The Offline Nanovisor

Raz Ben Yehuda [1] and * Nezer Zaidenberg[2]

[1]Department of Computer Science, University of Jyväskylä, Jyväskylä ,Finland ,
raziebe@gmail.com, +972-54-551-6861, ORCID *0000-0001-5712-0677*
[2]Department of Computer Science, College of Management Academic Studies,Rishon Le Zion, Israel, *scipioenterprises@gmail.com*, +972 54-553-1415

February 17, 2023

Abstract

Current real-time technologies for Linux require partitioning for running RTOS alongside Linux or extensive kernel patching. The Offline Nanovisor is a minimal real-time library OS in a lightweight hypervisor under Linux. We describe a Nanovisor that executes in an offline processor. An offline processor is a processor core removed from the running operating system. The offline processor executes userspace code through the use of a hyplet. The hyplet is a Nanovisor that allows the kernel to execute userspace programs without delays. Combining these two technologies offers a way to achieve hard real-time in standard Linux. We demonstrate high-speed access in various use cases using a userspace timer in frequencies up to 20 kHz, with a jitter of a few hundred nanoseconds. We performed this on a relatively slow ARM processor.

Keywords: Hypervisor; real-time; ARM; Virtualization; Embedded Linux.

1 Introduction

Obtaining predictable latency while processing data is a challenging task. That is especially true in general-purpose operating systems (GPOS). In GPOS the program becomes less predictable, in many cases, due to cache and TLB misses Ekman *et al.* [2002]; Bennett and Audsley [2001]. Software architects handle this unpredictability with various techniques. Such techniques include microkernels, microvisors and partitioning or real-time extensions to the GPOS. Other solutions include auxiliary processors, such as DSPs and even GPUs.

This paper offers an alternative approach to the above for multi-processor machines. By virtually removing a processor from the operating system and running a single program in it, it is possible to achieve predictable latency in a GPOS. Removing a process from the GPOS scheduler and assigning it to a single task is referred by Ben-Yehuda and Wiseman [2013] as the 'offline scheduler'. A difficulty with the offline scheduler design is that it can only run kernel code - in what we refer to as an 'offlet'.

We evolve the offline scheduler to process hyplets as introduced in Ben Yehuda and Zaidenberg [2018]. Instead of running in kernel mode, the offlet executes in hypervisor mode. Hyplet technology is an excellent fit for the offline scheduler because both provide complementary real-time advantages. Furthermore, both run without interrupts, so it was easy to combine them.

Microcontrollers trap real-world events. These microcontrollers send interrupts to the GPOS processor to inform about incoming events. Thus, the accuracy of the data also relies on the rate of the interrupts processed by the GPOS processor. In this context, therefore, we demonstrate the offline scheduler as a technique to acquire data from external devices through high-speed data sampling. We construct test cases in which the physics changes so fast that without a tight loop to access the data, it is not possible to observe these changes. For example, we show that we can measure the beginning and the return of an ultrasonic wave more accurately than a standard Debian Linux. Another essential use we demonstrate is a 20 kHz software timer. High-speed timers are widely used in many real-time applications and usually require hardware and software. An interrupt triggers the system's timer and, on Linux, the interrupt usually wakes a userspace process; this entire chain of events causes latency. While the hyplet eliminates kernel-to-user latency, the Offline Nanovisor eliminates the **offlet-to-user latency**. This motivates us to escalate privileges.

Therefore, we provide a hard real-time library in a Linux GPOS and a fast access to sensors that cannot generate interrupts, i.e. sensors that need to be polled. In the Offline Nanovisor, the kernel runs with interrupts disabled. Thus, the Offline Nanovisor guarantees latency as long as the program does not wait for another program or overflow the processor's L1 caches. An executing program never leaves the processor, and it is up to the programmer to yield the processor. As we show later, as long as the code and data remain in the processor's cache, real-time responsiveness is guaranteed. Furthermore, if the data size surpasses the processor's cache size, we can pre-fetch it (or part of it). The pre-fetch is possible because there is no other program in the processor. Therefore, it possible to predict which buffers are needed and when.

The outline of this paper is as follows:

- **Section 2** describes hyplet and offlet technologies and their combination.

- **Section 3** presents the Offline Nanovisor and its programming model.

- **Section 4** is an evaluation.

- **Section 5** demonstrates some use cases for the Offline Nanovisor.

- **Section 6** provides an overview of related work.

- **Section 7** presents a summary.

2 Background

This section describes the offline scheduler and presents ARM's permission model.

2.1 The offline scheduler

Algorithm 1: Offline scheduler typical main loop

```
..Drop the processor procedure;
while Processor in offline mode do
    user_callback();
    pause();
end
```

In many Linux devices today, unplugging a processor is a way to reduce power consumption and heat. An unplugged processor is stripped out of any resources it controls, such as memory and scheduling system, and then is moved to sleep mode. In Intel architectures, the unplugged processor runs a loop of the halt instruction. In ARM, the processor switches to EL3, the TrustZone, which relaxes the processor or executes the WFI (Wait For Interrupt) instruction. At this point, the offline scheduler appears. The processor invokes a kernel driver procedure instead of shifting to a relax mode. We depict offline scheduling in Algorithm 1. This kernel driver is the offline scheduler. The offline scheduler cyclically assigns the processor a single task. The offline CPU operates without interrupts and can access the entire kernel address space natively. Thus, a program executing in an offline mode may access most native APIs of the Linux kernel as long as these APIs do not rely on the processor to be online. For example, it is not possible to use the standard kernel memory allocation (kmalloc), but freeing memory is possible. The offline scheduler, depicted in Figure 1, shows that the offline processor, CPU3, can access the entire address space in the operating system like a program in the online processors. The online processors, in the light gray area, run the GPOS while the offline processor, in the darker gray area, does not run the GPOS. The GPOS does not control CPU3 in the gray area. In Linux, it is possible to unplug no more than N-1 processors in an N processors computer.

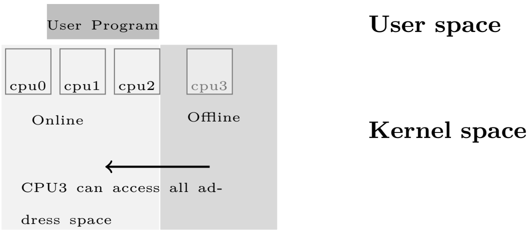


Figure 1: Offline scheduler

Waiting for an event using a tight loop is considered to be a bad technique. There are several reasons for this:

- The processor does not serve any other task, which violates the multiprogram paradigm described by Betz *et al.* [2009]. A single program cannot keep either the CPU or the I/O devices busy at all times.
- The processor's temperature increases.
- The processor must go through a quiescent state, as discussed in Bovet and Cesati [2005]. In operating systems, a quiescent state is when the program performs a system call or relinquishes the processor. If the program does not relinquish the processor, then the operating system would get into an error state and might hang.

For these reasons, we decided to hot-unplug the processor because unplugging relaxes some of the above as the processor is not part of the operating system kernel and, therefore, not subordinate to its heuristics. To mitigate the heat problem, we called the *pause* mnemonic instruction in x86 and its ARM equivalent mnemonic, *yield*. The duration of the delay instructions is measured so they can be used accurately.
Historically, the first use of the offline scheduler was as a real-time packet scheduler. We created the offline scheduler as a component of a high-performance video server. Thus we proposed a way to achieve real-time alongside the GPOS. Our method does not have the costs and efforts of an additional RTOS.

Ad hoc RTOS for power saving

Due to their nature, real-time operating systems usually consume more power Shalan and El-Sissy [2009]; Madhavapeddy *et al.* [2015]. Thus RTOS generates more heat than a general-purpose operating system. Vendors deal with these problems through the use of auxiliary processors such as DSPs and GPUs and power-saving software Datta *et al.* [2012]; Paul and Kundu [2010] and hardware. The Offline Nanovisor behaves like these auxiliary processors. the GPOS manages the processor as long as there is no need for real-time performance. In these periods (when real-time performance is not required), the processor is usually unplugged or running in reduced frequency. Once there is a need for real-time, the Offline Nanovisor unplugs the processor and executes a hyplet in it. We refer to this process as 'offlet booting'. Unplugging a processor takes about 100 milliseconds. For returning the CPU to the GPOS, the Offline Nanovisor unmaps the hyplet and then returns the processor to the GPOS. This process also takes approximately 100 milliseconds.

Resource sharing

Resource sharing between two VM guests, a host and a guest VM, or a complete unikernel with the GPOS may be costly in terms of engineering effort (device drivers) and synchronization (shared memory). The Offline Nanovisor, however, constantly switches between kernel mode and HYP mode. To utilize the host's services, the offlet accesses the entire Linux system resources natively and securely. The same applies for the hyplet, it also runs natively but in the user process's confined address space. In both cases, the offlet and hyplet programs access variables directly.

Learning Curve

The offline hyplet learning curve is small. It resembles setting a UNIX signal sig [2020]; it does not require any special compilers or tools, and communication between the Linux process to the real-time context is not needed as they share address space. In contrast, assimilating other RTOS systems in some cases is a big engineering effort that requires high expertise.

2.2 ARM permission model

ARM has a unique approach to security and privilege levels and is crucial to the implementation of the hyplet. In ARMv7, ARM introduced the concept of a secure world and a non-secure world, through the implementation of TrustZone and, starting from ARMv7-a, ARM presents four exception (permission) levels.

- **Exception Level 0 (EL0)** refers to userspace code. EL0 is analogous to 'ring 3' on the x86 platform.

- **Exception Level 1 (EL1)** refers to operating system code. EL1 is analogous to 'ring 0' on the x86 platform.

- **Exception Level 2 (EL2)** refers to HYP mode. EL2 is analogous to 'ring -1' or 'real mode' on the x86 platform.

- **Exception Level 3 (EL3)** refers to TrustZone. A special security mode that can monitor the ARM processor and may run a real-time security OS. There is no direct x86 analogous mode, Intel's ME or SMM are related concepts in the x86 platform.

Each exception level provides its own state of special-purpose registers and can access these registers at the higher but not lower levels. The general-purpose registers are shared. Thus, moving to a different exception level does not require the expensive context switch associated with the x86 architecture. ARMv8 architecture dictates distinct translation tables of the different exception levels; this separation is the pillar of the hyplet.

3 The hyplet

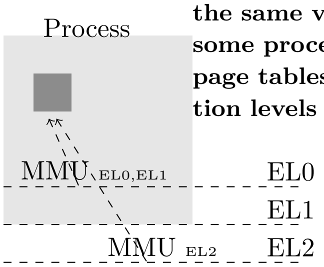
ARM8v-a specifications offer to distinct between user-space addresses and kernel space addresses by the MSB (most significant bits). The user-space addresses of Normal World and the hypervisor use the same format of addresses. These unique characteristics are what make the hyplet possible. The nanovisor can execute user-space position-independent code without preparations. Consider the code snippet at Figure 2. The ARM hypervisor can access this code's relative addresses (adrp), stack (sp_el0) etcetera without pre-processing. From the nanovisor perspective, Figure 2 is a native code.
Here, for example, address 0x400000 might be used both by the nanovisor and the user.

```
400610: foo:
400614: stp x16, x30, [sp,#-16]!
400618: adrp x16, 0x41161c
40061c: ldr x0, [sp,#8]
400620: add x16, x16, 0xba8
400624: br x17
400628: ret
```

Figure 2: A simple hyplet

So, if we map part of a Linux process code and data to a nanovisor it can be executed by it.
When interrupt latency improvement is required, the code is frequently migrated to the kernel, or injected as the eBPF framework suggests Corbet [2018]. However, kernel programming requires a high level of programming skills, and eBPF is restrictive. A different approach would be to trigger a user-space event from the interrupt, but this would require an additional context switch. A context switch in some cases is time-consuming. We show later that a context switch is over 10 μs in our evaluation hardware. To make sure that the program code and data are always accessible and resident, it is essential to disable evacuation of the program's translation table and cache from the processor. Therefore, we chose to constantly accommodate (cache) the code and data in the hypervisor translation registers Penneman *et al.* [2013]

(Figure 3) in EL2 cache and TLB. To map the user-space program, we modified the Linux ARM-KVM, Dall and Nieh [2014] mappings infrastructure to map a user-space code with kernel space data.



Two Exception Levels access the same physical frame with the same virtual address of some process. However, the page tables of the two exception levels are not identical.

Figure 3: Asymmetric dual view

Figure 3 demonstrates how identical addresses may be mapped to the same virtual addresses in two separate exception levels. The dark shared section is part of EL2 and therefore accessible from EL2. However, when executing in EL2, EL1 data is not accessible without previous mapping to EL2. Figure 3 presents the leverage of a Linux process from two exception levels to three.

The natural way of memory mapping, depicted in Figure 4, is that EL1 is responsible for EL1/EL0 memory tables and EL2 is responsible for its memory table, in the sense that each privileged exception level accesses its memory tables.

The boxes represent the page tables of EL1/EL0 and EL2. Each page table is managed by its exception level. The small box represent the page that contain the page table.

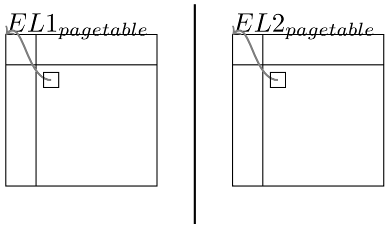


Figure 4: Memory Table access

However, this would have put the nanovisor at risk, as it might overwrite or otherwise garble its page tables. As noted earlier, on ARM8v-a hypervisor has a single memory address space. (unlike TrustZone that has two, for kernel and user). The ARM architecture does not coerce an exception level to control its memory tables. This makes it possible to map EL2 page table in EL1 (Figure 5). Therefore, only EL1 can manipulate the nanovisor page tables. We refer to this hyplet architecture as a Non-VHE hyplet. Also, to further reduce the risk, we offer to run the hyplet in injection mode. Injection mode means that once the hyplet is mapped to EL2, the encapsulating process is removed from the operating system kernel, but its hyplet's pages are not released from the nanovisor, and the kernel may not re-acquire them. It is similar to any dynamic kernel module insertion.

In processors that support **VHE** (Virtual Host Extension), EL2 has an additional translation table (Figure 6), that would map the kernel address space.

The boxes represent the page tables of EL1/EL0 and EL2. These page tables are distinct. Here, however, EL2 page tables are controlled by EL1 kernel. The small low box in EL1 page table contains the page table of EL2

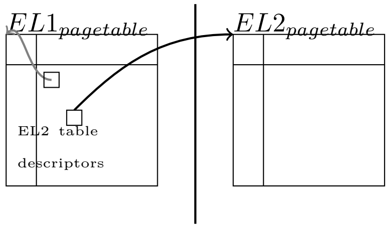


Figure 5: Non-VHE hyplet

In a VHE hyplet, it is possible to execute the hyplet in the user-space of EL2 without endangering the hypervisor. Figure 6 shows how a hyplet of a Linux process in $EL0_{EL1}$ (EL0 is EL1 user-space) is mapped to $EL0_{EL2}$ (EL2 user-space). Also, the hyplet can't access EL2 page tables because the table is accessible only in the kernel mode of EL2. VHE resembles TrustZone as it has two distinct address spaces, user and kernel. Operating systems such as QSEE (Qualcomm Secure Execution Environmen) Loftus *et al.* [2017] and OP-TEE opt are accessed through an upcall and execute the user-space in TrustZone. Unfortunately, at the time of writing, only modern ARM boards offer VHE extension Penneman *et al.* [2013] (ARMv8.2-a) and therefore this paper demonstrates benchmarks on older boards.

Hyplet in VHE mode. The hypervisor's kernel maps the hyplet (virtual address 0x400000) from Normal mode to HYP user mode. The nanovisor is secured.

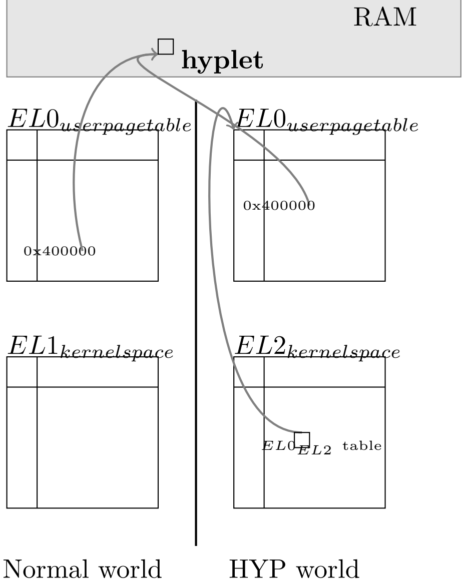


Figure 6: VHE Hyplet Architecture

3.1 The hyplet security & Privilege escalation in RTOS

As noted, VHE hardware is not available at the time of this writing, and as such we are forced to use software measures to protect the hypervisor.

On older ARM boards it can be argued that a security bug at hypervisor privilege levels may cause greater damages compared to a bug at the user process or kernel levels thus poising system risk.

The hyplet also escalates privilege levels, from exception level 0 (user mode) or 1 (OS mode) to exception level 2 (hypervisor mode). Since the hyplet executes in EL2, it has access to EL2 and EL1 special registers. For example, the hyplet has access to the level 1 exception vector. Therefore, it can be argued that the hyplet comes with security cost on processors that do not include ARM VHE.

The hyplet uses multiple exception levels and escalates privilege levels. So, it can be argued that using hyplets may damage application security. Against this claim, we have the following arguments.

We claim that this is risk is superficial and an acceptable risk, for processors without VHE support. Most embedded systems and mobile phones do not include a hypervisor and do not run multiple operating system.

In the case where no hypervisor is installed, code in EL1 (OS) has complete control of the machine. It does not have lesser access code running in EL2 since no EL2 hypervisor is present. Likewise code running in EL2 can affect all operating systems running under the hypervisor. Code running in EL1 can only affect the current operating system. When only one OS is running the two are identical.

Therefore, from the machine standpoint code running in EL1 when EL2 is not present has similar access privileges to code running in EL2 with only one OS running, as in the hyplet use case.

The hyplet changes the system from a system that includes only EL0 and EL1 to a system that includes EL0, EL1, and EL2. The hyplet system moves a code that was running on EL1 without a hypervisor to EL2 with only one OS. Many real-time implementations move user code from EL0 to EL1. The hyplet moves it to EL2, however, this gains no extra permissions, running rogue code in EL1 with no EL2 is just as dangerous as moving code to EL2 within the hyplet system. Additionally, it is expected that the hyplet would be a signed code; otherwise, the hypervisor would not execute it.

The hypervisor can maintain a key to verify the signature and ensure that lower privilege level code cannot access the key. This was shown by Resh and Zaidenberg [2013] on Intel platform.
Furthermore, Real-time systems may eliminate even user and kernel mode separation for minor performance gains. We argue that escalating privileges for real performance and Real-time capabilities is an acceptable on older hardware without VHE where hyplets might consist of a security risk. On current ARM architecture with VHE support the hyplet do not add extra risk.

3.2 Static analysis to eliminate security concerns

Most memory (including EL1 and EL2 MMUs and the hypervisor page tables) is not mapped to the hypervisor. The non-sensitive part of the calling process memory is mapped to EL2. The hyplet does not map (and thus has no access to) kernel-space code or data. Thus, the hyplet does not pose a threat of unintentional corrupting kernel's data or any other user process unless additional memory is mapped or EL1 registers are accessed.

Thus, it is sufficient to detect and prevent access to EL1 and EL2 registers to prevent rogue code affecting the OS memory from the hypervisor. We coded a static analyzer that prevents access to EL1 and EL2 registers and filters any special commands.

We borrowed this idea from eBPF. The code analyzer scans the hyplet opcodes and checks that are no references to any black-listed registers or special commands. Except for the clock register and general-purpose registers, any other registers are not allowed. The hyplet framework prevents new mappings after the hyplet was scanned to prevent malicious code insertions. Another threat is the possibility of the insertion of a data pointer as its execution code (In the case of SIGBUS of SEGFAULT, the hyplet would abort, and the process terminates). To prevent this, we check that the hyplet's function pointer, when set, is in the executable section of the program.
Furthermore, the ARM architecture features the TrustZone mode that can monitor EL1 and EL2. The TrustZone may be configured to trap illegal access attempts to special registers and prevent any malicious tampering of these registers.
Now, we present the Offline Nanovisor programming model.

3.3 Programming model

The offline hyplet requires modifications in the native C code. A hyplet'ed program that interacts with hardware devices accesses these devices through the kernel in the offline processor. A common hyplet is one that constantly exits the nanovisor to the kernel and from the kernel back into the nanovisor. Another possibility is for programmers who are acquainted with eBPF programming and can write an abstraction for the kernel part in eBPF. Thus, porting of a real-time program includes kernel programming and userspace programming. In general, whenever there is need for an operating service, we exit the nanovisor to the GPOS. Algorithm 2 demonstrates real porting; it is the simplified implementation of the ultrasonic sensor code we use later in this paper. The code accesses two different GPIOs (general purpose I/O), so it needs to perform some system calls. To replace system calls, the nanovisor exits to the kernel and performs the required service in kernel mode. An exit to the kernel is done through the use of *ERET* and the return back to the nanovisor is done by *HVC*. Therefore, the programming model requires that the program must change and be broken down to hyplets and offlets, which are then called sequentially, thereby maintaining the program state. For instance, in Algorithm 3, we perform the I/O in kernel mode in an offlet and then return to the hyplet to process the new data in user mode. Each line of code in Algorithm 3 is prefixed with the exception level the processor is in when it executes it. The programmer does not need to perform *ERET* or *HVC* by themselves. The programmer registers a user callback when they write the kernel portion. When the callback returns,

Algorithm 2: Native C program for an ultrasonic sensor

```
 // triggers the sound wave;
write(fd_trig, ”1”, 1)
do {
// Wait for the wave transmission
    read(fd_echo, &in, 1) ;
    t1 = get_time();
} while (in == 0);
do {
// Wait for the wave end of reception
    read(fd_echo, &in, 1);
    t2 = get_time();
} while (in == 0);
// Return the delta to the user program
return t2 - t1 ;
```

Algorithm 3: Hyplet ultrasonic

```
EL2: long hyplet_timer(long ts) {
// stash the time stamp and wait ..
EL2:      timestamps[i]=ts;
EL2:      wait(N microseconds); // Exit the
  nanovisor(EL2). Move to EL1;
EL2: }
...
EL1: long user_callback()
EL1 {
EL1:      gpio_set_value(gpio_trig, val);
EL1:      do {
EL1:         val = gpio_get_value(gpio_echo);
EL1:         t1 = get_time();
EL1:      } while (in == 0);
EL1:      do {
EL1:         in = gpio_get_value(gpio_echo) ;
EL1:         t2 = get_time();
EL1:      } while(in == 1) ;
// The callback finished. The framework enters
  the nanovisor and the arguments are passed to
  the hyplet.
EL1: }
...
EL2: long hyplet_timer(long ts) { // stash the
  time stamp and wait...
```

the framework returns to the nanovisor that runs the next hyplet.

The hyplet framework offers the following APIs:

- Memory mapping
  hyp_map(address,size)
  hyp_map_all()
  hyp_unmap(address)
  Maps or unmaps regions of code or data. It is also possible to map the entire process's address space, but we discourage it because usually the address space grows during the process life.

- Stack assignment
  hyp_set_stack(addr,size)
  A user should map some memory chunk as a stack.

- vma mapping
  hyp_map_vma(addr, size)
  Maps an address only if it is a vma. A vma (virtual memory area) is a virtual contiguous memory Bovet and Cesati [2005] with the same properties (read/write/execute).

- Offlet activation
  hyplet_drop_cpu(cpu_id)
  Unplugs a processor from the kernel. Uses Linux standard hotplugging API, i.e. we utilize Linux sysfs to remove a processor.

- Hyplet assignment
  hyp_assign_offlet(cpu_id, user function)
  hyp_unassign_offlet(user function)
  The userspace provides a function as an hyplet to execute in cpu cpu_id. Once the hyplet is assigned to the offline processor, it would run.

- Synchronisation
  hyp_lock(spinlock),hyp_unlock(spinlock)
  In cases where the programmer wishes to protect a resource from concurrent access from EL0 and EL2 or concurrent access from two offlined processors.

- Get time
  hyp_gettime()
  Returns the current time in nanoseconds. It is the value of the cntvct_el0 register that holds the current clock value.

- Printing
  hyp_print(const char* fmt,...)
  Prints in hyplet context.
  print_hyp()
  Records the print string and the values to a temporary buffer in EL2. When the program is in EL0, it should call print_hyp to print the data to the program's standard output as if it were a regular C's printf(3).

- Event
  hyp_wait()
  Waits for the completion of the offlet. *hyp_wait()* is similar to the UNIX wait(2) system call. There is no restriction on the number of callers.

EL1 data are passed to and from the hyplet through the function's arguments.
When the process exits, the hyplet is automatically removed from the nanovisor. At this point, the processor remains offline, waiting for a new assignment.

```
while (offlet_assigned == 0) {
    pause();
}
```

In the kernel, the offlet API uses *struct hyp_wait* that contains the user callback and some additional context information. We provide two APIs for offlets: *offlet_regiser(hyp_wait, cpu_id)* and *offlet_unregister(hyp_wait, cpu_id)*. These two APIs may be used after a processor is unplugged or before.

Delicate mapping
There are times when we want to map only certain global variables and functions to the nanovisor. For this, we used gcc sections. For example:

```
__attribute__ ((section("hyp"))) unsigned int a = 0;
unsigned int b = 0;
```

In this case, we want only to map the variable 'a' and not 'b'. So, we grab the ELF (Executable Linkable Format) section 'hyp' and map it to the nanovisor. The Offline Nanovisor offers a library that reads the program ELF structure and maps the required sections. For example, the below maps section 'hyp'.

```
Elf_parser_load_memory_map("myprogram")
get_section_addr("hyp", &hyp_sec, &sec_size);
hyplet_map_vma((void *)hyp_sec, cpu_id)
```

3.4 Runaway hyplet

If the hyplet is locked in an endless loop, We need to intervene in the code executing in the processor. For this, we offer to write on the entire EL2 user-mapped code through a different processor an opcode that generates an abort. Once the offline processor executes this code, the processor aborts, records the program counter and exits gracefully from EL2. For the programmer to understand an infinite loop exists, the hyplet calls a function that increments a counter. This counter is shared between the EL2, EL1 and EL0. The value of this counter and the value of the program counter are visible through the Linux procfs file system. So, if this counter is not updated, the programmer can quickly locate the infinite loop and its position in the program. The claim that a hyplet may change its own MMU tables is correct, but for the hyplet to access restricted data, the hyplet must know which pages to access and for this it usually needs access to the kernel.

3.5 Soft microcontroller

The Offline Nanovisor is a software microcontroller. Instead of having the operating system kernel serve an interrupt from an external microprocessor, the hyplet is the one that decides whether or not to interrupt the **user process**. In the phase of research and development, a soft microprocessor can be used during a proof-of-concept phase and before the electronics design, thus reducing costs.

Furthermore, some devices do not have any interrupts at all. Instead of receiving an interrupt, the device driver cyclically accesses the device. However, there is no way to know when data are available except by regularly accessing it in a tight loop. Cyclic pulling of the device data implies jitter when performed by the GPOS because we may not write a tight loop un-interrupted as noted previously.

3.6 AMP cache thrashing

AMP, or Asymmetric Multiprocessing Architecture, is when the system cores run different operating systems (a notable example is the Jailhouse microvisor Blackham *et al.* [2011]). AMP cache thrashing occurs when a GPA (guest physical address) for two different guests is mapped to different host physical addresses. The GPA might map to the same L2 or L3 cache lines due to cache associativity. Because the two guests do not have the same data in the GPA, they disturb each other, which degrades performance. This problem does not happen in the Offline Nanovisor because the hyplets use EL2 cache lines and not EL1's, and the offline processor is not virtualised.

3.7 Concurrency

Linux supports unplugging multiple processors. Thus, it is possible to load the Offline Nanovisor on multiple processors concurrently. The Offline Nanovisor API provides a synchronisation primitive, a spinlock, for the case where a process is shared between multiple processors' cores.

4 Evaluation

We demonstrate different types of experiments. The first is a software timer in which we show the accuracy of the Offline Nanovisor. In the next two sections, we demonstrate how some systems today do not reflect accurately enough the natural non-discrete physical world. The two experiments show that even when we use microcontrollers connected to a small computer that runs Linux, we lose accuracy due to the nature of the Linux operating system. We conduct our experiment with off-the-shelf devices, many of which are used in real products and can be reproduced.

We conducted measurements on a Raspberry Pi3 running a standard Linux OS. For practical reasons, we compare the Offline Nanovisor only to Linux and to technologies available in ARM. Dune Belay *et al.* [2012] and Rump Kernels Kantee and others [2012] are not available on ARM at the time of writing; RTAI (last released in Feb 2018) is not available for PI and not for 64-bit ARM.

Table 1 summarizes the Raspberry Pi3 main specifications:

Soc	Broadcom BCM2837
CPU	4 cores, ARM Cortex A53, 1.2GHz
RAM	1GB LPDDR2 (700 MHz)
Oscillator	19.2 Mhz

Table 1: Pi3 Specifications

It is also important to note that the Pi's clock inaccuracy is up to 140 ppm.

4.1 Latency

Linux handles interrupts Regnier *et al.* [2008] in two parts:

top half that acknowledges the interrupt

bottom half that handles the interrupt

So, before we start with the experiments, we first need to understand the variation in the Pi3 interrupt's latency before acknowledgement of the interrupt itself, i.e. the top half. As the source of the interrupt, we used an Invensense mpu6050 Fitriani *et al.* [2017] IMU (inertial measurement unit) to the Pi, and configured it to work in the i2c protocol. In i2c, for every 8 bits of data, there is an acknowledgement signal that generates an interrupt to the Pi. The acknowledgement signal is sent on the SDA line (the data line of the i2c) and received by the operating system kernel. We wanted to measure the time interval between the moment of the **i2c ACK** and the moment the processor runs the main interrupt routine. Therefore, we connected a logic analyzer probe to the SDA of the IMU device. We programmed one of the Pi's GPIOs to trigger a signal in the kernel's main interrupt routine. The results were an average of 3.9 μs, a maximum 9μs and a minimum 1.7 μs. Therefore, interrupts reach the service routine at varying times. Our results emphasise the fact that there are occasions when we cannot rely on an interrupt to reach the kernel in a predictable time. In other words, while a jitter of 10 μs in a 1 ms cycle may be considered tolerable in some cases, in a 100 μs cycle it is not.

5 Use cases

Next, we demonstrate several use cases.

5.1 Timer

We first evaluate other real-time operating system technologies for ARM and check their jitter at 1 kHz tick rate. We evaluate seL4 Wang *et al.* [2015] and Xvisor Patel *et al.* [2015] because they offer RTOS for ARM, Linux RT_PREEMPT RTOS, a standard Debian Linux and the Offline Nanovisor. Standard Linux is an off-the-shelf Debian Linux without any kernel modifications. The test is a simple 1-millisecond timer program written in C for each operating system.
In Figure 7, other than seL4 and the hyplet (the Offline Nanovisor), none of the other solutions can handle hard real-time. We note that we performed this test on an idle system without any real load, and took time samples from the system clock.

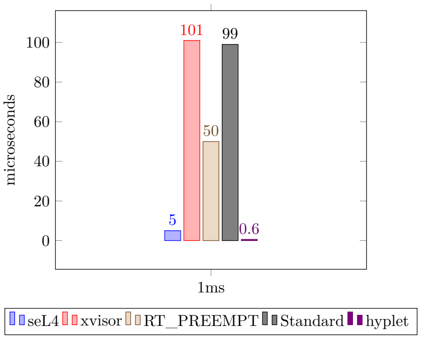


Figure 7: Maximum jitter in μs in idle mode of various operating systems

We did not continue the evaluations in higher frequencies, such as 20 kHz, because at these rates the operating systems are not responsive. We now want to evaluate the Offline Nanovisor. The Offline Nanovisor latency must resemble a real microcontroller latency. For example, Renesas MCU (microcontroller unit) MC16C/62P, has a 24 MHz processor, and according to Anh and Tan [2009], when it uses uTkernel, its latency from interrupt to task is about 1 μs. So, to be a competitive substitute, we aim for a few microseconds interrupt to task latency. As we will soon see, we reach this goal.

The offlet-hyplet test program is a timer function in HYP mode that waits for the time to expire, returns to EL1 to toggle a GPIO in EL1 and then returns to EL2 to wait again.

We measured latencies at various intervals. We constructed each test in two modes: An *idle mode* - While the processors are mostly idle and a *busy mode*. When in *busy mode* we flooded the network interface card (100 MbE) with network traffic to disturb the processors and consume bus cycles. The rate of interrupts was 3000 INTR/second, whereas in idle mode the network generates about 300 INTR/second.

We used an oscilloscope to achieve nanosecond accuracy. Figures 8 and 9 present the jitter in submicroseconds.

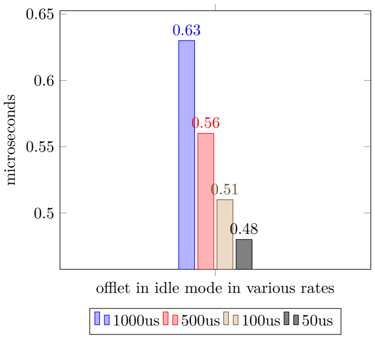


Figure 8: Maximum jitter in μs in idle mode

As noted, we toggle a GPIO for the measurements in EL1. From Figure 8, it appears that the transition from EL2 to EL1 and the GPIO toggle takes about 0.5 μs. Next, in Figure 9, we loaded the system by flooding the network card in order to examine the efficiency of the isolation.

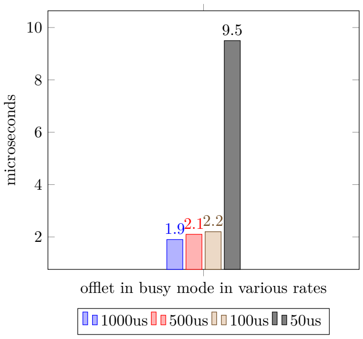


Figure 9: Maximum jitter in μs in busy mode

From Table 9 it is evident that a load in these frequencies is substantial. In the frequency of 20 kHz the worst case was 9.5 μs, which is nearly 20% of the cycle. The reason for the jitter is the error of the oscillator, which is 140 ppm and it is sensible that it is more observed at high frequencies. Table 2 presents some additional measures in busy mode.

Freq kHz	Avg	Stddev
1	1000165	170
2	500082	162
10	100016	150
20	50008	162

Table 2: Offline hyplet jitter (nanoseconds)

We see that bus sharing should be avoided as much as possible. Even if the process is entirely isolated from the GPOS there is still degradation in performance. The maximum jitter increases in general by 4 in low rates (up to 10 kHz). We also understand that the oscillator ppm deviation must be taken into consideration in high frequencies.

5.2 Ultrasonic distance

In this test, we used an ultrasonic sensor to measure the distance between the sensor (a consumer-grade HC-SR04) and an object. The sensor reports when the ultrasonic signal hits the echo sensor. Thus, by measuring the time elapsed between sending and receiving an ultrasound signal, we can deduce the distance. For this reason, it is important to measure the time as accurately as possible. Figure 10 shows an object located 30 cm from the sensor. The distance was chosen so that the duration would surpass 1 millisecond.

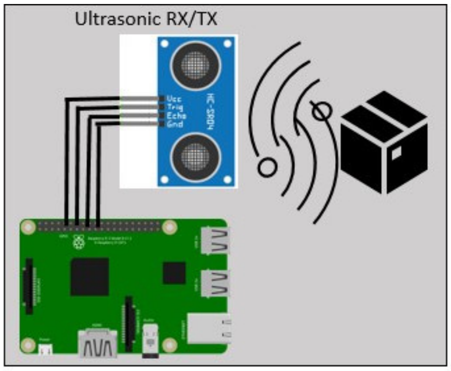


Figure 10: Ultrasonic schema

The speed of an ultrasonic sound wave is $34,300\frac{cm}{s}$. Thus, assign the following:
Δt as the time elapsed between triggering and receiving.
d as the distance from the sensor to the object.
The calculation of d is:
$d = \Delta t \times 34,300/2$
To measure a distance with a sensor, we applied this relation in Algorithm 4, and implemented a program as an offline hyplet and as a native C userspace code. Table 3 displays the results of the native C program in RT_PREEMPT and in a hyplet mode. In this experiment, we did not impose any CPU load.

Test	Average	Stdev	Min	Max
RT_PRPT	30.2	1.2	29.2	32.4
hyp-offlet	30.4	0.04	30.3	30.4

Table 3: Ultrasonic, idle mode (in cm)

Now, we repeat the test but with disturbances. As in the timer test, we generate a large number of interrupts while performing the test ($3000\frac{INTR}{sec}$). Table 4 shows the averages and standard deviations.

Algorithm 4: Ultrasonic distance algorithm

```
GPIO.Output(GPIO_TRIGGER) = True;
Sleep 100 us;
GPIO.Output(GPIO_TRIGGER) = False;
Result: Transmit for a 100us
StopTime = StartTime = time();
Result: Take start time
while(GPIO.input(GPIO_ECHO) == 0) {
    StartTime = time();
}
Result: When the echo GPIO is 1 the
        transmission began
while (GPIO.input(GPIO_ECHO) == 1) {
    StopTime = time();
}
Result: When ECHO is 0 again then the
        ECHO signal is received completely
TimeElapsed = StopTime - StartTime;
distance = (TimeElapsed * 34300) / 2;
```

Test	Average	Stdev	Min	Max
RT_PRPT	30.8	1.86	27.4	32.6
hyp-offlet	30.4	0	30.4	30.4

Table 4: Ultrasonic, busy mode (in cm)

The offset from 30 cm is probably due to the HCR sensor itself. Its oscillator is 40 kHz (25 μs), so the expected deviation is at most $0.000025 \times 343 = 0.00875 \approx 9mm$.

The Offline Nanovisor overcomes RT_PREEMPT.

5.3 Infrared sensor

An infrared sensor setup is composed of a light trigger and an echo object. Usually, the sensor is connected to a GPIO , and the program reads it constantly.
To evaluate the accuracy of the Offline Nanovisor, we designed the experiment depicted in Figure 11. A beam is projected to a photo-resistor connected to a GPIO pin in a Raspberry Pi, which reads the digital

value of the photo-resistor. In this experiment, we measured the interval between the moment we turn on the beam and the moment the photo-resistor raises the GPIO input to 'high'. It is hard to estimate the actual theoretical value of the expected result. We placed the beam projector 2 mm from the resistor. Light travels nearly at 300mm in 1 nanosecond. So, in theory we should have seen numbers less than one nanosecond; however, the actual numbers we got are much higher due to the PI's microcontroller delay. But we do see that the standard deviation differs.

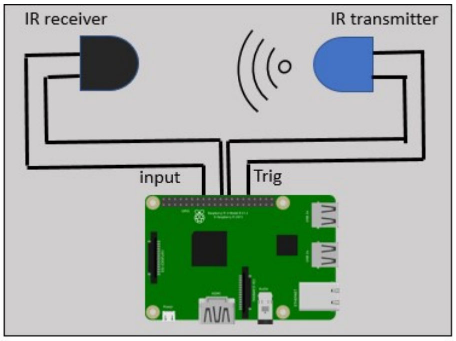


Figure 11: Infrared schema

Table 5 presents the difference between the native C program and its hyplet variant.

Test	Average	Stdev	Min	Max
RT_PRMPT	2077	20.5	2118	2056
hyp-offlet	2122	0.5	2122	2123

Table 5: Infrared (in μs)

Again, it is evident that the offline hyplet's programs are more accurate than native programs.

6 Related Work

Many groups have researched real-time operating systems (RTOS). Under the ARM architecture, other than RT_PREEMPT, we can find Jailhouse Baryshnikov [2016], Xen Barham *et al.* [2003], seL4 and OKL4 Heiser and Leslie [2010], Xenomai and RTAI, and many others. Of the six, OKL4 is a closed-source microvisor not available to us and we do not discuss it.

As noted earlier, this paper does not claim that hyplets or the Offline Nanovisor are kernels but rather a nanovisor extension to the GPOS. Moreover, the Offline Nanovisor may share the same computer with other technologies, such as seL4, as long as they do not use the same core. Nevertheless, we do provide some overview of related technologies, starting with the microkernel. A microkernel is defined as the minimum set of functionalities needed to implement an operating system. Jochen Liedtke best describes it: 'A concept is tolerated inside the microkernel only if moving it outside the kernel, i.e., permitting competing implementations would prevent the implementation of the system's required functionality'.

SeL4 is a microkernel for ARM8, ARM7, ARM6 and x86, and in some cases it is implemented as a microvisor. SeL4 architecture evolved from Liedtke's L4 microkernels family Elphinstone and Heiser [2013]. SeL4 provides a minimal set of functions to access a physical address space, interrupts, and processing time. It is also considered to be the only formally verified operating system Klein *et al.* [2009]. In general, seL4 is considered the fastest microkernel. Its support for SMP is considered experimental. SeL4 offers real-time Blackham *et al.* [2011] and security Sewell *et al.* [2011]. The weak part of seL4 is, however, its assimilation to existing hardware. It requires mastering CAmkES Kuz *et al.* [2007], a software component for microkernel-based embedded systems and a framework to build an operating system.

Jailhouse is a microvisor created by Siemens™. It uses partitioning to split the hardware into isolated compartments, or 'cells'. These cells are dedicated to executing programs called "inmates" Jailhouse does not emulate any device. Devices and processors are statically assigned to Jailhouse on creation. Unfortunately, Jailhouse, like KVM, does not run on Raspberry PI3. In Jailhouse, a single processor is assigned to perform the hard real-time tasks while the other processors are assigned to run Linux. Jailhouse is implemented as a Linux kernel driver and is a type-2 hypervisor, like the hyplets. Sebou Soltesz *et al.*

[2007] show that Jailhouse's performance surpasses Xen's. This is mainly due to its simple design.

Other known open-source real-time operating systems for Linux, are Xenomai Gerum [2004] and RTAI Mantegazza *et al.* [2000]. Both technologies employ a microkernel architecture, meaning that the Linux kernel is merely a background task. Both technologies run on most processor architectures, e.g. x86, ARM7 and Power ISA. According to Barham *et al.* [2003], these two technologies perform somewhat the same while RTAI is a bit faster. However, as noted earlier, RTAI is not available on Raspberry PI and, as such, we feel that its development lags.

Dune Belay *et al.* [2012] is a system that provides a process rather than a machine, an abstraction through virtualisation. Dune offers a sandbox for untrusted code, a privilege separation facility and a garbage collector and is implemented on the Intel architecture. It is intended more for security than for RTOS. We believe we can port hyplets to Dune for ARMv8-a and ARMv7-a.

Rump kernels Kantee and others [2012] are virtual lightweight containers for drivers in NetBSD Mewburn [2001]. Rump kernels run on top of the hypervisor and are processes running in hypervisor mode, and wrapped by containers that enable driver operations such as threads and synchronisation primitives. Rump kernels are designed for running drivers with little if any modification and still leave the kernel monolithic.

The Extended Berkeley Packet Filter, also known as eBPF Corbet [2018] Borkmann [2016] is described as an in-kernel virtual machine that provides the ability to attach a program to a certain tracepoint in the kernel. Whenever the kernel reaches the tracepoint, the program is executed without a context switch. eBPF is undergoing massive development and is mainly used for packet inspection, tracing and probing. EBPF supports x86 architectures and ARM (although we failed to compile eBPF for ARMv8-a). It runs in kernel mode, which is considered unsafe, but uses a verifier to check for illegal accesses to kernel areas or the tampering of registers. Access to the userspace is done through memory maps.

EBPF uses LLVM, requires clang to generate a JIT code and has a quite small instruction set. As a consequence, eBPF has substantial limitations as only a subset of the C language can be compiled into eBPF. EBPF has no loops, no native assembly, no static variables, no atomics, may not take a long time and is restricted to 4,096 instructions. Each eBPF instruction is 64-bit, so the biggest eBPF program may reach the size 4096 * 8 = 32 KB. But this is a byte code. Thus, in addition to the program size, there is also the overhead of LLVM. The hyplet runs native opcodes. Numerous vulnerabilities in an eBPF program might jeopardize the operating system. This is not the case with hyplets. The hyplet is not a program that executes in the kernel's address space but in the user's address space. Hence, there is no need for maps to share data between the user and the kernel. Hyplets do not require any particular compiler extensions, are much less restricted (what mapped prematurely can be accessed). Hyplets are meant to propagate events to a userspace program and process them in real-time, not just collect data as in eBPF.

7 Summary

The approach of embedding two distinct instances of two or more operating systems in a single computer has many justifications. However, this approach has some flaws. First, these operating systems require the programmers to master at least two operating systems: Linux and the microkernel's RTOS. The learnability of an operating system, its maintainability and portability are what make RT_PREEMPT so popular. RT_PREEMPT is easy and it is Linux. Just to shed some light, the RT_PREEMPT patch was evaluated (and thus promoted) by Cerqueira and Brandenburg [2013]; Regnier *et al.* [2008]; Arthur *et al.* [2007]; Mossige *et al.* [2007] , and is known today as the operating system taught in universities McLoughlin and Aendenroomer [2007].

Second, when we embed two operating systems on the same machine, we do that for a certain purpose. For instance, to have a GPOS that can run a GUI (graphical user interface) program that consumes data from the RTOS. Therefore some form of communication between the two operating systems is required. The communication between the two operating system needs to be maintained and synchronized. Thus, programmers need to be careful that it does not jeopardize the RTOS responsiveness. The Offline Nanovisor is intended to ease these challenges.

8 Conflict of Interest

The authors declare that they have no conflict of interest."

References

Tran Nguyen Bao Anh and Su-Lim Tan. Real-time operating systems for small microcontrollers. *IEEE micro*, 29(5):30–45, 2009.

Siro Arthur, Carsten Emde, and Nicholas Mc Guire. Assessment of the realtime preemption patches (rt-preempt) and their impact on the general purpose performance of the system. In *Proceedings of the 9th Real-Time Linux Workshop*, 2007.

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 *ACM SIGOPS operating systems review*, volume 37, pages 164–177. ACM, 2003.

Maxim Baryshnikov. Jailhouse hypervisor. B.S. thesis, České vysoké učení technické v Praze. Vypočetní a informační centrum., 2016.

Adam Belay, Andrea Bittau, Ali José Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe user-level access to privileged cpu features. In *Osdi*, volume 12, pages 335–348, 2012.

Raz Ben-Yehuda and Yair Wiseman. The offline scheduler for embedded vehicular systems. *International Journal of Vehicle Information and Communication Systems*, 3(1):44–57, 2013.

Raz Ben Yehuda and Nezer Zaidenberg. Hyplets-multi exception level kernel towards linux rtos. In *Proceedings of the 11th ACM International Systems and Storage Conference*, pages 116–117. ACM, 2018.

MD Bennett and Neil C Audsley. Predictable and efficient virtual addressing for safety-critical real-time systems. In *Proceedings 13th Euromicro Conference on Real-Time Systems*, pages 183–190. IEEE, 2001.

Wolfgang Betz, Marco Cereia, and Ivan Cibrario Bertolotti. Experimental evaluation of the linux rt patch for real-time applications. In *Emerging Technologies & Factory Automation, 2009. ETFA 2009. IEEE Conference on*, pages 1–4. IEEE, 2009.

Bernard Blackham, Yao Shi, Sudipta Chattopadhyay, Abhik Roychoudhury, and Gernot Heiser. Timing analysis of a protected operating system kernel. In *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, pages 339–348. IEEE, 2011.

Daniel Borkmann. On getting tc classifier fully programmable with cls bpf. *tc*, (1/23), 2016.

Daniel P Bovet and Marco Cesati. *Understanding the Linux Kernel: from I/O ports to process management.* " O'Reilly Media, Inc.", 2005.

Felipe Cerqueira and Björn Brandenburg. A comparison of scheduling latency in linux, preempt-rt, and litmus rt. In *9th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pages 19–29. SYSGO AG, 2013.

Jonathan Corbet. Bpf comes to firewalls, 2018.

Christoffer Dall and Jason Nieh. Kvm/arm: The design and implementation of the linux arm hypervisor. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 333–348, New York, NY, USA, 2014. ACM.

Soumya Kanti Datta, Christian Bonnet, and Navid Nikaein. Android power management: Current and future trends. In *Enabling Technologies for Smartphone and Internet of Things (ETSIoT), 2012 First IEEE Workshop on*, pages 48–53. IEEE, 2012.

Magnus Ekman, F Dahgren, and Per Stenstrom. Tlb and snoop energy-reduction using virtual caches in low-power chip-multiprocessors. In *Proceedings of the international symposium on Low power electronics and design*, pages 243–246. IEEE, 2002.

Kevin Elphinstone and Gernot Heiser. From l3 to sel4 what have we learnt in 20 years of l4 microkernels? In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 133–150. ACM, 2013.

Diah Ayu Fitriani, Wahyu Andhyka, and Diah Risqiwati. Design of monitoring system step walking with mpu6050 sensor based android. *JOINCS (Journal of Informatics, Network, and Computer Science)*, 1(1):1–8, 2017.

Philippe Gerum. Xenomai-implementing a rtos emulation framework on gnu/linux. *White Paper, Xenomai*, pages 1–12, 2004.

Gernot Heiser and Ben Leslie. The okl4 microvisor: Convergence point of microkernels and hypervisors. In *Proceedings of the first ACM asia-pacific workshop on Workshop on systems*, pages 19–24. ACM, 2010.

Antti Kantee et al. Flexible operating system internals: the design and implementation of the anykernel and rump kernels. 2012.

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220. ACM, 2009.

Ihor Kuz, Yan Liu, Ian Gorton, and Gernot Heiser. Camkes: A component model for secure microkernel-based embedded systems. *Journal of Systems and Software*, 80(5):687–699, 2007.

Ronan Loftus, Marwin Baumann, Rick van Galen, and Rachelle de Vries. Android 7 file based encryption and the attacks against it, 2017.

Anil Madhavapeddy, Thomas Leonard, Magnus Skjegstad, Thomas Gazagnaire, David Sheets, David J Scott, Richard Mortier, Amir Chaudhry, Balraj Singh, Jon Ludlam, et al. Jitsu: Just-in-time summoning of unikernels. In *NSDI*, pages 559–573, 2015.

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

Ian McLoughlin and Anton Aendenroomer. Linux as a teaching aid for embedded systems. In *Parallel and Distributed Systems, 2007 International Conference on*, volume 2, pages 1–8. Ieee, 2007.

Luke Mewburn. The design and implementation of the netbsd rc. d system. In *USENIX Annual Technical Conference, FREENIX Track*, pages 69–79, 2001.

Morten Mossige, Pradyumna Sampath, and Rachana Rao. Evaluation of linux rt-preempt for embedded industrial devices for automation and power technologies-a case study. In *9th RTL Workshop [Online]. Available: http://www. linuxdevices. com/files/article081/Sampath. pdf*, 2007.

Op tee , linaro limited: Open portable trusted execution environment.

Anup Patel, Mai Daftedar, Mohamed Shalan, and M Watheq El-Kharashi. Embedded hypervisor xvisor: A comparative analysis. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 682–691. IEEE, 2015.

Kolin Paul and Tapas Kumar Kundu. Android on mobile devices: An energy perspective. In *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*, pages 2421–2426. IEEE, 2010.

Niels Penneman, Danielius Kudinskas, Alasdair Rawsthorne, Bjorn De Sutter, and Koen De Bosschere. Formal virtualization requirements for the arm architecture. *J. Syst. Archit.*, 59(3):144–154, March 2013.

Paul Regnier, George Lima, and Luciano Barreto. Evaluation of interrupt handling timeliness in real-time linux operating systems. *ACM SIGOPS Operating Systems Review*, 42(6):52–63, 2008.

Amit Resh and Nezer Zaidenberg. Can keys be hidden inside the cpu on modern windows host. In *Proceedings of the 12th European Conference on Information Warfare and Security: ECIW 2013*, page 231. Academic Conferences Limited, 2013.

Thomas Sewell, Simon Winwood, Peter Gammie, Toby Murray, June Andronick, and Gerwin Klein. sel4 enforces integrity. In *International Conference on Interactive Theorem Proving*, pages 325–340. Springer, 2011.

Mohamed Shalan and Dina El-Sissy. Online power management using dvfs for rtos. In *Design and Test Workshop (IDT), 2009 4th International*, pages 1–6. IEEE, 2009.

Unix signals, 2020.

Stephen Soltesz, Herbert Pötzl, Marc E Fiuczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 275–287. ACM, 2007.

Xiaolong Wang, Masaaki Mizuno, Mitch Neilsen, Xinming Ou, S Raj Rajagopalan, Will G Boldwin, and Bryan Phillips. Secure rtos architecture for building automation. In *Proceedings of the First ACM Workshop on Cyber-Physical Systems-Security and/or PrivaCy*, pages 79–90. ACM, 2015.