Combining Predictable Execution with Full-Featured Commodity Systems

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

***Abstract*—Predictable execution of programs is required to satisfy real-time constraints in many use cases, including automation and controlling tasks. Unfortunately, background activities of the operating system may influence execution of such workloads in unpredictable ways, as do other applications running on the system concurrently. Separating time-critical workloads from unrelated activities is thus a common approach to ensure predictable execution.**

Different strategies are used to achieve this separation. On multi-core systems, developers typically assign work loads to dedicated cores, which then run completely separate software stacks. They often do not provide fault isolation nor security guarantees. Another approach is to co-locate a commodity operating system with a real-time executive, which hooks into the interrupt subsystem of the standard kernel to run real-time code at the highest priority in the system. There are also ongoing activities to modify commodity kernels such as Linux to enable more predictable execution. This pairing of the rich and versatile feature set of Linux with a real-time execution is very compelling, but it requires significant developer effort to ensure that the huge monolithic code base does not violate real-time requirements.

In this paper, we present a mechanism that combines predictable execution and all of Linux' functionality with much less effort. It allows unmodified programs to be started on top of a virtualized Linux kernel and then "pull them out" of the virtual machine to let them run undisturbed on the microkernel that hosts Linux. Whenever the program performs a system call or catches an exception, those are forwarded to Linux transparently. Experimental results show that execution-time variation is reduced by two orders of magnitude.

I. Introduction

Predictable execution, often also called real-time execution, is a required functionality for a broad range of uses cases. Common real-time operating systems (RTOS) are simple, and thus predictable, but lack features commonly offered by a full-featured commodity operating system (OS), such as Linux. Unfortunately, full-featured OSes typically cannot ensure predictable execution. Still, there is ongoing work on full-featured operating systems aiming to run real-time workloads. An example are the real-time extensions for Linux (Linux-RT [1]) which are merged step-by-step into mainline Linux. However, it is a constant challenge to keep the huge code base of the kernel preemptible, while hundreds of developers add new features all the time or rework entire subsystems; preemptibility and real-time capabilities are typically not the main concerns for most of these developers.

Another common approach is to use multi-core systems and to run a commodity OS and an RTOS on the same system. This provides good temporal isolation but lacks spatial isolation. Both OSes exist side by side and interaction between them is usually coarse grained, for example, through mailbox systems. As each of the two OSes runs with full system privileges, such setups do not offer effective fault containment as required for security and safety-critical usage scenarios. A hypervisor (or microkernel) that virtualizes the platform can contain faults within each of the two software stacks by deprivileging their OSes. Adding the virtualization layer may cause a slight performance degradation, but run-time overheads are not prohibitive in most cases.

The contribution of this work is a mechanism for combining the flexibility and feature set of a full-featured commodity OS with the real-time characteristics of an RTOS. By means of virtualization, we enable threads of a program to detach from the unpredictable commodity OS and run in the real-time capable environment. Whenever such a detached thread needs to call services of the feature-rich OS (e.g., system calls), those requests will be forwarded to the commodity OS transparently. In our prototype, we use the L4Re system, a microkernel-based operating system framework [2] for building customized systems. The L4Re microkernel serves both as a hypervisor and in the role of an RTOS to runs detached threads. We use L^{4}Linux, a paravirtualized variant of the Linux kernel that comes with L4Re. It has been adapted to run on the L4Re system as a deprivileged user-space application.

We are not the first to combine a real-time executive and a feature-rich commodity operating. However, our approach represents a new way of building this kind of split OS platform. We reach this goal without "reinventing the wheel" by enhancing existing microkernel technology with a simple mechanism. We believe that our approach is low-effort, maintainable, and it provides continuous access to the latest releases of the feature-rich OS. This paper builds on our previous work in the context of high-performance computing [3].

In the remainder of the paper, we will describe our system in more detail (Section II) and then discuss the detaching mechanisms we added in Section III. We evaluate our work in Section IV before we conclude.

II. Virtualization System

We build an OS for predictable execution based on the L4Re microkernel system, which hosts a virtualized Linux kernel called L^{4}Linux. To get an understanding of L4Re's capabilities

and the detaching mechanism described in Section III, we will now introduce the L4Re system architecture.

A. L4Re Microkernel and User Land

The L4Re microkernel is a small and highly portable kernel. It is the only component of the system running in the most privileged mode of the CPU. Its main task is to provide isolation among the programs it runs, in both the spatial and temporal domains. To do so, the kernel needs to provide mechanisms for all security-relevant operations, such as building up virtual memory for programs and scheduling them. To support virtual machines (VMs), the kernel also provides abstractions for those virtualization-related CPU instructions that can only be executed in the most privileged processor mode. Thus it also takes the role of a hypervisor. Functionality that is not required to enforce isolation of applications and virtual machines is built on top of the kernel in user-level components.

The L4Re system is a component-based operating system framework that provides a set of components on top of the microkernel. It can be tailored to the needs of applications. The set of components includes services and libraries for memory management, application loading, virtual machines, device drivers, and more. As the L4Re system provides functionality as components, applications need to rely only on those services they use, thereby minimizing their Trusted Computing Base (TCB). The TCB is also application-specific, as different application to may depend on different services. This is in contrast to monolithic designs, where, for example, a malfunction in a file-system leads to a kernel panic that concerns every application, including those that do not use that file-system at all.

The L4Re microkernel supports hardware-assisted virtualization such as Intel's VT and ARM's VE, as well as a pure software approach to hosting VMs. The latter only relies on the memory management unit, which also provides address spaces for isolating ordinary applications. The kernel provides interfaces specifically designed so that OS developers can port their kernels to L4Re with little effort. This paravirtualization support includes support for mapping guest processes and threads to the L4 tasks and L4 vCPUs that the microkernel provides: An L4 task encapsulates address spaces both for memory and kernel objects such as capabilities; a vCPU is a thread and thus a unit of execution, however, enriched with features beneficial for virtualization.

Besides providing address spaces through L4 tasks and execution through L4 thread and vCPUs, the microkernel provides a few more mechanisms. Interrupts are abstracted using Irq objects. Irqs are used for both physical device interrupts as well as for software-triggered interrupts. The microkernel also schedules the threads on the system and offers multiple, compile-time selectable scheduling algorithms.

The whole L4Re system is built around an object capability model. Any operation on an object outside the current L4 task must be invoked through a capability; this includes the objects that provide inter-process communication (IPC) and Irqs. Thus one can state that IPC is used to invoke capabilities. L4Re uses the same invocation method for all objects in the system, whether they are implemented in the microkernel itself or provided by user-level applications.

The L4Re microkernel always runs on all cores of the system and address spaces span all cores; threads can be migrated. The microkernel itself will never migrate a thread between cores on its own; however, user-level applications can request migrations.

B. L4Linux

In our work, we use L^{4}Linux, a paravirtualized variant of the Linux kernel that has been adapted to run on the L4Re system. L^{4}Linux is binary compatible to normal Linux and runs nearly any Linux binary [4]. We chose L^{4}Linux instead of a fully-virtualized Linux because L^{4}Linux is integrated more tightly with the underlying L4Re system and thus allows our approach to be implemented much more easily. In the following we will describe L^{4}Linux in sufficient detail to understand our approach to detaching thread execution.

The L^{4}Linux kernel runs in an L4 task and each Linux user process is assigned its very own L4 task, too. Thus, the L^{4}Linux kernel is protected from misbehaving applications like native Linux is, where user processes run in another privilege level. There are no dedicated L4 threads for the user processes as those are provided by the vCPU. A vCPU is like a thread, but provides additional functionality useful for virtualization. Such features include an asynchronous execution model with a virtual interrupt flag and also the ability of a vCPU to migrate between address spaces which is used to implement user processes. Thus, from the host's point of view, an L^{4}Linux VM comprises multiple vCPUs (one for each virtual CPU in the guest) and L4 tasks that provide address spaces for the L^{4}Linux kernel and each user process.

During operation, a vCPU executes both guest kernel code and the code of the user processes. When the L^{4}Linux kernel performs a *return-to-user* operation, the vCPU state is loaded with the register state of the user process as well as the L4 task of the user process. The vCPU will then continue execution in that task. For any exception that occurs during execution (e.g., system call invocations or page faults), the vCPU migrates back to the guest kernel task and resumes execution at a predefined entry vector, where the exception is analyzed and handled appropriately. Interrupts are handled similarly: After having bound a vCPU to an interrupt object, firing the interrupt will halt current execution and transfer the vCPU to the entry point of the L^{4}Linux kernel where the interrupt will be processed.

Memory for user processes is exclusively managed by the Linux kernel. To populate the address spaces of user processes, L^{4}Linux maps memory from the Linux kernel task into the respective L4 tasks using L4 system calls to *map* and *unmap* memory pages. When resolving page faults for user processes, L^{4}Linux traverses the page tables that Linux builds up internally to look up guest-kernel to user address translations. Note that those shadow page tables are not used by the CPU. Only the L4 microkernel manages the hardware page tables; the only

way to establish mappings in Linux user processes (or any other L4 task) is to use the microkernel's map functionality.

III. Detaching Work

Now we want to pursue how we can separate a thread of a Linux user program so that it can run undisturbed from the rest of the L^{4}Linux system. As described in the previous section, L^{4}Linux does not use separate L4 threads for user processes, but it multiplexes user threads onto a single vCPU. However, to isolate execution of a user thread from the unpredictable L^{4}Linux, we must create a dedicated L4 thread that is not managed by the Linux scheduler. This detached thread will run Linux user code, be scheduled by the L4Re microkernel independently from L^{4}Linux's scheduler. As a separate L4 thread, we can also move it to a different core, preferably one that does not share caches with L^{4}Linux. A schematic view of our architecture is depicted in Figure 1.

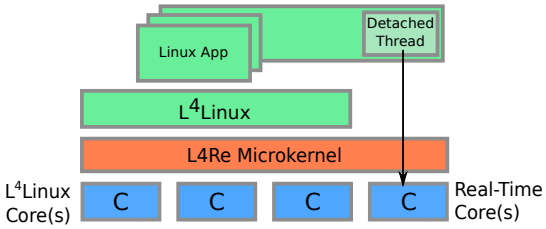

Fig. 1. A thread detached from L^{4}Linux running on a separate core.

To implement the creation of separate threads we can leverage infrastructure developed in previous versions of L^{4}Linux: the thread-based execution model [5], in which Linux threads are mapped one-to-one to L4 threads. This approach to threading in L^{4}Linux predates the superior vCPU execution model, but it is still being maintained. We build upon this older implementation to add creation of separate L4 threads to the vCPU model that is now used. Detached processes start as normal processes, for which a new L4 host thread is created and placed in the L4 task of the user process. Then, instead of resuming execution through the vCPU, the execution is resumed to the L4 thread by using L4 exception IPC. Exception IPC is a special type of IPC carrying the thread's register state and that is used to transfer the exception state between the causing thread and a handler thread, which is the L^{4}Linux kernel.

After launching the thread, L^{4}Linux puts the kernel-part of the user thread into *uninterruptible* state and calls `schedule()` so that another context is chosen. While a context is in state *uninterruptible* it is not chosen to be dispatched by the Linux scheduler. Thus, in L^{4}Linux's view, the context is blocked, however, it is running outside and independent of the virtual machine provided by L^{4}Linux.

While the detached program is running, it will eventually cause an exception, such as triggered by issuing a system call, or causing a page fault. In both cases the thread's state will be transferred to the L^{4}Linux kernel using L4 exception IPC. However, the context that will be active at that time in L^{4}Linux's kernel will not be the one of the detached thread as this one is in *uninterruptible* state. Thus the L^{4}Linux kernel will just save the transmitted state in the thread's corresponding kernel context and bring the thread out of the *uninterruptible* state via a *wakeup* operation. When L^{4}Linux's scheduler has chosen the thread again, the request will be handled. When done, execution is resumed by replying to the incoming exception IPC and setting the thread to *uninterruptible* again.

By using exception IPC, any request made by the detached user-level thread is transparently forwarded to the L^{4}Linux kernel. One may also describe that in a way that the user thread is being reattached while executing requests to the L^{4}Linux kernel.

A. L4 Interactions

When a thread is running detached, it is not restrained to run code only but it can also interact with other L4 components or the microkernel. For example, a control loop can be implemented using absolute timeouts of the L4 system or the thread can wait on other messages or interrupts, including device interrupts. Waiting directly for device interrupts in detached threads might be beneficial to avoid interaction with the Linux kernel and thus to achieve lower interrupt response latency.

For doing L4 IPC, the L^{4}Linux kernel needs to provide the thread information where its User Thread Control Block (UTCB) is located. The UTCB is a kernel provided memory area that is used to exchange data with the kernel and communication partners. The way of retrieving the address of the UTCB, as used in native L4 programs, does not work within an L^{4}Linux environment as the segment, as used on x86, registers are managed by L^{4}Linux and might be used by the libc. Thus an alternative approach must be provided, for example, by a specifically provided extra system call. As the UTCB address is fixed for a thread, it can be cached. When just using one thread in the application, the UTCB address is always the same and a well-known constant can be used as a shortcut.

For the program to communicate with other L4 services, the L^{4}Linux kernel needs to map a base set of capabilities into the task of the user process. L^{4}Linux must have been setup accordingly to receive those capabilities itself beforehand. Further the user program must be able to get information on where which capabilities have been mapped. In L4Re, this information is provided through the environment when the application is started. As application starting is done by the L^{4}Linux kernel, an alternative approach is required, such as a dedicated system call or a `sysfs` interface.

B. Implementation Details

In the following we will shortly describe interesting aspects of the implementation.

1) Signal Handling: As threads run detached from the L^{4}Linux kernel they are blocked by being in the *uninterruptible* state. This affects signal delivery, such as `SIGKILL`, to take effect, as the signal will just be processed when the thread is in the kernel or enters it. When the detached thread never enters

the L^{4}Linux kernel again ("attaches" again), any posted signal will have no effect. For that reason, we added a mechanism that periodically scans detached threads for pending signals, and if it finds any, the detached thread is forced to enter the L^{4}Linux kernel to have the signal processed eventually.

2) Memory: As already described, all memory of a detached application is managed by L^{4}Linux. Linux may do page replacement on the pages given to the application which in turn affect the undisturbed execution. Thus it is advised that applications instruct the L^{4}Linux kernel to avoid page replacement by means of `mlock` and `mlockall` system calls. Generally, using large pages to reduce TLB pressure is also recommended. L^{4}Linux and the L4Re microkernel support large pages.

With the possibility of a detached thread to call out to other L4 services, it could also acquire memory pages. This is possible, given the application is provided with appropriate service capabilities, however, care must be taken as the address space is managed by the L^{4}Linux kernel and Linux is unaware of other mappings in the address space. Reservations of regions of the address space can be done via `mmap`, and given no page faults are generated in those regions, the pages can be used. Using memory from elsewhere is useful, for example, to use shared memory with other L4Re applications.

3) Floating Point Unit: vCPUs also multiplex the state of the Floating Point Unit (FPU) on behalf of the virtualized OS kernel. FPU handling for vCPUs is built in a way that it matches the hardware's behavior and thus aligns well with how operating systems handle the FPU natively. Although a vCPU can handle multiple FPU states, only one at a time can be active per vCPU. However, with detached threads, there are additional L4 threads, and thus active FPU states, that need to be handled.

The FPU-state multiplexing is built in a way that an FPU state travels between different threads, that is, the set of L4 threads building up a virtual CPU just use one single FPU state. Additionally, the state is handled lazily so that an FPU state transfer must only be done when the FPU is actually used. Thus, when a detached L4 thread enters the L^{4}Linux kernel, its FPU state cannot be transferred automatically to the L^{4}Linux kernel because another FPU state might be active there. To resolve this situation, we extended the L4Re microkernel with an operation for explicitly retrieving a thread's FPU state. This way L^{4}Linux can save the FPU state of a thread to L^{4}Linux kernel's internal FPU state for other Linux activities to access it. An operation for setting the FPU state of an L4 thread is not required because the FPU state is transferred with the exception IPC upon the resume operation. This is possible because resumption is done out of the thread's context, contrary to the incoming operation, that is done on a different context.

4) Sysfs Interface: As already described, we use a `sysfs`-based interface to control detaching of threads. Contrary to using an extra system call, this gives use the possibility to easily use it in wrapper scripts without requiring to modify the application itself. Noteworthy characteristics is that the detached state is retained through the `execve` system call, allowing to build wrapper scripts that detach an application:

```
#! /bin/sh
SYSFS_PATH=/sys/kernel/l4/detach
echo $$ > $SYSFS_PATH/detach
echo $HOST_CORE_ID > $SYSFS_PATH/$$/cpu
exec "$@"
```

As seen, specifying the target host CPU of the detached thread is also possible via the `sysfs` interface. The `sysfs` interface will only detach the first thread of an application, thus multi-threaded programs will need to take care of detached threads themselves.

IV. Evaluation

In the following we will evaluate our detaching mechanism regarding undisturbed execution. First, we use the FWQ benchmark, which is famous in the high performance computing (HPC) area for measuring OS noise. Then we will implement a control loop and monitor results for timing deviations. With both experiments we will generate load in the L^{4}Linux VM.

For all benchmarks, we use the same x86 system, running an Intel® Core™ i7-4770 quad-core CPU clocked at nominally 3.4GHz, reported with 2993MHz.

A. FWQ Benchmark

First, we run the fixed-work quantum (FWQ) benchmark [6]. The benchmark measures a fixed amount of work multiple times. Ideally the time it takes to run the work loop is the same for all runs, however, due to preemptions and other activities in the OS and the hardware, the measured times fluctuate. Thus the degree of deviation shows the impact of those other activities. The benchmark executes the work 10,000 times.

Figure 2 shows a run of FWQ on native Linux-4.6 built with preemption enabled (`CONFIG_PREEMPT=y`) and run with `chrt -f 10` while I/O intensive work is running as well, comprising network and disk load.

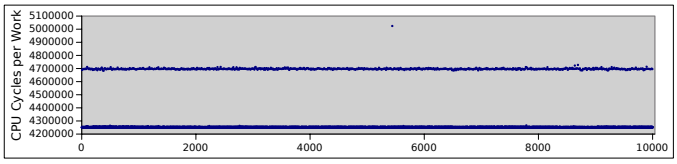

Fig. 2. FWQ results for Linux-4.6 PREEMPT with I/O load in Linux.

We see, although the FWQ benchmark is running as a real-time program and the Linux kernel uses its full preemption mode, deviation goes up to about 18%.

When running the same FWQ benchmark in L^{4}Linux using our presented mechanism, we measure results as seen in Figure 3. The maximum deviation is 1152 CPU cycles, or 0.027%.

When we run a Linux kernel compile instead of I/O load in L^{4}Linux, we see a pattern as in Figure 4 that has larger deviations: 6500 cycles, or 0.15%. When the L^{4}Linux is idle,

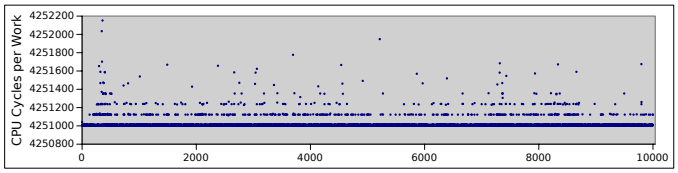

Fig. 3. FWQ results for detached mode with a I/O load in L^{4}Linux-4.6.

we see a behavior as seen in Figure 5 with just 21 cycles difference.

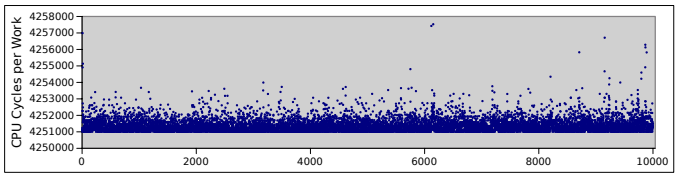

Fig. 4. FWQ results for detached mode with a build load in L^{4}Linux-4.6.

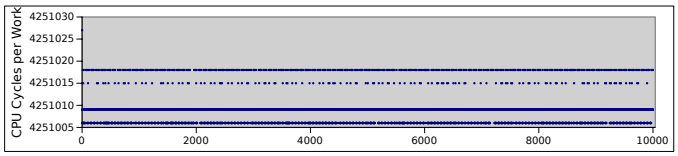

Fig. 5. FWQ results for detached mode with an idle L^{4}Linux-4.6.

Although the FWQ benchmark is so small that it is running out of L1 cache, effects can be seen in the results. Our speculation is that due to the inclusiveness of the caches in Intel's multi-level cache architecture, cache content can be evicted due to aliasing. However, whether this explains the different levels in Figure 3 is unclear and requires further investigations that are out of scope for this paper.

In summary, the results show for the FWQ benchmark that our detaching mechanism significantly improves the execution predictability of programs. It effectively isolates activities of the Linux kernel and unrelated background load from the detached real-time program, such that execution-time jitter is reduced by more than two orders of magnitudes.

B. Host-driven Control Loop

In our second experiment, we emulate a control loop that blocks repeatedly until an absolute time in the future to execute some task. In each iteration of the loop, we increment the programmed wake-up time by 1,000μs (`delta = 1,000`μs) as illustrated in the following code:

```
next = now() + delta;
while (1) {
  wait_for_time(next);
  /* do work */
  next += delta;
}
```

While the loop is running, we capture the time-stamp counter (TSC) and plot the delta of each consecutive loop iteration. Ideally, the measured delta between TSC-read operations should be constant, meaning that the `wait_for_time` call unblocks at precisely the specified time. The target for the delta is 2,993,000 cycles, as determined by CPU clock speed of 2,993MHz. We run the loop for 10,000 iterations so that the benchmark runs for 10 seconds.

On Linux, we implement the blocking using `clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME, ...)`. We see results as depicted in Figure 6 for I/O load and in Figure 7 for a build load. The way to generate the load has been the same as in the previous experiment. All Linux programs are pinned to a core and run with real-time priority (using `chrt -f 10`).

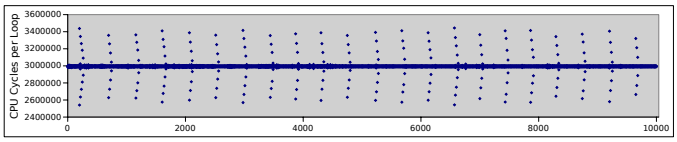

Fig. 6. Control loop results on native Linux with I/O load.

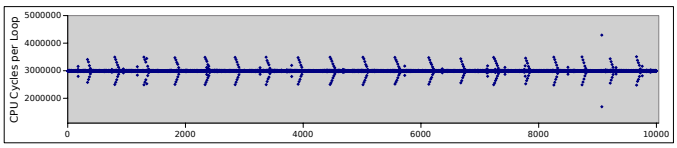

Fig. 7. Control loop results on native Linux with build load.

The two graphs show an interesting arrow-style pattern. With about 20 of such outlier events, one each half second, we suspect an internal activity in the Linux kernel that induces this result. We see a deviation from the target of about 500,000 CPU cycles in each direction, translating to about 167μs. The results for the I/O-load experiment look similar to the build-load case, however, there is an even larger outlier with about 1,300,000 cycles deviation (430μs).

With L^{4}Linux, using our detaching mechanism, the control loop uses L4 system calls to block until a specified point in time (absolute timeout). Thus, the blocking and unblocking is directly done by the microkernel and does not use or depend on Linux. We use the same background load as before; the results are shown in Figures 8 and 9. Note the change of range in the y-axis.

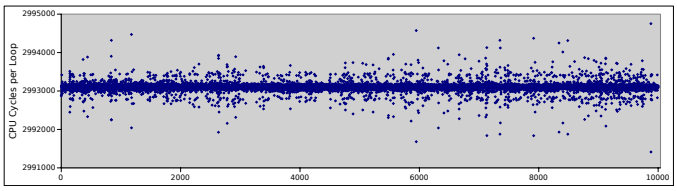

Fig. 8. Control loop results on L^{4}Linux with I/O load.

The major difference between Linux and L^{4}Linux is the significantly reduced deviation. With I/O load, we observe that the biggest outlier is about 1700 cycles away from the target

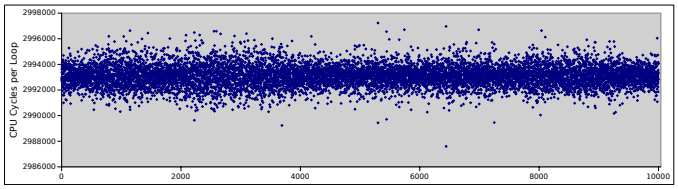

Fig. 9. Control loop results on L^{4}Linux with build load.

while the biggest outlier of the build load is about 4700 cycles away, translating to 600ns and 1.6μs deviation. This is a 2-fold improvement over the Linux results.

V. Related Work

There is plenty of work regarding the combination of real-time and general purpose operating systems (GPOS), using virtualization or co-location approaches. There are also efforts for enhancing the real-time capabilities of Linux itself [1].

In co-location approaches, a real-time executive is added to the GPOS that hooks into low-level functions to execute real-time tasks. Examples are Xenomai [7] and RTAI [8].

Xen-RT [9] adds real-time support to the Xen Hypervisor [10] by adding real-time schedulers. Jailhouse [11] is a recent development that uses the Jailhouse hypervisor to partition Linux and an RTOS to different cores on a multi-core system. Other hypervisors for real-time are Xtratum [12] and SPUMONE [13], and there are also commercial offerings, such as Greenhill's Integrity.

Similar work is also done in the HPC community. Although the real-time and HPC communities are typically disjunctive, they strive for similar goals. The focus in HPC is to minimize the disturbance caused by other software, such as the OS, and hardware, that is experienced while executing HPC applications. Uninterrupted execution is required because HPC application communicate over many nodes where a delay on a single node also has influences on other nodes. Thus disturbance must be minimized [14]. Proposed solutions are similar to what is done in the real-time area: Multi-core systems are partitioned into "OS Cores" and "Compute Cores". The OS core(s) typically run Linux to provide functionality that applications running on the compute cores require, but that the jitter-free "light-weight kernel" (LWK) does not implement. Several implementations of this approach exist, such as mOS [15] and McKernel/IHK [16], as well as our own work [3].

VI. Conclusion and Future Work

Our experiments show that our detaching mechanism is capable of improving the predictability of execution by at least two orders of magnitude compared to using a standard Linux. As the real-time programs on our system are unmodified Linux programs, existing development environments and tool can be used. This allows for an efficient use of developer's time when implementing timing sensitive functionality.

Implementing this or a similar mechanism using hardware-assisted virtualization promises to use any available Linux version, giving a broader access to platforms. We also plan evaluation on other architectures than Intel x86.

Acknowledgment

The research and work presented in this paper is supported by the German priority program 1500 "Dependable Embedded Software" and the German priority program 1648 "Software for Exascale Computing" via the research project FFMK [17]. We also thank the cluster of excellence "Center for Advancing Electronics Dresden" (*cfaed*) [18].

References

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

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

[3] 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.

[4] Chia-Che Tsai, Bhushan Jain, Nafees Ahmed Abdul, and Donald E. Porter. A Study of Modern Linux API Usage and Compatibility: What to Support when You'Re Supporting. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 16:1–16:16, New York, NY, USA, 2016. ACM.

[5] 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.

[6] Lawrence Livermore National Laboratory. The FTQ/FWQ Benchmark.

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

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

[9] 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.

[10] 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.

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

[12] 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.

[13] Tatsuo Nakajima, Yuki Kinebuchi, Hiromasa Shimada, Alexandre Courbot, and Tsung-Han Lin. Temporal and spatial isolation in a virtualization layer for multi-core processor based information appliances. In *Proceedings of the 16th Asia and South Pacific Design Automation Conference*, ASPDAC '11, pages 645–652. IEEE Press, 2011.

[14] 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.

[15] 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.

[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] FFMK Website. https://ffmk.tudos.org. Accessed 17 Jun 2016.

[18] cfaed Website. https://www.cfaed.tu-dresden.de/. Accessed 17 Jun 2016.