The hyplet - Joining a Program and a Nanovisor for real-time and Performance

Raz Ben Yehuda
University of Jyväskylä
Jyväskylä, Finland
raziebe@gmail.com

Nezer Jacob Zaidenberg
College of Management Academic Studies
Israel
scipio@scipio.org

***Abstract*—This paper presents the concept of sharing a hypervisor address space with a standard Linux program. In this work, we add hypervisor awareness to the Linux kernel and execute code in the HYP exception level through using the hyplet. The hyplet is an innovative way to code interrupt service routines and remote procedure calls under ARM. The hyplet provides high performance and run-time predictability. We demonstrate the hyplet implementation using the C programming language on an ARM8v-a platform and under the Linux kernel. We then provide performance measurements, use cases, and security scenarios.**

***Index Terms*—Hypervisor, real time, Linux, Virtualization, Security**

I. Introduction

There are various techniques to achieve real-time computing. One is to use a single operating system that provides real-time computing, such as standalone VxWorks or RT PREEMPT which is a Linux kernel extension. Another technique is the microvisor that co-exist with the general purpose operating system. A microvisor is an operating system that employs some characteristics of a hypervisor and some characteristics of a microkernel. A typical architecture of a microvisor. OKL4 [8] is an example for operating system that uses a microvisor. The hyplet (Figure 1) is a software, code and data, shared between a process and a nanovisor. It is a hybrid of a normal user program and a nanovisor that offers real-time processing and performance. A hyplet program (1) maps parts of code and data to the nanovisor, then it associates (2) the hyplet handler with an event, IRQ or RPC (or both). At this point (3), RPC that traps to the nanovisor, or IRQ (4) that upcalls the nanovisor would trigger an (5) hyperupcall to the hyplet.

We introduce the hyplet for the purpose of interrupt handling in user-space and for the purpose of efficient interprocess communication. The fast RPC and the user-space interrupts are done in a standard Debian, and with little modifications to the user program. This paper aims to provide Real-Time responsiveness to interrupts and fast RPC in cases where it is not cost-effective to run a standalone RTOS or Linux RT PREEMPT.

We present the hyplet ISR (Interrupt Service Routine) as a technique to reduce hardware to user-space latency. The hyplet is not aimed to replace a kernel space drivers in user-space, but to propagate the interrupt to a process. Thus, the hyplet can affect a driver's behavior. For example, should the driver process the packet or not. We will use the term hypISR to distinguish a normal ISR from an ISR in hyplet mode.

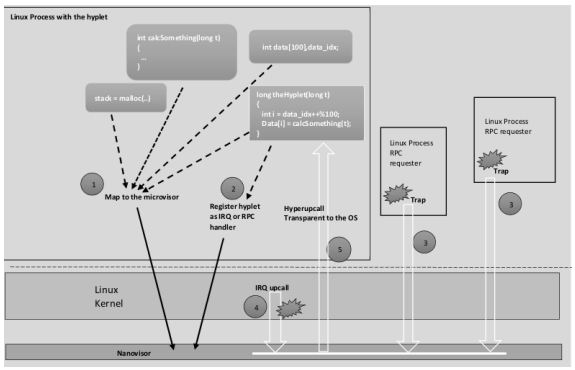


Fig. 1. The hyplet nanovisor

In addition to hypISR, we present hypRPC. HypRPC is a reduced RPC mechanism which has a latency of a sub-microsecond on average, and 4 microseconds worst case. Our RPC is a type of a hypervisor trap where the user process sends a procedure id to the hypervisor to be executed with high privilege without interrupts in another process address space. We use the term hypRPC for our RPC as a mixture between hyperuprcall and RPC.

One motivation for hypRPC is safety. In many cases, Real-Time programmers tend to encapsulate most of the software functionality in a single address space, so that the various threads would easily access shared data. However, this comes with the cost of a single thread crashing the entire process in case of an error in this thread. Through the use of the hypRPC we can separate a single multi-threaded process to multiple processes. Other RPC solutions, as we show later in the paper, are slower.

The hyplet is based on the concept of a delicate address space separation within a process. Instead of running multiple operating systems kernels, the hyplet divides the Linux process into **two execution modes: HYP and Normal**. Part of the process (HYP mode) would execute in an isolated, non-interrupted privileged safe execution environment. The other part of the process would execute in a regular user mode (Normal mode). To summarize, the hyplet is meant to reduce the latency of hardware interrupt to a user-space program,

and program to program local communication in user-space programs to sub microsecond order of magnitude.
In the taxonomy of virtualization, hyplets are classified as bare metal type 2 hypervisors. A type 2 hypervisor is a hypervisor that is loaded by the host operating system. A type 1 hypervisor is a hypervisor which is loaded by the boot loader, prior to the general operating system (GPOS). The hyplet does not require to be a virtual machine; thus it may execute in hardware that does not have support for interrupts virtualization. It is meant to be simple to use and adapt to the existing code. It does not require any modifications to the boot loader, only minor changes to the Linux kernel.

II. Background

ARM has a unique approach to security and privilege levels that is crucial to the implementation of the hyplet. In ARMv7, ARM introduced the concept of secured and non-secured worlds through the implementation of TrustZone, and starting from ARMv7a. ARM presents four exception (permission) levels as follows.
Exception Level 0 (EL0) Refers to the user-space code. Exception Level 0 is analogous to ”ring 3” on the x86 platform.
Exception Level 1 (EL1) Refers to operating system code. Exception Level 1 is analogous to ”ring 0” on the x86 platform.
Exception Level 2 (EL2) Refers to HYP mode. Exception Level 2 is analogous to ”ring -1” or ”real mode” on the x86 platform.
Exception Level 3 (EL3) Refers to TrustZone as a special security mode that can monitor the ARM processor and may run a real-time security OS. There are no direct analogous modes, but related concepts in x86 are Intel’s ME or SMM.

Each exception level provides its own set of special purpose registers and can access these registers of the lower levels, but not higher levels. The general purpose registers are shared.
Microvisors, Microkernels, virtualization and para virtualizion are all possible in this architecture. Microvisors are operating systems that execute in EL2, Microkernels virtual memory management (and some other parts) are kept mostly in EL1, while the user services are kept in EL0. Virtualization is kept in EL2 and para virtualization is kept both in EL1 and EL2.

III. The hyplet

The hyplet is a native C function that runs in a nanovisor and a Linux process. It does not require any special compilation or pre-processing. But before diving into the technical details, we describe our motivation through use cases.

Trusted Interrupts
The hyplet can be used to mask the handling of an interrupt so that it will not be visible by the OS driver. Interrupts handled by the hyplet can be verified by TPM or TrustZone, and pose an extra layer of protection to reverse or modify, unlike OS-based interrupt handler.
To modify a normal OS interrupt it is sufficient to elevate privileges to the OS level. To modify a hyplet one must first elevate permissions to the OS level, and then attack and subvert the hypervisor itself.

The hyplet a malware detector
We showed that the hyplet RPC is the fastest in Linux. For this reason, we used this technology for C-FLAT [1]. C-FLAT is a run time remote attestation technique that detects malware-infected devices. It does so by recording a program runtime behavior and monitoring the execution path of an application running on an embedded device. [1] presents C-FLAT through the use of the TrustZone. We implemented C-FLAT through the use of hypRPC. We replaced the various branch opcodes with the trap opcode. This effort is completed, and due to the low overhead of the hypRPC, we can present this technology in Linux and not in bare metal as in [1].

Protection against reverse engineering
On x86 platforms, TrulyProtect provides anti-reverse engineering , end-point security, video decoding, forensics etc. TrulyProtect relies on Dynamic Root of Trust Measurement (DRTM) attestation to create a trusted environment in the hypervisor to receive encryption keys [12]. We have used the hyplet to implement a TrulyProtect-like system on the ARM platform. We have encrypted parts of the software and used the hyplet in order to switch context and elevate privileges. Our systems then decode the code in the hypervisor context (a hyplet), so that the code or decryption keys will not be available to the OS. Our system for protection against reverse engineering has a cost affiliated with first execution and decryption of the code, but very low per iteration overhead as demonstrated in the table below.

Iterations	Encrypted	Clear
1	1185	1127
10	2737	2597
100	18022	18018
1000	173925	171251
10000	1758997	1670811

TABLE I
Duration of stack access in ticks

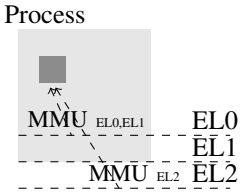
A. The hyplet explained

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 is 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
```

Fig. 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 [5]. 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 in EL2 cache and TLB. To map the user-space program, we modified the Linux ARM-KVM, [6] 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.

Fig. 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 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. 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. 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, that would map the kernel address space. In a VHE hyplet, it is possible to execute the hyplet in the user-space of EL2 without endangering the hypervisor. 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 Environment) and OP-TEE [18] 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 (ARMv8.2-a) and therefore this paper demonstrates benchmarks on older boards.

B. 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.
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.

C. 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.

D. The hyplet - User-Space Interrupt

In Linux and other operating systems, when an interrupt reaches the processor, it triggers a path of code that serves the interrupt. Furthermore, in some cases, the interrupt ends up waking a pending process.
The hyplet is designed to reduce the time from the interrupt event to the program. To achieve this, as the interrupt reaches the processor, instead of executing the user program code in EL0 after the ISR (and sometimes after the bottom half), a special procedure of the program (Figure 4) is executed in HYP mode at the start of the kernel's ISR.

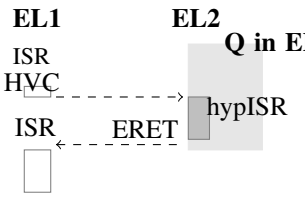


Fig. 4. HypISR flow

The hyplet does not change the processor state when it is in interrupt; thus, once the hyplet is completed, the kernel interrupt can be processed.
The hyplet does not require any new threads and because the hyplet is an ISR, it can be triggered in high frequencies. As a result, we can have high-frequency user-space timers in small embedded devices.
Some may argue that the hyplet should have been implemented as a virtual interrupt. However, many ARMv8-a platforms do not support VGIC (virtual Interrupt Controller). Raspberry PI3, for example, does not fully support VGIC (as a consequence, ARM-KVM does not run on a Raspberry PI3). Interrupts are then routed to the hypervisor by upcalls from the kernel main-interrupt routine, and the nanovisor communicates with the guest through a hyperupcall [2]. Nested hyplet interrupts are not possible.

E. Hypervisor based RPC

The remote procedure call is a type of interprocess communication (IPC) in which parameters are transferred in the form of function arguments. The response is returned as the function return value. The RPC mechanism handles the parsing and handling of parameters and the returned values. In principle, RPC can be used locally as a form of IPC and remotely over TCP/IP network protocols. In this paper, we only consider the local case.

IPC in real-time systems is considered a latency challenge. Thus system developers refrain from using IPC in many cases. The solution many programmers use is to put most of the logic in a single process. This technique decreases the complexity but increases the program size and risks.
In multicore computers, one reason for the latency penalty is because the receiver may not be running when the message is sent. Therefore, the processor needs to switch contexts.

HypRPCs are intended to reduce this latency to the sub-microsecond on average by eliminating the context switch (in some way the hyplet is viewed as a temporary address-space extension to the sending program).
If the receiving program exits, then the API immediately returns an error. If the function needs to be replaced in real-time, there is no need to notify the sending program; instead, the function in the hypervisor only needs to be replaced.

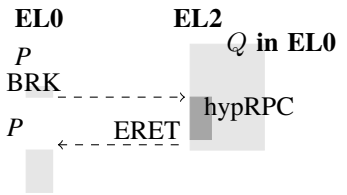


Fig. 5. HypRPC flow

Figure 5 demonstrates the hypRPC flow. Program P is a requesting program, and Q is a serving program. As program P loads, it registers itself as a hypRPC requesting program. A hypRPC program, unlike hypISR, is a program that when it executes the BRK instruction, it traps into HYP mode. The reason for that is that user-space programs are not permitted to perform the upcall instruction. When P calls BRK, the first argument is the RPC id, i.e; $x_0 = rpc_{id}$. As the processor executes BRK, it shifts to HYP mode. Then the hypervisor checks the correctness of the caller P and the availability of Process Q, and if all is ok, it executes in EL2 the function with this id.
The semantics of hypRPC is different from the common RPC. In a common RPC, the receiver is required to assign a thread to wait for the caller, and if a single thread is used, a serialized access provides protection. In Q, accessing a resource shared from the nanovisor and Normal world must be protected, even if Q has no threads. The protection is achieved by disabling context-switch and disabling access from another processor, using synchronization primitives we provide.

The hyplet has some additional benefits:
Safety
The hypISR provides a safe execution environment for the interrupt. In Linux, if there is a violation while the processor is in kernel mode, the operating system may stop. In the hyplet case, if there is a fault in the hypISR, the nanovisor would trigger a violation (for instance, a SEGFAULT). The nanovisor would send, through the kernel, a signal to the process containing the hyplet. This signal is possible because the fault entry of the nanovisor handles the error as if it is a user-space error. For example, if a divide-by-zero failure happens, the operating system does not crash, but the hyplet capable-program exits with a SIGFPE.
Some may claim that endless loops may hog the processor. For this, we argue that OP-TEE shares the same vulnerability because in OP-TEE the tee-supplicant blocks on a session until the TA (trusted application) finishes. However, we still offer to handle endless loops by replacing the hyplet code by the NOP (no operation) opcode. The hyplet will exit back to the nanovisor, the nanovisor will put back the original code and send a SIGKILL signal to the process.
We offer to handle code injection by caching the original code and overwriting it once an injection is detected. It is possible to checksum the hyplet code and see if it is changed, and if so re-write the original code [15].
Another facet of hypISR is sensitive data protection. We can use the hyplet to securely access data, and I/O data may be hidden from EL1 and accessible only in EL2.
Scope of Code Change
The hyplet patch does not interfere with the hardcore of the kernel code, and the hyplet patch does not require any modification to any hardware driver. The modifications are in the generic ISR routine. As a result, it is easy to apply this technology as it does not change the operating system heuristics. Microvisors, such as seL4 and Xvisor, are not easily applied to arbitrary hardware because they require a modified boot loader, making it impossible to apply the microvisors in some cases, for example, when the boot-loader code is closed (mobile phones). Jailhouse [3] and KVM do not even run on many devices because virtualization does not suffice (the GIC virtualization is incompatible) in some cases, and raspberry PI3 is an example of such virtualization hardware. In Android OS, it is undesirable to apply RT PREEMPT because it changes the entire operating system behavior. Our nanovisor does provide any service other than bridging to the hyplet. A user chooses to add services to the microvisor as part of the hyplet-utils library.

IV. Evaluation

We demonstrate that the hyplet is suitable for hard real-time systems. We provide synthetic microbenchmarks, and compare our solution to Normal Debian Linux, RT PREEMPT Linux(kernel version 4.4.92), seL4 microkernel (version 10.0.0.0) [10], and Xvisor (v0.2.11), all on a Raspberry PI3. PI3 main specifications are a 19.2 MHz, 4 ARM Cortex A53 1.2GHz cores and 1GB RAM 900 MHz.
We selected Xvisor because Xvisor is a thin microvisor. We chose RT PREEMPT because it is considered a free open source non-commercial RTOS Linux OS, and we chose seL4 because it is a hard real-time mathematically proven microkernel. The time units we use are due to the clock granularity, which is $\frac{10^9}{19.2*10^6} = 52$ nanoseconds per tick. We start by evaluating PI3 interrupt latency.

A. Interrupt Latency

To understand the possible time deviations in the Timer's test(in the next section), we start first by evaluating the latency of an interrupt in PI3.
We measured the delay from the attached hardware to the start of the hyplet. For this purpose, we connected an Invensense

mpu6050 to the PI, and configured this IMU to work in i2c protocol. In i2c, for every 8 bits of data, there is an acknowledgment signal, that generates an interrupt to the PI. We wanted to measure the time interval between the moment of the i2c ACK, to the moment the processor runs the main interrupt routine. So, we connected a logic analyzer probe to the SDA of the IMU and programmed one of the PI's GPIO to trigger a signal in the kernel's main interrupt routine. This way we could take the time of the IMU ACK signal, and the kernel ISR time. We generated the i2c signals in random times. The results were an average of 3.9 μs, a maximum of 9μs, and the minimum was 1.7μs.
First, we should not expect interrupts to be processed in deterministic times. A deviation of nearly 5 μs from the average (3.9 μs) is a lot. This can happen for various reasons, such as interrupts congestion or TLB latency and so on.
Also, this benchmark means that if we connect a device that ticks in a high frequency, such as 100Khz, two consequent interrupts may appear in 1.9μ delta. So, while the kernel can handle these interrupts in-accuracy, the user-space will miss the second interrupt.

B. Timer

We continue the evaluation and construct a hyplet timer. Table II presents the measured delay latencies of the timer programs in various operating systems. We conducted a delay of 1 ms for 5 minutes. In RT PREEMPT and Normal Raspbian Linux we used cyclictest, a real-time test suit for Linux. Cyclictest is a test software that measures timer latency in Linux. Cyclictest implements a sleeper thread, takes a time sample, goes to sleep, and when woken it records the time differences and goes back to sleep again. Cyclictest binds the waiting thread to a single processor.
Since Cyclictest is not available in seL4 and Xvisor, we wrote a timer microbenchmark that sleeps for 1 ms and records the time differences. In Xvisor, to make the test equal to the hyplet as much as possible, in terms of which privilege level the code was executed, our simple timer ran in HYP mode (not in the VM/guest OS).

ranges in μs	RT_ PRPT	Hyplet	Nrmal	Xvsr	seL4
0	0	99.9477	0	0	0
1	0	0.0523	0	0	0
2-5	0	0.0020	0	0	100
6-10	0	0	47.7	99.9	0
11-15	69	0	49.7	0	0
16-20	28	0	1.6	0	0
21-25	2	0	0.25	0	0
26-30	0.085	0	0.26	0	0
31-35	0.01	0	0.0874	0	0
36-40	0.05	0	0.034	0	0
41-45	0.001	0	0.034	0	0
46-50	0.0003	0	0.05	0	0
51-55	0	0	0.0321	0	0
56-100	0	0	0.18	0	0
101+	0	0	0.0014	.1	0

TABLE II
: LATENCIES DISTRIBUTION IN PERCENTAGE

Table II presents the delay deviations of each OS. The tests were conducted while the operating systems were idle. This is because it is not easy to load the operating systems and measure the load in all the operating systems we tested, and in some cases, the operating systems were not stable enough to sustain a load. For the analysis, we assume a deviation of approximately 5% soft real-time and below hard real-time.
In the hyplet case, 99.96% of the samples are below 1μs latency, and 100% are below 5μs. The deviation is probably because of the interrupt latency we showed earlier. The maximum latency of 9 μs is probably not reflected here, because, in this test, the interrupt source is the local timer. The deviation is below $\frac{5}{1000} = \frac{1}{2}\%$.
In the RT PREEMPT case, the upper boundary was 47μs, and 14μs on average. RT PREEMPT's deviation in PI3 is nearly $5\% = \frac{50}{1000}$. We consider RT PREEMPT on PI3 soft real-time.
In Normal Linux, the maximum value was 144μs, and the distribution of the values was higher. So the deviation is 14 %, which, as expected, shows that Normal Linux is a non-real-time OS.
Xvisor presents an impressive benchmark where 99.9% of samples jitter is less than 8 μs, the rest, unfortunately, were nearly 500 μs. Xvisor is not RTOS.
SeL4 is an RTOS. SeL4 produces a remarkable latency of less than 5 μs, approximately $\frac{1}{2}$ % maximum deviation.
To conclude, it is evident that hypISR can provide hard real-time in a regular Linux kernel, and since seL4 is not abundant software as Linux, hypISR can be used as a real-time solution in some cases.

C. Fast RPC

Here we demonstrate an RPC that eliminates context switches and therefore increases the remote call predictability. This section focuses on performance. We evaluated the round trip delay of calling a function that returns the time. For Xvisor, Native Linux and RT PREEMPT we used ptsematest, which is part of the Linux rt-test suite. Ptsematest measures interprocess latency communication with POSIX mutexes. Ptssematest starts two threads and synchronizes them with pthread_lock and pthread_unlock APIs. The receiving thread locks the mutex, and the sending thread releases the mutex. The time difference between the unlock to lock is the IPC duration.
In seL4 we used ptsematest-like test (sync.c) because ptsematest is not available in seL4. We used two threads, a consumer and a producer, the consumer waits on a semaphore (sync_bin_sem_t) and the producer signals the semaphore.
The hyplet test was a C program that made an RPC to a hyplet'ed process. The RPC returned the time stamp from the hyplet'ed process. The traveling time from the sender to the hyplet was recorded.
The reference test is to evaluate the cost of the function of the hyplet when not in HYP mode. We measured how much time it costs to call the function in the hyplet'ed process. Table III depicts the results. The tests were conducted on an idle system. The hyplet is the fastest RPC, even in the worst case. Xvisor

Name	Avg	Max
Ref	156ns	520ns
Hyplet	520ns	4.2μs
Normal	13μs	56μs
RT PRMT	15 μs	59μs
Xvisor	203μs	7067μs
seL4	8μs	17μs

TABLE III
ROUND TRIP RPC

results are the worst, it seems that its hypervisor preempts the OS for long durations. SeL4 RPC is on average is 13 times slower than the hyplet.
The maximum latency of the hyplet is may be due to the clock deviation, which is 140ppm. It is not cache misses or TLB EAT (Effective Access Time) because each exception level in ARM has its private cache and TLB, which is never evacuated.
Normal and RT PREEMPT results are close, which leads to the understanding that a context switch, on average, between two threads of the same process, on Linux in PI, is 14 μs compared to seL4's context switch which is on average 8 μs.

V. USABILITY

Due to the limitations of this paper, we do not present the full API. HypletUtil is a library that provides a services such as memory mappings to the nanovisor, synchronization primitives, events, printing and many others. Our library also provides **delicate mapping**. Delicate mapping is used when we want to map only certain global variables and functions to the nanovisor. For this, we use 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. For example, the below maps the section "hyp" to the nanovisor.
In non-inject mode, the hyplet can be removed the minute the process terminates, gracefully or not, or by explicitly unregisters the hyplet. Also, it is mandatory to lock the hyplet memory to the RAM to avoid relocation, invalidation, or swapping.

VI. RELATED WORK

The extended Berkely Packet Filter (eBPF) [5] is described as an in-kernel VM, and eBPF 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. It runs in kernel mode, which is considered unsafe, but it uses a verifier to check that there are no illegal accesses to kernel areas or some tampering registers. Access to the user-space is enabled through memory maps. Also, eBPF uses LLVM and requires clang to generate a JIT code and has a small instruction set. As a consequence, eBPF has serious limitations. Particularly, only a subset of the C language can be compiled into eBPF; as such eBPF has no loops, no native assembly, no static variables, and no atomics. Furthermore, using eBPF may not take a long time and is restricted to 4096 instructions. This is not the case with the hyplet. The hyplet is not a program that executes in the kernel's address space, but in the user's address space. So, there is no need for maps to share data between the user and the kernel. The hyplet does not require any special compiler extensions and is much less restricted (what mapped prematurely can be accessed) and less complicated to use compared to eBPF. In general, the hyplet is meant to process events in user-space while eBPF collects data and processes it in kernel mode.
Hyperupcalls [2], which are ePBF extensions for a hypervisor, are a means to run hypervisor code in the guest's kernel context. Hyperupcalls are intended mainly for monitoring the health of the guest VM and are available only for the x86 architecture. The hyplet, in contrast, only uses the hypervisor and is not intended for the control and management of VMs. Nevertheless, it is possible to combine eBPF and the hyplet technologies so an eBPF program invokes a hyplet directly.
There has been a significant amount of research on secure microkernels and microvisors. A prominent microvisor is the OKL4 by Open kernel labs. The OKL4 microvisor [8] is a secure hypervisor that is supported by Cog systems and General Dynamics. The OKL4 microvisor supports both paravirtualization and pure virtualization. It is designed for the IoT and mobile industries and supports ARMv5, ARMv6, ARMv7, and ARMv8. Unlike the hyplet, the OKL4 microvisor is a full kernel executing in HYP mode. OKL4 microvisor has an open source sister project microkernel called seL4. Installing seL4 and running it is a challenging task because seL4 requires expertise and the adoption of the hardcore of the code. Another L4 para-virtualization technology is the L^{4}Linux [7] para-virtualized Linux kernel, that runs on top of the L4Re [13] microkernel. This system demonstrates real-time when threads execute in the microkernel. It transparently migrates a Linux thread to an L4Re thread. This is possible since the L^{4}Linux reuses address spaces and threading APIs of the L4Re microkernel. [11] presents a hard real-time in x86 and ARM. However, this technology requires a special Linux variant kernel and an SMP machine. The hyplet was ported to several kernel versions, (3.18 (android), 4.4, 4.1, 4.10, and 4.17) and several SOMs (Mediatech phone, Hikey, or any other ARMv8a processors) and it can execute on a single processor. This is possible because we re-used many of Linux virtualization capabilities (KVM).
Dune [4] is a system that provides a process rather than a machine abstraction through virtualization. Dune offers a sandbox for untrusted code, a privilege separation facility, and a garbage collector. Dune is implemented on Intel architecture and can be implemented with hyplets. However, this implementation means coercing a VM on hyplets, which is not the intended use of a hyplet.
In the area of pure virtualization, some efforts, such as Jailhouse and Xvisor, were made to run a guest OS as an

RTOS. Jailhouse demonstrates that it is possible to run an RTOS guest on top of a thin hypervisor and still achieve low latencies.

In the Linux area, the topic of user-space drivers handling IO events and exists in the Linux kernel inside the Universal I/O (UIO) framework. The UIO device driver is a user-space driver that blocks until an interrupt arrives. UIO offers an easy way to interact with various hardware devices. However, UIO device drivers are not suitable for devices with a high interrupt frequency.

VII. Summary

A. Future work

We intend to implement the hyplet for PowerPC. PowerPC shares some ARM capabilities, and the results will determine whether using the hyplet is efficient. We expect that ARM virtualization host extension becomes available for commercial use, and test the VHE hyplet performance.

B. Conclusions

We have introduced a new way ARM hypervisor instructions can enhance Linux performance in real-time systems. These features allow for security and performance benefits. The hyplet allows coding interrupts with a predictable μs latency and highly efficient RPC. We have implemented hyplets variant as security solutions for ARM.

References

[1] Tigist Abera, N Asokan, Lucas Davi, Jan-Erik Ekberg, Thomas Nyman, Andrew Paverd, Ahmad-Reza Sadeghi, and Gene Tsudik. C-flat: control-flow attestation for embedded systems software. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 743–754. ACM, 2016.

[2] Nadav Amit and Michael Wei. The design and implementation of hyperupcalls. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 97–112, 2018.

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

[4] 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.

[5] Jonathan Corbet. Bpf comes to firewalls, 2018.

[6] 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.

[7] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The performance of μ-kernel-based systems. *ACM SIGOPS Operating Systems Review*, 31(5):66–77, 1997.

[8] 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*, APSys '10, pages 19–24, New York, NY, USA, 2010. ACM.

[9] Wataru Kanda, Yu Yumura, Yuki Kinebuchi, Kazuo Makijima, and Tatsuo Nakajima. Spumone: Lightweight cpu virtualization layer for embedded systems. In *Embedded and Ubiquitous Computing, 2008. EUC'08. IEEE/IFIP International Conference on*, volume 1, pages 144–151. IEEE, 2008.

[10] "Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood". sel4: formal verification of an os kernel. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 168–178, New York, NY, USA, 2008. ACM.

[11] Adam Lackorzynski, Carsten Weinhold, and Hermann Härtig. Predictable low-latency interrupt response with general-purpose systems. In *Proceedings of OSPERT2017, the 13th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications OSPERT 2017*, pages 19–24, 2017.

[12] William Rosenblatt, Stephen Mooney, and William Trippe. *Digital rights management: business and technology*. John Wiley & Sons, Inc., 2001.

[13] Alexander Warg and Adam Lackorzynski. The fiasco. oc kernel and the l4 runtime environment (l4re). avail.

[14] Bruno Xavier, Tiago Ferreto, and Luis Jersak. Time provisioning evaluation of kvm, docker and unikernels in a cloud platform. In *Cluster, Cloud and Grid Computing (CCGrid), 2016 16th IEEE/ACM International Symposium on*, pages 277–280. IEEE, 2016.

[15] Raz Ben Yehuda and Nezer Jacob Zaidenberg. Protection against reverse engineering in arm. *International Journal of Information Security*, 19(1):39–51, 2020.

[16] Jun Zhang, Kai Chen, Baojing Zuo, Ruhui Ma, Yaozu Dong, and Haibing Guan. Performance analysis towards a kvm-based embedded real-time virtualization architecture. In *Computer Sciences and Convergence Information Technology (ICCIT), 2010 5th International Conference on*, pages 421–426. IEEE, 2010.

[17] Baojing Zuo, Kai Chen, Alei Liang, Haibing Guan, Jun Zhang, Ruhui Ma, and Hongbo Yang. Performance tuning towards a kvm-based low latency virtualization system. In *Information Engineering and Computer Science (ICIECS), 2010 2nd International Conference on*, pages 1–4. IEEE, 2010.

[18] Op tee , linaro limited: Open portable trusted execution environment