Hyplets - Multi Exception Level Kernel towards Linux RTOS

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

Nezer Zaidenberg
University of Jyväskylä
Jyväskylä , Finland
nezer.j.zaidenberg@jyu.fi

ABSTRACT

This paper presents the concept of a Multi-Exception level operating system. We add a hypervisor awareness to the Linux kernel and execute code in hyp exception level. We do that through the use of Hyplets. Hyplets are an innovative way to code interrupt service routines under ARM. Hyplets provide high performance, security, running time predictability ,an RPC mechanism and a possible solution for the priority inversion problem. Hyplets uses special features of ARM8va hypervisor memory architecture.

1 INTRODUCTION

Available technologies today based on virtualization offer Microvisors that divide the computer into VMs, each VM encapsulating with its own hardware and has its own operating system. Many if not all of the above perceptions separates between higher exception levels to the lower exception levels.

This paper presents the hyplet ISR as a para-virtualization technique to reduce kernel to user latency to less than a microsecond on average.

We will also demonstrate a new RPC (Remote Procedure Call) functionality. Our RPC is a type of hypervisor trap where the user process sends a procedure id to the hypervisor to be executed in high privileges with no interruptions in another process address space. We use the term hypRPC for our RPC as a mixture between hypercall and RPC.

SYSTOR, 2018, Haifa, Israel
© 2018 Association for Computing Machinery.
ACM ISBN 123-4567-24-567/08/06. . . $15.00
https://doi.org/10.475/123_4

Hyplets are based on the concept of a delicate address space separation within a running process. Instead of running multiple operating systems kernels in order to segment and divide the system resources, the hyplet divides the Linux process into two execution modes. One part of the process would execute in an isolated, non-interrupted privileged safe execution environment while other parts of the process would execute in a regular user mode. However, both execution modes run in the same Application processors.

We believe that Hyplets are suitable for hard real time systems. We will provide benchmarks and compare our solution to Linux RT PREEMPT and to a standard Linux. We chose RT PREEMPT as it is considered an open source non-commercial RTOS on Linux.

2 HYPLET ARCHITECTURE

In ARM, each of the four exception levels provides its own state of registers and can access the registers of the lower levels but not higher levels. This architecture dictates that the translation tables of the different exception levels are distinct. This means , that EL2 (hypervisor mode) may point to any memory translation table while the generic operating system, running in EL1 uses another translation table. This way, we can map an executing process (program) or part of it to the hypervisor, and we automatically gain access to the program address space, without any need to perform context switches or relocations.

To make sure that the hyplet code is always accessible and evacuation of the hyplet code and data from the current translation table is disabled, we chose to use $TTBR0\_EL2$ register to constantly accommodate the hyplet code. When a process address space is mapped to EL1 and EL2 excpetion levels it is reffered as a dual mapping of the process.

2.1 Hyplet - User Space Interrupt

In Linux, when an interrupt interrupts the processor, it triggers a path of code that serves the interrupt and in some cases ends up waking a pending process. The time to wake up the process is the interrupt latency the hyplet reduces. To achieve this, as the interrupt reaches the processor, instead of executing the user program code in EL0 after the ISR, a special procedure of the program is executed in a hypervisor mode before the kernels ISR. This is possible due to the dual mapping. The hyplet does not require any special threads and should be implemented as a small procedure. Since hyplets are actually ISRs they can be triggered in high frequencies. This way we can have a high frequency user space timers in small embedded devices.

2.2 Hypervisor RPC

Interprocess communications (IPC) in real time systems is considered a latency challenge. One reason is because there is the possibility that the receiver is not running therefore the kernel needs to switch contexts, which is considered a penalty. IPC is also described as a possible priority inversion scenario problem. RPC (Remote procedure call) is a form of IPC in which parameters are transferred in the form of function arguments and response is returned in the form of function return value. The RPC mechanism handles the parsing and handling of parameters and return values. We will show that it is possible to define a procedure in the address space of a receiving process that is invoked as a callback through the hypervisor whenever a sending process triggers an RPC. In this paper hyp RPCs are a form of IPC, i.e they are local.

3 EVALUATION

All tests were performed on Raspberry PI 3.

3.1 Interrupt Latency

We tested Hardware to hyplet latency. The Interupt latency was $2.5\mu s$. Evidently,hyplets have a low latency and also are suitable for hardware that generates interrupts at different rates.

3.2 Timer

We compared the responsiveness of hyplets to Linux RT PREEMPT.

In the hyplet case, 99.96 % of the samples were bellow $1\mu s$ latency, and 100% were bellow $5\mu s$. In RT PREEMPT case, the upper boundary is $47\mu s$ and the average is over $14\mu s$. It is evident that ISR-hyplets provide hard real time in a regular kernel.

3.3 Fast RPC

We evaluated the round trip of calling a null function (it just returns the time). A common IPC usually involves two context switches in a full round trip. The below benchmark is between two processes, a receiver and a sender. The receiver maps a hyplet to a single core, and the sender calls it. There are four types of tests:

- **Ref** Duration of the function when called in the process.
- **Hyplet** Duration of the function when called by a hypRPC and the sender and receiver share the same core.
- **IPI-hyplet** Duration of the function when called by a hypRPC when the sender and receiver do not share the same core
- **Standard Linux** The sender and receiver exchange is undertaken by Posix semaphores. The receiver waits on a semaphore; the sender awakes it and then waits on a second semaphore; the receiver executes the null function, and releases the sender.
- **Linux RT_PREEMPT** Like Standard Linux but over RT PREEMPT.

	Min	Avg	Max
Ref	104ns	156ns	520ns
Hyplet	520ns	520ns	$4.2\mu s$
IPI-hyplet	$3.4\mu s$	$6\mu s$	$21\mu s$
Normal Linux	$2.3\mu s$	$102\mu s$	$208\mu s$
RT PREEMPT	$12\mu s$	$14\mu s$	$340\mu s$

It is evident that hyplets are the fastest.

4 OTHER FEATURES

4.1 Security

We provide a safe execution environment for the operating system kernel so that even if there is a fault in the hyplet or if malicious data (that somehow crashes the hyplet) arises in the computer we can choose not to stop (or panic in Linux terms) the operating system. This is possible because the fault entry in EL2 handles the error as if it is a user space error. We can access hardware and protect its data. This can be done by reading data into a secured memory that is not accessible from EL1.

4.2 Temporality

Interrupts service routines rarely change ,i.e. it is not easy to modify a behavior of an interrupt routine in real time (while the device is running) .In the hyplet case, instead of modifying the kernel drivers, we can kill the user space hyplet program and run a new hyplet. We also have an abort mechanism that protects the hyp-ISR from crashing the operating system when there is a failure. The hyplet will fault like any other user space task.