Towards Runtime Monitoring in Real-Time Systems

Martin Pohlack, Björn Döbel, and Adam Lackorzyński
Technische Universität Dresden
Department of Computer Science
Operating Systems Group
{pohlack, doebel, lackorzynski}@os.inf.tu-dresden.de

Abstract

In this paper we present the state of our work on runtime monitoring for real-time systems: a way to observe system behavior online without unpredictably disturbing real-time properties.

We discuss generic requirements to achieve these properties wherefrom we deduce our monitoring framework architecture. We describe this architecture in detail and discuss several challenges for our implementation called Ferret. We also explain why common operating system primitives, such as message passing or system calls, should not be used for monitoring in the general case and propose a very low-intrusive alternative. We also propose a way of measuring the intrusiveness caused by monitoring.

We applied our technique in different scenarios ranging from simple temporal debugging, resource requirement estimation, gaining behavioral information of peripheral hardware devices to build timing models for providing real-time capable service on top of them, up to whole-system views, such as the interaction between concurrently running system threads. Our research platform also contains a para-virtualized version of Linux that we use to run legacy applications. We discuss how to apply our framework to these components with real-time requirements being only one of several important aspects. We also show how to compare the behavior of our para-virtualized Linux kernel with the behavior of the native variant.

In this work, we demonstrate how to gain a continuous whole-system view by using only Ferret sensors in all layers of our system, starting from the underlying microkernel, basic microkernel programs, real-time applications, and the para-virtualized Linux kernel, as well as Linux user-space applications.

1 Introduction

Computer systems have been growing in complexity for decades, by introducing new software layers and by increasing functionality in established layers. Accordingly, the number of bugs increased as complexity grew. Some problems can be found with static analysis. Runtime monitoring can help where static analysis fails by live peeking into running systems.

As many current systems are faced with these problems, there are many solution attempts. In this work we explore how to apply runtime monitoring especially to real-time systems. Our research platform exhibits some interesting features which influenced our design, in addition to plain real-time properties.

We use a para-virtualized version of Linux [11] to run legacy applications, whenever no real-time functionality is necessary. Even if certain applications have real-time requirements, it is often not the whole application, but only a small part of it. We therefore also support real-time and non–real-time components in one system, interacting with each other [15, 7]. As a consequence, one system configuration might comprise many different software layers with varying requirements. We want to monitor the system in all those layers.

Of course, creating events and storing them must be fast and nonblocking for the monitored components. And last, but not least important, monitoring must be simple to use.

Therefore, we built our monitoring framework Ferret such that it only uses shared memory for transporting monitoring data. This approach does not require any system calls — and as such costly kernel entries — at runtime and also makes Ferret independent of the concrete interface of the layer where we want to monitor (microkernel system calls, Linux system calls, etc.).

The remainder of this paper is structured as follows: In Section 2, we describe related work, followed by Section 3, detailing our design. We describe our experience with several typical application scenarios in Section 4. In Section 5, we conclude our paper and discuss future work.

2 Related work

In this section, we describe existing monitoring approaches for operating systems and evaluate their applicability to real-time system. Thereafter, we describe available utilities related to monitoring and evaluation of obtained data.

The *Linux Trace Toolkit* (LTT) [19] consists of patches to the Linux kernel that instrument a fixed number of locations inside the kernel. These sensors can be switched on to collect subsystem-related events. A kernel module stores data and provides it to user-space applications for further processing. Our Ferret monitoring framework is similar in that it provides means to generate events, and collect and store data. It is different to LTT, because it does not contain predefined instrumentation for any part of our system environment. However, we can reuse the locations identified by LTT in our Linux version.

Dynamic Probes (DProbes) [12] allow users to dynamically instrument arbitrary locations within an application or the Linux kernel. Whenever a so-called probe point is hit, the framework executes user-defined probe handlers. DProbes require probe handlers to be written in a special language. Dynamic probes have the advantage that they do not pose any probe effect on the system, if monitoring is turned off. DProbes mainly provide means for dynamic instrumentation, whereas Ferret aims at providing an infrastructure for monitoring. Thereby, both means can complement each other.

Kernel Probes (kProbes) [10] introduce the DProbes concept to 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 trap event opcode. 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. kProbes provide means for dynamic instrumentation in the Linux kernel, which we use to place Ferret sensors.

Sun's Solaris operating system uses *DTrace* for tracing. DTrace also does not have any probe effect if tracing is turned off for dynamic probes. Static probes have no-ops in place if deactivated, which are patched with calls to DTrace trampoline code on activation. In addition to kProbes, DTrace is able to probe user-space applications. Probes are written in a specialized language called D. Although DTrace offers versatile aggregation support, it is not especially suited for real-time systems, for example, probe code is executed with interrupts turned off in the kernel.

Each of these approaches uses system calls to hand over data between in-kernel sensors and user-space monitors. Dynamic probing additionally uses exceptions to handle probes. These means add an additional overhead to the probe effect caused by the instrumentation. We therefore do not use dynamic probes in the real-time parts of our system, but only static probes. On the other hand, dynamic probes are an extremely flexibly way of gaining information about unforeseen problems in running systems. We therefore use kProbes in our Linux kernel version (a non–real-time part) to instrument it with Ferret sensors. We describe the kProbes port in Section 3.5.

In his thesis “Monitoring, Testing and Debugging of Distributed Real-Time Systems” [18] 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, we target a different class of systems where we can not even enumerate all potentially required sensors. Also, in our scenarios, the applied sensors and their type were quite problem specific. While the single sensors have a negligible overhead, always enabling all of them would slow down the system considerable. We can, of course, use Thane's approach for simple, hard real-time parts of our system, where functionality and monitoring requirements are known beforehand and are static. In this case type, number, and functionality of the sensors can be determined.

Magpie [2] is a toolchain that can, based on observed events of one or several connected systems, extract requests. Requests are typically the unit of interest for human interpretation or performance evaluation, for example, everything that belongs to a single `HTTP` request (potentially crossing machine boundaries on the server side for accessing a database), or everything related to displaying a single video frame. What comprises a request is highly problem specific and can be defined in schemata. Also other event post-processing is defined there.

Originally, Magpie worked only as a client for Event Tracing for Windows (ETW). It can process traces online and offline, but does not give real-time guarantees. However, online usage was shown to be feasible in a typical business scenario [2].

ETW however, has properties that make it unsuitable for real-time systems. Communication with clients works with a set of buffers that are handed to consumers once they are full; there is no upper latency bound (e. g., if buffers are filled slowly). In [2] Barham and colleagues mention an approximate processing delay of one second for the online version of Magpie. Additionally, ETW notifies event consumers with call-back functions that introduce additional causal relationships between the observed and the observing entity. This infringes the desired property of the minimal probe effect. We avoid this problem in FERRET by *not* synchronously notifying monitors.

As Magpie has interesting properties for finding, building, and post-processing requests, we modified the Magpie toolchain to work with FERRET and adapted its inner workings to our L4 architecture. In Section 4 we present example traces processed with our extensions to Magpie.

Different sources give different numbers about ETW overhead: [13] estimates 1500–2000 cycles and [14] about 1000 cycles. For posting ETW events from user-space, at least one kernel entry is involved. In FERRET we do not pay kernel-entry costs for each event.

3 Design

While designing FERRET we tried to bring in line general requirements for runtime monitoring systems (e. g., low overhead, ease-of-use, versatility, ...), as well as requirements derived from the real-time nature of the target system (DROPS [3]). In this design discussion we focus on the latter properties that are low intrusiveness, low probe effect, or Heisenberg's uncertainty principle for software.

We aim at a software-only solution, that is, we have no additional hardware devices snooping memory busses, CPU cycles, or unusual high precision timers, because we want to deploy our framework easily on standard machines, basically everywhere where our system runs.

DROPS consists of many heterogenous components, including L^{4}Linux and normal time-sharing software on top of it. Also, we have several scenarios where real-time and non–real-time components interact. Therefore, we want to be able to monitor all types of components and their interaction.

Many systems implement monitoring for user-level components with the help of system calls that care for sensor buffer management, taking timestamps, ordering, and atomic execution of monitoring code.

We have not chosen this approach for several reasons. First, it includes the additional overhead of a system call (kernel entry). Second, some components in our system cannot and must not directly interact with the microkernel using system calls (e. g., L^{4}Linux user-space programs). Third, using kernel primitives for monitoring might simply be too intrusive as it might change the system's behavior we currently want to observe (e. g., scheduling). Instead, we only use shared-memory buffers, which can be completely pre-mapped and pinned, so that monitoring itself only incurs memory access, which should be possible on all environments. Furthermore, we use superpages and cache-line aligned data structures to further reduce sensor overhead.

Instrumented real-time and non–real-time applications can post events by only using memory accesses. A non–real-time monitor can collect these events online or offline. In the offline case, sensor buffer memory must be large enough to hold all events. The monitor waits until the experiment is finished and collects data afterwards. It does not influence the experiment at all in that case. For longer running experiments, the monitor works online and periodically polls sensors and collects events from them. It can process, filter, or store events. Therefore, the monitor can either run with non–real-time priority if there is enough slack time in the system or it can be scheduled as real-time task. In both cases, the monitor runs concurrently to the experiment, but there is no observable relationship that would influence traces, as we refrain from synchronously notifying monitors of the availability of new data. This would be costly in the event creation path and might create new communication relations that we did not have in the unmonitored system. We can calculate the necessary amount of shared memory for storing events if the event rate is known beforehand [16].

3.1 Roles and nomenclature

In the following we define terms used throughout this paper:

Events are assumed to be instantaneous and have a timestamp associated. Events can carry additional payload, where the header typically contains major and minor numbers, as well as an instance identifier. Events are typically posted into sensors but may also be serialized into a persistent event stream for later evaluation.

Event types describe the payload data layout for a class of events and have a semantic attached as they stand for a certain action in the system. Events are instances of event types.

Sensors are unidirectional means of transportation

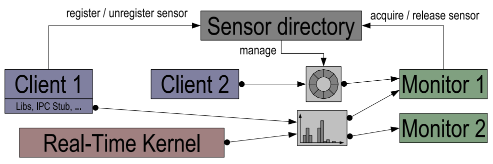


Figure 1: Shown is the interaction of the sensor directory, traced processes (clients), and monitors. The kernel is shown as well as a special event source. Also note that there is an n:m relation between event producers and consumers.

for events. They are writable for event producers and readable by monitors. Internally, sensors are composed of a shared-memory region with meta-data describing the data layout and identifying the type and instance of a sensor. Additionally, they contain a container area for events.

Probe points are locations in programs where events may be created and posted. Instrumentation places sensor code at probe points either statically or dynamically.

In Ferret, we have three roles involved in tracing, which roughly correspond to the three roles defined in the "POSIX Trace standard 1003.1q" [9] (PTs). We have traced processes (also called traced processes in PTs) as event producers, we have monitors as event consumers (called trace analyser processes in PTs), and we have one sensor directory in Ferret (the trace controller process in PTs). The interaction of these roles is shown in Figure 1.

In the following, we will discuss the responsibilities of each of these roles in Ferret and discuss differences to PTs.

Traced process With Ferret, the creation of sensors is triggered either by the traced process or by a third party on behalf of the traced process. In each case, the sensor itself is setup and configured in a memory region by the sensor directory that then maps the shared-memory region to the traced process.
Actually tracing events happens by memory accesses into the sensor area inside the traced process. Therefore, the traced process uses inline functions defined in Ferret header files.
Ferret comes with a set of common event types predefined. Customized types can easily be defined by the user as well.

Monitor 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 constants. After looking up sensors, monitors register themselves as consumers. The sensor directory gives them read-only access to the corresponding shared-memory region. These two mechanisms (the read-only access and the indirection over the sensor directory) prevents information flow back to monitored processes from the monitors through Ferret.
After acquiring access to all required sensors, monitors can periodically look into their sensors for new events. Evaluation can happen online or offline by storing event streams for later use.

Sensor directory The Ferret sensor directory corresponds roughly to the Trace Controller Process in PTs. It is responsible for creating and initializing new sensors, granting access for other producers and consumers, managing the sensor name space, and the handling of instances. Ferret supports several instances of a traced component running side by side, so, beside the sensor identifier, we 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 producing and consuming tasks, the sensor directory is not involved in the event flow. Filtering of relevant events can be done either in the producer, by just not posting certain events, or in the consumer by ignoring events. At a more coarse-grained level, filtering can also happen by subscribing to specific sensors. Ferret allows for very fine-grained sensor usage, also across processes, such that events might be grouped thematically, rather than by source. In the PTs, event filtering is done by the Controller Process, which is therefore involved in all event transportation.

3.2 L^{4}Linux

L^{4}Linux is a port of the Linux kernel to an L4 environment. It uses L4 kernel primitives as well as the basic software environment (L4Env) running on top of the microkernel to provide the necessary environment for running a Linux kernel. These services include communication primitives, execution contexts, interrupts, memory services and the like. For example, L^{4}Linux queries the memory server for a chunk of memory that it then uses as the main memory.

Internally, the L^{4}Linux kernel is divided into several threads within the same address space. The main thread runs the actual kernel code. L4 requires that threads that want to receive interrupt messages need to attach to an interrupt. Consequently, each interrupt that is used by L^{4}Linux requires an interrupts thread. These threads execute bottom half code and then notify the main thread. Besides the main thread and interrupts threads, special L^{4}Linux drivers may start own threads to receive asynchronous notifications. Additionally, L^{4}Linux has a so called tamer thread that is used to serialize access to critical sections in case of contention. There are additional management threads as well as service threads of L4Env.

3.3 Layers

From the monitoring point of view we have four different layers in our System: the microkernel (L4 Fiasco in our case), basic microkernel programs, a paravirtualized Linux kernel, and Linux programs. We support monitoring in those layers and describe this in the following.

For monitoring kernel and system behavior, we use Fiasco's built-in tracebuffer, which can be configured at runtime. We access the tracebuffer as shared-memory region and read events from it containing kernel entries. The tracebuffer supports a set of standard events, such as context switches, inter process communication (IPC), page faults, and so on, which can be selectively activated. The kernel as an event source is sometimes important as we cannot atomically take timestamps in user-space directly before or after IPC or other system calls. Using the kernel here eases accounting. Also, it is very convenient to use the kernel as event source for gaining a general system view as sensors in only one place have to be activated. The FERRET framework provides the tracebuffer as a normal event list sensor to monitors.

Basic microkernel programs, with our without real-time properties, can be monitored with normal FERRET sensors. FERRET uses two L4Env services for setting up sensors: names (name server) and dm_phys (physical memory manager). The sensor placement in the virtual address space is handled by FERRET if the L4Env region mapper is available or it can be specified manually. When instrumenting names and dm_phys themselves with FERRET sensors, we have bootstrapping problems, naturally. In names, we solved this by deferring sensors setup until dm_phys registered at names. For dm_phys we will use a preinitialized sensor to circumvent cyclic dependencies.

We could have prevented those two special cases by reimplementing a lot of functionality of names and dm_phys in the FERRET framework but decided not to do so, as the gains for all this code duplication seemed small.

The current version of the L^{4}Linux kernel is an L4Env program, so monitoring it with FERRET works normally. Additionally, the L^{4}Linux kernel sets up one event list sensor for its user-space programs and pages the sensor's memory into their address spaces on demand. It acts as a proxy between FERRET's sensor directory and L^{4}Linux user-space programs, as those programs are normally not allowed to contact other L4 programs. Another advantage of this architecture is that only one entity manages the virtual address spaces of Linux user-space programs, the L^{4}Linux kernel.

3.4 Sensor types

To address the different problem domains we are facing and to minimize intrusiveness, we provide different types of sensors in FERRET. The most simple sensor is a counter type we call *scalar*. This type is basically used for counting the occurrence of events, for example, the number of deadline misses in a time interval for a real-time task.

The next sensor type is the *histogram*. Histograms can be accessed either as arrays of scalars (using the bin number) or as real histograms with offsets and index scaling. Overflows and underflows are counted additionally. Amongst other things, we use histograms for acquiring the resource usage for various routines in the system, for example, the video and audio decoding steps in our video player Verner [17]. We use these numbers directly for CPU time reservation in Verner. In L^{4}Linux, we use the array mode of the histogram sensor for counting the calling frequencies of system calls [4].

Histograms are not restricted to one dimension, but can have multiple dimensions and layers.

The most general form of sensors are *event lists*. Events can be posted from all positions in the system and can contain arbitrary data. All events are timestamped and can therefore be totally ordered per processor. Using event lists 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.

In the general case, event lists are read from and written to by several parties concurrently, that is, there might be several producers in different address spaces. Our event list sensor is based on the Concurrently Invocable Sensors from [16] and uses lock-free algorithms for achieving synchronization.

We defined a common event header denoting the origin and type of events, followed by custom data. We describe the data layout for the custom parts of all events in the Magpie toolchain (cf. Figure 2).

3.5 Porting kProbes

The kProbes mechanism in Linux allows to insert trap points at arbitrary positions in the kernel. When the execution flow passes such a point, a

```
0 Kernel                           1 L4LXK
#type context_switch 10            #type atomic_begin 2000
{                                  {
  context,    ItemULong              l4tid, ItemULongLong
  eip,        ItemULong            }
  pmc1,       ItemULong
  pmc2,       ItemULong            #type atomic_end1  2001
  _pad0,      ItemChar             {
  _pad1,      ItemChar               l4tid, ItemULongLong
  _pad2,      ItemChar             }
  _pad3,      ItemChar
  dest,       ItemULong            #type atomic_end2  2002
  dest_orig,  ItemULong            {
  kernel_ip,  ItemULong              l4tid, ItemULongLong
  space,      ItemULong            }
  sched_cont, ItemULong
  from_prio,  ItemULong
}
```

(a) Context switch event *(b)* Atomic blocks

Figure 2: Exemplary data layout description for microkernel context switch events with detailed information (a) and simple wrapping events for atomic sections in the L^{4}Linux kernel, containing just a thread ID (b)

trap is raised and kProbe trap handler functions are called.

Having kProbes available in L^{4}Linux allows us to use legacy Linux instrumentations. The SystemTAP project [5] aims at providing a scripting language to create dynamic instrumentation. Its developers focus on kProbes and there is already a large range of instrumentations for common problems available.

Installing a kProbe works by saving the instructions at the kProbe point to a private area and overwriting the point to be probed with the shortest possible instruction that causes a trap or exception. When the point is hit, the kProbe module calls user defined handlers. Then, the original code, saved in the private area, is executed in single step mode and finally execution is resumed after the kProbe.

The i386 kProbes implementation uses the one byte opcode INT3 as the trapping instruction. However, INT3 is also used to call the Fiasco kernel debugger. We replaced the INT3 opcode in kProbes with HLT, that also causes a general protection fault when called in user mode. HTL is also a good choice as it is definitely not used in the Linux kernel otherwise, except in the idle loop, which we implemented differently in L^{4}Linux. Other alternatives would have been CLI or STI, but L^{4}Linux would then be restricted to run without full I–O privileges, which is convenient sometimes.

We also changed the exception handler within L^{4}Linux such that an exception on HLT is handled by kProbes. Additionally, the instruction pointer of the exception needed to be adjusted, as for the HLT instruction it points *onto* the instruction, whereas for INT3 it points *after* the instruction.

4 Application scenarios

In the following, we describe the application of the Ferret monitoring framework to typical scenarios we encountered while working with our system. We chose the scenarios to highlight different properties of the framework that we think are important.

4.1 Resource usage estimation

The first application scenario is centered around our video player Verner [17]. In [8] we describe how we achieve real-time guarantees for video decoding. In short, we adapt the quality of the post-processing step and thereby change its CPU-time demand. We count the number of deadline misses over the previous n frames and adapt the post-processing quality accordingly. We can either lower the quality (if many deadline missed occurred), keep it the same (if we had only few misses), or raise it (if there were none). We can tolerate several deadline misses because we buffer some frames.

Interesting from the runtime monitoring perspective is that we only need to embed very little sensor code into the application (the time slice overrun handler). Thereby, we can completely separate the adaptation decision from the functional core of the video decoder. We only need a simple scalar sensor in the decoding component.

We also augmented Verner with histogram sensors for measuring the CPU time demand for certain routines, most importantly, the video decoding step, post-processing video frames, and the audio decoding step. We can view the distributions live, while the video is playing for observing system behavior and we use collected execution time distributions for admission and scheduling of the whole video player application.

We also used event list sensors for verifying buffer fill levels between components in the decoder chain and for collecting information about the development of the decoding times over time (or stream position).

4.2 Resource usage modeling

With DOpE [6], 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. Time estimation for copy routines is currently based on area size (pixel count) to be copied.

To verify this estimation we instrumented the inner copying routing of DOpE and took the time in CPU cycles for copying rectangular areas.

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

We 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 also two layers, whereas the first layer contains the accumulated copy time, and the second layer counts the number of occurrences for this width-height combination.

We directly aggregate the information online to minimize the memory load in this experiment as we are a taking huge number of measurements. We create redraw requests with uniformly distributed width and height between 1 and 400 pixels using a small benchmark program. We also measure each point at least 100 times and compute average copy times.

We 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 byte cache lines. The machine has a 64 kB Level 1 Instruction-Cache (2-way associative), a 64 kB Level 1 Data-Cache (2-way associative), and a 64 kB Level 2 Unified-Cache (8-way associative).

Machine B has an older Intel Pentium-Pro with 200 MHz. First and second level caches have 32 byte cache lines. The machine has an 8 kB Level 1 Instruction-Cache (4-way associative), an 8 kB Level 1 Data-Cache (2-way associative), and a 256 kB Level 2 Unified-Cache (4-way associative).

In the experiments depicted in the Figure 3 we see that the assumption of fixed cost per pixel is a viable approximation for a large range of rectangular sizes. However, we also see diversions in several places, which we discuss in the following:

1. The mountainous area on the left side results from copy operations on rectangles with a small width. The huge increase in copy time per pixel for very short pixel rows probably stems from computing the pixel row addresses for source and destination buffer.
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 3a, 3b, and 3c were taken with a horizontal screen resolution of 1024 pixels, resulting in aliasing effects with the processor cache size. Effectively, every cache color[1] is bound to a small set of pixel columns. This leads to trashing the own cache set when copying areas with more than a certain height, independently of the width of the rectangle.
   Machine A has a data cache of 128 kB (L1 and L2 together as the cache hierarchy is exclusive). Running with a graphics mode of 1024 pixel per line, with 2 bytes per pixel results in a cache-trashing height of 64 lines, computed as follows:

   $$128\ \text{kB}/(2\ \frac{\text{B}}{\text{pixel}} * 1024\ \frac{\text{pixel}}{\text{line}}) = 64\ \text{lines} \quad (1)$$

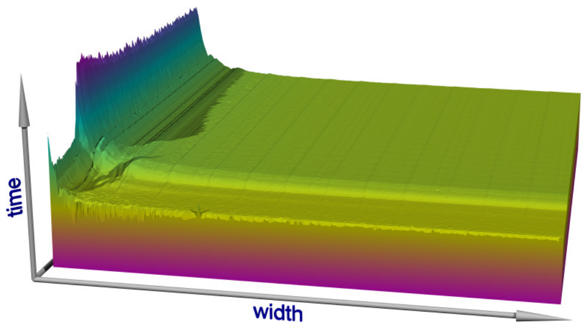
   This is also exactly what we see in Figures 3a and 3b.
   Compare Figures 3b and 3d. The most prominent difference between Figures 3b and 3d is that data for Figure 3d was taken with an `800x600` resolution, whereas data for Figure 3b was measured with an `1024x768` resolution. In Figure 3b, the cache trashing line can be clearly seen (parallel to the width axis), whereas in Figure 3d it is gone, as their is no aliasing between cache colors and pixel columns.
   Machine B has an effective data cache of 256 kB (maximum of L1 and L2 as the cache hierarchy is inclusive). Using equation 1 with otherwise equal settings we compute a cache-trashing height of 128 lines that can also be seen in Figure 3c.

For comparison we also ran our experiments with memory type range registers (MTRR) disabled, as this was the initial situation when DOpE was created in the year 2002. We see that enabling MTRRs for the framebuffer area results in an approximately three-fold speedup (compare Figures 3a and 3b). 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-cost-per-pixel assumption questionable.

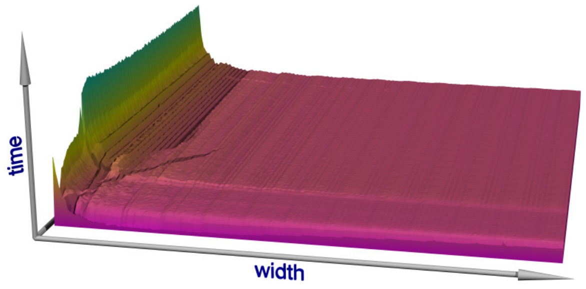
Also, in the measurements taken on machine B the differences in the copy times per pixel contrast even stronger as depicted in Figure 3c.

From the experiments we see 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 but on the length of both rectangle sides seems feasible from the mea-

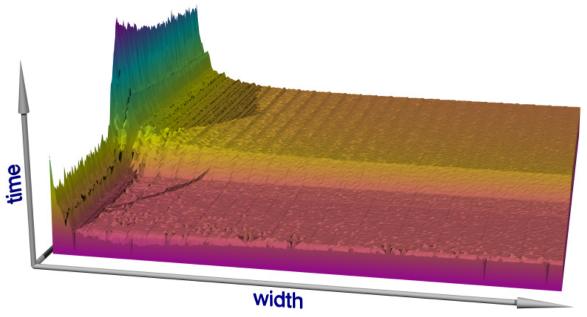
[1]A cache color is the set of cache lines that can cache the same physical address.



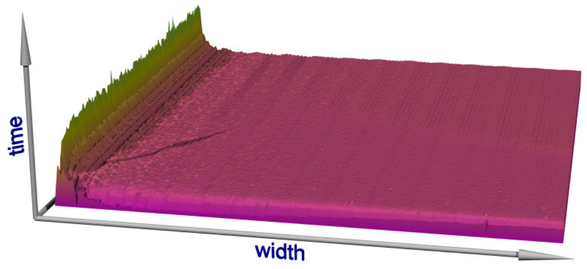
(a) Basic experiment with MTRRs disabled.



(b) For this experiment we used MTRRs 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 3b but the resolution is redcuded to `800x600`, eliminating the cache aliasing effect.

Figure 3: Depicted is the time for copying a single pixel to graphics card memory, depending on the width (horizontal axis) and height (axis "into" the paper) 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 3a, 3b, and 3d quantitatively, which were measured on *machine A*)

surements 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.

4.3 Taming L^{4}Linux

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

Native Linux uses CLI and STI instructions to protect critical sections by disabling and enabling interrupts. L^{4}Linux uses a replacement for CLI–STI as it runs without kernel privileges in user-mode where these instructions are not allowed. CLI–STI is replaced with a mutex, which is taken using atomic instructions in the noncontention case. In the contention case, a blocking IPC is sent to one synchronizing *tamer* thread *T*. *T* runs on the highest 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 leaving a critical section and another thread wants to enter the critical section, the leaving thread also notifies *T* that, in turn, wakes up one waiting thread according to its queuing policy. Again, the underlying assumption here is that by running T on the highest priority inside L^{4}Linux it can run its code atomically with respect to all other L^{4}Linux threads.

L4 kernels have a feature called donation, which optimizes a common communication case, where a client sends a short request to a server, which 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 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 separate subsystems.

In our case one L^{4}Linux-internal stub driver thread *A* was notified by an external thread *W* running on the same priority as L^{4}Linux's tamer thread. By notifying *A*, *W* temporarily transfered its priority to A. As a consequence, although extremely rarely, T was interrupted in its atomic sequence when A itself wanted to enter the CLI–STI critical section.

After identifying this problem we wrote a monitor that detects this problem at runtime. Therefore, we wrapped the tamer's atomic sequence with start–stop events. Additionally we enabled logging of context switch events in the microkernel. The monitor checks for the absence of the following condition: 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 for the atomic sequence. The monitor also keeps a history of the previous n events for later visualization and debugging of the problem. Figure 4 shows such a situation.

After we fixed the problem, this test can now be run after any changes we make 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 condition is checked 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.

4.4 Virtual and native Linux

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

We compared L^{4}Linux to native Linux running comparably configured kernels. We used kProbes to track control flow inside the Linux kernel. For Drops, we used kernel-level instrumentation to track scheduling information. In native Linux, an additional kProbe was used for this purpose. To reuse the instrumentation code from L^{4}Linux within native Linux, we implemented a Linux kernel module that implements parts of the Ferret framework.

One of our experiments showed a difference in scheduling behavior between L^{4}Linux and native Linux within the `vfork` system call. `vfork` is a special version of `fork` that assumes that only little work has to be 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 the parent's page tables. The parent process has to be blocked to prevent address space corruption until the child calls `exec` or `exit`.

While observing the behavior of `vfork` in L^{4}Linux, we found, that after the system call has

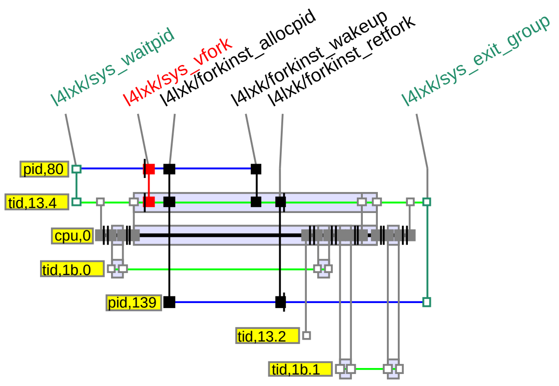


Figure 5: Depicted is a **`vfork`** system call, issued by pid 80 (tid,1.b0). After the L^{4}Linux kernel (tid,13.4) returned from the fork routine (**`l4lxk/forkinst_retfork`**), the parent process (tid,1.b0) is scheduled again for a short time, before finally the child (tid,1b.1) is scheduled.

been handled by the L^{4}Linux server, the Fiasco kernel switches back to the parent task before executing the child (see Figure 5). We further investigated this behavior, because this looked like a serious bug in either L^{4}Linux or the Fiasco kernel. As it turned out, this is only an optimization artifact in Fiasco, never visible to user-space. Fiasco indeed switches to the parent process, but only to find out that it is blocked. The parent process was 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 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 other case, where the parent was flagged and has now to be removed from the ready queue.

Although this bug turned out to be a false positive, this example shows that Ferret can be very helpful 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.

When evaluating performance, the overhead induced by monitoring needs to be considered in order to measure correct values. The two main sources of intrusion are:

1. The overhead caused by measuring and storing data, and calling functions provided by the monitoring framework. We used microbenchmarks, executing thousands of monitoring calls in a row, to determine the minimum time that is needed to run these functions. Additionally, we used macrobenchmarks, executing real applications, to measure the average and maximum effect monitoring calls have on the system.
2. The cache misses that are caused because

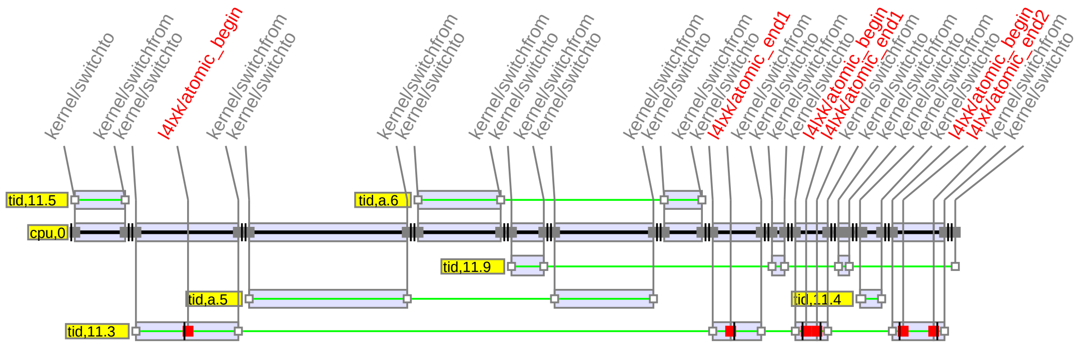


Figure 4: Visualization of the atomicity problem with the tamer thread (11.3). You see three atomic sections wrapped in the events: `l4lxk/atomic_begin` and `l4lxk/atomic_end`. The first of the three sections is interrupted with context switches to thread A.5, A.6 (both in the external driver), and 11.9 (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.

the above mentioned monitoring calls overwrite cached data that is used by the instrumented applications. To measure the cache overhead, we again conducted micro- and macrobenchmarks and used hardware performance counters to determine the number of cache misses.

By knowing the intrusiveness of instrumentation and the monitoring framework, we are able to reassess obtained timing data and approximate the real performance of the monitored system [4].

4.5 Whole-system view

To illustrate the continuity of the Ferret framework we demonstrate how to follow the control flow from a L^{4}Linux user-space program (`arping`), through the L^{4}Linux server, further on to an external L4 network server (ORe), using different and adequate means to place sensors in the different layers of the system.

To visualize the control flow for this scenario, we used the following sensors:

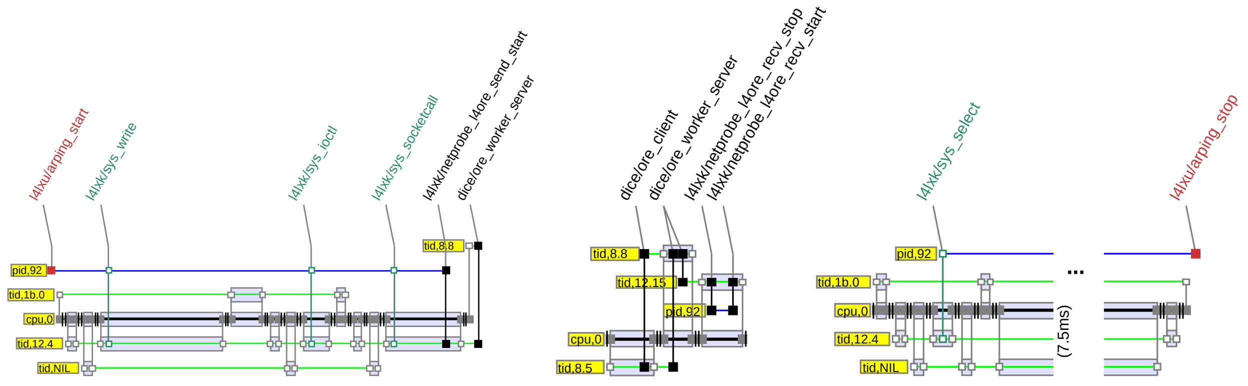
1. The `arping` program running in L^{4}Linux was manually instrumented to produce events before sending a request, before listening for an answer, and after processing the answer.
2. The L^{4}Linux kernel was instrumented using a kernel module installing kProbes at the system call entry, the `send` function, and the `receive` function of the L^{4}Linux ORe driver stub.
3. Drops applications usually provide an IPC interface to other components of the system. This interface is typically described using CORBA IDL. The Drops IDL Compiler (Dice, cf. [1]) translates this description into IPC code. We instrumented the server-side IPC code of the ORe network switch using a tracing plugin for Dice.
4. Information about context switches were obtained from the Fiasco kernel using events written to the Fiasco trace buffer.

Extracts of the resulting trace can be seen in Figure 6. Figure 6a shows how the ping request is sent from a user-space task to the L^{4}Linux server (tid,12.4), which then hands the packet over to the ORe network server. Figure 6b shows the answer coming in. The ORe IRQ thread (tid,8.5) notifies the server thread (tid,8.8) and the server thread then calls back the already waiting L^{4}Linux driver stub (tid,12.15). In Figure 6c we see the ping application doing a `select` system call on the socket and finally receiving the ping reply.

The example demonstrates that with the help of Ferret we are able to instrument and monitor applications at different system layers and use the obtained data as a basis for evaluating whole systems. The demonstrated approach is not constricted to our research system but may be applied to other layered architecture as well. For example, other virtual machine setups will require event from the low-level hypervisor, medium-level virtual machine monitors, and high-level guest operating systems and applications as well. Ferret provides a framework to create, transport, and process these events.

5 Conclusion

We created a monitoring system, solely based on shared-memory regions, not using any system calls in normal use. This results in very fast and predictable logging suitable for real-time and high-performance applications. We also achieve independency from the actual operating system the monitored process runs in, indicating the applicability in virtualization scenarios, as we can use this technique in all layers of our system, starting from the underlying microker-



(a) Ping start *(b)* Ping incoming *(c)* Ping done

Figure 6: Visualization of the system interaction in response to the ping request by the application. Depicted is only the send part wrapped in the two user-land events `l4lxu/arping`. You see the interaction from the user-space program (pid 92) with the L^{4}Linux server (task 12), where 12.4 is the main server and 12.15 being the driver stub communicating with the ORe network server. The network server is task 8.

nel, plain microkernel programs, real-time applications, and our para-virtualized Linux kernel, as well as Linux user-space applications. It is extremely helpful for continuous monitoring to use only one mechanism in all places and to be able to collect all monitoring data with one framework.

Runtime monitoring of real-time systems is possible and useful. However, we find that one size does not fit all. The problem range is just too large for a generic solution with one sensor type. Even if one could obtain monitoring data at zero cost (CPU cycles) the sheer amount of data would have to be handled. Therefore, we applied early aggregation in the forms of histogram and scalar sensors to reduce the amount of data to be stored, transported, and evaluated later on. At first glance this might sound counterintuitive as aggregation itself does cost some CPU time in the monitored process, but the whole system will be relieved of a lot more load that way, resulting in lower overall intrusiveness. Currently, the user must decide whether and how to aggregate.

6 Outlook

In microbenchmarks, the event list as the most complex sensor can create events with an approximate overhead (depending on the sensor configuration and the payload) of about 200–300 cycles on an 1.2 GHz AMD Duron. The biggest costs are the copying of the payload, taking timestamps (with `rdtsc`), and synchronization against concurrent writers with the help of `cmpxchg8b` instructions.

Specifying the intrusiveness in a larger scale makes only sense in the context of an observer, either a human judging the system's behavior or machine parts evaluating metrics. For a server environment, the most important metric might be throughput, so intrusiveness could be defined over the change in throughput. For a real-time capable video player, the number of deadline misses is important, so the change in this fraction is the intrusiveness. In a hard real-time system, there should be no new deadline misses — here the intrusiveness is binary. For a desktop system where humans might not notice small changes but penalize larger ones, responsiveness is important. So, intrusiveness might not be linear with additional CPU time required. We will next work on defining metrics for certain application classes and verify our framework within these metrics.

Currently, events can be totally ordered per processor, as we use processor timestamp counter, which are strictly monotonic. Totally ordering event across several processors would require perfectly synchronized clocks, which probably cannot be achieved for such high-resolution time sources as timestamp counters. Instead, we envision to a use partial ordering relation by taking two local timestamps for cross-processor communication. We also plan to support different timestamp source — for example, logical clocks which might be cheaper (just a counter in memory) — for situations where not the accurate timing but only the ordering is important.

References

[1] Ronald Aigner. DICE Documentation. http://os.inf.tu-dresden.de/dice/.

[2] Paul T. Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using magpie for request extraction and workload modelling. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, San Francisco, CA, 2004.

[3] DROPS Team. DROPS - The Dresden Real-Time Operating System Project. http://os.inf.tu-dresden.de/drops/.

[4] Björn Döbel. Request tracking in DROPS. Master's thesis, TU Dresden, June 2006.

[5] 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.

[6] 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)*, Cancun, Mexico, December 2003.

[7] Steffen Göbel, Christoph Pohl, Ronald Aigner, Martin Pohlack, Simone Röttger, and Steffen Zschaler. The COMQUAD component container architecture. In *Proc. 4th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, Oslo, Norway, June 2004.

[8] 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*, 2006.

[9] IEEE. *IEEE Std 1003.1-2004 Standard for Information Technology — Portable Operating System Interface (POSIX) System Interfaces, Issue 6.* IEEE, New York, NY, USA, 2004.

[10] R. Krishnakumar. Kernel Korner: Kprobes—a Kernel Debugger. *Linux J.*, 2005.

[11] Adam Lackorzynski. L^{4}Linux Porting Optimizations. Master's thesis, TU Dresden, March 2004.

[12] 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.

[13] 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.

[14] 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.

[15] 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. http://os.inf.tu-dresden.de/papers_ps/tr-rtnonrtcomp.pdf.

[16] Torvald Riegel. A generalized approach to runtime monitoring for real-time systems. Master's thesis, TU Dresden, 2005.

[17] Carsten Rietzschel. VERNER – ein Video EnkodeR uNd playER für DROPS. Master's thesis, TU Dresden, 2003.

[18] Henrik Thane. *Monitoring, Testing and Debugging of Distributed Real-Time Systems.* PhD thesis, Royal Institute of Technology, Stockholm, May 2000.

[19] Karim Yaghmour and Michel Dagenais. Measuring and characterizing system behavior using kernel-level event logging. In *USENIX Annual Technical Conference, General Track*, 2000.