Obtaining hard real-time performance and rich Linux features in a compounded real-time operating system by a partitioning hypervisor

Chung-Fan Yang
University of Tsukuba
Department of Computer Science
Tsukuba, Ibaraki, Japan
sonicyang@softlab.cs.tsukuba.ac.jp

Yasushi Shinjo
University of Tsukuba
Department of Computer Science
Tsukuba, Ibaraki, Japan
yas@cs.tsukuba.ac.jp

Abstract

In this study, we describe obtaining hard real-time performance and rich Linux features together in a compounded real-time operating system (cRTOS). This system creates two realms with a partitioning hypervisor: a normal realm of Linux and a hard real-time realm of a swift RTOS (sRTOS). A rich real-time process running in the real-time realm can use not only the hard real-time performance of the RTOS but also the rich features of Linux through remote system calls. Unlike existing approaches for real-time Linux including the PREEMPT_RT patch and using interrupt-dispatching layers, this approach requires no modifications to Linux.

We implemented the cRTOS by running Nuttx, a POSIX-compliant RTOS as an sRTOS and Jailhouse as the partitioning hypervisor. We ported base Nuttx to the x86-64 architecture and added support for multiple address spaces with MMU. This allows developers of rich real-time applications to use the same toolchains and executables with Linux, which reduces the cost and complexity of developing real-time applications.

We measured the timing accuracy and interrupt latency of the proposed cRTOS and other existing systems, the PREEMPT_RT patched Linux and Xenomai 3. The experimental results show that the proposed cRTOS could deliver a hard real-time performance with about 4 µs jitter and well bounded maximum latency, while the others could not. The experimental results also show that the proposed cRTOS with a real-time device yielded the best interrupt response in both latency and jitter. The RTOS could execute complex Linux executables with graphical user interfaces through the X window system.

CCS Concepts • **Computer systems organization** → **Real-time operating systems**; • **Software and its engineering** → *Virtual machines*; **Operating systems**; Embedded software; **Real-time systems software**;

Keywords Real-time operating system, Linux kernel, binary compatibility, virtualization, operating systems

ACM Reference Format:
Chung-Fan Yang and Yasushi Shinjo. 2020. Obtaining hard real-time performance and rich Linux features in a compounded real-time operating system by a partitioning hypervisor. In *16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '20), March 17, 2020, Lausanne, Switzerland*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3381052.3381323

VEE '20, March 17, 2020, Lausanne, Switzerland

ACM ISBN 978-1-4503-7554-2/20/03…$15.00
https://doi.org/10.1145/3381052.3381323

1 Introduction

Real-time operating systems (RTOSs) are typically used in embedded systems to support time critical applications [30]. With the evolution of embedded systems with the development of the Internet of things (IoT) and edge computing, RTOSs also need to evolve. Users and applications now are demanding rich features from RTOSs along with hard real-time performances. These include graphical user interfaces and complete TCP/IP networking, which are commonly available in general purpose operating systems (GPOSs).

Adding these rich features to an RTOS is difficult and unfavorable work because adding code often harms real-time performance and is often considered reinventing the wheel. In addition, the development of real-time applications is typically tedious, because the programming environments and toolchains for RTOSs are very different from those of widely adapted GPOSs. For example, the support for well-known programming environments, e.g. Portable Operating System Interface (POSIX) [51], libraries, e.g. GNU C library [14], and toolchains e.g. gcc [13], are not well adapted by many off-the-shelf RTOSs. This makes the development process of real-time applications inefficient and more expensive.

This inspires a lot of research challenges for converting GPOSs to fulfill hard real-time requirements and extending

RTOSs to be compatible with typical GPOSs. These challenges mainly aim to increase the preemptible regions in OS kernels. For example, a popular GPOS, Linux, is frequently modified as an RTOS. Linux has rich features and supports the standard POSIX interface. It has a large number of existing applications and libraries. It is also a familiar programming environment for developers. By translating Linux to an RTOS, these existing resources can reduce the development cost of real-time applications. Therefore, researchers and developers are working on real-time extensions of Linux. However, each of these approaches has limitations.

The first representative approach is to extend Linux. Early examples of real-time extensions to Linux are RT-Linux [60], Resource kernel for Linux [40], and Time-Sensitive Linux [19]. Currently, the most widely used extension is PREEMPT_RT [53]. This is a large set of patches that translate the non-preemptible parts of Linux into preemptible ones and add real-time capability to Linux. However, it is difficult to prove that this set of patches can achieve hard real-time performance. The existing studies only claim that the performance of the PREEMPT_RT patched Linux is statically bounded, but the permissiveness of this bound is unknown [46]. Moreover, this set of patches is maintained out of the upstream kernel source tree. In Linux 4.9, which contains about 25,000,000 lines of code, the set of patches contains about 14,000 lines of changes to the vanilla kernel. It is difficult to track down every single section of large and evolving Linux, rewrite it, and evaluate its real-time performance and preemptibility. This makes the effort of developing and maintaining this set of patches extremely high.

Another representative approach to real-time Linux is running a coscheduled RTOS with Linux and hijacking the interrupt requests (IRQ) from Linux by using interrupt-dispatching layers including microkernels [3, 16, 18, 21–24, 32, 59]. These methods achieve hard real-time performance but limit access to rich Linux features. For example, developers have to split their applications into real-time processes running in the RTOS and non-real-time processes running in Linux. Because real-time processes cannot use features of Linux directly, they must use a special inter-process communication mechanism with the non-real-time processes. Developers must use special toolchains for building applications of real-time processes. It is not trivial to port the evolving Linux kernel to an interrupt-dispatching layer. Thus, they fall in the same problems as typical RTOSs, which is tedious for development.

To address these limitations in usability and maintainability, we propose a new approach to a compounded real-time operating system (cRTOS). This system runs both a rich GPOS and a swift RTOS (sRTOS) in it with a hypervisor and creates a normal realm and a real-time realm. The GPOS requires no modifications and patching. Real-time applications in the real-time realm can not only gain hard real-time performance by the RTOS but can also use the rich features by the GPOS through same application binary interface (ABI) as the GPOS. Developers can write and test their applications with toolchains they are familiar with, e.g. gcc.

The contributions of this paper are the following:

1. We show the design and implementation of a cRTOS that provides both hard real-time performance and a Linux-compatible execution environment without modifications to Linux.
2. We provide the same efficient development process of rich and hard real-time applications as that of Linux without modifications to Linux. It also allows fast prototyping.

This paper is organized as follows. We describe the design of our cRTOS in Section 2 and its implementation in Section 3 and 4. In Section 5, we present the evaluation and experimental results, including real-time performance and Linux-compatibility on system calls. In Section 6 we compare our work with various previous studies. We conclude the paper in Section 7.

2 System design

We design a cRTOS that fulfills the following requirements:

- Short and bounded jitter:
  It has fully preemptive hard real-time performance.
- Good maintainability for kernel developers:
  It works without patching Linux but adding plugins, e.g. kernel modules.
- Good usability for application developers:
  It allows access to the rich features of Linux.

It is not trivial to achieve the first and second requirements by modifying existing GPOSs because they are designed for obtaining high throughput for common (non-real-time) applications. Many previous approaches tried to modify Linux which has architectures for high throughput, e.g. the interrupt handling architecture with the top and bottom half. Modifying such throughput-oriented architecture faced a maintenance problem, as discussed in Section 1. Therefore, we designed our cRTOS which has an isolated environment for real-time applications to achieve the first requirement as well as a common environment for the second and third requirement.

2.1 Normal and real-time realms

Our proposed cRTOS runs a GPOS and one or more instances of sRTOSs in parallel using a hypervisor on a multi-core platform, as shown in Figure 1. The hypervisor provides strong isolation of hardware resources. It also provides inter-VM collaboration facilities, such as inter-VM shared memory, messaging and event notification.

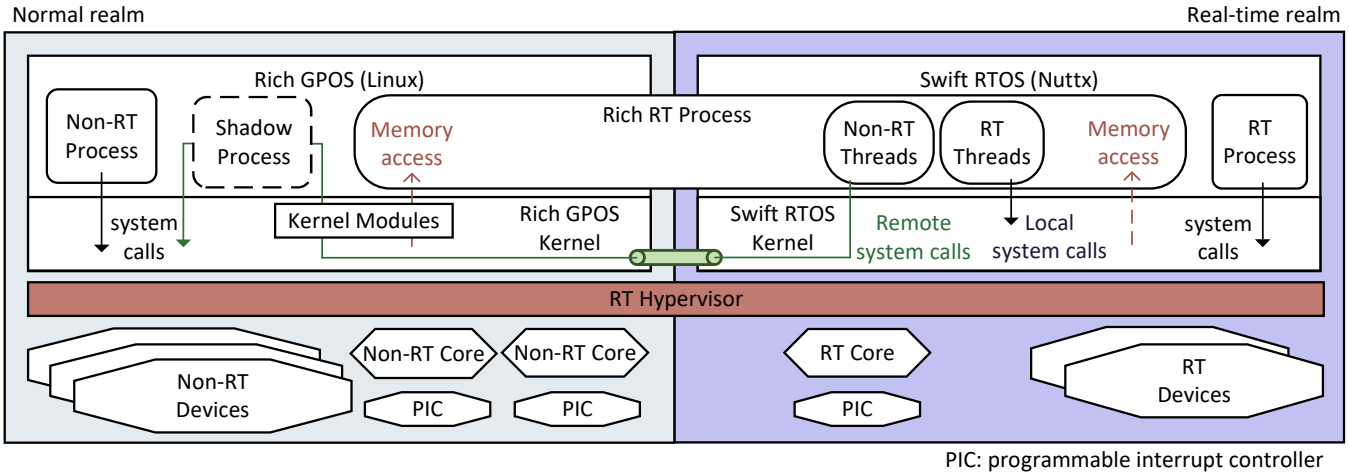

Figure 1. The system architecture of our cRTOS and the internals of a rich real-time process.

In Figure 1, we show two realms. One is the normal realm, which runs a GPOS, and provides the rich features. It executes non-real-time (non-RT) processes. This normal realm occupies most of the hardware resources including processor cores and non-real-time I/O devices. Each core has a programmable interrupt controller (PIC) with a timer.

The other is the real-time realm, which runs an sRTOS and acts as a subsystem of the GPOS. The sRTOS kernel is specially designed for real-time processes and provides a fully preemptible hard real-time environment. It also has a shorter and more consistent interrupt handling mechanism than typical GPOSs. This real-time realm has dedicated processor cores and real-time devices.

These two realms collaborate via inter-VM messaging queues and a large shared memory area. The GPOS loads a helper kernel module dynamically for this inter-VM collaboration.

2.2 Developing rich real-time applications

Developers can create real-time processes from the normal realm. A real-time process is visible in both the normal and real-time realm. We call such a process a *rich real-time process*, as shown in Figure 1.

A single rich real-time process has multiple threads in the real-time realm. The scheduling and synchronization of these threads are managed by the sRTOS kernel in the real-time realm with the real-time scheduler. This provides a fully preemptible environment and a short interrupt handling path to the rich real-time processes and prevents priority inversion among threads.

Each rich real-time process has *a shadow process* running in the normal realm, which is used to access the resources and features in the normal realm. Each thread of a rich real-time process is mapped to a thread of the shadow process. This avoids head-of-line blocking because multiple non-real-time threads do not interfere one another on accessing the resources in the normal realm.

In our cRTOS, a developer develops a rich real-time application as follows.

1. Choose a rich GPOS and sRTOS.
   Typically, Linux is chosen as a rich GPOS. The sRTOS should provide real-time performance through standard APIs of the GPOS, such as the POSIX thread and real-time extension [51]. If the sRTOS lacks support for multiple address spaces and other features, the developer needs to add them for reusing the rich toolchains of the GPOS. We will discuss this in Section 3.3. The developer also has to add system call handlers, which will be shown in Section 4.
2. Design a rich real-time process.
   It is composed by real-time and non-real-time threads. The real-time threads access real-time devices and timers and contain critical algorithms. The non-real-time threads use rich features of the GPOS, e.g. X Window. They use lockless algorithms over shared variables to avoid blocking real-time threads by non-real-time threads.
3. Develop and test the application in the GPOS.
   The developer can use the standard API and rich toolchains of the GPOS. The application uses a device driver of the GPOS in this (prototype) phase.
4. Find device drivers of real-time devices.
   The device drivers of the sRTOS of the real-time realm can be reused as is. If no drivers are available, the developer has to write them.
5. Run the executable in the real-time realm.
   The developer confirms the real-time performance of the rich real-time process.

2.3 System calls

The OS kernel of the real-time realm only provides a portion of system calls locally. When a rich real-time process triggers system calls, the system call handler executes the system call functions according to pre-defined types. There are following three types of system calls:

- Real-time system calls:
  These are related to scheduling and synchronization and are executed in the real-time realm.
- Remote system calls (RSCs):
  These are executed in the normal realm.
- Dual system calls:
  These are executed in both the normal and real-time realms. For example, `mmap()` and `exit()` are executed in both the realms.

2.4 Comparisons with existing approaches

This design of our cRTOS has advantages over the two representative approaches to real-time Linux described in Section 1. Because our design utilizes a partitioning hypervisor, modifications of the Linux kernel is not necessary. This enables a developer to use the newest vanilla kernel with new features.

Both the approach using the PREEMPT_RT patch and our cRTOS provide the same application binary interface (ABI) as vanilla Linux and allow reuse of the existing binary executables. However, our cRTOS includes a real-time kernel and a real-time interrupt handling path for real-time processes. This makes our cRTOS outperform the approach using the PREEMPT_RT patch in hard real-time tests regarding interrupt handling and preemptive scheduling.

The approach using interrupt-dispatching layers also uses a real-time kernel. However, this real-time kernel does not provide Linux APIs, and its ABI is not compatible with vanilla Linux. We cannot use existing binaries as-is, and we need special APIs to issue system calls to the Linux kernel. Furthermore, in our cRTOS, the real-time and non-real-time threads in a rich real-time processes can be synchronized by the standard APIs of the GPOS, i.e. the POSIX thread API, with priority-inheritance protocol. Our cRTOS out performs this approach on usability. Moreover, our cRTOS gives better real-time performance while under load because of better resource isolation and hardware usage for interrupt handling. We will show this in Section 5.

3 Implementation

In Section 2, we show the design of our cRTOS. Based on this design, we implement our cRTOS using Jailhouse as the partitioning hypervisor, Linux as a GPOS, and Nuttx as an sRTOS.

3.1 Jailhouse

We have chosen Jailhouse [45] as the partitioning hypervisor, because it provides the following features:

- Strong isolation of hardware resource and interrupts.
- Real-time performance.
- No VM exit under normal execution.
- Assigning PCI express (PCI-e) devices and direct routing of the related interrupts to the CPU cores.

Jailhouse does not perform dynamic core scheduling. This lowers the resource utilization but has an advantage for realizing hard real-time performance [12, 27, 57, 61].

Jailhouse utilizes typical virtualization extension of modern CPUs and creates isolated virtual machines called cells. Jailhouse provides shared memory (IVSHMEM), messaging queues based on virtio for inter-guest communication and virtual PCI devices [43, 47, 48]. We implement the shared memory facility by using IVSHMEM.

In our cRTOS, the GPOS, Linux, boots first. Next, Jailhouse is activated from Linux. Jailhouse creates two cells and allocates hardware resources to these two cells. The GPOS (booted Linux) is moved to a cell. Finally, the sRTOS, Nuttx boots in the other cell. Figure 2 shows the memory map of these Linux and Nuttx. Linux uses most of the physical memory. The memory of Nuttx consists of a boot loader, shared memory area, the kernel area, and virtio areas.

3.2 Nuttx x86-64

We use Nuttx [39] as the sRTOS because it provides the following features:

- POSIX-compatible multithread API and real-time thread scheduling.
- Real-time interrupt handling.
- Pseudo file systems, such as `/dev` and `/proc`.
- A TCP/IP networking stack.

While Nuttx is a simple RTOS targeting embedded systems, it supports a wide range of CPUs including ARM, MIPS, RISC-V, Zilog, and 8086. However, it does not support x86-64 at this time. Because we would like to run it together with Linux x86-64, we first ported it to the x86-64 long mode [25].

Nuttx consists of architecture-dependent and -independent parts. We only modified the architecture-dependent part of x86-64 and have added support for

1. time stamp counter (TSC) timers for nanosecond resolution and
2. local advance programmable interrupt controller (LAPIC) of x86-64.

We run this Nuttx x86-64 port as a guest in Jailhouse. Our current port does not support multi-core real-time scheduling. To utilize multiple cores, a developer can create multiple real-time realms running the Nuttx kernels.

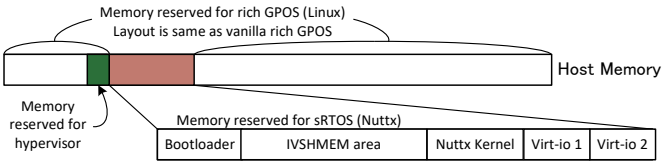

Figure 2. Memory map of our cRTOS.

3.2.1 Real-time device drivers

We can reuse the existing device drivers for Nuttx in the real-time realm. For example, we reused the upstream 16550 serial device driver without modifications.

Nuttx provides a C library and real-time version of Linux kernel API, e.g. work queues, spinlocks, etc. Real-time application developers can write their device drivers easily with this library and Linux-like API.

In our cRTOS, a rich real-time process can directly access the real-time devices that are attached to the real-time realm by Jailhouse. This can provide hard real-time performance.

3.2.2 Virtual Ethernet

In our cRTOS, we need a user-level inter-realm communication mechanism. For example, we need this to start rich real-time processes in the real-time realm from the normal realm, which will be described in Section 3.4. We decided to reuse the existing TCP/IP stack of Nuttx for this purpose.

To reuse the existing TCP/IP stack of Nuttx, we implemented a paravirtual Ethernet driver for Nuttx. The frontend driver of this driver runs in the real-time realm, uses the virtio mechanism of Jailhouse, and communicates with the backend driver in the normal realm. We used the backend driver for Linux that is provided by Jailhouse.

3.3 Using ELF binaries in the real-time realm

Base Nuttx uses binary large objects (BLOBs) as the executable format. This is quite different from Linux and prohibits use of the standard toolchains of Linux.

In our cRTOS, we use the Linux executables as the executable format in the real-time realm. We use executable and linkable format (ELF) [55] as the common executable format of the normal and real-time realms because ELF is the most-widely used executable format in Linux.

To use ELF as the executable format in the real-time realm, we add the followings to base Nuttx.

- Address space compatibility.
- Application binary interface (ABI) compatibility.

3.3.1 Address space compatibility

In base Nuttx, a real-time application includes both its code and the kernel code. A real-time application comprises threads that share the same flat address space with the kernel. Base Nuttx does not support memory mapping and multiple address spaces with memory management unit (MMU).

In our cRTOS, we changed this model, similar to Linux. A real-time application only includes its code. The kernel code is loaded at boot time.

Next, we implemented memory mapping and multiple address spaces with MMU. A single kernel runs multiple real-time processes. Each real-time process has its own address space similar to that in Linux. We implement `mmap()`, `munmap()`, `clone()`/`fork()`, and `execve()` in the Nuttx x86-64 port.

While we use MMU, we did not implement virtual memory with paging for hard real-time performance. In other words, before a real-time process runs, all pages are presented on memory and locked. Pages are physically copied when system calls `mmap()`, `fork()`, and `execve()` are executed. If a user process performs `mmap()` for a file, the kernel allocate memory pages and reads the contents of the file to the pages, immediately. This makes these system calls slower but suitable for real-time execution.

3.3.2 Application binary interface (ABI) compatibility

The ABI of base Nuttx depends on the compiler used, and the system calls ABI for x86-64 are missing. In our cRTOS, we made the Nuttx kernel use the same ABI as Linux. Our Nuttx x86-64 can execute regular Linux executables that are built with gcc, linked with Glibc, and issue the system calls of Linux. The arguments of function calls are stored in registers rdi, rsi, rdx, rcx, r8, and r9 of the x86-64 CPU. Any other arguments beyond these are pushed on the stack [33]. System calls are provided via the syscall instruction, and arguments are stored in the registers rdi, rsi, rdx, r10, r8, and r9 [33]. Our Nuttx x86-64 configures floating-point unit (FPU) in the same way as Linux. It also puts ELF auxiliary vectors[1] on the stack when a program starts execution [9].

System calls are executed by Nuttx, Linux, or both. We will describe this in Section 4.

3.4 Executing rich real-time processes in the real-time realm

To start the first real-time process, the corresponding shadow process first starts in the normal realm. This shadow process takes arguments of the `execve()` system call and creates a rich real-time process in the real-time realm as in a distributed system. After the first real-time process is created, more real-time processes can be created in this manner or the typical `fork()` and `execve()` Unix convention using existing real-time process.

Figure 3 shows the communication protocol between them. The shadow process in the normal realm establishes a TCP connection to the server. It marshals the arguments of the `execve()` system call and sends them via the TCP connection

[1] In Linux, both Glibc and the dynamic loader, ld.so, read these values to setup the runtime environment.

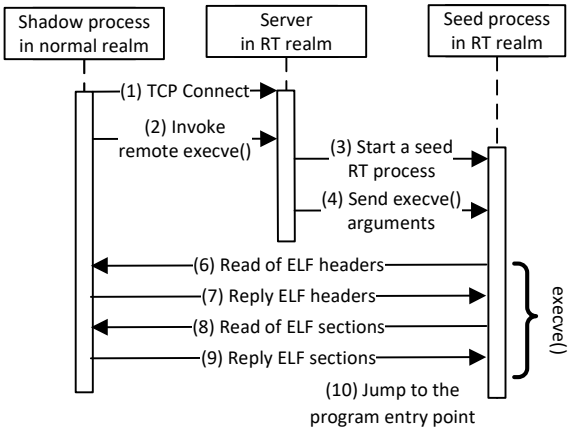

Figure 3. Executing a real-time process in the real-time realm from the normal realm

to the server. The first argument is the file name of the ELF executable in the normal realm. The server creates a *seed* process with a new system call `rexec()` of Nuttx. This seed process executes the pre-defined code in the Nuttx kernel and calls the `execve()` system call.

The `execve()` system call is mainly implemented by the user-level code that performs the following.

1. Communicates with the shadow process and reads the headers of the ELF executable.
2. Allocates the required memory pages for the `.text`, `.data`, and `.bss` segments from the shared memory area.
3. Communicates with the shadow process and reads the sections of the ELF executable into these memory pages.
4. Allocates memory pages for a new user stack from the shared memory area.
5. Maps the memory pages to the address space of the seed process.
6. Communicates with the shadow process and requests it to map the memory pages in the shared memory area to the same addresses of the seed process by the `mmap()` system call.
7. Stores the arguments and environment variables onto the user stack.
8. Stores the ELF auxiliary vectors onto the user stack.
9. Sets the stack pointer and jumps to the entry point of the executable.

The shadow process shares the same address space of the new rich real-time process. Figure 4 shows an example of the mapping between a rich real-time process and its corresponding shadow process. This is essential for the remote system calls, and its details will be described in Section 4.

Executables of Linux are classified into two types: statically linked and dynamically linked executables. Real-time

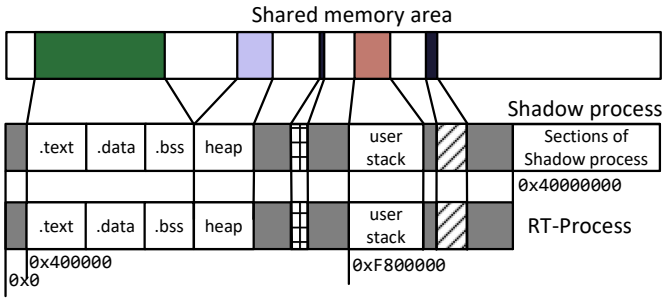

Figure 4. Memory layout of a rich real-time process and its shadow process.

applications often prefer statically linked executables because they have no run-time overheads. Typical real-time applications of Nuttx are also statically linked executables. In our cRTOS, the `execve()` system call can handle statically linked executables.

In contrast, it is convenient for rich real-time application developers to use dynamically linked executables because most Linux executables nowadays are dynamically linked ones. A dynamic linked executable requires a dynamic loader to load related dynamic shared objects (DSOs) during startup- and run-time. Most dynamically linked executables in Linux use the common dynamic loader, called ld.so [29].

To reuse this dynamic loader, we implemented the `mmap()` system call as a dual system call. Ld.so itself is a statically linked program. Therefore, it can be loaded as a statically linked executable with `execve()`. Ld.so loads the dynamically linked libraries with `mmap()` and jumps to the entry point of the dynamically linked executable. We will describe the implementation of the system call in the next section.

4 Handling system calls

4.1 Real-time system calls

In the real-time realm, the following system calls (and related APIs) are handled by the Nuttx kernel in a real-time way:

- Task-related:
  `clone()`, `fork()`, `sched_setparam()`, etc.
- Time-related:
  `clock_nanosleep()`, `gettime()`, etc.
- IO-related:
  `open()`, `close()`, `read()`, `write()`, `ioctl()` for real-time I/O devices and pseudo devices.
- IPC-related:
  `socket()` (both Unix domain sockets and TCP/IP), `pipe()`, `select()`, and `poll()`, etc. for communication with other real-time processes.

When we run Linux executables in Nuttx, we have to solve the following problems:

1. The numbering of system calls of Nuttx is different from that of Linux.

2. The flag values of various system calls are different. For example, the flag ''O_RDONLY'' of `open()` is `0x0` in Linux and `0x1` in Nuttx. Conforming to POSIX means only source level compatibility.

To solve these problems, we have implemented a translation table for the system call numbering and multiple stubs for all the system calls with different flag values. We did not modify the system call numbering and flag values of Nuttx.

A rich real-time process can use the POSIX Thread API to create real-time and non-real-time threads. In Linux, these are implemented mainly in the Glibc at the user level with a small number of non-standard system calls of Linux. We have added the following non-standard system calls of Linux to Nuttx.

- `clone()` (including `fork()` and `vfork()`)
  This is used for creating a kernel-level thread. Because multi-threading of both Linux and Nuttx is based on the 1:1 model [31], we create a single Nuttx thread for each invocation of `clone()`. In addition, we create a Linux thread in the shadow process for remote system calls.
- `futex()` (fast user-space locking)
  This is used in Glibc to reduce the overhead of the condition variable and mutex of Glibc.
- `arch_prctl()` (set architecture-specific thread state)
  This system call can set the FS register of x86-64 with an argument. This is used for implementing thread local storage (TLS) with gcc support on the x86-64 platform [6]. For example, Glibc utilizes TLS of gcc to implement the per-thread errno. Furthermore, TLS is used in Glibc to implement thread specific data with `pthread_getspecific()` [50].
- `set_tid_address()` (set thread ID address)
  This is used to implement `pthread_join()`, `pthread_exit()` and `pthread_create()`. When a thread calls `pthread_join()`, the thread is put to wait for a futex associated with the thread ID address that is set in `pthread_create()`. The futex is released when another thread call `pthread_exit()`.

4.2 Remote system calls

For non-critical tasks, a rich real-time process can use the following remote system calls of Linux:

- IO-related:
  `open()`, `close()`, `read()`, `write()`, and `ioctl()` for non-real-time I/O devices and filesystems mounted in the normal realm.
- IPC-related:
  Sockets (both Unix domain sockets and TCP/IP) for communicating with non-real-time processes.

These IO and IPC system calls make a real-time process have access to rich features. For example, a rich real-time process can open a DSO file in the normal realm and read it into the memory of the real-time realm to use additional libraries. A rich real-time process can use Unix domain sockets and connect to the X window server in the normal realm.

We have implemented remote system calls using shared-memory and virtio queues of Jailhouse as follows.

In the real-time realm:

1. A user thread of a rich real-time process issues a system call, switching to its kernel thread.
2. The kernel thread gets the arguments, makes a request message from these arguments and the priority of the thread, and puts the message into the virtio queue to Linux.
3. The kernel thread notifies Linux with an inter-processor-interrupt (IPI).
4. The kernel thread sleeps and yields the CPU to a next runnable thread.

In the normal realm:

5. The Linux kernel handles the IPI. The interrupt handler gets the request message and puts it to the memory of the corresponding shadow process. The interrupt handler wakes up the corresponding thread in the shadow process.
6. The corresponding thread extracts the request message and issues a system call to Linux.
7. The Linux kernel performs the system call.
8. The corresponding thread makes a reply message with the return value of the system call and goes back to the Linux kernel.
9. The Linux kernel puts the reply message into the virtio queue to Nuttx. The Linux kernel checks the current thread priority in the real-time realm, which is published on a shared page by the Nuttx kernel. The Linux kernel notifies the Nuttx kernel with an IPI if the priority of the returning thread is higher than the current one.

In the real-time realm:

10. The Nuttx kernel gets the reply messages from the queue and wakes up and schedules the sleeping corresponding kernel thread that has issued the remote system call in the following circumstance:
    - During receiving an IPI from Linux.
    - During a context switch. The Nuttx kernel polls the virtio queue for available messages.
11. The kernel thread gets the return value from the reply message and returns it back to the user thread.

In this implementation, we have chosen to use IPIs for better response with real-time performance. We took extra care on sending the IPIs to the real-time realm to prevent unwanted priority inversion. We also chose to use the user-level

code in the shadow process for simplicity. Similar implementations are found in device emulation of Qemu/KVM [28, 44].

We also have to multiplex system call handling for individual opened files. For example, if a file is opened in the real-time realm, the system call `read()` should be handled locally. If a file is opened in the normal realm, the system call `read()` should be handled as a remote system call.

To solve this problem, the current implementation uses a simple method for real-time performance. We split the address space of file descriptors into two spaces, the real-time space and the normal space, with a limit. If a file descriptor is smaller than the limit, the kernel deals with it as a file descriptor of the normal realm. Otherwise, the kernel deals with it as one of the real-time realm. The kernel deals with file descriptors 0 to 2 as those in the normal realm, and a rich real-time process can use `stdin`, `stdout`, and `stderr` with non-real-time threads.

4.3 Dual system calls

Because both a rich real-time process and its shadow process have to keep the same memory layout and terminate together, some system calls are executed in both the real-time and normal realms. These include `mmap()`, `munmap()`, `exit()`, `fork()`, and `vfork()`.

Because the parent process and the child process in `fork()` and `vfork()` have the same memory layout, a single shadow process cannot map the memory of both the processes. Therefore, the shadow process also forks itself in the normal realm.

5 Evaluation

In this section, we evaluate our cRTOS. First, we show the real-time performance of our cRTOS compared with the representative existing approaches, the PREEMPT_RT patched Linux and Xenomai 3. We measure the timing accuracy and interrupt latency. Next, we show the performance of the real-time and remote system calls. Finally, we discuss the compatibility of our cRTOS with Linux.

5.1 Real-time performance

5.1.1 Experimental setup

To evaluate our cRTOS, we used an x86-64 platform with an Intel Xeon 2630 v4 processor with 10 cores and 20 MB of the last-level cache (LLC) and 32 GB of RAM. We disabled the hyper-threading of the processor and the power management of operating systems to ensure accuracy of measurement. The normal realm ran vanilla or PREEMPT_RT patched Linux with version 4.9.84 in Ubuntu 16.04. The real-time realm ran our ported Nuttx version 7.3. The hypervisor was Jailhouse version 0.9.1.

We ran real-time benchmark programs in the following five different systems, which are all Linux-compatible environments.

- Vanilla Linux on bare-metal machine
- PREEMPT_RT patched Linux on bare-metal machine
- Xenomai 3 and I-Pipe patched Linux on bare-metal machine
- Our cRTOS with vanilla Linux on Jailhouse
- Our cRTOS with PREEMPT_RT Linux on Jailhouse

The first three served as baselines to compare the performance of our cRTOS. In our cRTOS, we allocated a single core and one half of the LLC to the sRTOS, and the rest of the cores and LLC to Linux. In the other systems, we allocated an exclusive core to a real-time benchmark program and one half of the LLC to the core. The rest of the cores and LLC were shared by other non-real-time programs.

To add loads to the system, we used stress-ng [26] with various built-in stressors, including the stream [34] stressor. In each experiment, we concurrently ran 10 stressors in the normal realm in our cRTOS. In other systems, we ran the stressors in Linux.

5.1.2 Timing accuracy

We evaluated the timing accuracy of our cRTOS using cyclictest [52] with various loads. Cyclictest measures the time between when a thread sets a timer and when the timer expires. Using this program, we measured the accuracy of the response to the timer interrupts. All the systems used TSC timers. We ran the same dynamically linked cyclictest executable of Linux, compiled from the source code of the main repository, in all the experiments except Xenomai. Because the cyclictest of the main repository did not work in Xenomai, we used the cyclictest port by Xenomai. Each test ran cyclictest with the SCHED_FIFO scheduling policy, with the highest priority, and produced 100,000 samples.

Figure 5 shows the results without and with load. The results of vanilla Linux in Figure 5 (d) and (e) have been cropped for better visualization. This figure contains modified box plots, which depict the first and third quantile of latency with maximum and minimum values. Figure 5 (a) shows the results without load. While threads running in vanilla Linux could achieve a very short response time to timer interrupts, the variation in response time was high. In contrast, threads running in the real-time realm of our cRTOS had not only shorter latency, but also higher stability and predictability.

Figures 5 (b) to (e) show the results with load. Compared with Figure 5 (a), the jitter of the real-time process was increased. These results show that our cRTOS could deliver hard real-time performance with about 4 µs jitter and well bounded maximum latency of 4 µs, while the others could not. In other words, with the same application executable, our cRTOS could provide better real-time performance than the representative Linux-compatible real-time environments. In addition, we noticed that the performance of our cRTOS was slightly better when using PREEMPT_RT patched Linux

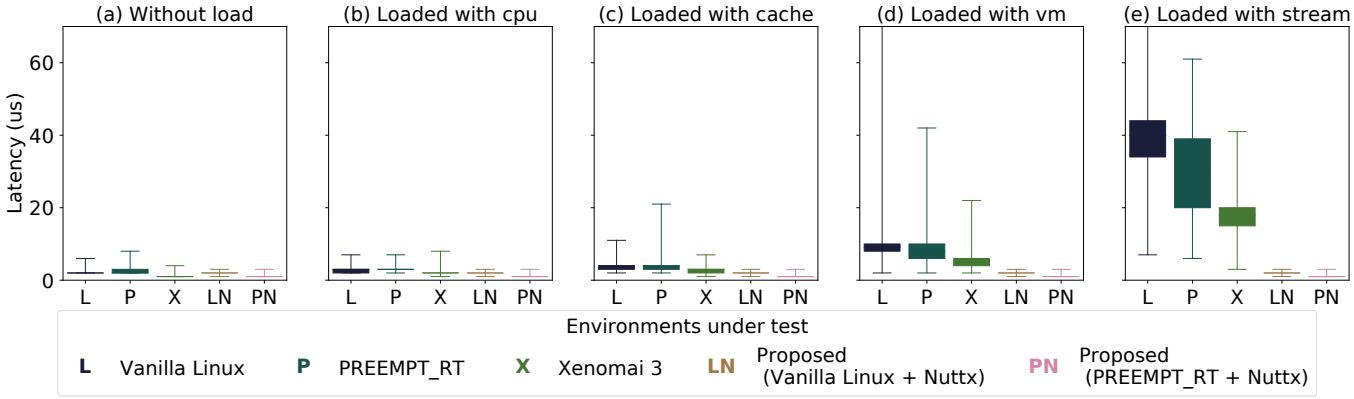

Figure 5. The latency jitter measured with cyclictest in Linux-compatible real-time environments under various loads.

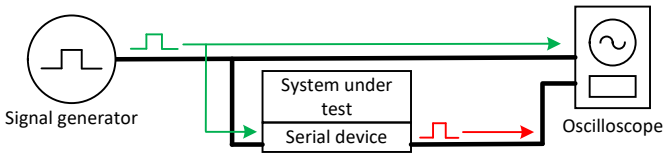

Figure 6. Measuring interrupt latency.

as the GPOS of the normal realm. However, the difference was not large. Therefore, some developers can use vanilla Linux as a rich GPOS.

5.1.3 Interrupt latency

To compare the interrupt latency, we designed the following experiment with a serial port device[2], as shown in Figure 6. The signal generator generated a 1-Khz 50% square wave on the clear-to-send (CTS) pin of the serial port device. The real-time process used the system call `ioctl()` to wait for a change of CTS and set the ready-to-send (RTS) pin. We measured the delay and jitter between the change of CTS and the corresponding change of RTS using an oscilloscope with 20 ns resolution. This included the latency and jitter of hardware, interrupt handling, and scheduling of the system.

Each test consisted of 1000 samples and ran with the SCHED_FIFO scheduling policy and the highest priority. In the PREEMPT_RT patched Linux, we pinned the device IRQ handler and the IRQ thread to the same CPU core as the process of the test program.

The results are shown in Figure 7 with the same type of modified box plots as the previous Section 5.1.2. Vanilla Linux produced a single measurement overflow in Figure 7 (c). Overall, our cRTOS yielded the best performance in both latency and jitter among the Linux-compatible environments.

[2]Typical x86-64 platform lacks general purpose input/output (GPIO) with interrupts. We utilized the modem status interrupt and the ability to sense and set hardware flow control lines and mimicked GPIOs with interrupts. We used a PCI-e serial port device with ASIX MCS9900 PCI-e to serial controller [2], because Jailhouse does not support the legacy interrupts of COM ports of x86 PCs.

5.1.4 Overhead of interrupt handling

We measured the basic overhead of interrupt handling of our cRTOS using the TACLe-benchmark [8]. This benchmark consists of 57 small CPU-bound programs that do not invoke any system calls during execution. While a benchmark program is running, the kernel handles interrupts from timer devices. We ran these benchmark programs in PREEMPT_RT patched Linux, Xenomai 3 and our cRTOS with PREEMPT_RT Linux. Each test consisted of 100 samples and ran with the SCHED_FIFO scheduling policy and the highest priority.

We measured the execution time of each program's `main()` function using the x86 time stamp counter with the method proposed in [42]. We observed no significant difference between PREEMPT_RT patched Linux, Xenomai 3 and our cRTOS.

5.2 System call latency

In this section, we evaluated the latency of real-time and remote system calls. We measured the latency using the "latency of system call" (lat_syscall) micro-benchmark in lmbench [35]. For this experiment, we used PREEMPT_RT patched Linux as a GPOS for our cRTOS. We pinned the IPI handler and the IRQ thread to the same CPU core as the shadow process. We tested the system calls `getpid()`, `read()`, `write()` and a combination of `open()` and `close()`. The filesystem related system calls were operated on pseudo devices, i.e. `/dev/zero` for `read()` and `open()` and `/dev/null` for `write()`. We collected the data of 100 tests for each system call. Each single test consisted of 10 warm-up and 100 measurement iterations. We took the maximum of these test results as the latency of each system call. We also conducted this experiment on PREEMPT_RT patched Linux and Xenomai 3 without Jailhouse for comparison. The virtual dynamic shared object (VDSO) feature in Linux was disabled for all tests to make all system calls be invoked with the `syscall` trap instruction of x86-64. This forced the execution of the actual system call handlers.

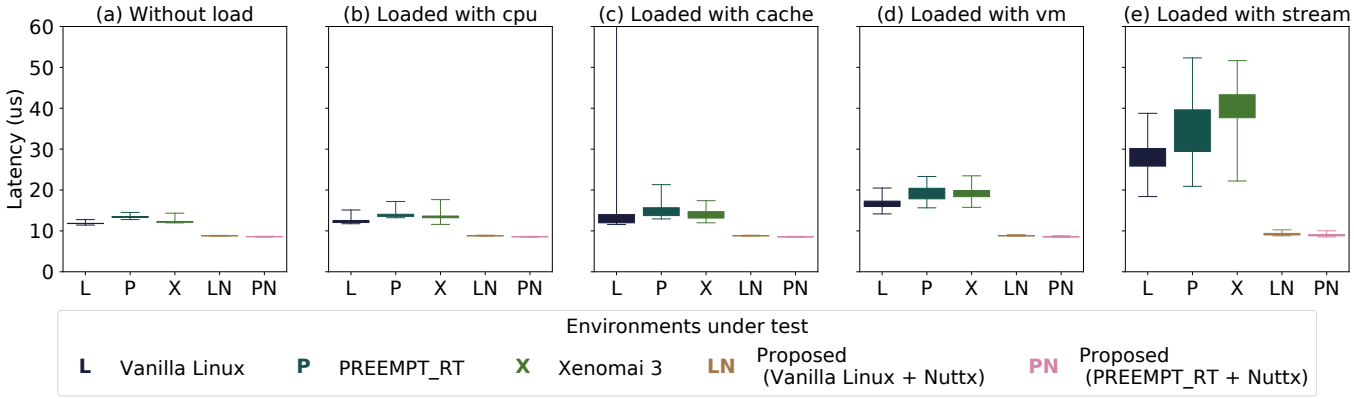

Figure 7. The interrupt latency and jitter of a serial port device in Linux-compatible real-time environments under various loads.

Table 1. The maximum latency of various system calls.
Measured by Lmbench in microseconds.

Environment	getpid	read	write	open and close
PREEMPT_RT native	0.306	0.406	0.338	2.23
Xenomai 3	0.456	1.14	1.07	4.16
Real-time system call	0.059	0.088	0.083	0.445
Remote system call	—	27.7	27.0	56.3

The results are shown in Table 1. We noticed that for real-time system calls, our cRTOS has smaller latency than Linux and Xenomai 3. This is because the real-time realm has a simpler system call handler and does not require switching between user and privilege levels. In general, we achieved 5 times speedup with respect to PREEMPT_RT patched Linux and Xenomai 3.

For remote system calls, the latency was large. It took 27 µs for a single system call. This was because of the overhead of IPIs, memory copy, cache contention, and scheduling in both the realms. We conclude that to achieve good real-time performance, developers should provide real-time drivers for real-time devices. Developers should use remote system calls for rich user interface, access to non-real-time devices, and initialization, i.e. dynamic loading.

5.3 Compatibility with Linux

The compatibility with Linux depends on the sRTOS running in the real-time realm. To evaluate the compatibility of our cRTOS using Nuttx as the sRTOS, we performed the following:

1. We statically analyzed the coverage of system calls and pseudo files in our cRTOS and compared it with vanilla Linux.
2. We executed X Window applications.

Table 2. Analysis of system call coverage.
A=Available, N/A=Not Available, O=Obsolete in Linux.

Classification	Importance	Linux 3.19	Real-time realm		
			A	N/A	O
Indispensable	100%	224	224	0	0
Important	10% - 100%	33	22	11	0
Low important	0% - 10%	44	38	6	0
Not used	0%	18	4	6	8
New in Linux 4.9		—	8	5	0
Total		319	295	29	8

5.3.1 System calls and pseudo files

We have shown the development process of our target real-time applications in Section 2.2. In this process, developers develop new real-time applications by using local real-time system call and remote system calls according to their needs. Although our cRTOS does not support all the system calls of Linux, this missing causes no problem in this development process. In the following, we evaluate our cRTOS's compatibility with Linux to run existing executables, including ones for PREEMPT_RT patched Linux.

Linux version 4.9 has 332 system calls for the x86-64 architecture. Among the 332 system calls, 8 are obsolete, and the rest of them, 324 are properly implemented in the Linux kernel. The real-time realm supports 295 of 324 system calls, and 29 system calls are not supported.

We evaluate the effect of these missing system calls by following the methodology in the paper [56]. They analyzed Ubuntu and Debian packages statically in 2.9 million installations, and found that not all APIs are equally important. They define a metric, *API importance* for a given API as the probability that an installation includes at least one application requiring the given API. Using this metric, they classified 319 system calls of Linux 3.19 into four classes: *indispensable*, *important*, *low important*, and *not used*. Table 2 shows the results.

We confirmed that the real-time realm supports all the indispensable system calls, as shown in Table 2. The real-time realm does not support 11 important and 6 low important system calls.

We analyze these 11 important system calls in detail. They are categorized as follows.

1. 5 system calls for Linux kernel extension, `init_module()`, `finit_module()`, `setns()`, `pref_event_open()` and `prlimit64()`
2. 1 system call for debugging, `ptrace()`
3. 5 Linux-specific system calls, `vhangup()`, `pivot_root()`, `unshare()`, `signalfd4()` and `process_vm_readv()`

If an existing application uses these system calls, it does not run in the real-time realm. As discussed in Section 2.2, our target real-time application developers can execute debuggers in the normal realm at the prototype phase, and do not use debuggers in the real-time realm at the production phase. Note that developers can use these system calls and existing executables in the normal realms in a non-real-time way.

The current implementation of system calls has several limitations. First, our cRTOS currently cannot handle cross-realm signals. While a real-time process can send signals to real-time processes in the real-time realm, it cannot send signals to processes in the normal-realm, and vice versa. Second, the kernel of real-time realm cannot interpret some arguments of vectored system calls, such as `ioctl()` and `fcntl()` of Linux.

The paper [56] also evaluates the API through pseudo files with the API importance as similar to system calls. It is very hard for a Linux-compatible system to implement a large number of pseudo files. The paper shows the API importance distribution of pseudo files under `/dev` and `/proc` has a small head and a long tail.

The real-time realm, using Nuttx as the sRTOS, supports the important pseudo files whose importance values are greater than 5%. Rich real-time processes can use `/dev/null`, `/dev/zero`, `/dev/urandom` and `/dev/random` in a real-time way[3].

The current implementation has a limitation in pseudo files. If an application opens a pseudo file that does not exist in the real-time realm, the process opens the file in the normal realm. This can cause a problem. For example, if an application opens `/dev/cpuinfo`, the application gets the CPU model, correctly. However, the application cannot get the number of online CPUs. Furthermore, accessing the pseudo file in the normal realm can violate hard real-time constraint. In such a case, real-time application developers should modify their applications or write device drivers of pseudo files in the real-time realm.

[3]Our Nuttx port uses the x86 `rdrand` instruction to generate random numbers for `/dev/random`.

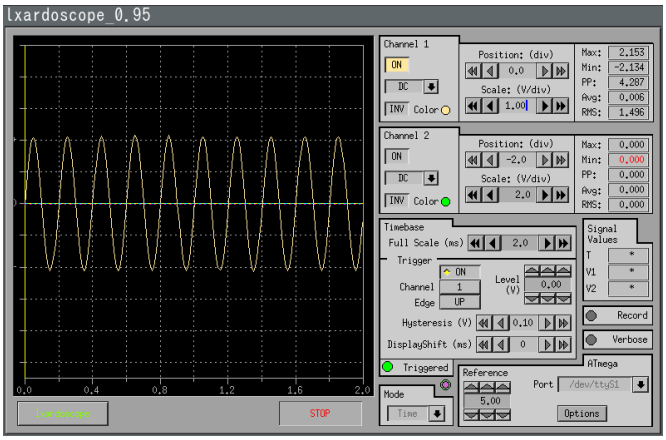

Figure 8. Screen shot of lxardoscope, an opensource- oscilloscope program running in the real-time realm.

Remote system calls of our cRTOS are similar to system call wrappers in Xenomai 3. Through this mechanism, Xenomai 3 allows a real-time process to call the other non-real-time kernel with sacrificing hard real-time performance [17]. Our cRTOS has two advantages over Xenomai 3 in terms of Linux-compatibility and usability. First, our cRTOS achieves higher Linux-compatibility by providing a larger number of system calls in the real-time realm. Second, our cRTOS supports the full GNU C Library, while Xenomai 3 uses a custom C library with limited functions.

5.3.2 X window applications

It is not trivial for existing hard real-time environments including Xenomai to run X window applications [5, 16]. We tested various X window applications of Ubuntu 16.04 in the real-time realm. The following applications could work properly.

- Basic X utilities: xeyes, xclock, glxinfo, and glxgears
- Image rendering: Feh, Image Magick and Ristretto
- PDF viewers: Xpdf, MuPDF, GSview and Zathura
- Editors: GVim and LeafPad
- Terminal emulator: xterm with dash

However, not all X window applications were supported in the real-time realm. For example, Emacs did not work because of lacking support of cross-realm signals.

We also tested a real-time program, *lxardoscope* [37]. This is an open-source oscilloscope program, gathers data from a serial device and plots the data as a waveform, as shown in Figure 8. This program could run in the real-time realm of our cRTOS. It received data via a serial device, which was directly attached to the real-time realm with a real-time driver.

6 Related Work

We have described two current representative approaches of real-time Linux in Section 1 and Section 2.4. We have compared the performance and Linux-compatibility of our cRTOS with those of the representative approaches in Section 5. In this section, we discuss other related work.

RTLinux [60], Resource kernel for Linux [40] and Time Sensitive Linux (TSL) [19] are early attempts to bring Linux as an RTOS prior to PREEMPT_RT. These methods required the vanilla Linux kernels to be modified. In addition, the programming environment is not rich as a normal GPOS. LITMUS^{RT} adds various types of real-time schedulers into Linux [4]. Its goal is to provide a scheduler testing platform for real-time systems in academic research. Its support is not active available and using it in real-world problems is difficult.

Similar techniques have been applied to hypervisors to acquire real-time performance. These include real-time KVM and RT-Xen [27, 57, 58, 61]. They also require patching the vanilla Linux kernel. Real-time KVM does not provide hard real-time performance. While RT-Xen provides guaranteed hard real-time performance, complicated compositional schedulability analysis (CSA) [20] is required to configure the system. RT-Xen does not support fixed priority scheduling. In this study, we proposed a cRTOS that overcomes these shortcomings.

Outsourcing and similar techniques delegate network and other I/O operations from a guest kernel to a host OS [7, 10, 36, 38]. Most of them focus on improving I/O throughputs by omitting message copying and shortening message processing paths. They require modifications to Linux. The work [11] provides only soft real-time performance.

McKernel is a co-kernel approach [15], and delivers a simple POSIX programming environment for high performance computing (HPC) applications. A shadow process in our cRTOS is similar to a proxy process in McKernel, and we applied the idea to real-time applications. We extended the idea to not only API compatibility but also to ABI compatibility. This allows reusing the rich toolchains of Linux. In addition, we allowed direct hardware access without relying on Linux system calls, which is important in real-time applications.

[41] demonstrated that building a unikernel with identical ABI to Linux is possible. This unikernel can yield the same level of performance for a single application by using Linux with a Type-II hypervisor. In this study, we extended an existing RTOS to adapt to Linux ABI. Our cRTOS yields better real-time performance than Linux and supports running multiple processes. We utilized a Type-I partitioning hypervisor instead of a Type-II hypervisor to ensure the real-time performance.

The implementation of remote system calls in our cRTOS is similar to that of FlexSC [49]. In FlexSC, user threads put requests to shared memory with a kernel, and kernel threads execute the requests asynchronously without exceptions. This architecture improves throughput, especially in multi-core processors. Our implementation uses shared memory and messaging queues between the normal realm and the real-time realm. We use inter-processor interrupts (IPIs) as the inter-realm notification mechanism to preserve the real-time property while avoiding priority inversion.

7 Conclusion

In this study, we achieved hard real-time performance and rich Linux features togethre in a compounded real-time operating system (cRTOS). This system creates two realms with a partitioning hypervisor: a normal realm of Linux and a real-time realm of a swift RTOS. A rich real-time process running in the real-time realm can use not only the hard real-time performance of the swift RTOS but also the rich features of Linux through remote system calls. Unlike existing approaches to real-time Linux including the PREEMPT_RT patch and using interrupt-dispatching layers, our cRTOS requires no modifications to Linux and provides hard real-time performance.

We have implemented our cRTOS by running Nuttx, a POSIX-compliant RTOS, as a swift RTOS and Jailhouse as the partitioning hypervisor. We ported base Nuttx to the x86-64 architecture and added support for multiple virtual addresses with MMU. This allows developers of rich real-time applications to use the same toolchains and executables with Linux. This reduces the cost and complexity of developing real-time applications while strongly guaranteeing the real-time performance.

We performed experiments and measured timing accuracy and interrupt latency of our cRTOS and other existing systems, the PREEMPT_RT patched Linux and Xenomai 3. The experimental results show that our cRTOS could deliver hard real-time performance with about 4 µs jitter and well bounded maximum latency, while the others could not. The experimental results also show that our cRTOS with a real-time device yielded the best interrupt response in both latency and jitter. Finally, we demonstrated that our cRTOS could execute complex Linux executables with graphical user interfaces through the X window system.

In the future, we would like to add debugging support to the real-time realm. We are also interested in running other RTOSs, such as FreeRTOS [1] and Zypher [54], in the real-time realm.

Acknowledgment

This work was partially supported by JSPS KAKENHI Grant Number 25540022 and 16K12410.

References

[1] Amazon Web Services. 2019. The FreeRTOS^{TM} Kernel. https://www.freertos.org/

[2] ASIX Electronics Corporationa. 2015. *MCS9900 PCIe to Multi I/O (4S, 2S+1P) Controller Datasheet.* ASIX Electronics Corporationa.

[3] Antonio Barbalace, Adriano Luchetta, Gabriele Manduchi, Michele Moro, Anton Soppelsa, and Cesare Taliercio. 2008. Performance Comparison of VxWorks, Linux, RTAI, and Xenomai in a Hard Real-Time Application. *IEEE Transactions on Nuclear Science* 55, 1 (Feb. 2008), 435–439. https://doi.org/10.1109/TNS.2007.905231

[4] John M. Calandrino, Hennadiy Leontyev, Aaron Block, UmaMaheswari C. Devi, and James H. Anderson. 2006. LITMUS^{RT} : A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers. In *Proceedings of 2006 27th IEEE International Real-Time Systems Symposium (RTSS'06)*. 111–126. https://doi.org/10.1109/RTSS.2006.27

[5] Chok Leong Chai. 2010. [RTAI] "XIO: fatal IO error 1..." when using Liunx server. http://mail.rtai.org/pipermail/rtai/2010-June/023338.html

[6] Ulrich Drepper. 2002. *ELF Handling for Thread-Local Storage.* Technical Report.

[7] Hideki Eiraku, Yasushi Shinjo, Calton Pu, Younggyun Koh, and Kazuhiko Kato. 2009. Fast Networking with Socket-outsourcing in Hosted Virtual Machine Environments. In *Proceedings of the 2009 ACM Symposium on Applied Computing (SAC '09)*. 310–317. https://doi.org/10.1145/1529282.1529350

[8] Heiko Falk, Sebastian Altmeyer, Peter Hellinckx, Björn Lisper, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Rasmus Bo Sørensen, Peter Wägemann, and Simon Wegener. 2016. TACLeBench: A benchmark collection to support worst-case execution time research. In *Proceedings of the 16th International Workshop on Worst-Case Execution Time Analysis (WCET '16)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2:1–2:10.

[9] The Linux Foundation. 2015. *Linux Standard Base Core Specification for X86-64.* http://refspecs.linuxbase.org/LSB_5.0.0/LSB-Core-AMD64/LSB-Core-AMD64.pdf

[10] Sahan Gamage, Ramana Rao Kompella, Dongyan Xu, and Ardalan Kangarlou. 2013. Protocol Responsibility Offloading to Improve TCP Throughput in Virtualized Environments. *ACM Transactions on Computer Systems* 31, 3, Article 7 (Aug. 2013), 34 pages. https://doi.org/10.1145/2491463

[11] Oscar F. Garcia, Yasushi Shinjo, and Calton Pu. 2018. Achieving Consistent Real-Time Latency at Scale in a Commodity Virtual Machine Environment Through Socket Outsourcing-Based Network Stacks. *IEEE Access* 6 (2018), 69961–69977.

[12] Marisol García-Valls, Tommaso Cucinotta, and Chenyang Lu. 2014. Challenges in real-time virtualization and predictable cloud computing. *Journal of Systems Architecture* 60, 9 (2014), 726 – 740. https://doi.org/10.1016/j.sysarc.2014.07.004

[13] GCC team. 2019. GCC, the GNU Compiler Collection. https://gcc.gnu.org/

[14] GCC team. 2019. The GNU C Library (glibc). https://www.gnu.org/software/libc/

[15] Balazs Gerofi, Masamichi Takagi, Yutaka Ishikawa, Rolf Riesen, Evan Powers, and Robert W Wisniewski. 2015. Exploring the design space of combining Linux with lightweight kernels for extreme scale computing. In *Proceedings of the 5th International Workshop on Runtime and Operating Systems for Supercomputers*. 5–12.

[16] Philippe Gerum. 2004. *Xenomai–Implementing a RTOS emulation framework on GNU/Linux.* Technical Report.

[17] Philippe Gerum. 2014. [Xenomai] POSIX application running under xenomai – what do wrapped functions do? https://xenomai.org/pipermail/xenomai/2014-June/031120.html

[18] Sourav Ghosh and Ragunathan Raj Rajkumar. 2002. Resource management of the OS network subsystem. In *Proceedings 5th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing. ISIRC 2002*. 271–279. https://doi.org/10.1109/ISORC.2002.1003728

[19] Ashvin Goel, Luca Abeni, Charles Krasic, Jim Snow, and Jonathan Walpole. 2002. Supporting Time-sensitive Applications on a Commodity OS. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*. 165–180. https://doi.org/10.1145/844128.844144

[20] Sriram Govindan, Arjun R. Nath, Amitayu Das, Bhuvan Urgaonkar, and Anand Sivasubramaniam. 2007. Xen and Co.: Communication-aware CPU Scheduling for Consolidated Xen-based Hosting Platforms. In *Proceedings of the 3rd International Conference on Virtual Execution Environments (VEE '07)*. 126–136. https://doi.org/10.1145/1254810.1254828

[21] Hermann Härtig, Robert Baumgartl, Martin Borriss, Claude-Joachim Hamann, Micheal Hohmuth, Frank Mehnert, Lars Reuther, Sebastian Schönberg, and Jean Wolter. 1998. DROPS: OS Support for Distributed Multimedia Applications. In *Proceedings of the 8th ACM SIGOPS European Workshop on Support for Composing Distributed Applications*. 203–209. https://doi.org/10.1145/319195.319226

[22] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. 1997. The Performance of μ-kernel-based Systems. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP '97)*. 66–77. https://doi.org/10.1145/268998.266660

[23] Gernot Heiser and Kevin Elphinstone. 2016. L4 Microkernels: The Lessons from 20 Years of Research and Deployment. *ACM Transactions on Computer Systems* 34, 1, Article 1 (April 2016), 29 pages. https://doi.org/10.1145/2893177

[24] Gernot Heiser and Ben Leslie. 2010. The OKL4 Microvisor: Convergence Point of Microkernels and Hypervisors. In *Proceedings of the First ACM Asia-pacific Workshop on Workshop on Systems (APSys '10)*. 19–24. https://doi.org/10.1145/1851276.1851282

[25] Intel. 2018. *Intel® 64 and IA-32 Architectures Software Developer's Manual.*

[26] Colin King. 2018. Stress-ng: a tool to load and stress a computer system. http://kernel.ubuntu.com/~cking/stress-ng

[27] Jan Kiszka and Rik van Riel. 2016. Two approaches to real-time virtualization - Jailhouse and KVM. In *Proceedings of the Linux Foundation Real-Time Summit.* https://wiki.linuxfoundation.org/realtime/events/rt-summit2016/schedule

[28] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. 2007. KVM: the Linux Virtual Machine Monitor. In *Proceedings of the 2007 Ottawa Linux Symposium (OLS '07)*.

[29] Linux man-pages. 2019. ld.so, ld-linux.so - dynamic linker/loader. http://man7.org/linux/man-pages/man8/ld.so.8.html

[30] C. L. Liu and James W. Layland. 1973. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *J. ACM* 20, 1 (Jan. 1973), 46–61. https://doi.org/10.1145/321738.321743

[31] Robert Love. 2010. *Linux Kernel Development* (3rd ed.). Addison-Wesley Professional.

[32] Paolo Mantegazza, EL Dozio, and Steve Papacharalambous. 2000. RTAI: Real time application interface. *Linux Journal* 2000, 72es (2000), 10.

[33] Michael Matz, Jan Hubicka, Andreas Jaeger, and Mark Mitchell. 2014. *System V Application Binary Interface AMD64 Architecture Processor Supplement, Draft Version 0.99.7.* Technical Report. https://www.uclibc.org/docs/psABI-x86_64.pdf

[34] John D McCalpin. 1995. Memory bandwidth and machine balance in current high performance computers. *IEEE computer society technical committee on computer architecture (TCCA) newsletter* (1995), 19–25.

[35] Larry McVoy and Carl Staelin. 1996. Lmbench: Portable Tools for Performance Analysis. In *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference (ATEC '96)*. 23–23. http://dl.acm.org/citation.cfm?id=1268299.1268322

[36] Jun Nakajima, Qian Lin, Sheng Yang, Min Zhu, Shang Gao, Mingyuan Xia, Peijie Yu, Yaozu Dong, Zhengwei Qi, Kai Chen, and Haibing Guan. 2011. Optimizing Virtual Machines Using Hybrid Virtualization. In

Proceedings of the 2011 ACM Symposium on Applied Computing (SAC '11). 573–578. https://doi.org/10.1145/1982185.1982308

[37] Nick. 2013. lxardoscope. https://sourceforge.net/projects/lxardoscope/

[38] Audun Nordal, Åge Kvalnes, and Dag Johansen. 2012. Paravirtualizing TCP. In *Proceedings of the 6th International Workshop on Virtualization Technologies in Distributed Computing Date (VTDC '12)*. 3–10. https://doi.org/10.1145/2287056.2287060

[39] Nuttx. 2019. NuttX Real-Time Operating System. http://www.nuttx.org

[40] Shuichi Oikawa and Raj Rajkumar. 1999. Portable RK: a portable resource kernel for guaranteed and enforced timing behavior. In *Proceedings of the 5th IEEE Real-Time Technology and Applications Symposium*. 111–120. https://doi.org/10.1109/RTTAS.1999.777666

[41] Pierre Olivier, Daniel Chiba, Stefan Lankes, Changwoo Min, and Binoy Ravindran. 2019. A Binary-compatible Unikernel. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '19)*. 59–73. https://doi.org/10.1145/3313808.3313817

[42] Paoloni, Gabriele. 2010. *How to Benchmark Code Execution Times on Intel® IA-32 and IA-64 Instruction Set Architectures*. Intel Corporation.

[43] Qemu. 2016. *Device Specification for Inter-VM shared memory device*. Technical Report.

[44] Qumranet Inc. 2006. *KVM: Kernel-based Virtualization Driver*. Qumranet.

[45] Ralf Ramsauer, Jan Kiszka, Daniel Lohmann, and Wolfgang Mauerer. 2017. Look Mum, no VM Exits!(Almost). In *Proceedings of Workshop on Operating Systems Platforms for Embedded Real-Time Applications*. 13–18.

[46] Federico Reghenzani, Giuseppe Massari, and William Fornaciari. 2019. The Real-Time Linux Kernel: A Survey on PREEMPT_RT. *Comput. Surveys* 52, 1, Article 18 (Feb. 2019), 36 pages. https://doi.org/10.1145/3297714

[47] Yi Ren, Ling Liu, Qi Zhang, Qingbo Wu, Jianbo Guan, Jinzhu Kong, Huadong Dai, and Lisong Shao. 2016. Shared-Memory Optimizations for Inter-Virtual-Machine Communication. *Comput. Surveys* 48, 4, Article 49 (Feb. 2016), 42 pages. https://doi.org/10.1145/2847562

[48] Rusty Russell. 2008. Virtio: Towards a De-facto Standard for Virtual I/O Devices. *ACM SIGOPS Operating Systems Review* 42, 5 (July 2008), 95–103. https://doi.org/10.1145/1400097.1400108

[49] Livio Soares and Michael Stumm. 2010. FlexSC: Flexible System Call Scheduling with Exception-less System Calls. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI '10)*. 33–46.

[50] Sun Microsystems, Inc. 2004. *Linker and Libraries Guide*.

[51] The IEEE and The Open Group. 2017. *IEEE 1003.1-2017 - IEEE Standard for Information Technology–Portable Operating System Interface (POSIX(R)) Base Specifications, Issue 7*. IEEE.

[52] The Linux Foundation. 2018. Cyclictest. https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/cyclictest/start

[53] The Linux Foundation. 2019. realtime:start [Linux Foundation Wiki]. https://wiki.linuxfoundation.org/realtime/start

[54] The Linux Foundation. 2019. Zephyr Project. https://www.zephyrproject.org/

[55] Tool Interface Standard (TIS). 1995. *Executable and Linking Format (ELF) Specification Version 1.2*. Technical Report.

[56] Chia-Che Tsai, Bhushan Jain, Nafees Ahmed Abdul, and Donald E. Porter. 2016. A Study of Modern Linux API Usage and Compatibility: What to Support when You're Supporting. In *Proceedings of the 11th European Conference on Computer Systems (EuroSys '16)*. 16:1–16:16. https://doi.org/10.1145/2901318.2901341

[57] Sisu Xi, Justin Wilson, Chenyang Lu, and Christopher Gill. 2011. RT-Xen: Towards Real-time Hypervisor Scheduling in Xen. In *Proceedings of the 9th ACM International Conference on Embedded Software (EMSOFT '11)*. 39–48. https://doi.org/10.1145/2038642.2038651

[58] Sisu Xi, Meng Xu, Chenyang Lu, Linh Thi Xuan Phan, Christopher Gill, Oleg. Sokolsky, and Insup Lee. 2014. Real-time multi-core virtual machine scheduling in Xen. In *2014 International Conference on Embedded Software (EMSOFT)*. 1–10. https://doi.org/10.1145/2656045.2656061

[59] Karim Yaghmour. 2001. *Adaptive domain environment for operating systems*. Opersys Inc.

[60] Victor Yodaiken. 1999. The RTLinux manifesto. In *Proceedings of The 5th Linux Expo*.

[61] Pei Zhang. 2017. See what happened with real time KVM when building real time cloud. In *Proceedings of the Linux Foundation LinuxCon China*. https://www.slideshare.net/LCChina