Predictable Low-Latency Interrupt Response with General-Purpose Systems

Adam Lackorzyński, Carsten Weinhold, Hermann Härtig
Operating Systems Group, Technische Universität Dresden
{adam.lackorzynski,carsten.weinhold,hermann.haertig}@tu-dresden.de

***Abstract*—Real-time applications require predictable and often low-latency response times when reacting to external events. Real-time operating systems allow applications to meet such timing requirements, but they offer less functionality and fewer APIs than a general-purpose operating system such as Linux. In this work, we present a virtualization layer that enables predictable, low-latency interrupt handling for Linux-based real-time applications, even if Linux itself or unrelated applications cause heavy load. The benefit of our approach is that developers can draw from the rich feature set and powerful infrastructure provided by Linux, but their applications can meet timing requirements as if they ran directly on a real-time operating system. Our benchmarks show a reduction of worst-case interrupt response times by more than two orders of magnitude compared to standard Linux, and by a factor of 3 on average.**

I. Introduction

Many workloads, like for example those in control systems, require both predictable execution times as well as low-latency response to external events. To satisfy these demands, the designers of such systems rely on real-time operating systems (RTOS). Compared to general purpose operating systems (GPOS) such as Linux, classical RTOSes are much simpler and they often run just a single application on dedicated hardware. This no-frills approach to system design provides timing-critical applications with a predictable execution environment and it can guarantee consistently low latency for event processing. However, it also means that RTOSes offer a smaller feature set, unfamiliar APIs, and even different development tools than commonly used general purpose systems.

If a use case requires both real-time processing and the rich functionality of a GPOS, one typically couples two different computing systems, for example, via a common memory. Alternatively, system designers can use a multi-core system, where each of the two OSes runs on a different core. In the multi-core setup, the two systems are not protected from each other. Thus, a malfunctioning GPOS, or one that has been compromised by an attacker, can negatively influence the RTOS. A solution to this problem is to run the real-time and the general-purpose software stacks in virtual machines (VMs), with a hypervisor providing shared memory for communication between the two isolated domains. However, in all the architectures described above, the RTOS is separate from the general-purpose system. An application scenario that requires services of both OSes must be explicitly developed as a split application without readily available mechanisms to let the two components cooperate.

In this work, we aim to combine both worlds by tightly integrating a real-time capable microkernel with a virtualized Linux kernel running on top. In this system architecture, we start real-time applications as ordinary Linux processes, but let their threads execute directly on the microkernel. Spatial isolation is still provided by Linux, but the microkernel can enforce temporal isolation even if Linux is heavily loaded or stops. This way, these programs can execute under real-time guarantees of the microkernel, while their non-real-time parts can benefit from the feature-rich environment offered by Linux and the huge amount of software available for it. This architecture builds on previous work on *decoupling* the execution of user threads from the Linux kernel scheduler [1], [2] in order to eliminate execution-time jitter caused by Linux housekeeping tasks and concurrently running applications. In this paper, we extend this mechanism to support interrupt handling in user-level threads, thereby enabling predictable and low-latency reaction to external events in Linux-based real-time programs. Using two different hardware architectures, we evaluate interrupt latency for both standard Linux on bare-metal hardware and our virtualization-based architecture, where events are handled by threads running *decoupled* from Linux.

The remainder of the paper describes the decoupling mechanism and how to use both Linux and L4Re system calls. It then evaluates interrupt latency characteristics, before we conclude.

II. System Architecture

A detailed description of the decoupling mechanism can be found in [2]. In this paper, we only summarize how the basic building blocks of our system architecture work together and how they enable noise-free and predictable execution of Linux-based programs. We then describe how we extended the decoupled execution model to support predictable and low-latency interrupt service routines in a Linux user-space program.

A. Decoupling

The decoupling mechanism is based on the L4Re microkernel system [3] and L^{4}Linux [4], a paravirtualized Linux kernel that runs on top of it. The L4Re microkernel can run unmodified OS kernels in hardware-supported virtual machines. However, L^{4}Linux has been specifically adapted to run on top of L4Re

as an unprivileged user-level application. A key property of this tight integration is that L4Linux reuses address-space and threading APIs of the underlying L4Re microkernel to implement Linux processes and threads. In our previous work on decoupled thread execution, we modified L4Linux such that the execution state of a Linux user thread can transparently migrate to a dedicated L4Re thread that is controlled directly by the L4Re microkernel. Such a decoupled thread is running in the same address space as the Linux process, but it is not subject to scheduling decisions of L4Linux. Since a Linux process' address space is ultimately controlled by the L4Re microkernel, it can exist on all cores of the system, not just those known to L4Linux. Thus, by moving a decoupled thread to a core on which L4Linux does not run, we drastically reduce any disturbance and noise that Linux can cause due to in-kernel housekeeping tasks or other Linux processes. Figure 1 shows an architectural overview of this decoupling architecture.

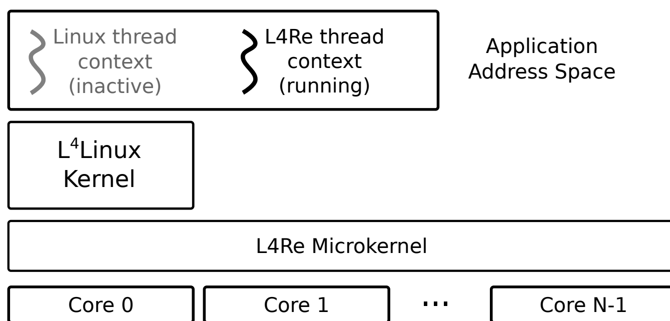

Fig. 1. Decoupling architecture, with Linux thread context inactive in the L4Linux kernel, while application code is executed as an L4Re thread on a dedicated core under control of the L4Re microkernel.

Decoupled threads can still do Linux system calls. The L4Re microkernel will forward these "non-L4Re" system calls to L4Linux, which temporarily migrates the decoupled thread's execution state back to the original Linux thread context, handles the call, and then resumes execution in the decoupled thread. In this work we extend the decoupling mechanism to also allow native L4Re system calls, such that a decoupled thread can implement an interrupt service routine (ISR) that will be invoked directly by the L4Re microkernel. The expected benefit is that external events signaled by the interrupt can be handled with consistently low latency.

B. Kernel Objects, Capabilities, and I/O Resources

L4Re is a capability-based system, where the microkernel exposes entities such as address spaces, threads, or facilities for inter-process communication (IPC) through an object-like interface. An L4Re application can make use of a kernel object only if it possesses a *capability* to name it. This is the case, if the kernel-protected capability table that is associated with each address space contains a pointer to the kernel object in question. The application can then invoke the object by doing a system call that specifies the corresponding index into the capability table. Additional parameters passed to the invocation system call indicate the type of operation to perform on the object.

The L4Re microkernel also provides abstractions for hardware devices. Specifically, it exposes hardware interrupts through an `Irq` object. Thus, any L4Re application that possesses a capability to an `Irq` object can implement an interrupt service routine (ISR) in user-space. To do so, it needs to `attach` a handler thread to the `Irq`, which can then `wait` on it for incoming interrupts. In all but the simplest cases, the application also must be able to talk to the hardware device that generated the interrupt. This access is typically provided by mapping the I/O registers of the device into the application's address space.

C. Passing Capabilities and I/O Resources

Since a decoupled thread is in fact a native L4Re thread that is independent of the L4Linux scheduler, we can implement an ISR in a Linux program using the same L4Re microkernel primitives: The decoupled thread attaches to and then waits on an `Irq` object directly, thereby achieving much lower latency and more predictable response times than Linux can guarantee. The problem to solve is how the decoupled thread of the Linux program gets access to the resources it needs for that: the `Irq` capability and the I/O memory pages of the device. Both resources are already held by the L4Linux kernel, which is also responsible for managing the virtual address space of all Linux user processes. In L4Re, two cooperating programs can voluntarily transfer object capabilities and memory mappings, if they already have established an IPC channel through an `Ipc_gate`. Unfortunately, a Linux user program is not aware of running on top of L4Re and the L4Linux kernel. It can therefore receive neither the `Irq` capability nor the I/O memory mappings through this channel. However, as the creator of all Linux processes, L4Linux possesses a capability to the respective address-space objects (called `Task`). By invoking the `map` operation on a process's `Task` object, the paravirtualized Linux kernel can map the I/O memory pages of a device directly to the user program without its cooperation. The `map` operation also allows L4Linux to pass capabilities to Linux programs and is therefore a suitable mechanism to provide them the resources needed to implement L4Re-supported ISRs.

D. Making Linux Programs Aware of L4Re

In practice, though, the L4Linux kernel still needs to know which `Irq` and I/O resources to pass to which program (if any). This decision should be made by the system designer or the application developer who implements the ISR. A simple and ad-hoc solution is to have L4Linux `ioremap` the device register regions and then let the Linux user program request this I/O memory by `mmap`'ing those parts of `/dev/mem` that contain them. However, this approach is not desirable from a security point of view. A minimally invasive and – from a Linux application developer's perspective – idiomatic way to request I/O memory is to use the POSIX `mmap` system call on a file descriptor pointing to a device node. For example, an already existing Linux device driver could be extended to hand

out its I/O memory regions via `mmap`. During this operation, it could also map the `Irq` capability to the user program. New device drivers should be written with user-space I/O in mind from the start; investigating how our approach can be combined with Linux UIO framework [5] is subject to future work.

In our prototype, we use a rather simple device for evaluation: the periodic timer of the system. As the handling of the timer event itself is done by the microkernel, we rather only need to block in the microkernel using an IPC operation. Still we need to read out the counter values of the timer and thus need to have those accessible to user-level code. Luckily this is easy in our prototype, as the timer's counters can be read from user-level directly when configured this way (TSC on x86, counter values on ARM). On x86 we additionally need to know the compare value of the timer which we can read with a specifically crafted interface to read this particular MSR value. When waiting for device interrupts, the code is basically similar, however we use the IRQ capability to block on incoming interrupts.

III. Evaluation

The main objective of this work is to assess the potential of the *decoupling* approach for *low-latency* and *predictable* event handling in a Linux-based user-space program. We therefore focus our evaluation on the latency of waking a thread that has been waiting for an incoming hardware interrupt; further interaction with the device that caused the interrupt (e.g., to obtain sensor readings) is not considered in this paper.

A. Hardware Setup and Interrupt Source

We perform our experiments on two different platforms: a desktop system with an Intel CPU (quad-core Core i7-4770, clocked at 2,993 MHz) and an embedded system based on the ARM architecture (NXP Layerscape LS1021A-TWR board with dual-core Cortex-A7, clocked at 1,200 MHz). Both systems operate in 32-bit mode.

On the x86 system, we considered using the High Precision Event Timer (HPET), because it is an independent device and it is known when its interrupt will fire. However, it turned out that the HPET's interrupt latency is significantly higher than for the local APIC (approximately 5 μs vs 2 μs, respectively). Based on this insight, we decided not to implement an HPET driver for L4Re that supports blocking on an `Irq` object. Instead we chose to use the local APIC, which is already supported as the system timer in both Linux and the L4Re microkernel. The situation is similar on the ARM platform, where both systems use the generic timer of the CPU. Since the system timer's interrupt also triggers wakeup of application threads that requested to sleep until some timeout, we have a simple and portable way to measure interrupt latency across OS and hardware platforms: We just let the benchmark application sleep until an absolute timeout that can be compared against the time stamp counter of the CPU. A downside of using the system timer for our experiments is that the kernel performs some additional work before waking up the user-level thread. It needs to program the next timeout, and remove all threads with expired timeouts (just one in our case) from a queue. However, the overhead to perform this small amount of in-kernel work is negligible both in Linux and the L4Re microkernel.

B. Benchmark Configuration

We compare latencies for waking a user-level thread on bare-metal Linux and on L^{4}Linux, where the decoupled thread of the Linux program blocks in the underlying L4Re microkernel. On both hardware platforms, we run a version of L^{4}Linux that is based on Linux 4.10. All bare-metal Linux runs on the x86 system use this version, too. On ARM, we had to use a vendor-provided Linux kernel based on Linux 4.1 for reasons explained in Section III-E. All Linux kernels were configured with *high-resolution-timers* and *PREEMPT*.

We measure interrupt latency for a simulated real-time ISR. On native Linux, we use *cyclictest* from the *rt-tests* suite [6] to evaluate the timer latency. *Cyclictest* continuously measures the wakeup latency after periodic timeouts. It records measured latencies and summarizes them in a histogram. On L^{4}Linux with decoupled threads, we implemented a benchmark similar to *cyclictest*, which uses an L4Re mechanism to sleep until the timeout hits.

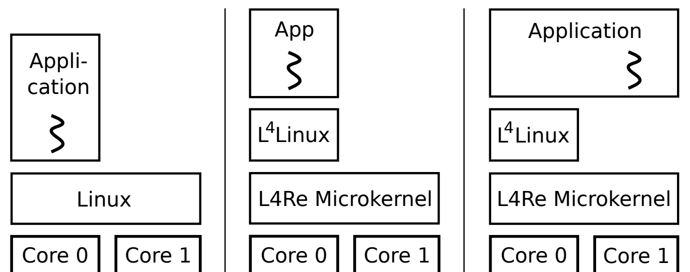

Fig. 2. Thread handling in our benchmark configurations: The *cyclictest* thread is either scheduled and woken by Linux running on bare-metal hardware (left), it is a decoupled thread that share a core with L^{4}Linux, or it is placed on a dedicated core that L^{4}Linux does cannot use (right).

All experiments are done both with and without a background load in Linux. As a load generator, we use *hackbench* from the *rt-tests* suite. *Hackbench* is a scheduler stress test, where process or thread pairs send data back and forth between them. Unless stated otherwise, we configured *hackbench* to exchange 10,000 messages of size 10,000 bytes through pipes (`hackbench -s 10000 -l 10000`). It thus creates a high system load, which ideally should not increase response time of the real-time work triggered by the interrupt. For L^{4}Linux, we also vary the placement of the decoupled thread that runs the benchmark. We measure latencies when it is placed on a dedicated core, as well as when it shares a core with the L^{4}Linux kernel. Note that in the latter case, the decoupled thread has a static priority that is higher than for the virtual CPU threads of L^{4}Linux. See Figure 2 for all possible placement and scheduling options (background load not shown for better readability). In total, there are 12 different configurations, which we evaluate in the following sections.

C. x86 – Bare-metal Linux

Figure 3 presents the results of the bare-metal Linux runs on the x86 machine with the Intel CPU. The x-axis shows the

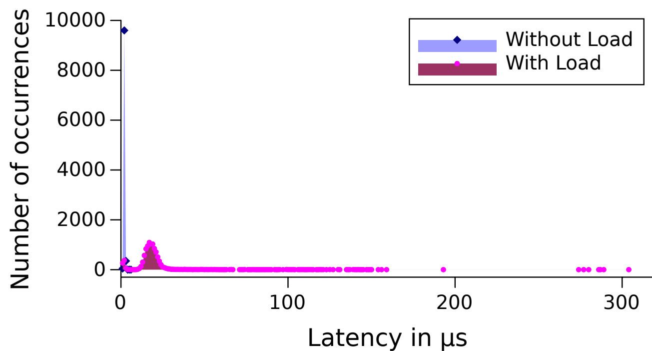

Fig. 3. Cyclictest results on bare-metal Linux with and without load generated by hackbench. The worst-case latencies are 6 μs without load and 304 μs with load.

latency and the y-axis indicates the number of occurrences of each latency. *Cyclictest* latencies without background load are shown in blue, those with *hackbench* running in parallel are red. Hardly visible in the diagram due to intentionally wide scaling of the x-axis, the maximum latency without background load is just about 6 μs, with most of the measurements clustered around 2 μs. With *hackbench* in the background, the maximum latency increases dramatically by more than two orders of magnitude to 304 μs. The majority of latency values is between 15 μs and 22 μs, which is about three times as high as on the unloaded system.

D. x86 – L⁴Linux with Decoupling

When we run the benchmark on L⁴Linux, with the decoupling mechanism we described in Section II-A, we achieve much lower latencies and significantly less variance. Figure 4 visualizes the results when the decoupled thread is placed on the dedicated core of the quad-core CPU. The measurements for running the decoupled thread on the same core as L⁴Linux are shown in Figure 5. In this case it is crucial that the decoupled thread runs at a higher priority than L⁴Linux under L4Re's fixed-priority scheduler. We changed the scale of the x-axis to just 3 μs for better readability.

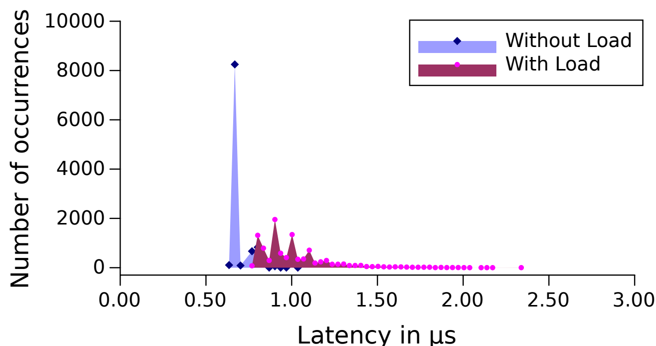

Fig. 4. Cyclictest results on x86 running L⁴Linux with and without load generated by hackbench. The decoupled thread runs on a different core. The worst-case latencies are 1.1 μs without load and 2.4 μs with load.

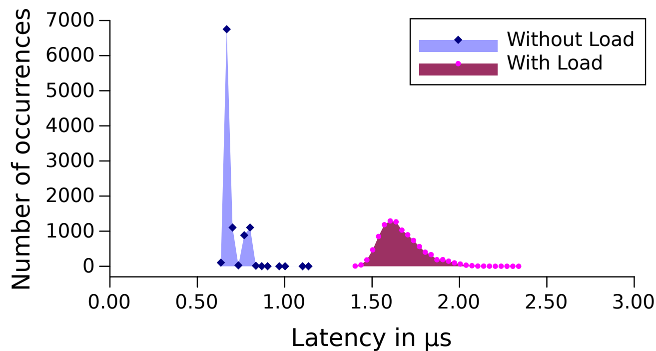

Fig. 5. Same setup as shown in Figure 4 however the decoupled thread runs on the same core as L⁴Linux. The worst-case latencies are 1.2 μs without load and 2.5 μs with load.

For both the dedicated-core and the shared-core configurations, we observe on an idle system a maximum timer latency of slightly more than 1 μs. The majority of observed latencies cluster around 0.7 μs, which is about a third of what we measure with bare-metal Linux. When loading the system with *hackbench* as described previously, the maximum timer latency increases to approximately 2.4 μs, irrespective of the placement of the decoupled benchmark thread. The majority of latencies are at around 1 μs on the dedicated core, and about 1.7 μs when the L4Re-aware *cyclictest* replacement (see Section III-B) shares a core with L⁴Linux.

The results demonstrate that our decoupling approach is highly effective for reducing average and tail latencies. They also indicate that the L4Re microkernel has a more efficient interrupt-to-wakeup path than Linux. As for the increased latencies under load, we suspect that they are the result of cache and TLB misses. The difference is more pronounced in the shared-core configuration. This could be attenuated in a dual-socket system, where no caches are shared [1]. However, this would also significantly increase the cost of the system.

E. ARM – Bare-Metal Linux

Since our decoupling mechanism also works for the ARM port of L⁴Linux, we repeated the experiments on that platform. We chose an NXP Layerscape 1021A-TWR system with a dual-core ARM Cortex-A7 CPU and attempted to build a vanilla Linux 4.10 kernel for it; Linux 4.10 is the version we used on x86 and also the one on which the latest L⁴Linux with decoupling support is based on. Unfortunately, we did not succeed in finding a kernel configuration where the generic ARM timer of the CPU could operate in high resolution mode; instead it only supported a resolution of 10 ms, which is prohibitively inaccurate for our experiments. We therefore installed the vendor-supplied kernel, which did not have this problem. This kernel from NXP is based on Linux version 4.1 and has the real-time patch-set applied applied to it ("*Linux-rt*"); our ARM build of L⁴Linux is still based on Linux 4.10. Using different kernel version is acceptable in our benchmark setup, because threads decoupled from L⁴Linux run on the L4Re microkernel, which is completely different anyway.

The next problem we encountered was that *cyclictest* reported incorrect results due to a bug: The load generated by *hackbench* caused response time jitter that was so high that an integer overflow occurred in *cyclictest's* measurement logic. We could prevent the bug from triggering by changing the message-size parameter of *hackbench* (`-s 100` instead of `-s 10000`). However, even with the reduced load, we can see from Figure 6 that bare-metal Linux suffers from extremely high interrupt latencies on this ARM platform. *Cyclictest* reports latencies of up to 147ms, which is why we have to use logarithmic scale for the x-axis.

F. ARM – L^{4}Linux with Decoupling

To evaluate L^{4}Linux with decoupled threads on the ARM platform, we reverted the *hackbench* parameters to those we used for the x86 runs. The results are shown in Figures 7 and 8 for dedicated-core and and shared-core placement, respectively. We observe results that are a vastly different from the bare-metal configuration on the NXP-provided Linux kernel. When the decoupled thread with our L4Re-aware version of *cyclictest* runs on its own core, we measure a maximum latency of 5.1 μs for the wakeup. When it shares a core with the L^{4}Linux kernel, the highest latency we observed is 31 μs. This increase over the dedicated-core configuration is relatively larger than on the Intel platform. We believe that this is due to shared-cache usage, especially of the L1 caches.

G. Summary of Evaluation Results

The evaluation of our decoupling mechanism with regard to interrupt latency shows that our approach of putting a fully independent scheduler in charge of time-critical ISRs is highly beneficial. By decoupling user threads from Linux's scheduling regime, we can significantly improve their response time to external events. While the difference on x86 is about two orders of magnitude, on the ARM platform, the difference can be even larger with over four orders of magnitude.

IV. Related Work

We are not the first who aim to combine the properties of an RTOS with the rich feature set of commodity, off-the-shelf general-purpose environments. There is related work in both the real-time community and in the context of high-performance computing. In principle, there are two approaches to improve the responsiveness of user programs. Either one enhances the operating system such that its interrupt latency improves, or latency-sensitive programs are run next to the general-purpose operating system, with only a loose coupling for data exchange between the two worlds.

An example for the first approach is the real-time Linux project. Most of their enhancements have been merged into the mainline Linux kernel [7]. Other work aims to separate latency-constraint programs from the rest of the general-purpose operating system, for example, Xenomai [8] and RTAI [9]. Both follow a co-location approach, where they hook into the low-level interrupt handling to branch execution away to low-latency handling routines. There has also been work to evaluate the use of additional protection through address spaces for real-time work [10].

More recent work focuses on using virtualization techniques to improve isolation. For example, Xen-RT [11] added real-time schedulers to the Xen hypervisor [12]. The ability of hypervisors to provide temporal as well as spatial isolation is also used to separate execution of the real-time and non-real-time workloads. Examples are Jailhouse [13], which uses virtualization technology to assign different cores in the system to different operating systems. Another example is Xtratum [14] and there are also commercial systems such as Greenhill's Integrity. These approaches cannot provide developers with a tightly integrated system architecture as we do with L^{4}Linux and decoupling.

The high-performance community also aims at running their applications with predictable performance such that, for example, bulk-synchronous programs do have to wait unnecessarily on global barriers [15]. This becomes increasingly important as the core-count increases on the path to Exascale systems. IBM's BlueGene systems are a prime example of noise-free execution. All nodes of the system are controlled by a compute-node kernel (CNK) that only runs one application. They suffer from no interference as there are no other activities on the nodes. However, BlueGene is a proprietary system that is no longer available anymore. Other projects such as McKernel/IHK [16] and mOS [17] aim to build systems with similar properties, but on standard hardware with x86 and ARM processors. They inject so-called light-weight kernels into Linux that take control of most of the cores and memory. We have also been pursuing this approach in the HPC context [1] with L4re and L^{4}Linux.

V. Conclusion

In this work, we have combined the microkernel-based L4Re system and L^{4}Linux, a paravirtualized variant of the Linux kernel, into an operating system that enables low-latency interrupt handling from Linux user-space applications. The key property of our system is that it decouples timing-sensitive threads of a process from the Linux scheduler by executing their code in a separate thread directly on the L4Re microkernel. Application developers can use all features and APIs of Linux for non-real-time work as needed. With the extensions to decoupling that we described in this work, they can also write interrupt service routines that register directly with the microkernel in order to respond to external events with low latency. The majority of the wakeup times we measured for decoupled threads on L^{4}Linux are at least 3 times shorter than on bare- metal Linux. Maximum latencies are reduced by two orders of magnitude and more when there is heavy system load caused by other processes running in parallel.

Acknowledgments

The research and work presented in this paper is supported by the German priority program 1648 "Software for Exascale Computing" via the research project FFMK [18] and project "microHPC" funded by ESF and the Free State of Saxony. We

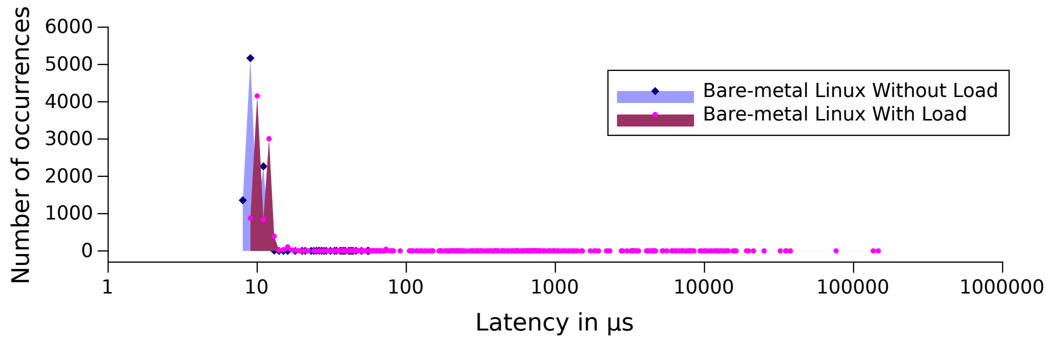

Fig. 6. *Cyclictest* results on bare-metal Linux on ARM with and without load generated by *hackbench*. The worst-case latencies are 56 μs without load and 146,500 μs with load. Note that we use a logarithmic scale for the x-axis in this figure.

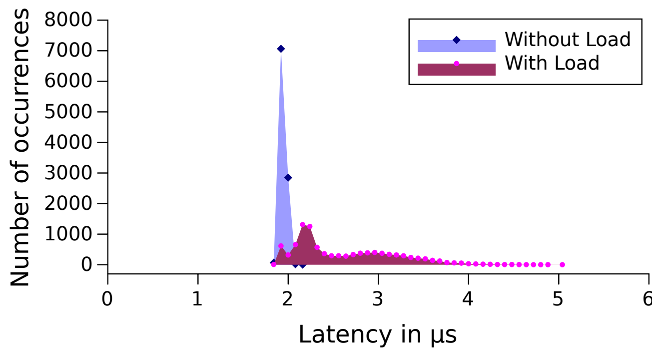

Fig. 7. *Cyclictest* with L[4]Linux results with and without load generated by *hackbench*. The decoupled thread runs on a different core. The worst-case latencies are 2.2 μs without load and 5.1 μs with load.

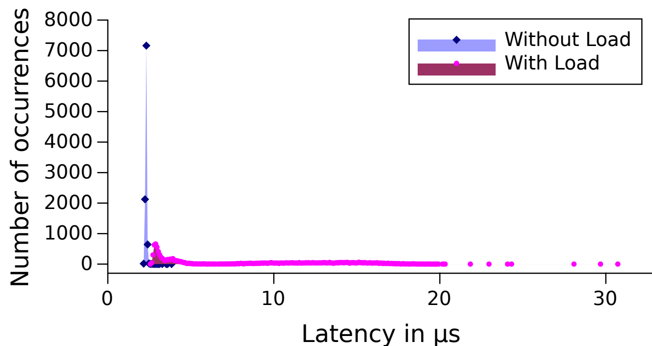

Fig. 8. Same setup as shown in Figure 4, however, the decoupled thread runs on the same core as L[4]Linux. The worst-case latencies are 3.9 μs without load and 31 μs with load.

also thank the cluster of excellence "Center for Advancing Electronics Dresden" (*cfaed*) [19].

References

[1] Adam Lackorzynski, Carsten Weinhold, and Hermann Härtig. Decoupled: Low-Effort Noise-Free Execution on Commodity System. In *Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers*, ROSS '16, New York, NY, USA, 2016. ACM.

[2] Adam Lackorzynski, Carsten Weinhold, and Hermann Härtig. Combining Predictable Execution with Full-Featured Commodity Systems. In *Proceedings of OSPERT2016, the 12th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, OSPERT 2016, pages 31–36, 2016.

[3] Alexander Warg and Adam Lackorzynski. The Fiasco.OC Kernel and the L4 Runtime Environment (L4Re). avail. at https://l4re.org/.

[4] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The performance of μ-kernel-based systems. In *Proceedings of the 16th ACM Symposium on Operating System Principles (SOSP)*, pages 66–77, Saint-Malo, France, October 1997.

[5] UIO: user-space drivers. https://lwn.net/Articles/232575/. Accessed 14 Apr 2017.

[6] RT-Tests, v1.0. https://www.kernel.org/pub/linux/utils/rt-tests/.

[7] Real-Time Linux Project. Real-Time Linux Wiki. https://rt.wiki.kernel.org.

[8] Xenomai Project. https://xenomai.org.

[9] RTAI – Real Time Application Interface. https://www.rtai.org/.

[10] 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.

[11] Sisu Xi, Justin Wilson, Chenyang Lu, and Christopher Gill. RT-Xen: Towards Real-Time Hypervisor Scheduling in Xen. In *Proceedings of the ninth ACM international conference on Embedded software*, EMSOFT '11, pages 39–48. ACM, 2011.

[12] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 164–177. ACM, 2003.

[13] Jan Kiszka and Team. Jailhouse: Linux-based partitioning hypervisor. http://www.jailhouse-project.org/.

[14] A. Crespo, I. Ripoll, and M. Masmano. Partitioned Embedded Architecture Based on Hypervisor: The XtratuM Approach. In *Dependable Computing Conference (EDCC), 2010 European*, pages 67–72, April 2010.

[15] T. Hoefler, T. Schneider, and A. Lumsdaine. Characterizing the Influence of System Noise on Large-Scale Applications by Simulation. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'10)*, Nov. 2010.

[16] T. Shimosawa, B. Gerofi, M. Takagi, G. Nakamura, T. Shirasawa, Y. Saeki, M. Shimizu, A. Hori, and Y. Ishikawa. Interface for heterogeneous kernels: A framework to enable hybrid os designs targeting high performance computing on manycore architectures. In *High Performance Computing (HiPC), 2014 21st International Conference on*, pages 1–10, Dec 2014.

[17] R.W. Wisniewski, T. Inglett, P. Keppel, R. Murty, and R. Riesen. mOS: An Architecture for Extreme-scale Operating Systems. In *Proc. ROSS '14*, pages 2:1–2:8. ACM, 2014.

[18] FFMK Website. https://ffmk.tudos.org. Accessed 14 Apr 2017.

[19] cfaed Website. https://www.cfaed.tu-dresden.de/. Accessed 14 Apr 2017.