4}Linux' stability problems. I wrote a monitor to verify this problem in a running system (its state machine is shown in Figure 5.9). I therefore wrapped the tamer's atomic sequence with start–stop events. Additionally, I enabled logging of context-switch events in the microkernel. The monitor checks for the following constraint: There must never be a context switch to an L^{4}Linux kernel thread (except the tamer thread itself) in-between a start event and a stop event of the atomic sequence. The monitor also keeps the recent event history for later visualization and debugging of the problem. Figure 5.10 shows one such captured constraint-violation situation.

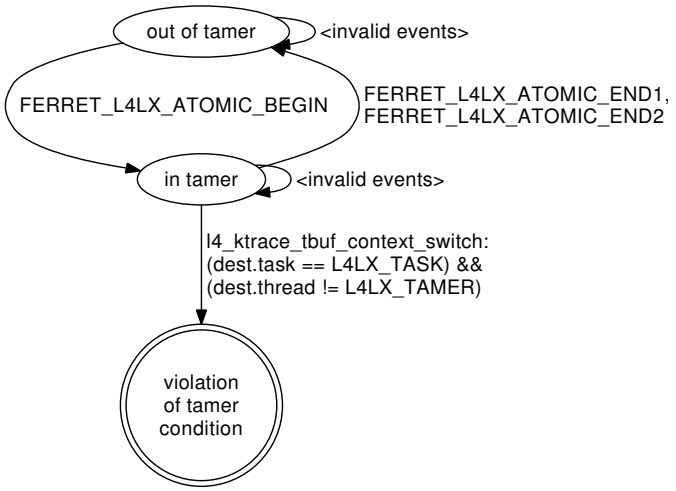


Figure 5.9: State machine for tamer monitor l4lx_verify_tamed_mon.

In our case one L^{4}Linux-internal stub driver thread *A* (`tid,11.9` in Figure 5.10) was notified by an external thread *W* (`tid,a.6`) running on the same priority as L^{4}Linux' tamer thread. By notifying A, W temporarily transferred its priority to A. As a consequence, although extremely rarely, the tamer was interrupted in its atomic sequence when A itself wanted to enter the `CLI`–`STI` critical section.

I describe how we fixed the tamer problem with the help of model checking in [Poh06]. Now that the problem is fixed, this test can be run after any changes made to L^{4}Linux. The bug would be hard to identify with other means, as it requires a whole-system view as events from several threads and the kernel are required and their exact order matters. Also, the monitor checks the atomicity condition completely outside of the L^{4}Linux kernel's address space, which makes the checking immune to, for example, memory corruption by other potential bugs in L^{4}Linux.

I would like to note here that after formulating the constraint, writing the monitor only took approximately one hour. After few minutes of runtime, I could capture a violation case. In contrast, the transformation of the tamer algorithm into Promela code for model checking with the Spin model checker [Hol06] took considerably more time than a week.

Summary of experiment properties Sensors in two system levels are used in the Taming-L^{4}Linux setup: Fiasco's tracebuffer for scheduling events and instrumentation in the tamer thread. Events from both sensors must be fully ordered, as such, a logical clock would be sufficient for event timestamps. For convenient visualization and to aid debugging, the monitor keeps a recent history of events and makes it available in the error case.

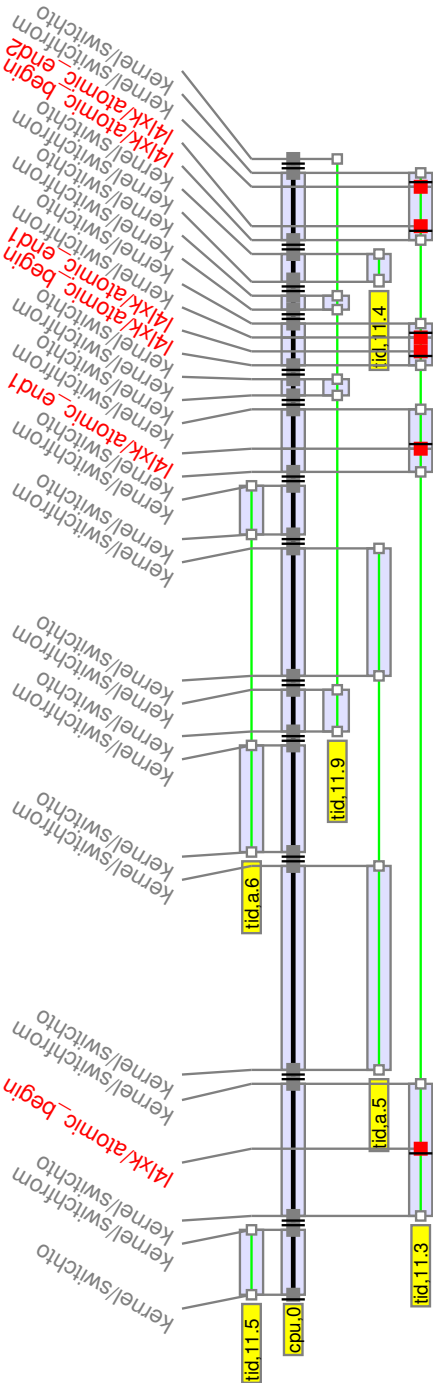


Figure 5.10: Timeline visualization of the atomicity problem with the tamer thread (tid,11.3). Time flows from left to right. Single events are typically associated with a processor timeline (e. g., `cpu,0`) and a thread timeline (e. g., `tid,11.3`). You see three atomic sections wrapped in the events `l4lxk/atomic_begin` and `l4lxk/atomic_end[12]`. The first of the three sections is interrupted with context switches (denoted by `kernel/switchfrom`–`kernel/switchto` event pairs) to thread A.5, A.6 (both in the external driver), and 11.9 (an L^{4}Linux-internal worker thread). The context switch to 11.9 violates the atomicity condition as the tamer thread is preempted by a normally lower-prioritized thread.

5.1.2.3 Constraint checking in drivers

DROPS' software network switch ORe [Döb08] once showed two problems, *first*, it crashed sometimes and persistently resisted debugging attempts. *Second*, it showed a performance regression after switching from DDELinux2.4 to DDELinux2.6 [Hel08].

We (ORe's author Björn Döbel and me) used FERRET to exclude several assumed crash reasons by instrumenting ORe and by applying a constraint checker in the form of a Magpie module to recorded traces. We successfully checked for four constraints:

1. There should never occur an RNR (receive no resource) interrupt,
2. The functions `e100_intr()` and `e100_poll()` must never be executed in parallel (top- and bottom-half interrupt handlers),
3. `e100_intr()` and `e100_poll()` each must always be executed in the same thread, and
4. `e100_intr()` is never called with NIC (network interface controller) interrupts disabled.

After that, a visual inspection of the recorded communication trace quickly hinted to the problem's cause: The memory used as DMA (direct memory access) target for the driver was of the wrong type, it was not pinned and not necessarily physically contiguous.

Furthermore, we could observe communication with the memory server at runtime, not only in the setup phase. The memory backend used in DDELinux2.6 is based on a slab-cache implementation that is optimized for runtime growth and shrinkage of memory pools. This may lead to a high runtime overhead if unfavorable usage patterns lead to a high communication frequency with the memory server.

This problem was already mentioned in [Fri06] and can easily be verified by a runtime monitor that monitors the communication frequency with the memory server.

Summary of experiment properties For this scenario again, events from several system layers are required. Context-switch events and IPC events from the Fiasco kernel denote communication patterns. Instrumentation events from within the network driver code (including the interrupt handler) denote higher-level semantics and allow a more fine-grained behavior control for visual trace inspection and constraint specification. Event representation should be compact as a hot system path is instrumented heavily.

The previously mentioned constraints do only require event order. For visual inspection and the mentioned online-checking of communication patterns, real-time timestamps for events are desirable. Visual event browsing played a key role in identifying the nature of the problems, a clear and intuitive event and communication-pattern representation in the form of the Magpie visualization program proved to be very helpful.

5.1.2.4 Comparison of native and virtualized operating system kernel (fork)

When (para-)virtualizing operating-system kernels, such as Linux, it is important to detect behavioral changes. This applies not only to the functional correctness of the

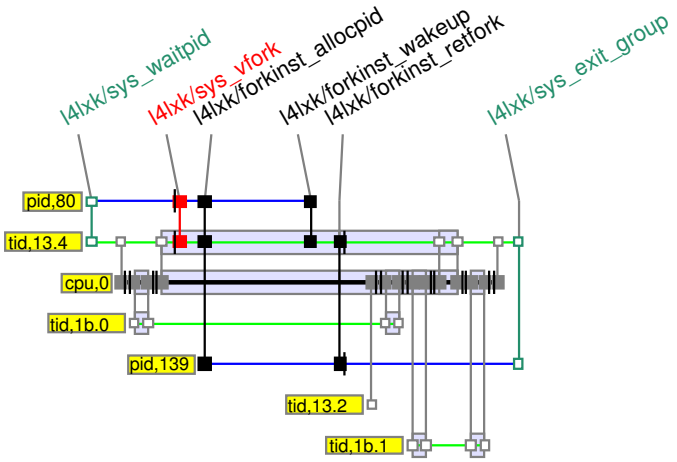


Figure 5.11: Depicted is a timeline view of a `vfork` system call, issued by pid 80 (`tid,1.b0`). After the L4Linux kernel (`tid,13.4`) returned from the fork routine (`l4lxk/forkinst_retfork`), the parent process (`tid,1.b0`) is scheduled again for a short time, before finally the child (`tid,1b.1`) is scheduled.

virtualized version but also to non-functional properties, such as speed and memory requirements.

In [PDL06] we compare L4Linux to native Linux running comparably configured kernels. We use kProbes [Kri05] to track control flow inside the Linux kernel. For L4Linux, we use Fiasco kernel-level instrumentation to track scheduling. In native Linux, an additional kProbe is used for this purpose. To reuse the Ferret instrumentation code from L4Linux also within native Linux, we implemented a Linux kernel module that provides parts of the Ferret framework.

One of our experiments showed a difference in scheduling behavior between L4Linux and native Linux within the `vfork` system call. `vfork` is a special version of `fork` that assumes that only little work is done between `fork` and `exec` (or `exit`). Therefore, parent and child can share the same address space (including the stack), which saves the overhead of copying (or marking read-only, for copy-on-write) the parent's page tables. The parent process has to be blocked to prevent address space corruption until the child calls `exec` or `exit`. In essence, parallelism is bypassed by blocking one process as anticipated parallelism is very short and the copying overhead would not pay off.

While observing the behavior of `vfork` in L4Linux, we found, that after the system call has been handled by the L4Linux server, the Fiasco kernel switches back to the parent task before executing the child (see Figure 5.11). We further investigated this behavior, because this looked like a serious bug in either L4Linux or the Fiasco kernel. As it turned out, this behavior is only an optimization artifact in Fiasco, never visible to user space. Fiasco indeed switched to the parent process, but only to find out that it is blocked. The parent process is still in Fiasco's ready queue but flagged as blocked. Fiasco uses an optimization called lazy queuing that does not always remove IPC senders from the ready queue but only marks them with a flag. Often, the IPC receiver answers fast and Fiasco can directly switch back to the sender, thereby saving two queue operations.

`vfork` triggers the contrary case, where the parent was flagged as blocked and must now be removed from the ready queue.

Although this bug turned out to be a false positive, it shows that FERRET can be used in finding and debugging behavioral differences between native and (para-)virtualized variants of Linux. Timing information from the events can also be used to point out performance differences.

Summary of experiment properties For this comparison experiment, we use two sets of sensors. For the L^{4}Linux case it is the Fiasco tracebuffer and the L^{4}Linux internal sensor. The native Linux system uses a FERRET version implemented as a kernel module.

The kProbes used in L^{4}Linux caused additional context switches as the trap handler runs in a separate thread. We did filter out those predictable context switches in a Magpie schema. In general, the usage of a trapping technology such as kProbes is not advantageous in a time-sensitive environment that FERRET ultimately targets. However, kProbes allowed us to place and refine the probes in running systems without having to go through complete system-rebuilding cycles. Furthermore, it demonstrates that orthogonal technologies can be used in conjunction with FERRET (but without removing their disadvantages, of course).

For pure structural comparisons, a logical clock as timestamp source would be sufficient. For additional performance assessments, highly accurate timestamps are required. As with previous multi-sensor use cases, events from the different sensors have to be fully ordered.

To actually compare L^{4}Linux with Linux, we had to port parts of FERRET to the native Linux kernel.

5.1.3 Other use cases

I will describe two heterogeneous use cases in the following section. The first details how FERRET can be used for collecting dynamic call-graph data from device drivers and similar environments. In the second part, I discuss Event Tracing for Singularity.

FERRET was also used by Hoffmann in [Hof08] in the context of an intrusion-detection system, but I will not further discuss this in the context of this work.

5.1.3.1 Dynamic function-call tracing

Dynamic call graphs as described in [GKM82] are used for obtaining an overview over a complex piece of software and for detecting hot spots. They are usually obtained by instrumentation of the target program. The gcc compiler supports this with function-level instrumentation (`-finstrument-functions` command-line switch). One can define call-back functions to be executed for each function entry and exit.

For gcc, these functions have the following signature:

```
void __cyg_profile_func_enter (void *this_fn, void *call_site);
void __cyg_profile_func_exit  (void *this_fn, void *call_site);
```

On each call the caller's and the callee's address are passed along for further processing.

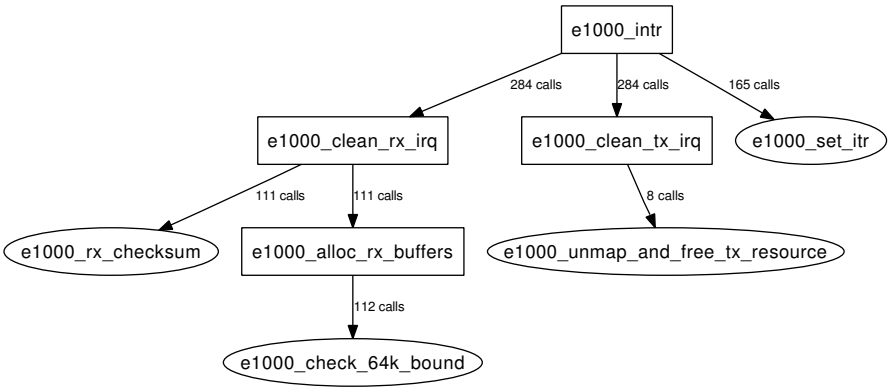


Figure 5.12: Dynamic call graph for the E1000 interrupt thread.

I wrote a very small library (below 50 lines of C code) that basically just posts an event inside both of these functions containing the addresses and the current thread ID into a FERRET sensor. Additionally, I used a small constructor function to set up this sensor at program startup.

Post-processing is done with Magpie using a simple schema that unpacks the binary trace and splits it up for each thread. I then feed the per-thread text traces through a slightly modified version of pvtrace [Jon05] to generate per-thread dynamic call graphs.

As a test case, I applied this approach to the E1000 ethernet driver inside the ORe software network switch [Döb08]. Figure 5.12 shows the dynamic call graph for the interrupt thread. The noteworthy part here is, that all I had to do was to globally activate the compiler flag and link my library to otherwise nontrivial Linux kernel code.

Summary of experiment properties For this experiment, FERRET is only one of several tools involved. Its role is only to store and transport calling information as unobtrusively as possible as it is called *twice per function call*. For the pure call graph, event order is only used for offline determination of calling sequences, hence a logical clock would be sufficient as timestamp source. It is, however, important in this scenario that FERRET uses only minimal system infrastructure for collecting events, as potentially every single function in the target program (including libraries) is instrumented and cannot be used by FERRET without causing recursive calls to itself.

5.1.3.2 Event Tracing for Singularity

I designed and implemented a tracing infrastructure for Microsoft's research operating system Singularity, which I called Event Tracing for Singularity (ETS). Magpie [BDIM04,BIMN03] was designated as tool for trace post-processing.

I first discuss several particularities of Singularity in the context of event tracing. Following which, I explain design goals and decisions for ETS. I conclude this description with a discussion of results and problems.

In *Singularity* [HLA+05,HL07], all normal processes and the kernel share one hardware address space. Isolation is provided at the language level by so called Software Isolated

Processes. Nearly all system communication happens through channel contracts that are also explicitly supported at the language level. Channels are strongly typed and stateful. All applications are precompiled to native code and linked statically before being started in Singularity. Because of that, whole-program optimization can be applied (bartok compiler). There are no means for runtime patching or dynamic code generation in Singularity. A Singularity system consists of many small components connected with channel contracts. There is no support for shared memory for user-space programs.

From a high level, *design goals* for ETS were facilities for fine-grained resource accounting on request level and observing channel protocols. Therefore, a compact and low-overhead tracing framework for fine-grained events was required. Instrumentation should, of course, perturb the system as little as possible. It must especially not interfere with the system's causal control flow.

I decided to use a single global event sensor for all applications and the kernel, based on the design from [Rie05]. The sensor is nonblocking, uses no locks, and is multiprocessor safe (Singularity did not fully support multiprocessors at that time). Sensor access is encapsulated in a library and happens from within `unsafe` code blocks (the C# language construct). Usual application code is not allowed to directly contain such code, because language safety can be circumvented that way. For tracing, it allows system-wide access to the global sensor without expensive context switches or kernel entries. Consequently, using a single sensor for the whole system is a safe approach. Only system-controlled code (the library) has access to the sensor and there is no way that a simple application can corrupt the sensor's state or arbitrarily read from or write to its memory.

ETS is loosely modeled after Event Tracing for Windows (ETW) so as to ease Magpie's adaptation (Magpie was an ETW-only client before). But unlike with ETW, event consumers are not notified synchronously by ETS. They have to use polling, similarly to Ferret.

ETS events are usually 48 byte large and have a common header format (28 byte) containing timestamp, thread ID, process ID, processor ID, provider, type, version designation, and the instruction pointer. The remaining 20 byte are application specific, for instance, channel IDs can be stored there. Event creation takes approximately 500 cycles. Larger events (e. g., containing strings) are also supported but are assumed to be rare.

Two instrumentation approaches lend themselves for establishing actual requests. They differ in the instrumentation level that is used.

For the *first* approach several *low-level* primitives need to be instrumented directly in the kernel. `EventHandle` and its derivatives `AutoResetEvent`, `ManuelResetEvent`, and `Mutex` cover channel communication, interrupt delivery, timeouts, mutual exclusion, and custom synchronization primitives. They also play a role in blocking and unblocking of the thread class. In addition to that, several methods for waiting and waking up are relevant (`AcquireOrEnqueue` and `NotifyOne`). Thread scheduling and de-scheduling on timer interrupts and `yield` also need to be covered.

Instrumentation can directly be performed in the kernel's source code and only the few central points mentioned previously have to be touched. This approach is simple to realize and covers an overwhelming majority of all cases. However, the amount of events generated is huge and each event carries very little semantics. Writing schemata

at this event level is very cumbersome, if not impossible. In essence, schemata have to contain a model of the kernel that recreates few high-level–semantic events out of a flood of meaninglessness.

The *second* approach targets the high-level contracts with their built-in valuable semantics. Executed contract code is not available in source-code form. Instead, generated binaries have to be instrumented. I described this approach in Section 4.2.2 in more detail. This high-level approach comfortably enables referring to communication partners and message types in schemata without any preprocessing. In particular, messages for initiating communication sessions can be handled specially. The disadvantage of this approach is that not all relevant system activities are covered (e. g., hardware interrupts and custom thread synchronization, as used in the NIC driver).

Consequently, I used a *mixed approache*, relying as much as possible on high-level events, using low-level instrumentation for the small rest. I am confident that large parts of the low-level instrumentation can be covered at higher layers in the system, given more time. This will eliminate many of the then superfluous but still numerous low-level events in traces.

Summary of experiment properties To summarize, I designed and implemented ETS, a tracing framework for Singularity and I extended and adapted Magpie to process ETS traces. As a first test case, I wrote a schema for requests of Singularity's web-server Cassini. The schema covers five system components: NIC driver, network stack, Cassini web server, Cassini extensions (e. g., BrowserWebApp), and NameServer (for Singularity's namespace). The schema is intuitively readable due to the high-level semantics in contract event names. Components can easily be separated in traces at contract boundaries. In [BIMH06], Barham et al. use ETS for inferring state machines about a Singularity system and its components.

5.2 Qualitative evaluation

In this section, I summarize qualitative key properties of different event-sensor types and their approaches to allow assessing for which environments they are suited. I omit a discussion of specialized sensors such as histograms or scalars here because of the vast design space for such special solutions.

AList and *VList* sensors were introduced in detail in Sections 4.1.6.1 and 4.1.6.2. The *GList* sensor represents GRTMon's Concurrently Invocable Sensors, which I discussed in Section 3.2.4.2. *SLists* and *ULists* use blocking semaphores for synchronization. They serve purely as a performance reference here. SLists use L4Env user-level semaphores (which use IPC in the contention case). ULists use Fiasco's User Locks for synchronization (they block in the kernel in the contention case). *DPLists* use an experimental delayed-preemption implementation in Fiasco. Fiasco's *tracebuffer* [Wei03] is an in-kernel implementation used mostly for in-kernel events (e. g., context switches and interrupt occurrences).

I discuss the following properties (named P. 1–8 in Table 5.1):

Table 5.1: Qualitative properties P. 1–8 of the different sensors compared (cf. to Section 5.2 for a detailed explanation of the properties).

Sensor Type	P. 1	P. 2	P. 3	P. 4	P. 5	P. 6	P. 7	P. 8
AList (rollf.)	RF	yes	yes	yes	no	∞ (∞)	no[b]	yes
AList (rollb.)	RB	no	yes	yes	no	∞ (∞)	no[b]	yes
VList	RF	yes	yes	yes	yes	∞ (∞)	no[b]	yes
GList	CAS	yes[a]	yes	no	no	bounded (∞)	no	yes
SList	S	no	no	no	no	∞[d] (∞[d])	no[c]	no
UList	U	no	no	no	no	∞[d] (∞[d])	no[c]	no
DPList	DP	yes	yes	yes	no	∞ (∞)	no[b]	?
Tracebuffer	K	yes	yes	N/A	no	∞ (∞)	yes	no

[a] If number of threads (activities) is bounded and known
[b] Not in the common case, a kernel entry is only required if a hardware interrupt hits
[c] A kernel entry is only required in case of contention
[d] Limited by lock implementation

1. Which synchronization method is used: atomic execution in the kernel (K), rollforward (RF), rollback (RB), compare-and-swap (CAS), L4Env Semaphore (S), Fiasco's User Lock (U), or delayed preemption (DP)?
2. Can a safe upper bound be determined for the execution time of the sensor code?
3. Is the sensor code real-time capable from a system perspective (it does not block, it does not indefinitely delay the system or other sensor code)?
4. Can multiple sensors be used safely and efficiently in the system (a view on a sensor is always current or becomes current after bounded time, cf. to Section 3.2.4.2)?
5. Are events of variable size supported?
6. How many event producers (and consumers) are supported for a single sensor?
7. Is a kernel entry required?
8. Are different software environments supported (cf. to Sections 4.2.1 and 4.3)?

As can be seen in Table 5.1, several sensors are suitable for usage in real-time situations: AList (RF), VList, GList, DPList, and the Tracebuffer. Each approach has restrictions:

- For the rollforward approach (AList and VList) kernel support for atomic sections is needed.

- The GList sensor is somewhat limited in its size[4] and several sensors cannot be used safely for online evaluation. Only fixed event sizes are supported.
- The delayed-preemption approach (DPList) is complicated to realize and probably impractical (cf. to Section 3.2.4). Here too, kernel support is required.
- For the tracebuffer, kernel entry costs have to be paid for each event (bad for throughput-oriented applications), the tracebuffer supports only fixed event sizes (overhead if smaller events are used, larger events are impossible), it is limited in size (currently to 32 768 entries), and it cannot be reached from all applications — in short, it is inflexible for anything but in-kernel events.

I think that both rollforward sensors (AList and VList) provide a good compromise for many scenarios if kernel support is available.

5.3 Quantitative evaluation

I will present and discuss quantitative results in the following so as to allow comparing and estimating monitoring costs for the different sensor approaches. In the context of this work with its real-time focus, I restrict myself to execution time as the key cost. I start by describing the four test machines in the next section. The succeeding section is devoted to the System Management Mode and real time. Section 5.3.3 presents microbenchmark results. Macrobenchmark findings are depicted in Section 5.3.4.

5.3.1 Hardware description

For the low-level measurements, I used these four different test machines:

Machine A Intel Celeron (Willamette), 1.7 GHz

Machine B Intel Pentium 4E (Prescott/Nocona), 3.2 GHz

Machine C AMD Opteron (Santa Rosa), 2.0 GHz

Machine D Intel Core 2 (Merom), 2.66 GHz

5.3.2 System Management Mode

The System Management Mode (SMM) is a special processor mode for x86 processors designed to run below everything else in the system (including the operating system) (cf. to Chapter 24 in [Int06] and Chapter 6 in [Adv06]). It is meant to be undetectable, uninterruptible, and transparent, such that any legacy software on top can be dealt with. The SMM is invoked by SMIs (system management interrupts) and is, for example, used

[4] The List sensor uses `CMPXCHG8` for atomically updating 8-byte structures. It therefore currently has the restriction that the bits for the current sensor index and the global event counter can only use 64 bit altogether (e. g., if a sensor size of 2^{16} event is used, 2^{48} events are supported over the sensor's lifetime; with a sensor size of 2^{32}, only 2^{32} events can be submitted to the sensor over its lifetime).

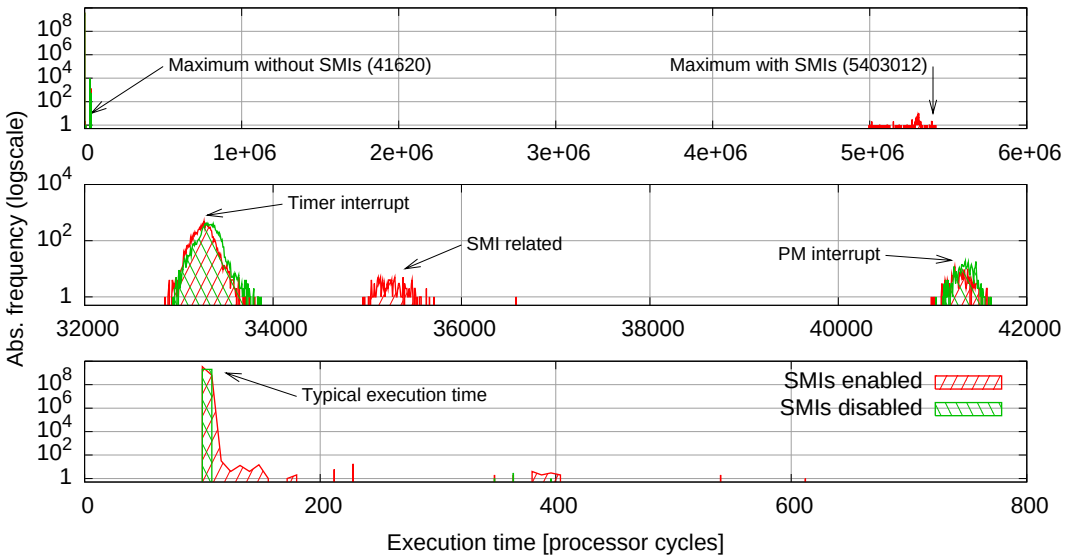


Figure 5.13: Depicted are different views (time scales) on a histogram of the execution times of a small counting loop for both enabled and disabled SMIs. The overall histogram with the maximum execution times is shown on top. In the middle, disturbances by hardware interrupts are shown. The bottom graphs details the common behavior.

to emulate a legacy keyboard for old software if only a USB keyboard is available and for power management (fan control etc.).

However, this transparency does not cover real time and therefore represents a massive problem for real-time applications. For my test machines, I regularly encountered *time holes* in the range of several million cycles for a setup with a single active and highly prioritized thread. Also context switches or system calls did not occur in these time holes.

A software solution cannot mask a real-time insufficiency of the underlying hardware. Therefore, one has to either use suitable hardware (hard to obtain) or use highly system-specific methods for disabling the SMM (e. g., as discussed in [Mar06]).

I use such platform-specific code to disable sources for SMIs for Machine B. Figure 5.13 shows the execution time of a simple counting loop (to 1 000) for both enabled and disabled SMIs. With enabled SMIs, the maximal measured execution time is about 5.4 million cycles. Also an SMM-caused anomaly at 35 000 cycles can be observed. Without SMIs, the observed maximum is 41 620 cycles. Also the typical execution time shows a more deterministic behavior with SMIs disabled. The measurement was executed 4 billion times in a loop.

5.3.3 Sensor microbenchmarks

This section on microbenchmarks is subdivided into three parts. I first show a throughput survey to enable a general comparison between different sensor approaches and for depiction of the average case. Following that, I present a more detailed evaluation for

the AList and the VList sensors, especially targeting the worst-case behavior that is relevant in the real-time domain.

5.3.3.1 Throughput survey

Table 5.2 contains throughput measurements for several different sensor types. Throughput numbers are relevant for my work for three reasons: *First*, throughput determines large-scale intrusiveness for non–real-time parts of the system. *Second*, it is useful for roughly gauging different implementations. *Third*, it allows to establish baseline performance comparisons with methods that use otherwise unsuitable synchronization mechanisms (e. g., the locking in SList and UList sensors).

Desnoyers et al. discuss LTTng microbenchmarks in [DD06] for a 3 GHz uniprocessor Pentium 4. They measured the average time spent in kernel probes as 288.5 cycles, the maximum is 6 997 cycles. Fast user-space probes (without system calls) take 297 cycles on average and a maximum of 88 913 cycles. Each probe writes 20 byte. These results can roughly be compared to the Machine-B column in Table 5.2 and show that Ferret is a competitive approach for best-effort system components.

In [KBD+08], Knüpfe et al. briefly discuss the function-call overhead of VampirTrace for a 1.6 GHz Intel Itanium II processor. The minimal overhead mentioned (for filtered-out events) is 0.82 µs (1 312 cycles). Recorded events cost 0.92 µs (1 472 cycles). I interpret the difference of 160 cycles between both numbers as an indication to the actual event-storage time which is roughly comparable to the Ferret results in Table 5.2.

5.3.3.2 AList

In the following I will describe the typical and the worst-case behavior of the AList sensor.

Figure 5.14 shows distribution times for posting events to an AList sensor. For the first measurement *AList (outside)*, I execute event-posting code, wrapped within timestamps, in a loop.

The two clusters labeled *Hardware interrupts* originate from two different points at which hardware interrupts hit the measurement. For the left cluster, interrupts hit *before* or *after* the actual rollforward section but inside of taking the two timestamps. Here, just the interrupt handling overhead adds to the execution time. For the right cluster, interrupts hit *inside* the rollforward section. The interrupt is enqueued, interrupt delivery is disabled temporarily, and rollforward execution is resumed on the instrumented version (cf. to Section 4.1.3). The final instruction of the instrumented version traps into the kernel again, where the queued interrupt is handled and hardware interrupts are enabled again. Therefore, slightly more time is required (lazy interrupt disabling *and* interrupt handling). This measurement does not reflect the real rollforward costs as it erroneously adds the interrupt handling costs.

In the second measurement *AList (inside)*, I therefore move both timestamp-taking code snippets into the rollforward section[5]. Consequently, hardware interrupts cannot

[5] I use the first experiment for factoring out the timestamping overhead.

Table 5.2: Shown are single-thread throughput results (in processor cycles per event) for different processor types, sensor types, and configurations (10 000 000 repetitions). The VList sensors were configured for a sensor size of 65 536 byte, all other sensors were configured to hold 1 024 events of size 64 byte. All events posted were of size 24 byte, except where noted otherwise. Numbers in parentheses were obtained with call-stack reuse, as described in Section 4.1.6.1. A qualitative discussion for the different sensor types is provided in Section 5.3. An introduction to single sensor types is given in Section 4.1.6 for ALists and VLists, in Section 5.2 for SLists and ULists, and for GList Sensors in Section 3.2.4.2.

Sensor Configuration	Machine A	Machine B	Machine C	Machine D
AList (rollforward)	200 (200)	172 (176)	43 (43)	110 (109)
AList (rollback)	212 (200)	184 (184)	50 (48)	139 (121)
VList	244	228	78	154
VList (64-byte events)	296	259	95	180
VList[a]	246	232	81	158
VList[b]	251	232	84	160
GList	657 (653)	273 (273)	167 (165)	215 (217)
SList	108 (108)	100 (100)	33 (32)	87 (87)
UList	112 (116)	100 (100)	41 (41)	91 (91)

[a] Page-aligned stack, as described in Section 4.1.6.2
[b] Page-aligned stack and event data on stack, as described in Section 4.1.6.2

disturb the measurement. Only the *Lazy interrupt blocking* costs can be observed. The measured maximum is at 12 120 cycles.

For the third measurement *AList (inside + cache flooder)*, I execute cache-flooding code before each event posting to trigger the worst-case. Of course, the observed maximum execution time increases (28 720 cycles). Interestingly, executions with *Lazy interrupt blocking* perform better than in the previous measurement.

5.3.3.3 VList

As the VList sensor supports variable event sizes, I measure its execution time depending on the event size. Again, the typical and the worst-case behavior are discussion targets. Figure 5.15 shows throughput results for the test Machines A–D. Execution time increases with event size. These results are relevant for non–real-time components.

The worst case for the VList sensor is constructed as follows: The sensor needs to be filled with the smallest possible events first, such that the first loop for advancing the tail pointer (cf. to Figure 4.5) needs maximal iterations. After that, the remaining space in the sensor until the end needs to be slightly smaller than the to-be-written event, such that a filler event is created and new room has to be created in the sensor again, this time at the beginning. Then the event itself needs to be copied to the sensor (this depends on the event size). Additionally to that, a hardware interrupt has to hit the

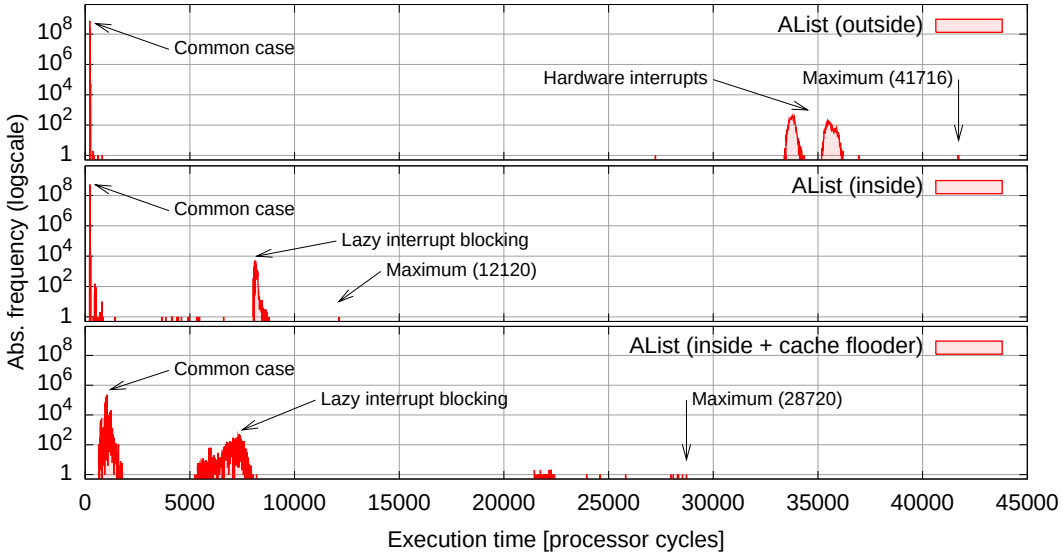


Figure 5.14: Distributions for AList event posting with three time-measurement setups: *First* outside of the post routine, *second* inside the rollforward section, and *third* inside and a cache flooder is called before *each* event. All measurements were done on Machine B.

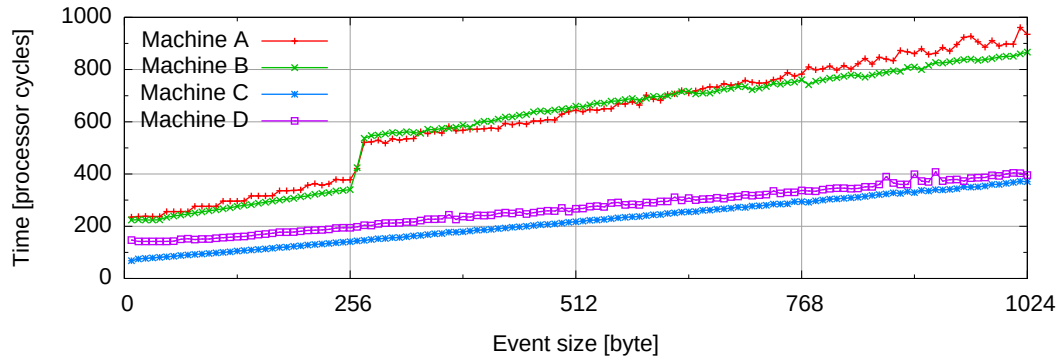


Figure 5.15: Throughput processor time vs. event size for the VList sensor.

rollforward section, such that the lazy–interrupt-disabling case is triggered (I repeat the experiment 100 000 times to trigger this case).

Figure 5.16 shows four experimental results: the average and the worst case for posting events, each with and without cache flooding performed before posting the events. All measurements were performed with the worst-case sensor setup described previously.

5.3.4 Macrobenchmarks

In the previous section, I investigated the behavior for the creation of single events in isolation. In the following, I analyze the overall influence that monitoring has on the basis of two concrete application scenarios — a non–real-time example in Section 5.3.4.2

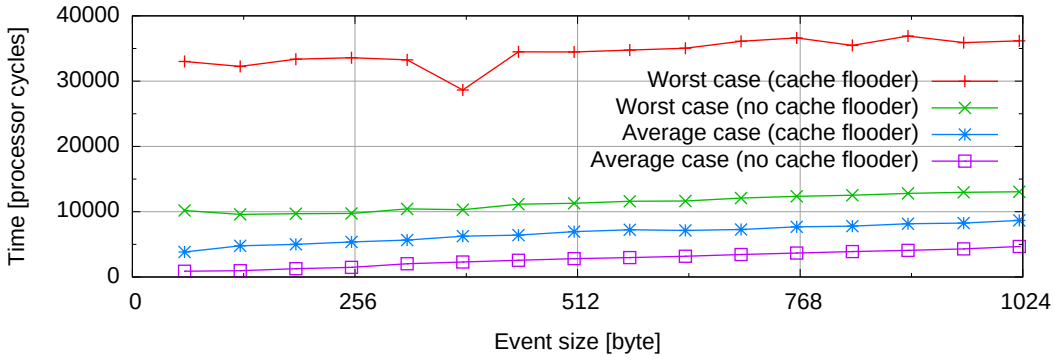


Figure 5.16: Worst-case execution times on Machine B.

and a real-time example in Section 5.3.4.3. Before that, I want to outline methods for quantifying intrusiveness in the next section.

5.3.4.1 Quantifying intrusiveness

I see two ways for quantifying the intrusiveness in the context of monitoring that I term the *objective* and the *subjective approach*. First, for the *objective approach*, the behavior of the instrumented version has to be compared to the behavior of the noninstrumented version using an objective metric. I assume that the behavior of each version is defined in the form of a distribution of execution times. The objective metric basically reduces the two distributions to a single scalar.

Second, for the *subjective approach*, important properties of the target system influence how the two distributions are compared. The subjective approach does not necessarily yield scalar results.

Objective approach I discuss general requirements for an objective metric in the following. Then, after introducing a simple approach, I briefly review metrics used in related work.

For an objective metric to be usable here, it has to fulfill the following four properties:

1. It should be *defined over the whole relevant domain* (in our case all positive real numbers, i. e., time durations, or at least the parts where one of the two distributions to be compared has nonzero values).

2. It should not require any *specific properties* of the two distributions (e. g., that the distributions are nonzero anywhere).

3. It should be *sensitive* to all kinds of differences between the two distributions (e. g., blurring, loss of compactness, or shifting).

4. Its result should be a *scalar* whose value corresponds to the amount of difference between the two distributions.

A very simple way to compare distributions is to compare their mean value and their standard deviation. The rationale here is, that the change in the mean value corresponds to the additional work required for monitoring and that the change in the standard deviation corresponds to additional jitter introduced by monitoring.

Different approaches are used in related work:

- The *Kolmogorov-Smirnov test* [Wei06] allows to compare two distributions. It allows to answer the question of whether two distributions are equal with a given certainty, but not *how* much they diverge. Furthermore, the test is only dominated by the maximum deviation between two distributions. It is therefore not suitable to, for example, capture the amount of shift.

  The *Chi-squared test* [Wei09] is related in that it also targets certainty of equality, not amount of difference.

- In [SG07], Schroeder et al. rely on *manual visual inspection* (and several chi-squared tests) for gauging the similarity of distributions. This approach does not scale and does not deliver objective and scalar similarity results.

  Techniques, such as Q-Q-Plots, represent data in a form such that differences in distributions can be easily spotted visually (cf. to Figure 5.18).

- The *Kullback-Leibler divergence* [KL51] for discrete random variables and everything that builds on it (*cross entropy* and *perplexity*) do not fulfill previously described properties. The Kullback-Leibler divergence is undefined, when one of the two distributions has a zero value. Also, the amount of shifting does not necessarily influence the metric's results.

  The Kullback-Leibler divergence for continuous random variables uses the probability density functions (PDF) and has the same limitations.

- In [KSXW04], Kumar et al. use the *Mean Relative Difference (MRD)* and the *Weighted Mean Relative Difference (WMRD)* to compare estimations of flow distributions with the actual distributions. MRD has the (theoretical) problem of being undefined for an area where *both* distributions are zero. Theoretical, because one could simply exclude these areas.

  Both methods, however, do not fulfill the second property from above (distributions need to be "zipfian") and the third property (the metrics are not sensitive to shifting in all cases).

- The *Earth mover's distance* [LO07] provides a robust metric that captures changes in shift and form of distributions.

All objective metrics have the limitation that they do not consider application-specific requirements. The subjective approach discussed in the following addresses this limitation.

Subjective approach Software-based monitoring techniques will always have influence on the system to be observed one way or the other. The decision whether an influence is negligible is case specific and cannot be rendered globally.

Therefore, for the subjective approach I define intrusiveness towards an observer. If there is no observer in a system, intrusiveness is not perceived and consequently unimportant (however, there are no important systems without observers). The observer may be a human directly using the system or it may be another system part that measures a quantity (e. g., the execution time of a function or request throughput).

Different measures are relevant to different observers. For a server system, for example, *throughput* may be the relevant metric. Hence, intrusiveness is a relative quantity here that describes the fraction of loss in throughput if monitoring is activated.

For a single-user desktop system, *latency* may be the critical measure. Here, everything that, for example, reduces GUI responsiveness is felt to be intrusive. As the observer is human for this scenario, small changes in latency are probably not noticed, whereas larger changes are seen as highly irritating. So, the *felt* intrusiveness does not correspond linearly with, for example, the measured average increase in latency.

For a soft–real-time system, intrusiveness may be defined as the change in the number of *lost deadlines* or the change in position for the highest percentile (99-% quantile).

For hard–real-time systems intrusiveness is boolean — are *all deadlines* still kept?

5.3.4.2 A non–real-time workload

Following the previously described *subjective approach*, I use the Re-AIM benchmark [REA08] in order to measure FERRET's intrusiveness in a typical non–real-time scenario. Re-AIM is a modern successor of the famous AIM Multiuser Benchmark and simulates a multi-user workload. Its results are measured in jobs per minute. I run the benchmark on Machine B (cf. to Section 5.3.1), which is equipped with 512 MiB Ram and a Seagate Barracuda hard disk with 160 GB and 7 200 rpm (ST3160815AS). I freshly installed Ubuntu 8.04.1 (*hardy*) on the disk.

I run Re-AIM with the following settings: `reaim -c reaim.config -f workfile.alltests -s 50 -e 50 -r 200` (all tests, 50 users in parallel, 200 repetitions). I carried out the measurements for both a normal L^{4}Linux kernel and an instrumented version (for each L^{4}Linux system call a FERRET event is posted, which contains the system-call number). Figure 5.17 shows the measurement results.

The *noninstrumented version* has an average throughput of 2 764 Jobs/min (50 Jobs/min standard deviation) and the *instrumented version* performed at 2 779 Jobs/min (44 Jobs/min standard deviation).

The instrumented version is slightly *faster* (15 Jobs/min, 0.55 %) and the distribution is slightly more compact. However, the difference between both versions is well within measurement accuracy for this macrobenchmark. I conclude that FERRET has no significant intrusion effect in this setup.

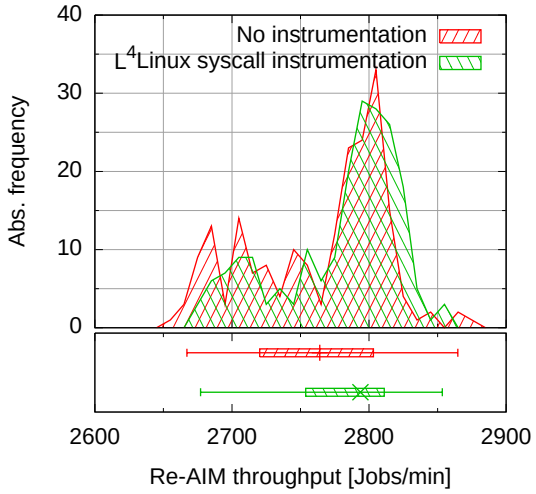


Figure 5.17: Re-AIM 7 throughput (reaim -c reaim.config -f workfile.alltests -s 50 -e 50 -r 200) for both the noninstrumented version and the instrumented version of L4Linux (each Linux system call creates an AList event). The whisker bars show lowest and highest percentiles, lowest and highest quartiles, and the median.

Table 5.3: Key properties of both the noninstrumented and the instrumented run compared (all values are in µs).

Quantile or Position	min.	0.01	0.25	0.5	0.75	0.99	max.
No instrumentation	500	569	717	787	854	1 540	2 727
Event instrumentation	506	570	718	788	855	1 516	2 700

5.3.4.3 A real-time workload

I use the video-player–application Verner [Rie03] as a soft–real-time workload. Again, Machine B is used for measuring here. Verner comprises several components (demultiplexer, video core, audio core, and synchronizer) of which the video core is the most resource-demanding part. It executes a loop that periodically decodes single packets delivered from the demultiplexer (frames). I wrap this loop with a histogram sensor to measure its execution time for each iteration. For the actual Ferret instrumentation, I create events inside the loop's body: in the beginning, before post-processing, and at the end (i. e., three events per loop iteration).

Figure 5.18 shows the results of this benchmark setup for both deactivated and activated Ferret event code inside the loop. I show the Q-Q Plot to demonstrate the similarity between both distributions. Table 5.3 compares key properties of both distributions to allow gauging changes in real-time behavior. The Ferret instrumentation results in an estimated average slowdown of 1 µs. The 0.99 quantile and the observed maximum are slightly better *with* Ferret instrumentation — probably a measurement artifact.

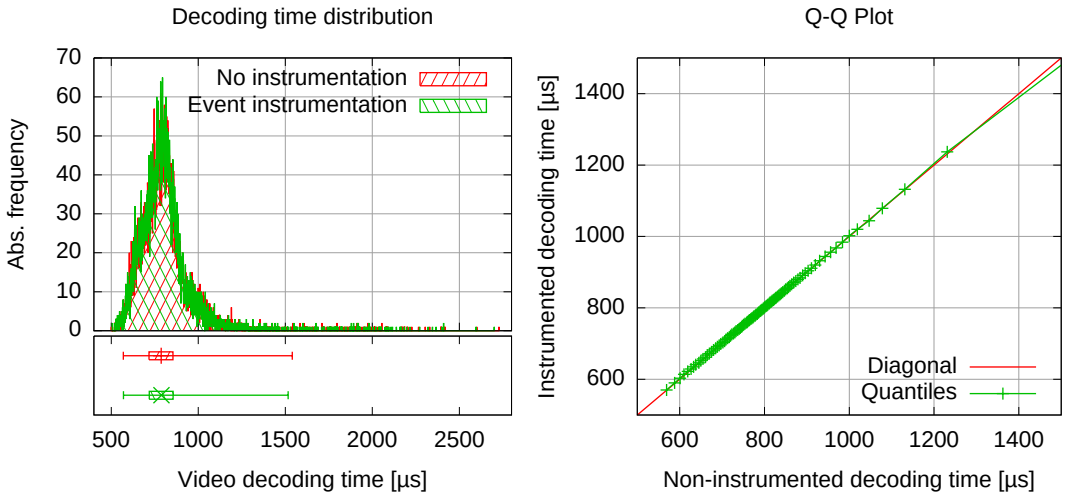


Figure 5.18: Execution-time distribution for both noninstrumented and instrumented video-decoding loop in Verner (the whisker bars show lowest and highest percentiles, lowest and highest quartiles, and the median).

5.4 Summary

I discussed evaluations of three types in this chapter, starting with a discussion of ten different use cases in Section 5.1. These use cases demonstrate Ferret's broad applicability and highlight what kind of scenarios drove Ferret's development.

Following that, in Section 5.2, I presented a qualitative evaluation of properties of different sensors with regard to their suitability for different scenarios. I showed that Ferret's rollforward sensors provide a very good compromise for many scenarios.

The last section of this chapter contained a quantitative evaluation including a discussion of real-time and the System Management Mode, throughput and worst-case microbenchmarks, and results of a real-time and a non–real-time macrobenchmark together with an outline about measuring intrusion. From this qualitative evaluation as well as the whole evaluation chapter combined I conclude that Ferret is suitable for use in my envisioned target systems described in Section 3.1.1.

Chapter 6

Conclusion

I started my work on runtime monitoring for open real-time systems with the single use case of modeling hard disk behavior in the context of real-time environments. I embedded the first measurement infrastructure into the Linux kernel. Low latency and high accuracy for measurements where driving factors for these experiments. Based on the results obtained in [Poh03] we build real-time disk-request scheduling for DROPS [RP03b,RP03a]. Accurate timestamping and resource accounting where added as requirements. I continued with modeling DOpE, DROPS's GUI component. I could verify DOpE's per-pixel model for larger regions, but also found deviations for smaller areas (cf. to Section 5.1.1.1). I adapted my monitoring infrastructure to do online aggregation for this experiment, as tracking each single request would have resulted in huge overheads. I then targeted high-level components in DROPS, for example a video player, and interaction of real-time and non–real-time components [APRZ03,HZP+07b,GPPZ04a,APPZ04,GPA+04,PAH04]. Video playback shows dynamic runtime behavior, for example, scene-specific or quality-profile specific processor-time demand, for which I adapted my monitoring infrastructure. Also the communication interaction between real-time and non–real-time parts where monitored. Based in these results, Riegel generalized a specific class of event sensors using lock-free techniques in his Master's thesis [Rie05].

I used the previously described experience for building event tracing for Microsoft's research operating system Singularity [HLA+05] and adapted the Magpie toolchain [BDIM04,BIMN03] for evaluation of measurement results. I there first used external schemata for event trace post-processing.

With the creation of FERRET, I tried to synthesize all of the previously discussed experiences into one tool. I targeted FERRET at DROPS (and in parts Linux) and adapted the Magpie toolchain for evaluating FERRET traces.

FERRET's own sensors are based on atomic sections for user-space code which allows for excellent throughput and very good worst-case behavior. I designed FERRET to be portable and to use only a very limited API of host systems in order to allow using it in very different situations and system layers. This allowed me to conduct several more experiments that I described in Section 5.1, for example, cross-component resource accounting (Section 5.1.1.2) and adaptation of FERRET for dynamic function-call tracing (Section 5.1.3.1).

Over the years, I evolved FERRET and its predecessors to match newly identified requirements, starting from an empty sheet several times. In the course of this process I identified several lessons that are now integral parts of FERRET:

- A single monitoring approach is desirable to use in all system layers and different components, in both real-time and non–real-time parts of a system. Otherwise, gaining a whole-system view is cumbersome. To this end, the monitoring infrastructure should be portable in that it uses little API functions of specific systems. I designed Ferret to use a limited set of functions in the setup phase and work solely on shared memory afterwards. Consequently, porting to a new environment mostly means to adapt the small setup parts.

  The architecture is designed to require minimal support in the kernel, basically only the mechanism for the atomic sections. This conforms to the microkernel philosophy of only having the striclty necessary support in the kernel — usually the mechanisms, not the policy — and leads to easier adaptability to new requirements as mostly user code will have to be touched.

- Due to its potential wide deployment in systems, the monitoring infrastructure becomes a critical part of them. Therefore, additional crosstalk through the monitoring infrastructure must be prevented.

  In more detail, stability, intrusiveness, and security concerns play a role here. For *stability*, the monitoring infrastructure must be stable itself and must not allow bugs in monitored parts to spread through itself in the system. For limiting *intrusiveness*, the system should be influenced as little as possible, especially, the communication and scheduling behavior. For *security*, security domains must be separable and several independent evaluations must be runnable in a system at the same time.

  To address these concerns, I designed Ferret such that no synchronous notification happens from monitored system parts to monitors, which prevents directly caused reactions in the monitoring system. No communication is possible from monitors back to monitored parts through Ferret by design. Independent system parts can use completely independent sets of sensors, which allows to separate them.

- The application of the monitoring infrastructure in potentially low system layers requires a low-overhead solution in terms of processor cycles and memory requirements as many invocations may be seen. In Ferret I address the memory-cost problem by using a very compact binary event representation, by providing sensors with statically configurable event sizes, and by providing sensors with variable event sizes. Also filtering in monitors and online evaluation and aggregation is possible with Ferret. The atomic sections for user-space code, used by Ferret have a very low average time demand, which allows to use them in throughput-oriented non–real-time components.

- For monitoring real-time components, good worst-case behavior is important. The rollforward variant of the atomic sections for user-space code show good worst-case behavior, which makes them applicable for deadline-driven real-time components.

- Monitoring in higher system levels typically involves fewer invocations but much richer data has to be stored per event. FERRET supports this scenario by providing sensors with variable event sizes.
- I developed wrapper functions for using the atomic sections for user code in limited environments where arguments on the stack or the heap cannot be pinned and on which access cannot be protected against page faults. I aggregate them on the call stack and prefault the active stack page atomically with entering the atomic section in FERRET using a side effect of the function call.
- These special wrapper functions or calling conventions also allow reusing compiler-generated code that accesses the local stack frame for local variables and as a register-spilling area. Without this mechanism, FERRET sensors were limited to relatively simple, hand-written assembler sensor access functions. With this mechanism, sensor access code can be conceived and be maintained in a higher-level language as C, which greatly simplifies these tasks.
- The system-wide deployment of a monitoring infrastructure exposes it to different assumptions about threading models and reentrancy. FERRET's use of atomic sections for user-space makes it compatible with these assumptions without unduly serializing or deferring monitored code. FERRET's access to Fiasco's in-kernel trace buffer adds in-kernel events to FERRET's view.
- Obtaining trace data alone has no value. I adapted the *Magpie* toolchain to evaluate and post-process FERRET's event traces and used an adapted and extended version of *Magpyvis* extensively for interactive investigation of traces.

To conclude, the developed architecture and the prototypical implementation fulfills the requirements for runtime monitoring in modern open desktop systems that support classic time-sharing applications as well as real-time tasks.

6.1 Suggestions for future work

I see several directions in which research and development could be carried on.

FERRET could be ported to other systems to further allow comparative studies. While finishing the work on this thesis, work was started in our research group to adapt FERRET to the current development version of our research operating system.

FERRET can always be used in more experiments and studies and grow with them. Porting FERRET to a 64-bit architecture may simplify sensor access code, as larger memory regions can be manipulated atomically (e. g., using `CMPXCHG16B` on the x86-64 architecture).

I proposed to use independent sensor buffers for each processor on multiprocessor systems. Implementing this feature is future work.

I assume that my adaptation of *atomic sections for user-space code* may be applicable to other problem domains, for example, a library for low-level data-structure manipulation may prove useful.

Bibliography

[AAD+02] Jonathan Appavoo, Marc Auslander, Dilma DaSilva, David Edelsohn, Orran Krieger, Michal Ostrowski, Bryan Rosenburg, Robert W. Wisniewski, and Jimi Xenidis. K42's Performance Monitoring and Tracing Infrastructure, August 2002.

[AAP06] Fernando H. Ataide, Alan C. Assis, and Carlos E. Pereira. Automotive X-by-Wire System Based on Linux — An Open Source Project. In *Proceedings of the Eigth Real-Time Linux Workshop*, Lanzhou, China, 2006.

[ABD+97] 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? *ACM Trans. Comput. Syst.*, 15(4):357–390, 1997.

[Adv05] Advanced Micro Devices Inc. Software Optimization Guide for AMD64 Processors, September 2005. Revision 3.06.

[Adv06] Advanced Micro Devices Inc. BIOS and Kernel Developer's Guide for AMD Athlon™ 64 and AMD Opteron™ Processors, February 2006. Revision 3.30.

[Adv07] Advanced Micro Devices Inc. Software Optimization Guide for AMD Family 10h Processors, December 2007. Revision 3.05.

[Adv09] Advanced Micro Devices Inc. Advanced Synchronization Facility — Proposed Architectural Specification, March 2009. Revision 2.1.

[Aig07] Ronald Aigner. DICE *Version 3.1.1 User's Manual*, 2007. Available at `http://os.inf.tu-dresden.de/dice/manual.pdf`.

[ALR01] A. Avizienis, J. Laprie, and B. Randell. Fundamental Concepts of Dependability, 2001.

[AMW+03] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance Debugging for Distributed Systems of Black Boxes. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 74–89, New York, NY, USA, 2003. ACM.

[APPZ04] Ronald Aigner, Christoph Pohl, Martin Pohlack, and Steffen Zschaler. Tailor-Made Containers: Modeling Non-functional Middleware Service. In *Workshop on Models for Non-Functional Aspects of Component-Based Software (NFC'04) colocated with UML 2004*, Lissabon, Portugal, October 2004.

[APRZ03] Ronald Aigner, Martin Pohlack, Simone Röttger, and Steffen Zschaler. Towards Pervasive Treatment of Non-Functional Properties at Design and Run-Time. In *Proceedings of 16th International Conference "Software & Systems Engineering and their Applications (ICSSEA 2003)"*, Paris, France, December 2003.

[ARJ97] James H. Anderson, Srikanth Ramamurthy, and Kevin Jeffay. Real-Time Computing with Lock-Free Shared Objects. *ACM Trans. Comput. Syst.*, 15(2):134–165, 1997.

[ARM07] ARM Limited. RealView Emulation Baseboard HBI-0140 Rev C — User Guide, 2007.

[BA07] Björn B. Brandenburg and James H. Anderson. Feather-Trace: A Light-Weight Event Tracing Toolkit. In *Third International Workshop on Operating Systems Platforms for Embedded Real-Time applications*, 2007.

[BDIM04] Paul T. Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using Magpie for Request Extraction and Workload Modelling. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 259–272, Berkeley, CA, USA, 2004. USENIX Association.

[BDS07a] Sumit Basu, John Dunagan, and Greg Smith. Why Did My PC Suddenly Slow Down? In *SYSML'07: Proceedings of the 2nd USENIX workshop on Tackling computer systems problems with machine learning techniques*, pages 1–6, Berkeley, CA, USA, 2007. USENIX Association.

[BDS07b] Martin Bligh, Matthieu Desnoyers, and Rebecca Schultz. Linux Kernel Debugging on Google-sized clusters. In *Proceedings of the Linux Symposium*, Ottawa, Ontario, Canada, June 2007.

[Bel05] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *ATEC'05: Proceedings of the USENIX Annual Technical Conference 2005*, page 41, Berkeley, CA, USA, 2005. USENIX Association.

[BH00] Bryan Buck and Jeffrey K. Hollingsworth. An API for Runtime Code Patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, Winter 2000.

[BIMH06] Paul Barham, Rebecca Isaacs, Richard Mortier, and Tim Harris. Learning communication patterns in Singularity. In *First Workshop on Tackling*

Computer Systems Problems with Machine Learning Techniques (SysML), March 2006.

[BIMN03] Paul T. Barham, Rebecca Isaacs, Richard Mortier, and Dushyanth Narayanan. Magpie: online modelling and performance-aware systems. In Michael B. Jones, editor, *HotOS*, pages 85–90. USENIX, 2003.

[Bir08] Tim Bird, et al. TracingCollaborationProject. `http://tree.celinuxforum.org/CelfPubWiki/TracingCollaborationProject`, 2008.

[Bru05] Rich Brunner. TSC and Power Management Events on AMD Processors. Usenet post, Nov 2005. comp.unix.solaris.

[Cag06] Andrew Cagney. The Frysk Execution Analysis Architecture. In *Proceedings of the Linux Symposium*, Ottawa, Canada, July 2006.

[Cag07] Andrew Cagney. Frysk 1, Kernel 0? In *Proceedings of the Linux Symposium*, Ottawa, Ontario, Canada, June 2007.

[CAK+04] Mike Y. Chen, Anthony Accardi, Emre Kiciman, Jim Lloyd, Dave Patterson, Armando Fox, and Eric Brewer. Path-Based Failure and Evolution Management. In *NSDI'04: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation*, Berkeley, CA, USA, 2004. USENIX Association.

[cCH02] Cristian Ţăpuş, I-Hsin Chung, and Jeffrey K. Hollingsworth. Active Harmony: Towards Automated Performance Tuning. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–11, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[CCZ07] Anupam Chanda, Alan L. Cox, and Willy Zwaenepoel. Whodunit: Transactional Profiling for Multi-Tier Applications. *SIGOPS Oper. Syst. Rev.*, 41(3):17–30, 2007.

[CO94] R. Carrasco and J. Oncina. Learning stochastic regular grammars by means of a state merging method. In *Proc. 2nd International Colloquium on Grammatical Inference - ICGI '94*, volume 862, pages 139–150. Springer-Verlag, 1994.

[CRSBPC06] D. W. Carr, R. Ruelas, H. Salcedo-Becerra, and G. A. Ponce-Castaneda. A Linux Based System to Monitor Train Speed and Doors for the Light Rail System in Guadalajara, Mexico. In *Proceedings of the Eigth Real-Time Linux Workshop*, Lanzhou, China, 2006.

[CSL04] Bryan Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic Instrumentation of Production Systems. In *USENIX Annual Technical Conference, General Track*, 2004.

[DD06] Mathieu Desnoyers and Michel R. Dagenais. The LTTng tracer: A low impact performance and behavior monitor for GNU/Linux. In *Proceedings of the Linux Symposium*, Ottawa, Canada, July 2006.

[DD08] Mathieu Desnoyers and Michel R. Dagenais. LTTng: Tracing across execution layers, from the Hypervisor to user-space. In *Proceedings of the Linux Symposium*, Ottawa, Canada, July 2008.

[Des08] Mathieu Desnoyers. Using the Linux Kernel Markers, 2008. `linux-2.6.25.5/Documentation/markers.txt`.

[Dil] Matthew Dillon. Page Coloring. Design elements of the FreeBSD VM system. `http://www.freebsd.org/doc/en_US.ISO8859-1/articles/vm-design/page-coloring-optimizations.html`. Retrieved on 2009-01-06.

[Döb06] Björn Döbel. Request tracking in DROPS, 2006. Diplomarbeit (Master's thesis).

[DS] MAXIM Dallas Semiconductor. Datasheet: Real-Time Clock DS12885, DS12887, and DS12c887. `http://www.maxim-ic.com/getds.cfm?qv_pk=2680`. Rev 3; 2/07, Retrieved on 2009-05-27.

[DTI08] Linux Driver Tracing Interface. `http://sourceforge.net/projects/dti`, 2008.

[Döb08] Björn Döbel. ORe - a software network switch for L4. `http://os.inf.tu-dresden.de/l4env/doc/html/ore/index.html`, Aug 2008.

[EPC+05] F. Ch. Eigler, Vara Prasad, William Cohen, Hien Nguyen, Martin Hunt, Jim Keniston, and Brad Chen. Architecture of systemtap: a Linux trace/probe tool. `http://sourceware.org/systemtap/archpaper.pdf`, 2005.

[FH03] Norman Feske and Hermann Härtig. Demonstration of DOpE — a Window Server for Real-Time and Embedded Systems. In *24th IEEE Real-Time Systems Symposium (RTSS)*, pages 74–77, Cancun, Mexico, December 2003.

[Fid88] C. J. Fidge. Partial orders for parallel debugging. In *PADD '88: Proceedings of the 1988 ACM SIGPLAN and SIGOPS workshop on Parallel and distributed debugging*, pages 183–194, New York, NY, USA, 1988. ACM Press.

[Fid91] Colin Fidge. Logical Time in Distributed Computing Systems. *Computer*, 24(8):28–33, 1991.

[FPK+07] Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. X-Trace: A Pervasive Network Tracing Framework. In *4th Symposium on Networked Systems Design and Implementation (NSDI)*, Cambridge, Massachusetts, USA, April 2007.

[Fri06] Thomas Friebel. Übertragung des Device-Driver-Environment-Ansatzes auf Subsysteme des BSD-Betriebssystemkerns, March 2006. Diplomarbeit (Master's thesis), Technische Universität Dresden.

[Gai85] Jason Gait. A Debugger for Concurrent Programs. *Softw. Pract. Exper.*, 15(6):539–554, 1985.

[GC304] GCC team. *GCC 3.4.6 Manual*, 2004. `http://gcc.gnu.org/onlinedocs/gcc-3.4.6/gcc/`.

[GC405] GCC team. *GCC 4.2.2 Manual*, 2005. `http://gcc.gnu.org/onlinedocs/gcc-4.2.2/gcc/`.

[GEGW02] Thomas Gschwind, Kave Eshghi, Pankaj K. Garg, and Klaus Wurster. WebMon: A Performance Profiler for Web Transactions. In *WECWIS '02: Proceedings of the Fourth IEEE International Workshop on Advanced Issues of E-Commerce and Web-Based Information Systems (WECWIS'02)*, page 171, Washington, DC, USA, 2002. IEEE Computer Society.

[GKM82] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. Gprof: A call graph execution profiler. *SIGPLAN Not.*, 17(6):120–126, 1982.

[GML06] Jan Glauber, Frank Mehnert, and Jochen Liedtke. Fiasco Kernel Debugger Manual, November 2006. Version 1.1.6.

[GPA+04] Steffen Göbel, Christoph Pohl, Ronald Aigner, Martin Pohlack, Simone Röttger, and Steffen Zschaler. The COMQUAD Component Container Architecture. In *WICSA '04: Proceedings of the Fourth Working IEEE/IFIP Conference on Software Architecture*, page 315, Washington, DC, USA, 2004. IEEE Computer Society.

[GPPZ04a] Steffen Göbel, Christoph Pohl, Martin Pohlack, and Steffen Zschaler. The COMQUAD Component Container Architecture and Contract Negotiation. Technical Report TUD-FI04-04-April-2004, TU Dresden, 2004. Available from URL: `http://os.inf.tu-dresden.de/papers_ps/tr-comqarch.pdf`.

[GPPZ04b] Steffen Göbel, Christoph Pohl, Martin Pohlack, and Steffen Zschaler. The COMQUAD Component Container Architecture and Contract Negotiation. Technical Report TUD-FI04-04-April-2004, TU Dresden, 2004. Available from URL: `http://os.inf.tu-dresden.de/drops/-doc.html#tr-comqarch`.

[GS95] Rajiv Gupta and Madalene Spezialetti. Dynamic Techniques for Minimizing the Intrusive Effect of Monitoring Actions. In *ICDCS '95: Proceedings of the 15th International Conference on Distributed Computing Systems*, pages 368–376, Washington, DC, USA, 1995. IEEE Computer Society.

[GS06] Rakesh Kumar Gupta and Jitendra Kumar Sasmal. Linux in Polyester Automation — A Case Study of Reliance Industries Ltd, Polyester Manufacturing. In *Proceedings of the Eigth Real-Time Linux Workshop*, Lanzhou, China, 2006.

[Ham97] Cl.-J. Hamann. On the quantitative specification of jitter constrained periodic streams. In *MASCOTS*, Haifa, Israel, January 1997.

[HBB+98] H. Härtig, R. Baumgartl, M. Borriss, Cl.-J. Hamann, M. Hohmuth, F. Mehnert, L. Reuther, S. Schönberg, and J. Wolter. DROPS: OS support for distributed multimedia applications. In *Proceedings of the Eighth ACM SIGOPS European Workshop*, Sintra, Portugal, September 1998.

[Hel08] Christian Helmuth. Linux Device Driver Environment Manual. `http://os.inf.tu-dresden.de/l4env/doc/html/dde_linux/index.html`, Aug 2008.

[HK99] Jeffrey K. Hollingsworth and Peter J. Keleher. Prediction and Adaptation in Active Harmony. *Cluster Computing*, 2(3):195–205, 1999.

[HL07] Galen C. Hunt and James R. Larus. Singularity: Rethinking the Software Stack. *SIGOPS Oper. Syst. Rev.*, 41(2):37–49, 2007.

[HLA+05] G. Hunt, J. Larus, M. Abadi, M. Aiken, P. Barham, M. Fahndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, B. Steensgaard, D. Tarditi, T. Wobber, and B. Zill. An Overview of the Singularity Project. Technical report, Microsoft Research, 2005.

[HLR+01] C.-J. Hamann, J. Löser, L. Reuther, S. Schönberg, J. Wolter, and H. Härtig. Quality Assuring Scheduling - Deploying Stochastic Behavior to Improve Resource Utilization. In *22nd IEEE Real-Time Systems Symposium (RTSS)*, London, UK, December 2001.

[HO07] Masami Hiramatsu and Satoshi Oshima. Djprobe—Kernel probing with the smallest overhead. In *Proceedings of the Linux Symposium*, Ottawa, Ontario, Canada, June 2007.

[Hof02] Sarah Hoffmann. Kleine Adressräume für Fiasco, 2002. Großer Beleg (Undergraduate thesis), Technische Universität Dresden.

[Hof08] Jörn Hoffmann. Intrusion Detection and Response for L^{4}Linux, 2008. Diplomarbeit (Master's thesis), Universität Leipzig.

[Hol06] Gerard J. Holzmann. *The SPIN MODEL CHECKER — Primer and Reference Manual*. Addison-Wesley, December 2006. 3rd Printing.

[HZP+07a] Hermann Härtig, Steffen Zschaler, Martin Pohlack, Ronald Aigner, Steffen Göbel, Christoph Pohl, and Simone Röttger. Enforceable Component-Based Realtime Contracts — Supporting Realtime Properties from Software Development to Execution. *Springer Real-Time Systems Journal*, 35(1), January 2007.

[HZP+07b] Hermann Härtig, Steffen Zschaler, Martin Pohlack, Ronald Aigner, Steffen Göbel, Christoph Pohl, and Simone Röttger. Enforceable component-based realtime contracts: Supporting realtime properties from software development to execution. *Real-Time Systems*, 35(1):1–31, January 2007.

[Int04a] Intel Corporation. IA-32 Intel Architecture Optimization — Reference Manual, 2004.

[Int04b] Intel Corporation. IA-32 Intel Architecture Software Developer's Manual — Volume 1: Basic architecture, 2004.

[Int04c] Intel Corporation. IA-32 Intel Architecture Software Developer's Manual — Volume 2A: Instruction Set Reference, A-M,, 2004.

[Int04d] Intel Corporation. IA-PC HPET (High Precision Event Timers) Specification, October 2004. Revision 1.0a.

[Int06] Intel Corporation. IA-32 Intel Architecture Software Developer's Manual — Volume 3B: System Programming Guide, Part 2, January 2006.

[Int07] Intel Corporation. Intel 64 and IA-32 Architectures Optimization Reference Manual, 2007.

[Int08] Open Source Technology Center Intel Corporation. LessWatts.org - Saving Power on Intel systems with Linux. `http://www.lesswatts.org/projects/powertop/powertop.php`, 2008.

[J. 00] J. Consortium. Real-Time Core Extensions for the Java platform, September 2000. International J Consortium Specification, Revision 1.0.14.

[Jah95] Farnam Jahanian. Run-Time Monitoring of Real-Time Systems. pages 435–460, 1995.

[Jon05] M. Tim Jones. Visualize function calls with Graphviz. `http://www.ibm.com/developerworks/library/l-graphvis/`, June 2005.

[JR81] S. C. Johnson and D. M. Ritchie. Computing Science Technical Report No. 102: The C Language Calling Sequence. Technical report, Bell Laboratories, September 1981.

[Kan07] Antti Kantee. puffs - Pass-to-Userspace Framework File System. In *Proceedings of the Asia BSD Conference (AsiaBSDCon)*, March 2007.

[KBD+08] Andreas Knüpfer, Holger Brunst, Jens Doleschal, Matthias Jurenz, Matthias Lieber, Holger Mickler, Matthias S. Müller, and Wolfgang E. Nagel. *Tools for High Performance Computing: Proceedings of the 2nd International Workshop on Parallel Tools for High Performance Computing, July 2008, HLRS, Stuttgart*, chapter The Vampir Performance Analysis Tool-Set, pages 139–155. Springer Berlin Heidelberg, 2008.

[KFT08] Kernel Function Trace. `http://elinux.org/Kernel_Function_Trace`, 2008.

[KL51] S. Kullback and R. A. Leibler. On information and sufficiency. In *Annals of Mathematical Statistics 22*, pages 79–86. 1951.

[Kri05] R. Krishnakumar. Kernel Korner: Kprobes—a Kernel Debugger. *Linux J.*, 2005.

[KSXW04] Abhishek Kumar, Minho Sung, Jun (Jim) Xu, and Jia Wang. Data streaming algorithms for efficient and accurate estimation of flow size distribution. In *SIGMETRICS '04/Performance '04: Proceedings of the joint international conference on Measurement and modeling of computer systems*, pages 177–188, New York, NY, USA, 2004. ACM.

[L4K06] L4Ka Team. L4 eXperimental Kernel Reference Manual, Version x.2. Technical report, 2006. Revision 6.

[Lam78] Leslie Lamport. Time, Clocks, and the Ordering of Vvents in a Distributed System. *Commun. ACM*, 21(7):558–565, 1978.

[LCM+05] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, New York, NY, USA, 2005. ACM.

[LDSP85] Carol H. LeDoux and Jr. D. Stott Parker. Saving traces for Ada debugging. In *SIGAda '85: Proceedings of the 1985 annual ACM SIGAda international conference on Ada*, pages 97–108, New York, NY, USA, 1985. Cambridge University Press.

[Lie95] J. Liedtke. On micro-kernel construction. *SIGOPS Oper. Syst. Rev.*, 29(5):237–250, 1995.

[Lie96a] Jochen Liedtke. Colorable Memory. November 1996.

[Lie96b] Jochen Liedtke. Toward Real Microkernels. *Communications of the ACM*, 39(9):70–77, 1996.

[LO07] Haibin Ling and Kazunori Okada. An Efficient Earth Mover's Distance Algorithm for Robust Histogram Comparison. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29(5):840–853, 2007.

[Mar06] Larry Mart. disabling SMI, 2006. Usenet post on comp.arch.embedded, comp.realtime, comp.sys.intel, alt.comp.periphs.mainboard.aopen, Message-ID: <1149202266.495177.211530@i40g2000cwc.googlegroups.-com>.

[Mat89] Friedemann Mattern. Virtual Time and Global States of Distributed Systems. In Cosnard M. et al., editor, *Proc. Workshop on Parallel and Distributed Algorithms*, pages 215–226, North-Holland / Elsevier, 1989. (Reprinted in: Z. Yang, T.A. Marsland (Eds.), "Global States and Time in Distributed Systems", IEEE, 1994, pp. 123-133.).

[Mcd77] Gene Mcdaniel. METRIC: a kernel instrumentation system for distributed environments. In *SOSP '77: Proceedings of the sixth ACM symposium on Operating systems principles*, pages 93–99, New York, NY, USA, 1977. ACM.

[McK04] Chris McKillop. User-Space Debugging Simplifies Driver Development. In *Proceedings of the embedded world 2004 Conference*. Caspar Grote and Renate Ester, February 2004.

[MDP96] David Mosberger, Peter Druschel, and Larry L. Peterson. Implementing Atomic Sequences on Uniprocessors Using Rollforward. *Software—Practice and Experience*, 26(1):1–23, 1996.

[MDP00] P. Mantegazza, E. L. Dozio, and S. Papacharalambous. RTAI: Real Time Application Interface. *Linux J.*, page 10, 2000.

[MH89] Charles E. McDowell and David P. Helmbold. Debugging Concurrent Programs. *ACM Comput. Surv.*, 21(4):593–622, 1989.

[MHH02] F. Mehnert, M. Hohmuth, and H. Härtig. Cost and benefit of separate address spaces in real-time operating systems. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS)*, pages 124–133, Austin, Texas, USA, December 2002.

[MHJM09] Michael Matz, Jan Hubička, Andreas Jaeger, and Mark Mitchell. *SYSTEM V APPLICATION BINARY INTERFACE, AMD64 Architecture Processor Supplement, Draft Version 0.99*, May 2009.

[MHSH01] Frank Mehnert, Michael Hohmuth, Sebastian Schönberg, and Hermann Härtig. RTLinux with address spaces. In *Proceedings of the Third Real-Time Linux Workshop*, Milano, Italy, November 2001.

[Mica] Microsoft. Event Tracing. `http://msdn2.microsoft.com/en-US/library/aa363787.aspx`.

[Micb] Microsoft. How To Use Event Tracing For Windows For Performance Analysis. `http://download.microsoft.com/download/f/0/5/f05a42ce-575b-4c60-82d6-208d3754b2d6/EventTrace_PerfAnalysis.ppt`.

[Mic02] Microsoft. Guidelines For Providing Multimedia Timer Support. `http://www.microsoft.com/whdc/system/sysinternals/mm-timer.mspx`, September 2002.

[MLCS89] D. C. Marinescu, J. E. Lumpp, Jr., T. L. Casavant, and H. J. Siegel. A Model for Monitoring and Debugging Parallel and Distributed Software. In *Proc. 13th Annual International Computer Software and Application Conference (COMPSAC '89)*, pages 81–88, October 1989.

[MM03] Alexander V. Mirgorodskiy and Barton P. Miller. CrossWalk: A Tool for Performance Profiling Across the User-Kernel Boundary. In *Proceedings of the international Conference on Parallel Computing (ParCo)*, Dresden, Germany, 2003.

[Moo01] Richard J. Moore. A Universal Dynamic Trace for Linux and Other Operating Systems. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, Berkeley, CA, USA, 2001.

[MRCR04] M. Masmano, I. Ripoll, A. Crespo, and J. Real. TLSF: A New Dynamic Memory Allocator for Real-Time Systems. In *ECRTS '04: Proceedings of the 16th Euromicro Conference on Real-Time Systems*, pages 79–86, Washington, DC, USA, 2004. IEEE Computer Society.

[MSSP02] Daniel Mahrenholz, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. Program Instrumentation for Debugging and Monitoring with AspectC++. In *ISORC '02: Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, page 249, Washington, DC, USA, 2002. IEEE Computer Society.

[Mye05] Ryan Myers. Funny, It Worked Last Time: Event Tracing for Windows (ETW). `http://blogs.msdn.com/ryanmy/archive/2005/05/27/422772.aspx`, May 2005.

[Nar05] Dushyanth Narayanan. End-to-end tracing considered essential. In *Proceedings of High Performance Transaction Systems — Eleventh Biennial Workshop (HPTS '05)*, Asilomar Conference Center, Pacific Grove, CA, September 2005.

[NS07] Nicholas Nethercote and Julian Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. *SIGPLAN Not.*, 42(6):89–100, 2007.

[OMCB07] Marek Olszewski, Keir Mierle, Adam Czajkowski, and Angela Demke Brown. JIT Instrumentation: A Novel Approach to Dynamically Instrument Operating Systems. *SIGOPS Oper. Syst. Rev.*, 41(3):3–16, 2007.

[OPr08] OProfile. `http://oprofile.sourceforge.net/`, 2008.

[OS] OS Group, TU Dresden. OS-Group–internal bugzilla database: Bug 95 — Delayed Preemption (kleine kritische Abschnitte). State: 2007-07-02.

[PAH04] Martin Pohlack, Ronald Aigner, and Hermann Härtig. Connecting Real-Time and Non-Real-Time Components. Technical Report TUD-FI04-01-Februar-2004, TU Dresden, 2004. Available from URL: `http://os.inf.tu-dresden.de/papers_ps/tr-rtnonrtcomp.pdf`.

[PB07] Insung Park and Ricky Buch. Event Tracing: Improve Debugging And Performance Tuning With ETW. In *MSDN Magazine*, April 2007.

[PDGQ05] Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. Interpreting the data: Parallel analysis with Sawzall. *Sci. Program.*, 13(4):277–298, 2005.

[PDL06] Martin Pohlack, Björn Döbel, and Adam Lackorzynski. Towards Runtime Monitoring in Real-Time Systems. In *Proceedings of the Eigth Real-Time Linux Workshop*, Lanzhou, China, 2006.

[PKFH02] David J. Pearce, Paul H. J. Kelly, Tony Field, and Uli Harder. GILK: A Dynamic Instrumentation Tool for the Linux Kernel. In *TOOLS '02: Proceedings of the 12th International Conference on Computer Performance Evaluation, Modelling Techniques and Tools*, pages 220–226, London, UK, 2002. Springer-Verlag.

[Poh02] Martin Pohlack. Ermittlung von Festplatten-Echtzeiteigenschaften, 2002. Großer Beleg (Undergraduate thesis), Technische Universität Dresden.

[Poh03] Martin Pohlack. Plattenscheduling für Quality-Assuring-Scheduling, March 2003. Diplomarbeit (Master's thesis), Technische Universität Dresden.

[Poh06] Martin Pohlack. Ein praktischer Erfahrungsbericht über Model Checking in L4Linux (A real-world experience talk about model checking L4Linux's internals). `http://os.inf.tu-dresden.de/~mp26/download/tamer.pdf`, 2006.

[Poh08] Martin Pohlack. RT_Mon — runtime monitoring documentation. `http://os.inf.tu-dresden.de/l4env/doc/html/rt_mon/index.html`, Mar 2008.

[Pro] LTTng Project. Notes on Asynchronous TSC Architectures (and workarounds). `http://ltt.polymtl.ca/svn/trunk/lttv/doc/developer/tsc.txt`, Retrieved on 2009-05-21.

[PT04] Daniel Price and Andrew Tucker. Solaris Zones: Operating System Support for Consolidating Commercial Workloads. In *LISA '04: Proceedings of the 18th USENIX conference on System administration*, pages 241–254, Berkeley, CA, USA, 2004. USENIX Association.

[PW93] Sharon E. Perl and William E. Weihl. Performance assertion checking. *SIGOPS Oper. Syst. Rev.*, 27(5):134–145, 1993.

[REA08] Re-AIM_7. `http://sourceforge.net/projects/re-aim-7`, 2008.

[Rie03] Carsten Rietzschel. VERNER – ein Video EnkodeR uNd playER für DROPS, 2003. Master's thesis.

[Rie05] Torvald Riegel. A generalized approach to runtime monitoring for real-time systems, 2005. Diplomarbeit (Master's thesis), Technische Universität Dresden.

[RKW+06] Patrick Reynolds, Charles Killian, Janet L. Wiener, Jeffrey C. Mogul, Mehul A. Shah, and Amin Vahdat. Pip: Detecting the Unexpected in Distributed Systems. In *NSDI'06: Proceedings of the 3rd conference on 3rd Symposium on Networked Systems Design & Implementation*, pages 9–9, Berkeley, CA, USA, 2006. USENIX Association.

[RP03a] Lars Reuther and Martin Pohlack. Rotational-Position-Aware Real-Time Disk Scheduling Using a Dynamic Active Subset (DAS). In *24th IEEE Real-Time Systems Symposium (RTSS)*, pages 374–385, Cancun, Mexico, December 2003.

[RP03b] Lars Reuther and Martin Pohlack. Work-in-Progress Report: Using SATF Scheduling in Real-Time Systems. *Work-in-Progress Report: 2nd USENIX Conference on File and Storage Technologies (FAST-2003), San Francisco, CA, USA*, March 2003.

[San97] The Santa Cruz Operation, Inc. *SYSTEM V APPLICATION BINARY INTERFACE, Intel386™Architecture Processor, Supplement (Fourth Edition, Draft Copy)*, March 1997.

[Sch91] Werner Schütz. On the Testability of Distributed Real-Time Systems. In *SRDS*, pages 52–61, 1991.

[Sch93] Werner Schütz. *The Testability of Distributed Real-Time Systems*. Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[SE94] Amitabh Srivastava and Alan Eustace. ATOM: A System for Building Customized Program Analysis Tools. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 196–205, New York, NY, USA, 1994. ACM.

[SEV01] Amitabh Srivastava, Andrew Edwards, and Hoi Vo. Vulcan: Binary Transformation in a Distributed Environment. Technical Report MSR-TR-2001-50, 2001.

[SG07] Bianca Schroeder and Garth A. Gibson. Understanding disk failure rates: What does an MTTF of 1,000,000 hours mean to you? *Trans. Storage*, 3(3):8, 2007.

[SGG+99] David C. Steere, Ashvin Goel, Joshua Gruenberg, Dylan McNamee, Calton Pu, and Jonathan Walpole. A Feedback-driven Proportion Allocator for Real-Rate Scheduling. In *OSDI '99: Proceedings of the third symposium on Operating systems design and implementation*, pages 145–158, Berkeley, CA, USA, 1999. USENIX Association.

[Sha99] Murray Shanahan. The Event Calculus Explained. *Lecture Notes in Computer Science*, 1600, 1999.

[SKA07] George Spanoudakis, Christos Kloukinas, and Kelly Androutsopoulos. Towards Security Monitoring Patterns. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*, pages 1518–1525, New York, NY, USA, 2007. ACM.

[Ste04] Udo Steinberg. Quality-Assuring Scheduling in the Fiasco Microkernel. Master's thesis, TU Dresden, March 2004.

[Stö05] Jan Stöß. Using Operating System Instrumentation and Event Logging to Support User-level Multiprocessor Schedulers, March 24 2005.

[SW92] Amitabh Srivastava and David W. Wall. A Practical System for Intermodule Code Optimization at Link-Time. *Journal of Programming Languages*, 1(1):1–18, December 1992.

[SW94] Amitabh Srivastava and David W. Wall. Link-Time Optimization of Address Calculation on a 64-bit Architecture. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 49–60, New York, NY, USA, 1994. ACM.

[TFCB90] J. J. P. Tsai, K.-Y. Fang, H.-Y. Chen, and Y.-D. Bi. A Noninterference Monitoring and Replay Mechanism for Real-Time Software Testing and Debugging. *IEEE Trans. Softw. Eng.*, 16(8):897–916, 1990.

[Tha00] Henrik Thane. *Monitoring, Testing and Debugging of Distributed Real-Time Systems*. PhD thesis, Royal Institute of Technology, Stockholm, May 2000. `http://www.mrtc.mdh.se/index.php?choice=publications&id=0242`.

[TIoEEE] The Institute of Electrical and Electronics Engineers. IEEE Standard for Information Technology — Portable Operating System Interface (POSIX) — Part 1: System Application Program Interface (API) — Amendment 6: Tracing [C Language]. Std 1003.1q-2000.

[TM99] Ariel Tamches and Barton P. Miller. Fine-grained dynamic instrumentation of commodity operating system kernels. In *OSDI '99: Proceedings of the third symposium on Operating systems design and implementation*, pages 117–130, Berkeley, CA, USA, 1999. USENIX Association.

[Tre86] R. K. Treiber. Systems programming: Coping with parallelism. rj5118, April 1986.

[Uni08] Computer Sciences Department — University of Wisconsin. Paradyn Parallel Performance Tools. `http://www.cs.wisc.edu/paradyn/`, 2008.

[Val96] John David Valois. *Lock-free data structures*. PhD thesis, Troy, NY, USA, 1996.

[Wei03] Andreas Weigand. Tracing unter L4/Fiasco, 2003. Großer Beleg (Undergraduate thesis), Technische Universität Dresden.

[Wei06] Eric W. Weisstein. Kolmogorov-Smirnov Test. `http://mathworld.wolfram.com/Kolmogorov-SmirnovTest.html`, July 2006.

[Wei09] Eric W. Weisstein. Chi-Squared Test. `http://mathworld.wolfram.com/Chi-SquaredTest.html`, December 2009.

[Wik] Wikipedia. Addressing mode: Memory indirect, autoincrement. `http://en.wikipedia.org/w/index.php?title=Addressing_mode&oldid=282493599#Memory_indirect.2C_autoincrement`. Retrieved on 2009-03-30.

[WR03] Robert W. Wisniewski and Bryan Rosenburg. Efficient, Unified, and Scalable Performance Monitoring for Multiprocessor Operating Systems. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 3, Washington, DC, USA, 2003. IEEE Computer Society.

[WSG96] Wanqing Wu, Madalene Spezialetti, and Rajiv Gupta. On-line Avoidance of the Intrusive Effects of Monitoring on Runtime Scheduling Decisions. In *ICDCS '96: Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS '96)*, page 216, Washington, DC, USA, 1996. IEEE Computer Society.

[XlRsT+06] Fan Xiao-liang, Zhang Rui-sheng, Ning Ting, Du Jin, and Zhang Chunyan. Investigating the Feasibility and Designing of Telecom Billing Systems Based on Real-Time Linux. In *Proceedings of the Eigth Real-Time Linux Workshop*, Lanzhou, China, 2006.

[YB97] Victor Yodaiken and Michael Barabanov. A Real-Time Linux. In *Proceedings of the Linux Applications Development and Deployment Conference (USELINUX)*, Anaheim, CA, January 1997. The USENIX Association.

[YD00] Karim Yaghmour and Michel Dagenais. System Administration: The Linux Trace Toolkit. *Linux J.*, page 22, 2000.

[YLW+06] Chun Yuan, Ni Lao, Ji-Rong Wen, Jiwei Li, Zheng Zhang, Yi-Min Wang, and Wei-Ying Ma. Automated known problem diagnosis with event traces. *SIGOPS Oper. Syst. Rev.*, 40(4):375–388, 2006.

[ZYW+03] Tom Zanussi, Karim Yaghmour, Robert Wisniewski, Richard Moore, and Michel Dagenais. relayfs: An Efficient Unified Approach for Transmitting Data from Kernel to User Space. In *Proceedings of the Linux Symposium*, pages 519–531, Ottawa, Ontario, Canada, 2003.

Index