2020 NDIA GROUND VEHICLE SYSTEMS ENGINEERING AND TECHNOLOGY SYMPOSIUM
CYBER TECHNICAL SESSION
AUGUST 11-13, 2020 - NOVI, MICHIGAN

Performance Impacts from the seL4 Hypervisor

Jesse Millwood[1], Robert VanVossen[1], Leonard Elliott[2]

[1]DornerWorks, Grand Rapids, MI
[2] CCDC-GVSC, Warren, MI

ABSTRACT

Hypervisor technologies are often presented as offering a high degree of separation at the cost of performance. Is this too expensive for embedded systems? The cost of performance has been shrinking year after year as new advancements in virtualization technologies are baked into processors. When a hypervisor couples hardware assisted virtualization with device emulation, it makes current systems portable, future proof, and extends the life of legacy systems.

seL4 is a perfect fit for the high assurance embedded hypervisor space. The open source seL4 microkernel is the first formally verified microkernel built with security and performance in mind. The mathematical proof of seL4 provides unprecedented assurance at the lowest, most critical software level.

This paper investigates the overheads associated with using seL4 as a hypervisor on ARM and x86 platforms, providing synthetic and real-world benchmarking methodology and results.

Citation: J. Millwood, R. VanVossen, L. Elliott, "Performance Impacts from the seL4 Hypervisor", In *Proceedings of the Ground Vehicle Systems Engineering and Technology Symposium* (GVSETS), NDIA, Novi, MI, Aug. 13-15, 2020.

1. INTRODUCTION

United States Department of Defense (DoD) programs continue to leverage and expand the use of hypervisor technologies to harden all manner of computer systems ranging from enterprise platforms down to embedded cyber-physical systems. The ability to isolate software components and partition functionality supports fundamental principles for improving the security of a system and has the added benefit of enabling

hardware consolidation. Hypervisors serve a foundational role for accomplishing this while underpinning a defense-in-depth approach, and although there is a performance penalty incurred with the use of hypervisors, modern processor architectures are incrementally minimizing this overhead and decreasing the impact on the user.

DoD weapon system programs are increasingly looking to improve the robustness of their cyber-physical systems. A recent report by the Government Accountability Office [1] [2] indicates that this is of the utmost importance. Traditionally the deployment of embedded hypervisors in cyber-physical systems has been approached through the acquisition of prohibitively expensive proprietary embedded hypervisors or microkernels, and corresponding certification artifacts, on a per project basis. Procurement of proprietary embedded hypervisors and microkernels results in vendor-lock and, despite similar system requirements, typically does not permit DoD agencies to reuse their investments. In the case of ground vehicles, and likely for many other applications, the cost of embedded hypervisors inhibits many commercial and military systems from leveraging them simply because they do not have the budget to procure these specialized products. The use of embedded hypervisors remains reserved for specialized applications, such as in aerospace, where the cost can be justified.

The genesis of the seL4 kernel and the co-developed mathematical proofs of correctness [3] represent a rare, and potentially unparalleled opportunity, for commercial and military systems to achieve levels of robustness that until now was unattainable. A secure hypervisor is the product of combining the seL4 kernel and the CAmkES Virtual Machine Monitor (VMM). When compared with proprietary offerings, the seL4 based CAmkES VMM is open-source, built on a kernel that claims to be “the world’s most secure OS” [4] freely providing formal proofs which are potentially stronger than those provided in proprietary offerings, and also claims to be the “the world’s fastest microkernel.” This combination of attributes makes seL4 a candidate for improving the security posture of not just military components, but commercial products including Internet-of-Things (IoT) appliances, medical devices, and critical infrastructure components to name just a few. The open-source nature of seL4, while prompting special consideration due to the sensitive applications that are targeted, may enable reuse and sharing between projects in a way that is not usually achievable with proprietary products. The openly published and peer-reviewed proofs of correctness guarantee the absence of bugs in the formally verified components and can be used to support system certification against the highest level of rigor including the Common Criteria for security, DO-178C for airborne systems, IEC 62304 for medical device software, and IEC 61508 and ISO-26262 for road vehicle safety. This picture is in stark contrast with proprietary offerings which typically charge additional fees for certification artifacts before much can even be ascertained about the nature of the guarantees they provide.

Recent work has examined the suitability of using seL4 in ground vehicle systems [2] [5]. Work has been done to port seL4 to ruggedized hardware and develop the necessary features which range from specific guest support to virtualized drivers. The presented work has never analytically examined the performance impacts of running the seL4 hypervisor in such a system, instead relying on preconceived notions, subjective measures such as user-acceptability testing, and qualitative assessments of performance inferred from virtualization trends in hardware architecture,

and known characteristics of the seL4 kernel and CAmkES VMM hypervisor itself. It is the objective of this paper to present a quantitative analysis of the seL4 hypervisor performance overhead that we conducted on a commercial ARM embedded device and a representative, military-grade embedded device. We will also discuss potential impacts on cyber-physical system development.

2. HYPERVISOR BACKGROUND

In the last decade hypervisors have proliferated to the level of a commodity product in the enterprise space and hypervisor technology stands as one of the foundational elements for the explosion and success of cloud-computing. The hypervisor provides the ability to virtually partition hardware and software resources so that businesses no longer require dedicated hardware and can take advantage of the economies of scale offered by the cloud computing providers. The abstraction and isolation enabled by hypervisor technology has matured to the degree that rival businesses can happily be running mission critical software on the same cloud server. Although there are indeed major differences between an enterprise system and a weapons platform, there are many similarities in the design patterns and benefits that we seek to achieve by applying hypervisor technology in a weapon system. For this reason, it is useful to discuss hypervisors in a general context and how it relates to what we refer to as an "embedded hypervisor."

2.1. Hypervisors

Hypervisors are fundamentally a layer of software that enables multiple guest operating systems or processes to run alongside each other and share the same hardware resources. The hypervisor, sometimes called a Virtual Machine Monitor (VMM), manages the guest virtual machines (VM) and provides an interface between the VMs and the host hardware. There are two broadly recognized types of hypervisors and we will introduce a third type, to which we consider an seL4-based system to belong.

2.1.1 Type-1 Hypervisor

Also known as bare-metal hypervisors, Type 1 hypervisors are relatively small pieces of software that generally do not comprise of a fully functional operating system by themselves. They are usually more efficient because they have direct access to the hardware, however they often need a separate VM or privileged process integrated to monitor and manage the guests.

2.1.2 Type-2 Hypervisor

Sometimes referred to as a "hosted" hypervisor, Type 2 hypervisors run as an application within the hosting Operating System (OS). Type 2 hypervisors are generally easier to use and deploy, however they access the underlying hardware resources through the host OS and so can suffer performance problems and have the potential to be compromised by vulnerabilities that are inherited from the larger, more feature rich, host OS.

2.1.3 Embedded Hypervisor

This type of hypervisor is less widely recognized and is generally a subset or customization of the Type 1 hypervisor. Embedded hypervisors may be used to host specialized guests such as Real-Time Operating Systems (RTOS) and an example configuration is depicted in Figure 1.

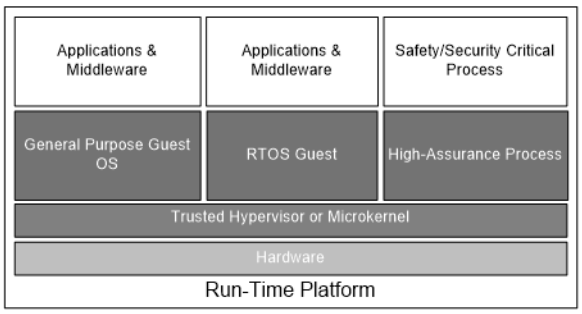

Figure 1: Typical Type-1 Embedded Hypervisor

Embedded hypervisors are specially tailored for security and performance in real-time applications. They often need to provide hard real-time determinism to the processes they run with worst-case execution time guarantees. They may lack some features that are baked into enterprise server hypervisors in an attempt to shrink attack surfaces and keep the code small and efficient. We will also assert, in the context of this paper, that this type of hypervisor must be small enough, in source lines of code (SLOC), that it is feasible to assess fitness for purpose to host applications that require high levels of robustness for safety and security.

2.2 seL4 VMM

While seL4 is a microkernel whose main purpose is to manage memory, interrupts, application threads, and system calls; it also provides some interfaces to support running VMs. The majority of the code needed for virtualization is implemented in user-space and is called the VMM. This initializes memory and capabilities for the VM and provides handlers for various expected faults, such as emulated device drivers. The seL4 microkernel does need some code specific to running VMs, which can be configured at build time. The user-space VMM code was originally implemented as a regular seL4 application. More recently, it has been re-implemented as a series of CAmkES components by Data61 [6] [7]. DornerWorks has added support for the ZCU102 and 64-bit x86 military-grade board with the intent of contributing upstream. It is important to note that the seL4 VMM implementations used in this study contain code in performance critical areas that DornerWorks developed for the porting effort.

2.2.1 CAmkES Background

CAmkES is the Component Architecture for Micro-Kernel-based Embedded Systems [8]. Developing simple applications on top of the seL4 microkernel is somewhat straight-forward; however, as the size of the application becomes larger, the complexity of developing on top of seL4 increases exponentially. The CAmkES framework allows developers to focus on functionality without being bogged down by the specifics of seL4 and its APIs.

CAmkES supplies a component-based framework by providing an Architecture Description Language (ADL) to describe software components and the interfaces between them. The seL4 microkernel guarantees these components are isolated from each-other; the only methods to communicate between them consist of user-defined interfaces and shared memory. During compilation, the CAmkES tool generates initialization code and the glue code required to tie the components together into a functional system built on seL4 without having to directly use any of the seL4 APIs. When developing a system without CAmkES, a root thread is developed that is given access to all of the resources in the system. Using system calls, the root thread creates and configures other user-space threads. It then splits up access to hardware resources to those threads and configures connections between them. Fault handlers are then setup and the threads are started. When developing with CAmkES, all of this is done with the CapDL Loader application. This application is a root thread that takes the CapDL [9] generated output from the CAmkES configuration to setup the system.

Using CAmkES for the user-space, VMM implementation makes sense because of the inherent modularity. A VMM just needs to supply the same features for almost all VMs and then it is up to each specific VM what applications it runs and how it uses them. It also makes it easier to add and remove features like servers and companion applications.

2.2.2 seL4 VMM Architecture

Due to differences in architecture, the VMM uses different library functions and methods to provide virtualization support on both ARM and x86.

On ARM, the hardware virtualization extensions and Exception Levels (EL) are utilized to provide most of the functionality that the VM needs. The hardware virtualization extensions provide features such as, multi-staged Memory-Management Unit (MMU) support, virtualized interrupt registers, virtualized system timer, hypercall support, and EL-based fault handling. This provides all of the low-level platform support needed to get a virtual machine running correctly. The system runs in components in different exception levels, as shown in Figure 1.

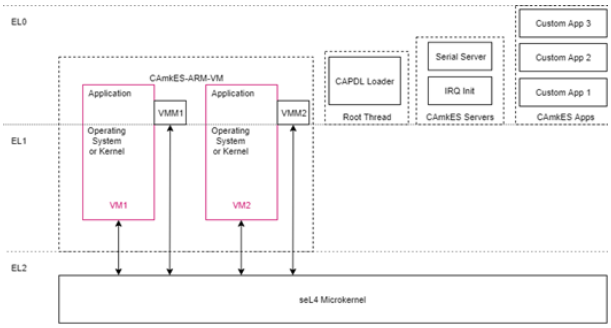

Figure 2:CAmkES-based VMM on ARM

On x86, VT-x is utilized to accomplish the same goals. Extended Page Tables (EPT), virtualized interrupts (APICv), IOMMU, and Peripheral Component Interconnect (PCI) pass-through and emulation are all used to provide what is needed for a VM.

For either platform, some devices need to be virtualized with software or passed-through. Serial devices are generally virtualized since one is likely to run more VMs than the number of available physical UARTs. This is done by trap and emulate. The memory regions related to the device are marked for no access. When the VM tries to read or write from it, a fault is generated. The seL4 hypervisor catches this fault and branches to the correct handler in the VMM thread. This handler interfaces with a server that interacts with the device on the VM's behalf. The way that these devices are defined for the VM is different between architectures. On x86, the virtualized device is enumerated to the PCI bus so that it can be discovered by the VM. On ARM, the device needs to be added to the device tree that is compiled and provided on boot to the VM.

On both platforms, hardware virtualization is used whenever possible as it will provide the best performance possible. When it isn't feasible to use hardware virtualization, such as virtualized devices, software virtualization is used to give the functionality that is needed at the cost of a performance hit.

3 METHODOLOGY

Several benchmark suites were used in this study to shed light on the performance overhead of utilizing specific resources in a virtualized environment. Performance overhead is calculated as shown in Equation (1).

$$P_{overhead} = \frac{P_{virt} - P_{native}}{P_{native}} * 100\% \quad (1)$$

Here, the performance overhead is calculated relative to the baseline, native, and performance. Performance here can be any measurement.

A Linux image was generated for each platform. The same Linux image was used for the native benchmarks as well as the virtualized one. PetaLinux 2019.2 [10] and a

CentOS derived Linux distribution were used to build and install reproducible images that contained the same benchmark software. The two images ran different Linux kernels. The PetaLinux image and the CentOS derived image used for this study ran different versions of Linux kernels. The differences in kernel versions is not of concern here since advanced features of Linux are not being used and the overhead introduced by the hypervisor is the real focus of our measurement. Due to hardware and software architectural differences, the actual measure of impacts between architectures are not what is of interest in this paper. This paper uses a ratio between the native performance and the delay incurred in the virtualized environment so that fairer comparisons can be made. For each platform, one CPU is virtualized per VMM.

3.1 Benchmarking Setup

The chosen platforms to perform benchmarks on were:

- 64-bit x86 Military-Grade Single Board Computer
- ZCU102 [11]
    - Manufacturer: Xilinx
    - SoC: ZU9EG

These platforms were chosen because they are powerful embedded platforms that could be used as development environments for military applications. No consideration was made to equate these boards in any way, such as clock-scaling or any other processor configuration. The minimal configurations that were performed on the boards were to support virtualization, otherwise they were used out of the box. The x86 board's BIOS was configured to enable VT-X extensions.

The boards were both connected to a computer via a Prolific serial-to-USB cable. The majority of interaction with the boards happened over this interface such as issuing terminal commands and collecting data. Besides flash memory, no other instruments or interfaces were used.

The x86 processor used had multiple cores and multiple threads, however only a single core was used. A CentOS-derived Linux distribution was used on the military grade x86 platform. Since the Linux is an RPM based distribution, the YUM package manager was used to install the benchmark software. This was booted with a Syslinux configured USB device, which held the images for the virtualized and native Linux systems. Syslinux menu entries were then used so that the operator could choose which system to boot into. The power to the system was reset in order to switch between running the native and virtualized.

The ARM Xilinx PetaLinux framework produced a BOOT.bin file, which contains a first stage boot loader (FSBL), U-Boot, and a bitstream. The ZCU102 board was booted with an SD card that contained the BOOT.bin, the virtualized Linux system, and the native one. Once the ZCU102 booted to the U-Boot prompt, commands were used to load the images to the proper memory locations and to jump there. The ZCU102 also required a power reset to run the other image. The ZCU102 has 4 ARM Cortex-A53's and dual Cortex-R5F's, however, a single A53 was used in this study. The FPGA fabric on the ZCU102 allowed connecting GPIO pins together in the fabric to facilitate simple and accurate interrupt latency measurements.

3.2 Benchmark Suites

Synthetic benchmarks were used here in order to target specific system resources. This allows us to have a more accurate idea of system impacts when running in a virtual machine. These benchmarks run workloads that otherwise do not have a useful outcome in the real world other than causing increased usage of a particular system component.

Several synthetic benchmarks were chosen to offer comparable performance numbers.

The Sysbench [12] benchmark suite offers some resource targeting synthetic workloads that are very useful for these comparisons. The benchmark sub programs that were used in this study were:

- **CPU**: Verifies prime numbers through division of the number between 2 and the square root of the number.
- **Memory**: Read and writes to an allocated buffer multiple times.
- **Threads**: A number of threads are moved through the run queue by locking their mutex and yielding.
- **Mutex**: Takes a mutex, increments a global number, and releases.

The Sysbench suite has more workloads available but these were chosen so that the impact to resources could be compared.

MiBench [13] is a "free, commercially representative embedded benchmark suite" that has been put together by the University of Michigan. This is not so much a synthetic benchmarking suite as it uses real programs to produce real workloads. It is meant to be used in a simulated environment in order to provide validation to microarchitectures. This is done by measuring metrics like cache misses, branch misses, code size, etc. The programs included in the MiBench suite are categorized into automotive, consumer, networking, office, security, and telecom. The programs included in the automotive subset that were used are:

- Basic Math
- Bit Count
- Qsort
- Susan – Smoothing
- Susan – Edges
- Susan - Corners

These workloads were used in this experiment to offer some benchmarks closer to real world usage. Since these systems were not instrumented to monitor metrics such as cache misses, the execution times were analyzed and used as the interested metric.

The GPIO IRQ Latency test kernel module [14] was ported to work with the ZCU102. The kernel module is used to configure one GPIO pin as an output and one as an input that triggers an IRQ handler when input is received. This was also combined with an AXI GPIO block in the FPGA fabric that allowed connecting the two pins in the fabric instead of with a wire. This ensured no environmental impacts could alter the measurement and improved reproducibility. The test then simply measures the time between setting a GPIO pin high and when the interrupt fires on the input pin.

Cyclic Test [15] is a benchmark developed under the Linux Foundation for the real time Linux variant. It "accurately and repeatedly measures the difference between a thread's intended wake-up time and the time at which it actually wakes up" [15]. This is used as a proxy measurement for interrupt latency on x86. The x86 board that was used in this experiment did not have suitable GPIO pins that allowed interrupts to be associated with them in the same way that the ZCU102 did. Therefore, this benchmark was used because when the Cyclic Test is run with the "nanotime" argument, the timer interrupt is utilized to determine when to wake up the thread. This measurement is very similar to what is being achieved on the ZCU102 platform.

3.3 Benchmarking a Hypervisor

The main measurement that is of interest when benchmarking a hypervisor is overhead of running natively as opposed to virtualized. Synthetic benchmarks were used in the study to show where overheads occurred and break out the impact to CPU usage, memory usage, and interrupt latency.

When benchmarking a hypervisor, care must be taken to ensure that there is little that differs between the operating system running

natively versus in a virtualized environment. A CentOS-derived Linux distribution and PetaLinux help with the software side of this equation.

For this study the following steps were performed to collect benchmarking data per platform:

1. Prepare boot media
2. Boot target with prepared media
3. Boot into either native or virtualized image
4. Perform tests manually at Bash shell
5. Record results in spreadsheet
6. Repeat steps from step 3 if needed for other environment.

The spreadsheet was then used to calculate overheads.

In systems that would be deployed to security critical environments, or systems that have more VMs than specific resources, some devices and resources must be virtualized. Resource virtualization introduces its own overhead, which is not considered by this study. This study is concerned with the basic overhead associated with using the CAmkES VMM hypervisor.

4 RESULTS

The percent difference between running these benchmarks, natively and in a virtualized environment, are presented in this section. In every case, performance is worse in a virtualized environment, so the figures here show the performance decrement suffered on the chosen platforms.

4.1 Benchmark Suite: Sysbench

The overheads associated with the Sysbench benchmarks can be seen in Figure 3 and Figure 4.

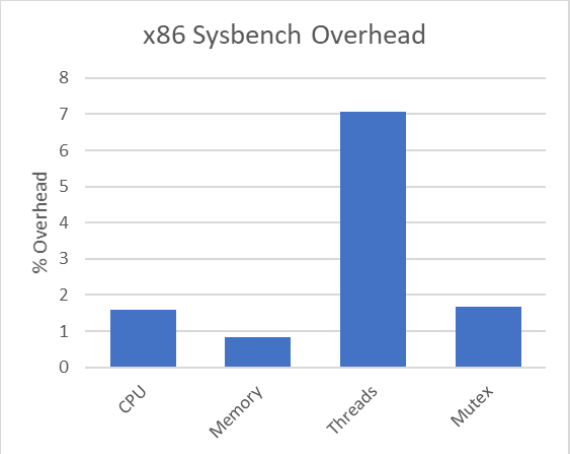

Figure 3: x86 Sysbench Overhead Results

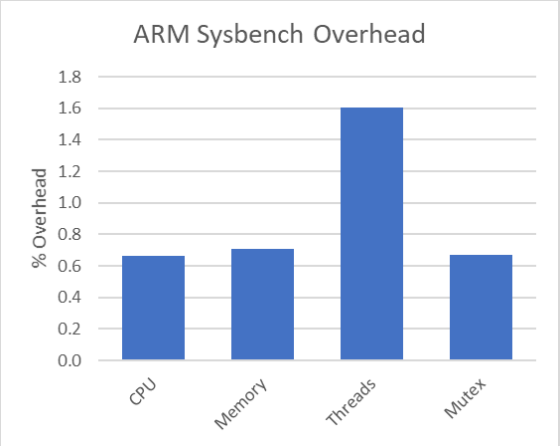

Figure 4: ARM Sysbench Overhead Results

There appears to be additional overhead associated with the x86 platform. On the ARM platform, there is a maximum of 1.6% overhead induced by the seL4 VMM. On the x86 platform, the overheads range from 0.82% to 7% overhead. The difference in overheads could be due in part to the difference in seL4 VMM implementations between the architectures and the less costly VM exits on ARM.

4.2 Benchmark Suite: MiBench

As mentioned before, the MiBench suite is intended to be used to provide validation for microarchitectures in instrumented settings. In this study the runtime of each of the chosen MiBench tests was recorded. The overheads

associated with each platform is shown in Figure 5 and Figure 6.

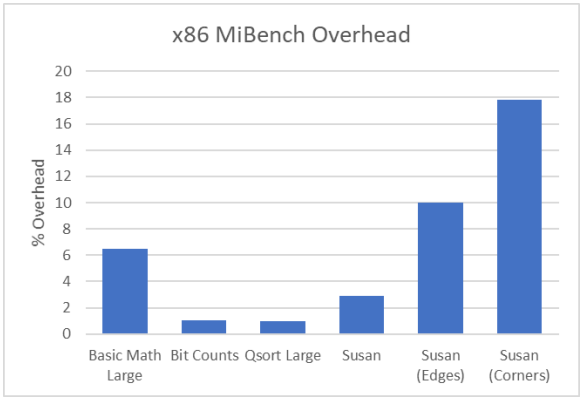

Figure 5: x86 MiBench Overhead

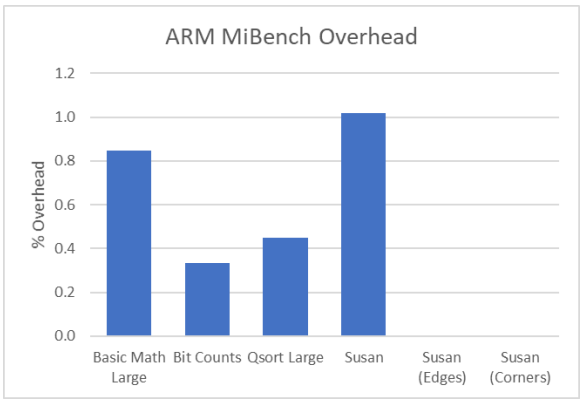

Figure 6: ARM MiBench Overhead

Again, it is interesting to note that the ARM platform suffered from significantly less overhead, compared to the x86 platform. There was a 1% to 17% overhead when running the workloads in a virtualized environment on x86 and only a maximum of 1% overhead on ARM.

4.3 Interrupt Latency

Interrupt latency was measured in different ways between the platforms due to the difference in available resources on the boards. The simple GPIO/Interrupt overhead results for the ARM ZCU102 board are shown in Figure 8, while the Cyclic Test overhead results for x86 board is shown in Figure 7.

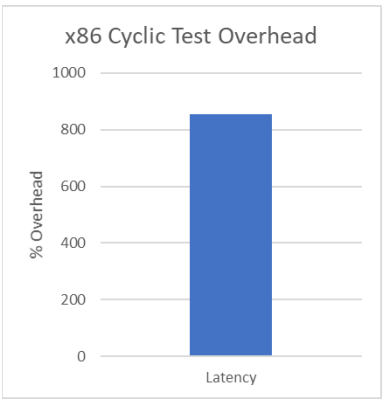

Figure 7: x86 Cyclic Test Overhead

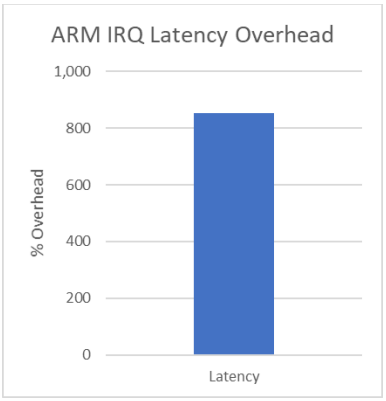

Figure 8: ARM IRQ Latency Overhead

Unlike the previous benchmarks, the x86 and ARM platforms both had very large overheads with the interrupt latency measurements, 855% and 851% respectively. This may look alarming but the actual time differences are 445µs and 13µs for x86 and ARM respectively. The overhead associated with latency overhead here is primarily due to how the CAmkES VMM handles interrupts. Every interrupt that occurs, is first handled by the seL4 kernel, dispatched to the VMM and then injected into the proper guest OS.

5 FUTURE WORK

The benchmarking presented in this paper is a baseline study on the overhead associated with running a single-guest OS in the CAmkES VMM on seL4. This can be expanded upon in many ways. Specifically, studying how different workloads in a multi-VM setup affect the performance of the

system and how time-slice settings tie into this.

The baseline set by this study could also be used to drive optimizations to servers, the hypervisor, the scheduler, or components of the CAmkES VMM. Using this work as a framework could provide empirical data that shows how performance changes with different system improvements and optimizations.

While DornerWorks ported the CAmkES VMM project to the ZynqMP and the particular x86 processor used in this study, further architectural changes are intended to be made when the code is up-streamed and reviewed by Data61. Collaboration with other members in the seL4 community may result in higher-performance virtualization with seL4.

This study focused on two architectures that are still adding features: x86 and ARM. The new instruction set architecture (ISA) on the scene, RISC-V [16] continues to be an interest to DornerWorks and other DoD research initiatives. The lack of licensing requirements for the ISA provides a basis for collaborative secure computing research and development. The current ratified RISC-V privileged ISA contains a draft version of hypervisor extensions. However, since the extensions aren't ratified yet there are not any silicon or FPGA implementations available. While a port of the CAmkES VMM has not occurred yet, RISC-V is a primary focus of Data61's verification efforts. Once the hypervisor extensions are ratified, benchmarking synthetic and representative workloads would provide valuable information to system designers when considering a virtualized environment based on a RISC-V platform.

6 CONCLUSION

The baseline benchmarking performed in this paper can help two main groups of people: system designers considering an seL4 based hypervisor and developers working on the CAmkES VMM. System designers can use these benchmarks to determine if the performance overheads measured here are acceptable for their uses. Developers can use these measurements to guide optimization efforts.

These benchmarks have shown that for computation-related overheads such as CPU, memory, and threads, the CAmkES VMM adds very little in overhead, while exposing seL4's world class isolation mechanisms. For many applications, this performance overhead would be worth the assurance and security gained by building on seL4 and the CAmkES ecosystem.

DornerWorks has talked about the virtues of virtualization for many years. These benchmarks show that the seL4-based CAmkES VMM hypervisor is a suitable lightweight and secure solution when paired with the proper hardware.

7 REFERENCES

[1] C. Chaplain, "Weapon Systems Cybersecurity: DoD just beginning to grapple with scale of vulnerabilities," *Washington, DC, USA, GAO Report No. GAO-19-128,* 2018.

[2] R. VanVossen, J. Millwood, C. Guikema, L. Elliott and J. Roach, "The seL4 Microkernel--A Robust, Resilient, and Open-Source Foundation for Ground Vehicle Electronics Architecture," in *the Ground Vehicle Systems Engineering and Technology Symposium*.

[3] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski and M. Norrish, "seL4: Formal verification of an OS kernel," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009.

[4] G. Heiser, "The seL4 Microkernel: An Introduction," 2020.

[5] C. Guikema, "Virtualization on seL4 – Expanding the CAmkES-ARM-VM," in *seL4 Summit*, Dulles, VA, 2019.

[6] Data61, "CAmkES ARM VM," Data61, [Online]. Available: https://github.com/SEL4PROJ/camkes-arm-vm.

[7] Data61, "CAmkES VM," Data61, [Online]. Available: https://github.com/seL4/camkes-vm.

[8] I. Kuz, Y. Liu, I. Gorton and G. Heiser, "CAmkES: A component model for secure microkernel-based embedded systems," *Journal of Systems and Software,* vol. 80, no. 5, pp. 687-699, 2007.

[9] I. Kuz, G. Klein, C. Lewis and A. Walker, "capDL: A language for describing capability-based systems," in *Proceedings of the first ACM asia-pacific workshop on Workshop on systems*, 2010.

[10] Xilinx, "PetaLinux Product Page," [Online]. Available: https://www.xilinx.com/products/design-tools/embedded-software/petalinux-sdk.html.

[11] Xilinx, "Zynq Ultrascale + MPSoC ZCU102 Evaluation Kit Product Page," [Online]. Available: https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html. [Accessed 2020].

[12] A. Kopytov, "Sysbench Github Repository," [Online]. Available: https://github.com/akopytov/sysbench. [Accessed 2020].

[13] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin and R. B. B. Trevor Mudge, "MiBench Home Page," University of Michigan, 2001. [Online]. Available: http://vhosts.eecs.umich.edu/mibench//. [Accessed 2020].

[14] gkaindl, "Linux GPIO IRQ Latency Test," [Online]. Available: https://github.com/gkaindl/linux-gpio-irq-latency-test. [Accessed 2020].

[15] C. Williams and J. Kacur, "Realtime: Cyclictest," The Linux Foundation, May 2016. [Online]. Available: https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/cyclictest/start. [Accessed 2020].

[16] A. Waterman and K. Asonovic, *The RISC-V Instruction Set Manual,* 20190608-Priv-MSU-Ratified ed., RISC-V Foundation, 2019.

[17] SuperMicro, "X10SDV-12C+-TLN4F Product Page," [Online]. Available: https://www.supermicro.com/en/products/motherboard/X10SDV-12C+-TLN4F. [Accessed 2020].

[18] Concurrent Real-Time, "RedHawk Linux Product Page," [Online]. Available: https://www.concurrent-rt.com/solutions/linux/.