ZxOS: Zephyr-based Guest Operating System for Heterogeneous ISA Machines

Ashwin Krishnakumar

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Engineering

Binoy Ravindran, Chair

Tam Chantem

Ruslan Nikolaev

02/18/2022

Blacksburg, Virginia

Keywords: Heterogeneous computing, hybrid computing, Puzzlehype , Heterongneous-ISA CMP

ZxOS: Zephyr-based Guest Operating System for Heterogeneous ISA Machines

Ashwin Krishnakumar

(ABSTRACT)

With the fast-approaching limits of single-threaded CPU performance, chip vendors are manufacturing an array of radically different computing architectures, including multicore and heterogeneous architectures, to continue to accelerate computer performance. An important emerging data point in the heterogeneous architecture design space is heterogeneity in instruction-set architecture (ISA). ISA-heterogeneity is emerging in many forms. An exemplar case is Smart I/O devices such as SmartNICs and SmartSSDs that incorporate CPUs of the RISC ISA family (e.g, ARM64, RISC-V), which when integrated with a high-performance server with CPUs of the CISC ISA family (e.g., x86-64) yields a single machine with heterogeneous-ISA CPUs. This thesis presents the design of a shared memory OS for a cache-coherent, shared memory heterogeneous-ISA hardware. The OS, called ZxOS, is built by modifying the open-source ZephyrOS, including its architecture-specific code and page mapping mechanism to create a memory region that can be shared across heterogeneous-ISA CPUs. Since existent heterogeneous-ISA hardware has physically discrete memory for ISA-heterogeneous CPUs, ZxOS targets a software emulation environment that emulates cache-coherent, shared memory heterogeneous-ISA hardware. Our experimental evaluation using a set of micro- and macro-benchmarks demonstrate ZxOS's functionality. In particular, they show that a multithreaded application's threads can be split across (simulated) ISA-heterogeneous cores for parallel execution and that thread's concurrent access of shared memory variables is consistent.

ZxOS: Zephyr-based Guest Operating System for Heterogeneous ISA Machines

Ashwin Krishnakumar

(GENERAL AUDIENCE ABSTRACT)

The computing industry saw a growing difficulty with power management and heat dissipation in single-core CPUs with high clock speeds. Chip vendors adapted multi-core chip strategy because it improved power and performance characteristics. This idea evolved into heterogeneous computing paradigm, which uses heterogeneous instruction-set-architecture(ISA) cores in a single system. Many server-grade specialized devices were provided with RISC-based general-purpose CPU to efficiently manage requests from the server. SmartNICs and SmartSSDs are the best examples of these smart devices. When integrated with generally present CISC(x86_64) processors in server space, these devices resulted in machine-level heterogeneous-ISA CPUs. This thesis presents a novel design of a shared memory OS for a cache-coherent, shared memory heterogeneous-ISA hardware. The OS, called ZxOS, is built by extending the ZephyrOS. Many of the architecture-specific codes were modified to create a memory region that can be shared across the cores. ZxOS is aimed at a software emulation environment for cache-coherent, shared memory heterogeneous-ISA systems. ZxOS has demonstrated the capability to run multi-threaded applications in a split fashion across heterogeneous-ISA cores. The heterogeneous-ISA cores execute threads parallelly, and the thread's shared memory interactions were observed to be consistent.

Acknowledgments

First, I want to thank my advisor Dr. Binoy Ravindran for allowing me to work in SSRG Lab and guiding me. Without this experience, my education at VT would not be complete.

I want to thank Dr. Antonio Barbalace and Dr. Pierre Oilver for providing technical advice and guidance in clarifying the task.

I am grateful to Dr. Ruslan Nikolaev for being a part of my thesis committee and for his Advanced Operating Systems and Virtualization Course. This course helped me understand a standard operating system in bare bones.

I want to thank Dr. Tam Chantem for the Advanced-RTOS course, which helped me understand RTOS fundamentals. Without the classes, I do not believe the work would have been a reality.

A special mention to Narayanan Rao and Ho-Ren Chuang for their technical discussion sessions and regular talks.

I want to thank my life mentor Mr.Dhanraj for his guidance. Finally, I would like to thank my parents and teachers who have guided me to achieve everything in life.

This work is supported by the US Office of Naval Research under grants N00014-18-1-2022 and N00014-19-1-2493.

Contents

List of Figures

List of Abbreviations

APIC Advanced Programmable Interrupt Controller

DSM Distributed Shared Memory

GIC Generic Interrupt Controller

IP Intellectual Property

ISA Instruction Set Architecture

ivshmem Inter-VM Shared Memory

LLVM Low Level Virtual Machine

PCIe Peripheral Component Interconnect Express

PID Process Identifier

PTE Page Table Entry

RDMA Remote Direct Memory Access

VMA Virtual Memory Area

Chapter 1

Introduction

In the early 2000s, the semiconductor industry observed that further increases in transistor count – to increase clock speeds and thus single-threaded CPU performance – resulted in heat dissipation and power consumption issues. This phenomenon, widely called "the end of Moore's law" [25], forced chip vendors to explore alternative designs to increase computer performance. An important design that emerged was chip multiprocessors (or CMPs) [45] [34], wherein multiple identical computing cores were integrated into the same chip. CMPs offered hardware parallelism as the means to improve software performance, as opposed to higher clock speeds of a single CPU.

Another important design that emerged around the same time is heterogeneous computing. In this space, special-purpose computing units such as GPUs [22], FPGAs [42], and ASICs [39] were integrated with a general-purpose CPU (or a CMP). Such special-purpose units can perform a specialized task, e.g., SIMD-style parallelism for GPUs, synthesis of compute-intensive logic into programmable logical devices for FPGAs, at significantly higher efficiencies than general-purpose CPUs, enabling application performance to be accelerated. Special-purpose compute units also have other advantages, such as FPGA's greater customizability and lower TDP [28].

An interesting data point in the heterogeneous computing design space that has recently emerged is the integration of general-purpose CPU cores of different instruction-set-architecture (ISA) families in the same hardware. This trend has emerged in different compute granular-

ities. Examples include PCIe-compatible Smart I/O devices such as SmartNICs [36], and SmartSSDs [40] that incorporate general-purpose CPUs of the ARM or RISC-V ISA family. Such devices can be integrated into a high-performance server composed of the x86 ISA family cores, which yields a single compute unit composed of heterogeneous-ISA cores. ARM-based servers (e.g., Amazon's Graviton [8]) are increasingly integrated into high-performance data centers (e.g., AWS [9]), which are traditionally composed of x86-based servers, which yields a loosely coupled distributed system of heterogeneous-ISA CMPs. Another interesting trend is FPGA fabrics with ARM-based CPUs that are integrated with x86 cores in the same chip using cache-coherent interconnects (e.g., Intel Agilex [3]).

Several academic research efforts have focused on the design space of heterogeneous-ISA cores. An exemplar case is the Popcorn Linux project [14]. Popcorn Linux enables applications written for a traditional shared-memory programming model (e.g., POSIX, OpenMP) to run on heterogeneous-ISA hardware without significant modifications. This improved programmability is accomplished through innovations across the system software stack (i.e., operating system, compiler, run-time), which hides complexities such as ISA/ABI differences, ISA-different CPU's discrete physical memory, and cross-ISA scheduling and resource management. Some of the other research efforts include DeVuyst et al. [26], Cho et al. [23], and Li et al. [30].

1.1 Motivation

In the academic research efforts on heterogeneous-ISA computing, heterogeneous-ISA cores have physically discrete memory, let alone cache-coherent shared memory. This is simply because CMP hardware with shared memory and heterogeneous-ISA cores are not available at the commodity scale. In such a design, cores with heterogeneous ISAs must be available

in the same die. These cores must possess the capability to uniformly share all of the system resources. The cores must also have cache-coherent interconnects and provide the capability to interact with each other through micro-architectural primitives. The factor restricting such a shared memory heterogeneous-ISA multiprocessor is that low-level cache-coherent interconnects such as CCI [2] and CAPI [41] are yet to be successfully realized. The lack of such chips hinders research in the design space of operating systems for such architectures. For example, a shared memory OS model (as in monolithic OSs such as Linux, Windows, and BSD), or a replicated kernel OS model (as in OSs such as Popcorn Linux [14], K2 [32], and Barrelfish [19]), or a partitioned OS core/application core model (as in OSs such as NIX [13], Helios [35], and M3 [11]) the most efficient design for such architectures? This has motivated the design of software-based simulation frameworks that enable the emulation of shared memory cache-coherent heterogeneous-ISA systems. At the time of writing this thesis, hetQEMU [17] is the only such environment.

HetQEMU is a simulation environment that is based on the QEMU [20] machine simulator and virtualizer. HetQEMU simulates the RAM device for the QEMU instances through a file-backed memory interface. A file-backed memory is an OS feature that maps a memory section onto a file in the virtual file system. Multiple applications can get a filehandle, and all applications consistently i.e., writes to a filehandle/memory section by an application, is seen by all other applications in the same order. Guest kernel instances that run on file-backed memory can use shared memory to interact with other kernels running on ISA-different cores simulated on top of hetQEMU. The shared RAM sections present across the heterogeneous QEMU instances enable the kernels to interact through them. This shared memory interface mimics a heterogeneous-ISA CMP in terms of on-chip memory. However, sharing memory resources across the machines is insufficient to enable a heterogeneous-ISA CMP simulation environment. The virtualizer's design is motivated for providing cache-

coherent shared memory across heterogeneous VM instances. HetQEMU also provides a generic interrupt controller to enable the systems to utilize shared interrupts. It also provides an Inter-Processor Interrupt(IPI) mechanism that enables the simulated cores to trigger the heterogeneous cores. These IPIs can perform TLB shootdown functioning across dissimilar cores. Using such a simulator, a guest OS can be made to run on a (simulated) heterogeneous ISA CMP.

HetQEMU thus provides a simulated heterogeneous ISA CMP with coherent shared memory enabling seamless memory sharing across VMs. This enables exploration of the OS design space for shared memory cache-coherent heterogeneous ISA systems. Many of the design principles of shared memory OS models can be adapted and re-implemented in a heterogeneous-ISA CMP OS. However, it is not straightforward to directly port these mechanisms. To begin with, the capability to run the same binary across ISA-heterogeneous cores requires building the binary for the different ISAs and having them present in the local ISA execution planes. In addition, the OS must manage applications to enable heterogeneous ISA execution. For example, the OS must realize task-sharing specific to individual applications. The OS must have the capacity to run threads, which can share the heap memory and execute the jobs concurrently across heterogeneous cores.

This thesis presents such a guest OS design, called ZxOS, for heterogeneous ISA CMPs. ZxOS is developed by extending the open-source Zephyr OS [7]. Zephyr is a single application OS, and this application uses the entire system memory. ZxOS utilizes the shared memory interface provided by hetQEMU to interact with ISA-heterogeneous cores. Since this shared memory is built using the file-backed mechanism, virtual memory coherence between ISA-heterogenous cores is realized through the host processor's consistency model. We use an x86 CPU as the host processor, which defaults the consistency of ZxOS to x86's Total-Store-Order (TSO) [6] model. The ZxOS provides a shared heap across the machines using

the shared RAM area. With this shared heap, an application, during this initialization stage, allocates memory chunks that are available for both cores to access. With such heap synchronization, application threads can be split across heterogeneous cores. ZxOS follows the same single application model of Zephyr. However, it enables application execution on ISA-heterogeneous cores. This capability opens up a prototype for a simple guest OS that can schedule threads across ISA-heterogeneous cores.

1.2 Thesis Contributions

This thesis makes the following contributions:

- Design of an OS for heterogeneous ISA CMPs, called ZxOS. ZxOS modifies the fundamental building blocks of Zephyr OS to support its execution on hetQEMU, a software simulator for cache-coherent shared memory, heterogenous-ISA CMP. hetQEMU provides shared memory across the heterogeneous cores at an offset of 1GB. In particular, ZxOS updates Zephyr OS's architecture-specific code to enable the OS to access extended memory. ZxOS modifies individual OS's page mapping and enables them to access memory. ZxOS utilizes a section of the shared RAM to provide a globally-consistent shared heap for heterogeneous cores. ZxOS can run multi-threaded applications by allocating cores to threads and sharing the workload across the heterogeneous kernels.

- We evaluate ZxOS's functionality using an experimental study involving a set of micro and macro-benchmarks. The macro benchmarks include Black-Scholes from the PARSEC [24] benchmark suite and MD5 from the Starbench benchmark [10] suite. The micro-benchmarks include data structure implementations from the Linux ker-

nel's codebase, including that of red-black tree [5] and linked-list [4]. The evaluations demonstrate ZxOS's functionality. In particular, they show that a multi-threaded application's threads can be split across (simulated) ISA-heterogeneous cores for parallel execution. These thread's concurrent access of shared memory variables are observed to be consistent.

1.3 Thesis Organization

The rest of the thesis is organized as follows. Chapter 2 describes background of the thesis. It covers topics like the hetQEMU tool, available OSes platforms, simulation environments, system software challenges in heterogeneous ISA CMPs, and the baselining challenges of ZxOS. Chapter 3 describes the past and related work in thesis's problem space. The topics covered include QEMU, various OS concepts like page tables, memory management in VMs, cache coherency in chip memory, and file-backed memory. The next half of the chapter discusses Popcorn Linux, the PriME simulator, and how the hetQEMU simulates RAM for the guest OSes.

Chapter 4 discusses the design of ZxOS, including adaptations that enable it to work on hetQEMU, memory management modifications in Zephyr OS, and ZxOS's boot-up synchronization sequence. Chapter 5 describes ZxOS's key implementation details such as stack, heap synchronization, the global heap memory pool, and the thread offloading process. Chapter 6 discusses the experimental study that we conducted to evaluate ZxOS's functionality. The thesis concludes in Chapter 7 by drawing lessons learned, outlining the work's limitations, and describing future prospects.

Chapter 2

Past and Related Work

This chapter will discuss various available research platforms and simulation environments to realize heterogeneous ISA Chip Multiprocessors (CMP). Currently available systems are either primitive in their scope of testing due to limited support for architectures or the systems claiming heterogeneity do not have real heterogeneous cores in principle. Available heterogeneous ISA CMP simulation systems and a discussion on their shortcomings are studied. The chapter gives a road map to why hetQEMU is the right candidate for simulating cross-ISA CMP. ZxOS also demonstrates a no-cost heterogeneous ISA CMP system using the open-source Zephyr project.

Section 2.1 will detail the QEMU variant built by another research group whose work is not yet published. This QEMU functions as the backbone of ZxOS. Section 2.2 describe about the OS and platforms available for heterogeneous systems. Section 2.3 will elucidate the simulation and chip bring-up environments reported. Exploring this will set a foundation for understanding the intricacies present in heterogeneous ISA chips and provide valuable insights to delineate the choice of infrastructure in this study. Section 2.4 will discuss the infrastructures built in the system software for facilitating heterogeneous ISA environments. This section will explain the challenges in integrating various ISAs into a single SMP environment. The final Section 2.5 discusses the possible baselines to compare the results of this thesis and details the reasons challenges in baselining.

2.1 The hetQEMU Emulation Framework

The hetQEMU was built by another research group, while Zephyr porting for the hetQEMU was simultaneously taken up. This simulator was developed to explore the design space of hardware and system software for heterogeneous ISA platforms with cache-coherent (CC) shared memory. The traditional CC shared memory platforms have been studied in detail. The advantage of these platforms is the ready availability of prototypes and hardware to conduct research, and the development of system software to function on top of these systems is time tested. These conventional platforms have identical CPUs. Researches like Popcorn Linux[14] has explored non cache-coherent heterogeneous ISA systems as well. However, a CC shared memory heterogeneous ISA system is still not successfully verified and reported; thus, the full extent of hardware-software design is still not completely explored. The hetQEMU is not a contribution of this thesis, and it is developed by other students/collaborators to realize heterogeneous-ISA CC platforms. The group has not published the work yet. The hetQEMU system runs heterogeneous small-footprint OS to realize heterogeneous-ISA CMP in ZxOS.

2.2 OSes and Platforms for Heterogeneous Systems

Evidence about heterogeneous systems providing performance[43] [44] and security enhancements[44] [18] are previously well documented. Such developments have not come by without a fair share of operating system design modifications, to name a few Helios[35], The Multikernel[19], lightweight cross ISA calls[23], and coherent domain OSes[32]. These developments in operating system paradigms contribute to removing barriers in heterogeneous computing. Apart from new OS designs, research on energy efficiency in heterogeneous computing[21] is a

preeminent field of interest.

The primary focus for evaluating the simulation environments will comprise features like hardware coherence, security, and an appropriate platform for applications to utilize the heterogeneity. In general, heterogeneous systems utilize some form of nontransparent generic communication bridges to interface between different ISA processors. For example, this energy efficiency in heterogeneity project[21] utilizes a PCIe to connect other ISA processors. As discussed earlier, introducing a PCIe-based interconnect does not involve any modification in the hardware and modifying the software component drastically. Even though this allows us to harness the advantages of heterogeneous cores, the system software complexity brings other challenges in application development. Besides, the number of software overheads during the execution can degrade the overall performance.

Another exciting approach proposed by a performance research work[29] where the team managed to run static analysis on the plethora of software to funnel down the instructions that are most used and least used by a general-purpose processor. This technique creates the possibility of designing cores with reduced instruction support and running applications that do not require SIMD or other complex instructions. While running these applications on reduced cores, energy efficiency improved 12% and application speedups 15%. Though this system does not represent a heterogeneous-ISA CMP system, significant speedups could be obtained by reducing the functionality of a native core and providing cache coherence. This literature demonstrates the promising capabilities of hybrid cores with cache coherence.

The fundamental takeaway from the above projects is that cache coherence is a rudimentary necessity in the field of heterogeneous core computing for breaking the barriers efficiently across ISAs. ARM LITTLE_big[1] endian processors have also demonstrated similar characteristics. The Little big-endian architecture alternates the core of execution based on the demand of the application. When intensive performance is needed, the application executes

on the Big-endian core. Conversely, when power-efficient execution is required, the little-endian core kicks in. The critical aspect to observe is that though the processor's nature is different, we still have the same microarchitectural features of the ISA. The caching mechanisms and coherency across cores in ARMLITTLEbig is not a significant challenge, as they hail from the same silicon design and combine without any software effort. The key takeaway is that the micro-architectural nature of the processors alone is different in both the above examples. Instead, ZxOS addresses the melding of entirely different architectures to provide the same cache coherence in simulations.

2.3 Heterogeneous-ISA CMP Simulations

Coherence in heterogeneous hardware simulations has attracted the research community in recent times. BYOC[12] designed an FPGA-based simulation suite and open-sourced the solution to provide a cache-coherent heterogeneous chip simulation environment. The FPGA prototypes the core, allowing the user to decide the ISA and evaluate the performance. The project currently supports SPARC, RISC-V, and x86 32 bit architectures. There have been some positive results and observations using the BYOC environment[31]. However, widely used chips in mobile phones and IoT devices are usually from the ARM64 family, while server clusters exclusively use Intel and AMD x86-64 (64 bit x86). Unfortunately, the BYOC project does not support these ISAs. Complete environment support for these ISAs on the FPGA requires specific company support. Furthermore, ARM, a licensing entity, does not have easily accessible support for developers. Even if the architecture is present, it requires a separate group of engineers to work on the FPGA to make modifications, increasing lead time and cost. Currently, there are no solutions that offer heterogeneous-ISA CMP in silicon.

2.4 System Software for Heterogeneous ISA CMP

While we have discussed the disadvantages of hardware simulation in BYOC[29] models are, just bringing up the capability to share memory and interconnects across hybrid cores do not guarantee flexibility of cross ISA core chips. Stable system software is essential to run on top of heterogeneous systems to facilitate harmonious functioning. Cache coherence and flexible interconnects across the ISA boundaries require special software design to engage cores of different ISAs to work smoothly. Recently researchers have been reporting the capabilities of migrating or offloading a thread into another machine of different ISA like H-container[16] and Popcorn Linux[14]. The H-container internally uses the Popcorn toolchain to compile the same application in multiple ISA-compatible instances. When offloading threads to edge computers, this compatible binary is advantageous.

The Popcorn compiler, a flavor of LLVM-clang with extra code passes, is used to compile the LLVM Intermediate Representations of the source code into applications of respective ISAs. Once the application is compiled in their respective ISA binaries, the compilation moves into post-processing. The symbols are extracted and aligned across ISA binaries at identical offsets. This alignment script generates a linker script to space the symbols at similar offsets in all the architecture-specific binaries. Once the linker script is available, the binaries are re-linked using the object files to generate a set of binaries of each ISA that are now aligned.

In the case of Popcorn, the target machines are connected through the network through IP/sockets, or RDMA high-speed interconnects. When the application migrates onto another machine of different ISA, sharing code data across nodes through a manual page request mechanism. The DSM gives the illusion of shared memory, although the memory is present in different machines. These projects are excellent examples of independent systems sharing a process space. Popcorn also involves a communication overhead in fetching code, data, and

stack metadata, reducing performance. The extra effort of compiling applications using a different tool and hunting for dependencies that are not compliant with the Popcorn compiler can be challenging to resolve.

Another vital aspect to consider is the system's security in a heterogeneous setup. Allowing an external module to have the liberty of copying pages of processes in one architecture and exposing it on air while sending over the interconnect can potentially compromise the entire system's security. RDMA, for example, is powerful hardware capable of copying the page table entries of the process in a node and delivering it to the other system without the original machine's knowledge. Exposing sensitive data structures through these interconnects can open a massive surface for spurious attacks. Bypassing RDMA Security Mechanisms[38] has detailed potential threats in RDMA security.

2.5 Discussions and Baselines

This thesis demonstrates heterogeneous-ISA CMP simulation, the first of its kind. The chapter discusses related work in the same faculty. Baselining is crucial to estimate a system's capability and shortcomings. Comparing the Zephyr OS against simulated x86 and ARM Linux VMs does not provide a valid baseline. The Linux OSes have mature hardware management subsystems and better support for simulations. The simulation environment for Zephyr OS has its additional layer of processing and overhead because it is not time-tested. Furthermore, the Zephyr kernel cannot match the features and capabilities of Unix-like systems in terms of system software stability.

Another possible baselining is a comparison against a real heterogeneous-ISA CMP hardware. However, no such platform exists to test our applications. The hetQEMU requires additional effort to port any existing system stack to evaluate. Therefore, this thesis will not compare

Zephyr's OS performance against any environment.

To summarize, analyzing the existing research around heterogeneous computing helps us understand each model's merits and shortfalls. The above models either have extra layers of software and hardware that facilitate coherency[21] or have not yet reached the level of cc that a homogeneous setup provides. Heterogeneous simulation of core and cache coherence were the essential factors considered while designing ZxOS. The next chapter will discuss the various software components that build up the final ZxOS using hetQEMU.

Chapter 3

Background

This work focuses on harnessing hetQEMU's memory-sharing capability and cross-ISA possibilities to simulate heterogeneous chip multiprocessing (CMP). It is essential to understand the underlying mechanisms used in hetQEMU to enable the guest system to use cache coherence (cc) and demonstrate heterogeneous computing capabilities. This chapter will explore essential software components used in this thesis and their background.

Section 3.1 will briefly explore QEMU and hypervisors used for virtualization. Section 3.2 will present the shortcomings of using non-cache-coherent heterogeneous ISA CMP. The following Section 3.3 will explain paging mechanisms in operating systems to understand memory management in OS. Section 3.4 will evaluate how memory virtualization happens and how the host machine manages VM's memory. In Section 3.5 we will discuss how cache coherence is vital in multicore systems. Section 3.6 will cover the fundamental design components of hetQEMU and explain how virtual memory is created for the guest VMs. Section **??** will present an example of the management of heterogeneous programmability in multicore systems using Popcorn Linux. Section 3.8 discusses the PriME simulator and how it has inspired the design of hetQEMU. Section 3.9.

3.1 QEMU and Hypervisors

Virtualization is emulating or simulating vital resources such as filesystem, memory, network, storage. The most common form of virtualization discussed is OS virtualization. Virtualization enables sharing of single hardware among various OSes and fragmenting hardware and software resources. Hypervisors run on the host machine and handle the internals of virtualizing the hardware. The application running on top of the virtual hardware is called the guest system. Hypervisors can be software, hardware, or firmware that work with the host to bring up a guest system like QEMU Bellard [20]. The hypervisor provides various resources and interfaces to the guest system and facilitates the guest OS to run virtually, mimicking its physical execution. The guest OS traps when the VM performs a privileged operation(page table update, halting the CPU), and the hypervisor takes control to manage the trap. This active management of hardware interfaces maintains the guest system as a process in the userspace. The guest kernel cannot affect the host system unless specified through unique interfaces. This complete abstraction helps users run multiple OSes on top of the hypervisor. In the development industry, this helps the users reduce the hardware cost and help them test OSes frugally.

3.2 Simulations of CC Heterogeneous ISA CMP

Research like BYOC [12] focuses on heterogeneous ISA CMP and provides the possibility of running SMP Linux on them. FPGA-based simulation environment and booting SMP Linux are milestones in moving towards cross-ISA chips software development. However, building FPGA hardware to replicate such a setup is costly. It is not easy to replicate such platforms despite BYOC being an open-sourced project. The projects also do not support more

commercially available ISAs. In the past we have had researches like Popcorn Linux [14] , and Heterogeneous thread migrations [33] showcasing performance improvements in data center through heterogeneous computing. [15] has provided evidence about improvements gained through heterogeneous computing paradigms. However, these systems lack hardware like cache-coherent mechanism that helps in seamless memory sharing across heterogeneous cores. The DSM takes care of memory sharing, and it comes at a performance cost. When SMP applications run on such machines, it is vital to have multi-layer cache coherence to ensure semantic correctness.

Simulation environments are essential to developing hardware for heterogeneous ISA chips efficiently. Simulations provide insights into the challenges of running applications on heterogeneous ISA CMPs. Moreover, a simulation demands no extra cost, and it can evolve into a go-to solution for heterogeneous CMP researchers to innovate hybrid chip systems.

3.3 Page Tables

ZxOS uses the memory management unit of ARM64 and x86-64 to access memory beyond Zephyr's default embedded guest OS's limits. Zephyr being an embedded OS, cannot access memory beyond its primitives. Although Zephyr supports memory management unit functionality, it restricts the OS in its reach for extended memory. Understanding the internals of the memory management subsystem will enable us to modify Zephyr OS's memory subsystem to access the shared memory area. The Memory Management Unit (MMU) unit is responsible for resolving the linear address access made by the processes and converting them into a physical address. The conventional ISAs organize memory in blocks called pages of size 4KB or 2MB. This organization of pages facilitates the MMU to access memory efficiently. Page tables are an exciting concept due to their flexibility in reusing the same

virtual address across different processes.

For example, x86-64 systems support up to 4 levels: Page Table Entry, Page Directory, Page Directory Pointer Entry, and PLM4E. When the application accesses a virtual address, the CPU presents the address onto the data bus. The MMU then converts the linear address into a physical address. However, the process that accesses the memory is unaware of the conversion. This powerful concept allows OS developers to use the same virtual address across multiple applications. However, these virtual addresses point to different physical pages. Since the mapping is different for each process, the address resolution likely results in a different physical page for the same linear address.

Similarly, ARM has equivalent translation tables that convert virtual addresses to physical addresses. Though implementation varies significantly across ISAs, the principles remain the same. The key takeaway from this section is to understand that an entry must be present in the system's page table for a virtual address to access memory. Without an entry, the memory access becomes illegal and results in a system trap.

3.4 Memory Management in Virtual Machines

Virtual machines are a set of physical hardware that mimics a physical machine that enables the user to run an unmodified OS on top of it. This is typical of KVM-based virtualization. The hypervisor provides the virtual RAM to place and execute the guest kernel. A virtual machine has the exact mechanism of memory translation of the same ISA physical machine. The memory translation happens using the extended hardware MMU in the host OS, which converts the virtual address into a physical address. However, the guest OS functions on the virtual memory allocated by the virtualization drivers. When the guest operating system tries to access an address, the address is virtual for the guest OS. However, after the virtual

MMU translation, the address is translated as an intermediate address for the hypervisor. This intermediate address again goes through the MMU of the host OS and translates into the actual physical memory. Virtualization software manages this through a concept called Extended Page Tables (EPT).

The EPT is a hardware virtualization mechanism that generates memory virtualization. When EPT is enabled, address resolution for the guest OS happens in two stages. The guest's linear address translates into the host's linear address. Then the host MMU converts them into a physical address. This resultant address is a physical RAM location. The guest machine also contains a CR3 register, and its responsibility is to convert the guest's linear address into the guest's physical address. In x86 architecture, the CR3 register holds the base address of the PLM4E (base address of level 4 paging structure). The EPT's base address acts as the CR3 register for the guest MMU. It goes through 2 iterations of the page walk to finally reach the machine address.

3.5 Cache Coherent Memory

This section will explore cache coherence and why it is crucial in multicore systems with SMP. In microarchitecture, the designers need to prioritize CC. CC is the mechanism that ensures any shared data modified by one computational unit of the system must reflect on all the other parts of the system in a timely fashion. The caches store entries to enable fast access to a previously accessed memory location. Now it becomes interesting when multiple CPUs are caching the exact memory location. When one of the CPUs attempts to modify the shared data, it is the responsibility of the CPU to flush out the cache lines for the same memory in other processors. Having different copies of the same data becomes a semantic error in computing. After writing, we will have irregular data across cores if the architecture

fails to clear all the other caches. Also, if there are consequent writes and reads from multiple cores, the Translation Lookaside Buffer (TLB) needs to be cleared frequently when the local address space of the CPU is changed. This constant cache miss can result in a performance loss.

In this thesis, when two cores simulated exist in unison, it is mandatory to maintain memory consistency. HetQEMU provides this memory consistency through its file-backed RAM virtualization. The following section will explore the concept of file-backed memory used in hetQEMU.

3.6 File-backed Memory

File-backed memory is a mechanism that maps virtual address space to a file, and if the write mode is enabled, it reflects the contents on the memory on the file system. File-backed shared memory can be a handy feature sharing the contents of the mapped memory across multiple processes. When multiple processes share the same file, any modifications performed by one process reflect on all other processes reading the file. When a process opens a file, we get a unique file descriptor. Although the file descriptor is different for each process, the inode instance for the file is the same across the processes. This technique ensures that the mapped memory for the file is consistent across the sharing processes. HetQEMU uses the /dev/shm present in Linux to allocate virtual RAM to be accessed by all hetQEMU instances. When processes call the do_mmap call, the kernel allocates vm_area_struct for that process and points it to the physical memory containing the shared file. As explained in the previous section, since the physical addresses are the same, only the independent process's page table has different virtual addresses in its context to access the same shared memory.

3.7 Popcorn Linux

Popcorn Linux is a flavor of Linux that provides heterogeneous ISA programming capability Olivier et al. [37]. The Popcorn project supports the migration of threads from one ISA machine to another through an IP socket or RDMA. The goal is achieved by providing a Single System Image (SSI) for processes and replicating the process address space across different ISA nodes. Popcorn Linux design philosophy aims to migrate applications on different ISA machines and provide transparency for the user space to view the migrated process's memory locally. This requires kernel modification to implement this functionality. Page caching mechanism ensures coherence in Popcorn. When the remote node executes an application that the current node previously was executing, the remote node will require stack data and previously updated global variables to be available locally. When the application accesses any page containing process data and is not available locally, the remote node's kernel requests the corresponding pages from the previous node. Popcorn uses the LLVM compiler to generate aligned binaries for each executing ISA. The alignment ensures that the validity of pointers is preserved across migration and facilitates correctness in execution across nodes.

3.8 PriME Simulator

The PriME: A Parallel and Distributed Simulator for Thousand-Core Chips [27] functions as the background for hetQEMU design. The PriME project involves creating thousands of x86 cores across several machines. The machines communicate using OpenMPI. The PriME project's goals are to simulate thousands of x86 cores through software simulation and demonstrate the capability to test Cache hierarchy, coherent memory, and NoC designs.

PriME demonstrates the capability to integrate vast clusters of machines and provide a simulation layer across machines with cache coherence. The hetQEMU project uses several of the design principles of PriME and demonstrates, first of its kind heterogeneous many-core simulators.

HetQEMU provides a reliable, coherent shared memory that the guest VMs can communicate without any software stack. hetQEMU targets in achieving more than simulate heterogeneous CMP with interconnects. The QEMU system functions as the consistency monitor and enables the system to map to memory regions and freely communicate with other VMs. HetQEMU also enables the OSes to use the inter-processor-interrupts (IPI) mechanism to forward interrupts to the other cores by modifying the interrupt controller simulation in QEMU. Device sharing capabilities and parallel bootup of guest OSes are other features. Since hetQEMU inherits all the features of QEMU, it is possible to boot up any compatible operating system and utilize the shared memory and IPI capability. HetQEMU instances communicate during boot up, and the virtual memory created is partitioned during this stage for each instance.

3.9 RAM Emulation in hetQEMU

QEMU supports multiple methods of host memory allocation for virtual RAM devices like anonymous memory, host RAM-backed memory, and file-backed RAM allocation. The file-backed memory allocation method is the same as POSIX standard-based shm_open for userspace applications. HetQEMU uses file-backed memory for guest memory simulation. The x86 and ARM QEMU instances share this memory in the host Linux kernel through the /dev/shm interface. The QEMU instances are allocated with 8GB of shared space. Since hetQEMU utilizes the file-backed memory for emulation, it is crucial to ensure that

this memory can maintain coherency across VMs. The x86 host Linux system ensures cc through the Total Store Ordering (TSO) consistency model. Host TSO model can ensure the coherency in-between the machines. The ARMv8 machines follow a weaker model for consistency, TSO coupled with reordering of writes-before-writes. This seamless memory model will ensure coherent shared memory in between machines.

Chapter 4

ZxOS Design

This chapter will describe the design of Zephyr-based OS customization to simulate heterogeneous-ISA Chip Multiprocessors (CMPs). The chapter discusses the challenges in Zephyr OS, and the constraints hetQEMU poses for booting the VMs with the desired parameters. Since the hetQEMU demands strict memory partitioning during the boot time, the chapter details how the kernels map pages to provide heterogeneous programmability. The modifications to the kernel and adaptations within the Zephyr build system are detailed.

The chapter organization is as follows. Section 4.2 discusses the requirements of the guest kernel to work with hetQEMU. Section 4.3 discusses the techniques to enable the Zephyr OS to expand its horizon of memory access. Freedom in memory access is crucial for access to shared areas. Section 4.4 discusses the startup synchronization of heterogeneous cores. Section 5.1 discusses the synchronization of automatic data in between cores. Finally, Section 5.2 details issues and solutions for synchronization of heap data across cores.

4.1 ZxOS Design

The core design of ZxOS is the modification performed on the Zephyr's sources and build system. This section will describe the changes to boot the Zephyr OS with the hetQEMU. The subsections will brief the steps to integrate the Zephyr kernels into the system. The following sections will detail the challenges in adopting the kernels in depth.

4.1.1 x86 Zephyr OS Adaptation

The x86 Zephyr kernel runs in the kernel mode in this thesis. Kernel-mode enables us to develop or integrate thread-based applications freely. Currently, Zephyr does not support thread creation in userspace at the current state. Once the execution context is in place, the next step in integrating the kernel with hetQEMU is memory access modifications. The hetQEMU does not restrict the address space for the x86 kernel. The x86 kernel has architectural limitations to execute from the address 0x0. The Zephyr kernel binary also gets generated at the exact location. However, since the OS builds to execute within a small memory footprint, the system does not have the reach to access memories outside the range of x86 binary. The build system only aids in generating the memory map within the ranges of the ELF fine generated. The Zephyr build system was tweaked to generate different memory mappings, and the device tree sources were modified to accommodate the new pages required for memory mapping. After all the changes, the hetQEMU was unable to boot the system. The build system leaves traces of unused sections that were causing the binary to get corrupted. The corruption required the post-build steps to remove unwanted sections from the ELF file. After removing these sections, the kernel becomes bootable with the hetQEMU.

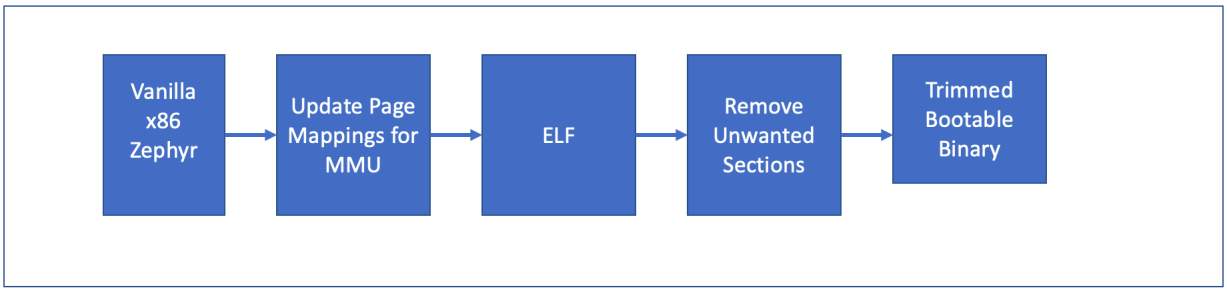

Figure 4.1: x86 modifications for hetQEMU

4.1.2 ARM Zephyr OS Adaptation

The ARM64 Zephyr kernel works in user mode. Similar to the x86's memory requirements, the ARM's translation tables had to be updated to enable access for the kernel to shared memory. Zephyr, by default, builds the memory mappings only for the ELF file's ranges. The hetQEMU's ARM64 boot parameters demand the ARM binary be present in the start address 0xA0000000. The Zephyr's build system provides us with an offset configuration to move the kernel to a different location. The build system still hardcoded the startup code at the address at 0x40000000. The start address that Zephyr provides violates the condition of hetQEMU. Consequently, the linker section was updated, and a particular Zephyr patch was applied for the ARM64 files to allow the movement of the kernel as per the user's requirements. To boot the kernel, remove the unused parts of the generated ELF. With these changes, the kernel moved the startup code to 0xA0000000 and successfully boots on the hetQEMU.With this kernel movement, the requirements of hetQEMU were obliged. These changes successfully allowed the kernels to boot, and shared memory areas became accessible.

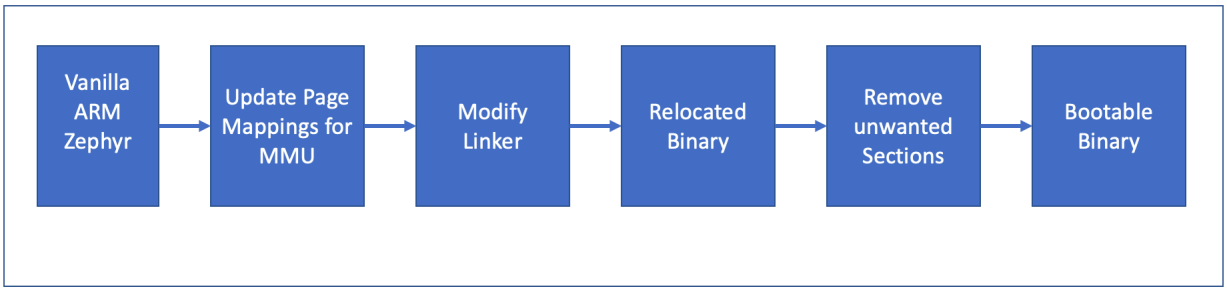

Figure 4.2: ARM modifications for hetQEMU

4.2 Kernel design Requirements from hetQEMU

HetQEMU's firmware performs a boot time partition of virtual RAM, and this partitioning must be obliged by the guest OSes for the proper functioning of VMs. Failure to do this results in kernel memory corruption by other VM instances. This section will detail the kernel requirement from hetQEMU and how the kernels are modified to fit into the system. Figure4.3 explains the system emulation of hetQEMU.

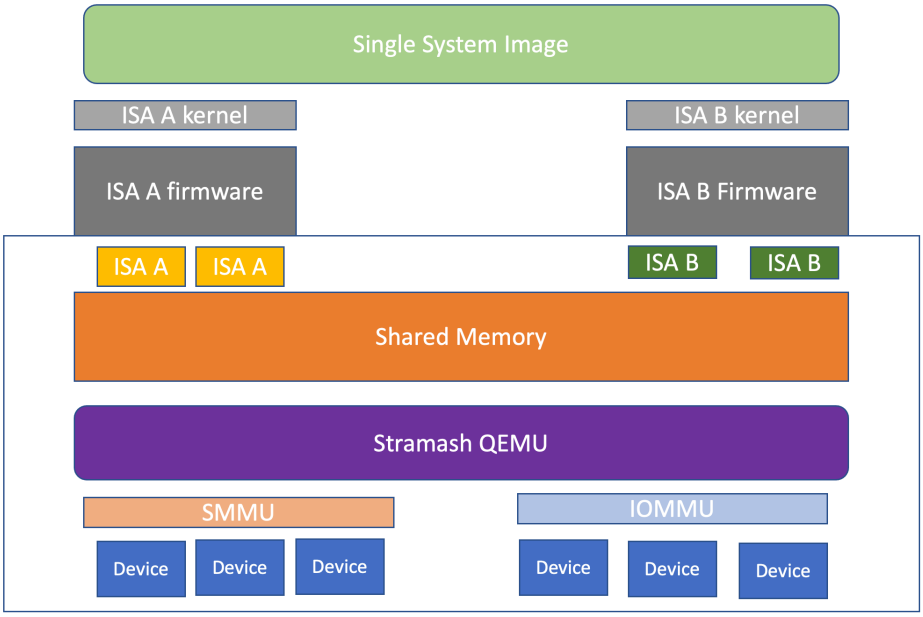

Figure 4.3: Generic hetQEMU design

The hetQEMU dictates the guest kernel to be built in a specified fashion and expects it to execute from the partitioned memory. The build system works using Zephyr's internal West tool, and its modifications are complex to perform. Zephyr kernels are small footprint OS and do not violate the size requirements of hetQEMU. However, the kernel also has minimal available memory and subsystem support. Zephyr OS is compatible with the memory footprint requirements of Stramash.

As discussed earlier, hetQEMU performs RAM partitioning at the boot time of each ISA VM. It is crucial to adopt the kernels to specified locations on the hetQEMU. The Zephyr OS has specific location constraints of its own. The x86 kernel, due to its inherent design primitives, cannot be altered with its boot locations. However, the build system has the flexibility of modifying the linker scripts to adopt the memory requirements of hetQEMU. Zephyr builds its ARM kernel at the static address of 0x40000000. For this thesis, we modified several Zephyr configurations and settings to remove any constraints in relocating the kernel. Relocation constraints even meant not using several subsystems, which have hard-coded memory locations in the default memory map. Once the system limitations were out of the way, the linker script was modified to adopt the necessary memory blueprint. The linker modifications makes the Zephyr kernels acceptable for hetQEMU. hetQEMU currently expects the ARM kernel to be booted at 0xA0000000 as the load address, although it creates memory from 0x40000000 for other purposes. Figure4.4 depicts the memory partitioning performed on 4GB of simulated RAM.

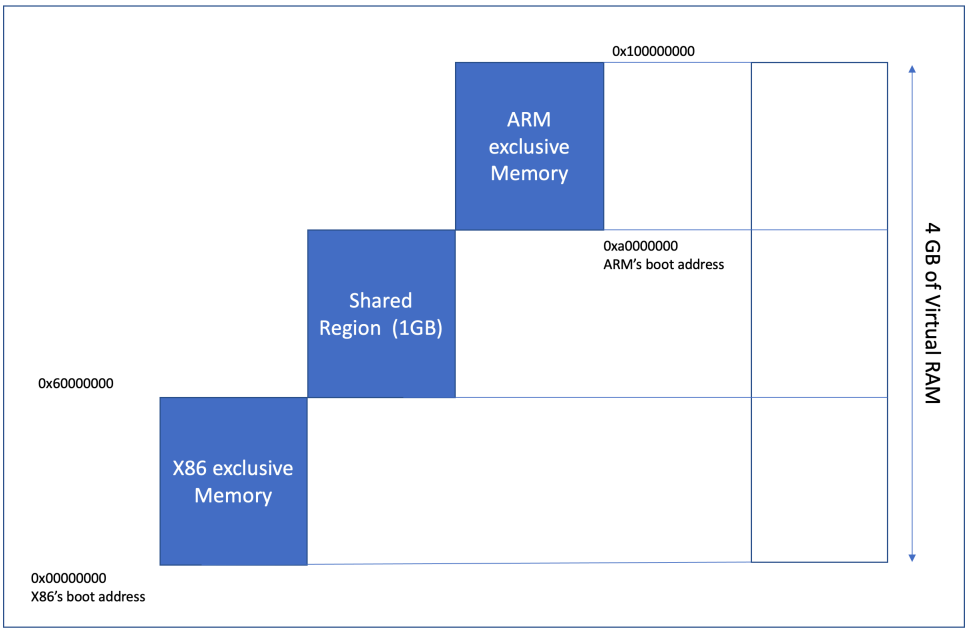

Figure 4.4: Memory organization in hetQEMU

Another essential design feature of hetQEMU is the offset in shared memory regions within the kernel. Enabling the movement of the guest kernel through linker scripts allowed the OS to own and use the Shared region as a common pool of memory. The x86 VM's shared memory ranges from 0x20000000 to 0x60000000. Similarly, the ARM VM's kernel ranges synchronizing with the x86's shared ranges are 0x60000000 to 0xA0000000. For example, any data written at the x86 VM's physical address 0x20000000 is coherently visible at the physical address 0x60000000 of ARM VM. This mechanism seems harmless, but we discuss the issues caused by this design and how this thesis handles the offset in the following sections.

4.3 Memory Mapping Modifications in Zephyr OS

The addresses of the kernels are strictly hardcoded by hetQEMU's design. The Zephyr's x86 and ARM instances being OS, the build system by default does not provide the capability to access large areas of memory such as 1GB in our case. We modify the page table creating mechanism in hetQEMU to handle this.

The first method that we attempted was to modify the linker sections to add additional memory ranges and map these ranges for accessing the shared memory. This technique provided shared sections. However, this does not allow these shared regions beyond the 32KB. The Zephyr project creates page mappings only until the Zephyr kernel's image ranges. The decision to modify and add additional mappings for the extended memory proved productive. The page tables are generated uniquely for each architecture within Zephyr. A script generates the x86 page table to map the memory regions for access. This thesis aimed to modify the external build steps that manage memory instead of writing our page tables. The design decision reduced the development time drastically. With the successful adaption

of page tables in the x86 kernel, the build requires additional 513 system pages for accommodating the newly mapped pages. Finally, the Device Tree Sources (DTS) were updated to provide more SRAM area to accommodate the extra 2MB of space required by the new page mapping tables.

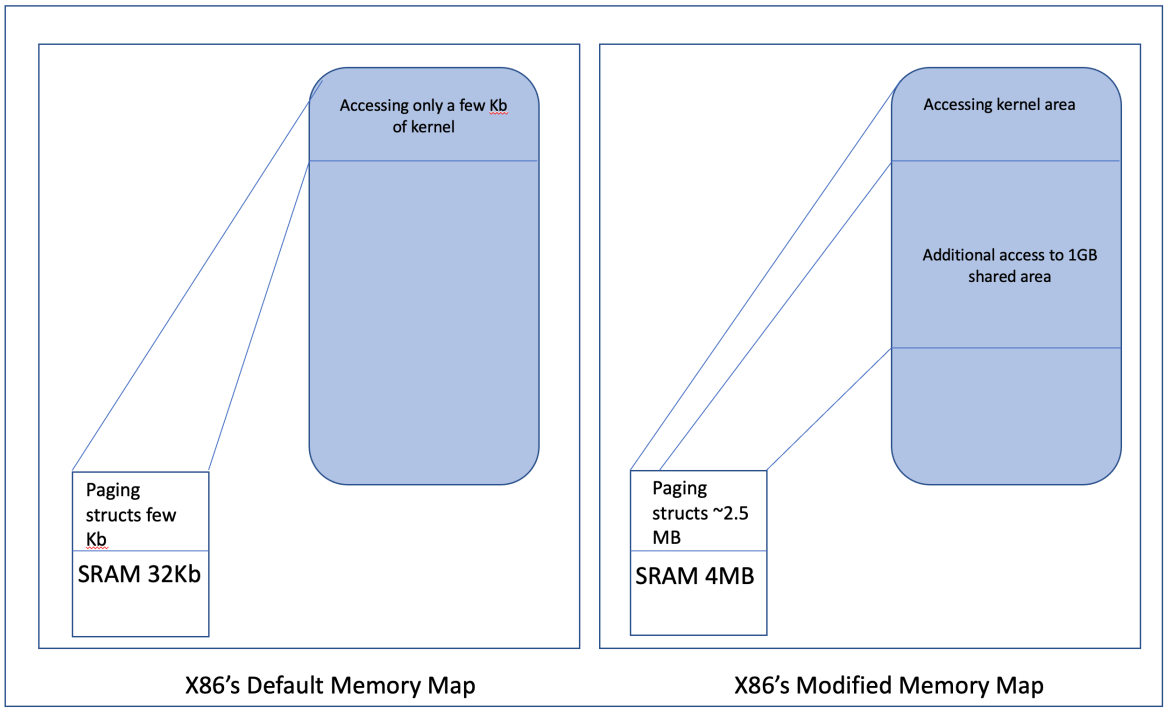

Figure 4.5: Memory map modifications in x86

Contrary to the x86 kernel, the ARM Zephyr kernel does not post-build the page table setup and integrates the mapping within the kernel build environment. The ARM64 kernel builds the page tables following the section mapping of the kernel. The ARM version only creates a mapping for code, image, rodata, and other drivers integrated within the kernel. This compilation step reduces the scope of change outside the kernel and requires architecture-specific code changes. The Zephyr ARM's architecture-specific MMU program provides an interface to add any number of additional regions in its latest release. With this support, we could comfortably map the 1GB area from 0x60000000 to 0xA0000000 into the memory map.

The ARM version of Zephyr, by default, has all infrastructure in place to map extra memory and requires no changes from the SRAM extension standpoint. With modifications in place with the MMU-related infrastructure, the kernels are fully capable of accessing the shared memory ranges. One of the main motives of this thesis is to maintain the originality of the Zephyr micro-kernel while enabling us to realize heterogeneous chip emulation. Utmost care is essential to not modify Zephyr's core kernel and design aspects. Designing ZxOS enabled the system to have large chunks of a shared free memory pool. It is not by design within Zephyr to support dynamic thread creation. This limitation brings roadblocks in running all the conventional benchmarks in Zephyr. It is possible to change the core scheduling and threading model and enable the system to run heavier applications using ZxOS's extended heap. However, this removes the originality of the Zephyr OS and its OS nature.

4.4 Kernel Handshaking and Synchronization

This section describes the integration of the Zephyr OS with coherent shared memory. With the modifications in the page table, it is possible to read and write data across kernel instances. A simple handshaking mechanism at the kernel entry function ensures systematic application execution with a dedicated memory range for this purpose. hetQEMU provides symmetric booting of the system, and partitioning on physical RAM per-VM enables the kernels to not use the entire memory area. The partitioning helps restrict memory access and avoids cluttering of opposite kernel areas. When the kernels startup, the first step is to check for a valid flag in the isolated shared memory. Flags indicate that the other core is already present in the system. If no valid handshake flag is present, the system is considered first to enter the hetQEMU system, writes the handshaking flag, and waits for the other core to come online.

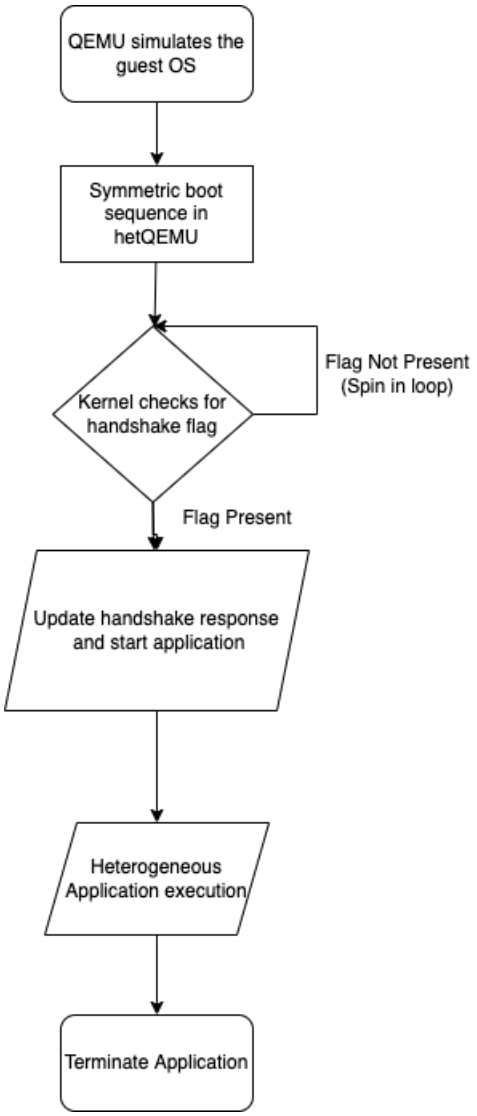

Figure 4.6: Startup flow in ZxOS

The startup mechanism is simplified as the Zephyr OS has limited support in terms of features such as worker threads. The Zephyr project does not support dynamic thread creation as the thread's metadata should be present at compile time. Although the POSIX thread implementation in Zephyr follows the standard, Zephyr's "pthreads" are not as versatile as we would encounter on a Unix-like kernel. Every thread created in the kernel needs to have a static stack area on which the threads need to limit their function. Such limitations naturally exist when choosing an OS. However, the focus of the thesis is to design a guest operating system for a heterogeneous-ISA CMP. The ZxOS is the first such guest OS design.

Chapter 5

Implementation

This chapter will discuss the implementation details of simulating a heterogeneous ISA chip multiprocessor (CMP) using Zephyr and hetQEMU. The previous chapter elaborated on the adaptation of Zephyr kernels for hetQEMU, kernel synchronization during startup, and memory constraints while using Zephyr with hetQEMU. This chapter will detail the Zephyr-hetQEMU system's implementation details and the heterogeneous execution environment implementation. This chapter elaborates on ZxOS's internal implementation fundamentals and its sharing of threads across heterogeneous cores.

This chapter is organized as follows. Section 5.3 details the information on the shared global heap implementation. This memory pool acts as the primary source for heap allocation. Section 5.4 describes the mechanism used to offload threads across nodes. The offloading mechanism enables the application to share the workload across heterogeneous nodes. Section 5.5 discusses why integrating applications with external dependencies are challenging. It also details the restrictions on the capability of Zephyr's POSIX thread library and challenges in integrating a sophisticated POSIX threading library to improve the system's capabilities.

5.1 Stack Synchronization

Once booted, the heterogeneous cores enter the main application built into the kernel. The benchmarks tested on this system are multi-threaded and provide the advantage of executing

the work on the thread's stack. Good designs do not use stack variables as input parameters for spawning threads. Stack isolation in threads prevents the necessity of synchronizing local variables. Furthermore, threads usually accept a void pointer as the input parameter for the thread. However, the independent VM's heap is not shared symmetrically across cores. This asymmetry can bring challenges by introducing constant overhead in translating the heap addresses. The global memory pool cannot be freely used across the cores, thus requiring copying heap data across machines which increases overhead to offload the thread. The following section describes how this thesis handles the heap memory asymmetry problem and, by doing so, considerably reduces the overhead for heap data sharing.

5.2 Heap Synchronization

We now describe the mechanism used to handle the issue of overheads possible due to irregular heap organization in hetQEMU. The previous sections describe the memory partitioning in hetQEMU and the requirement of strict adherence in booting the kernel at specified locations. The x86 kernel can access memory from 0x0000000 to 0xC0000000 and the ARM can access memory from 0x40000000 to 0xFFFFFFFF. The memory ranges in both the kernels map to each other at an offset. There are memory holes that align with the location of the kernel. Due to the 0x400000000 bytes offset for accessing identical memory, a page table modification makes it efficient to access memory identically.

The page table mapping flexibility provides the kernels to map the physical address to virtual addresses. In the initial consideration of developing the kernel, it is debatable whether the MMU would be needed as it brings complexity and restriction with memory access. With MMU disabled, the kernels can access the entire 8GB of memory without restrictions. However, it becomes difficult for heap allocations to happen favorably to share across VMs.

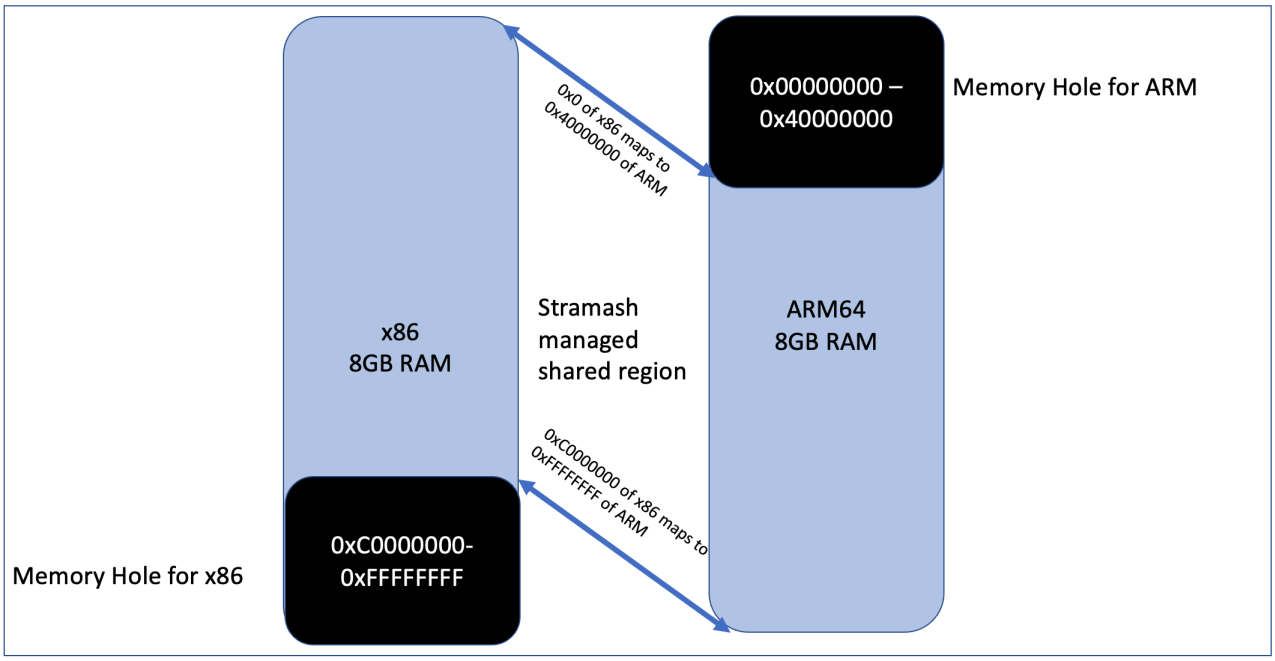

Figure 5.1: Virtual RAM mapping in hetQEMU

To illustrate this, when the x86 machine uses the inbuilt malloc, it allocates the memory pool from the compiled memory chunks that are limited. Size is not the only limitation, as the memory allocated will be inaccessible to the other core since it is allocated within a memory hole, as discussed.

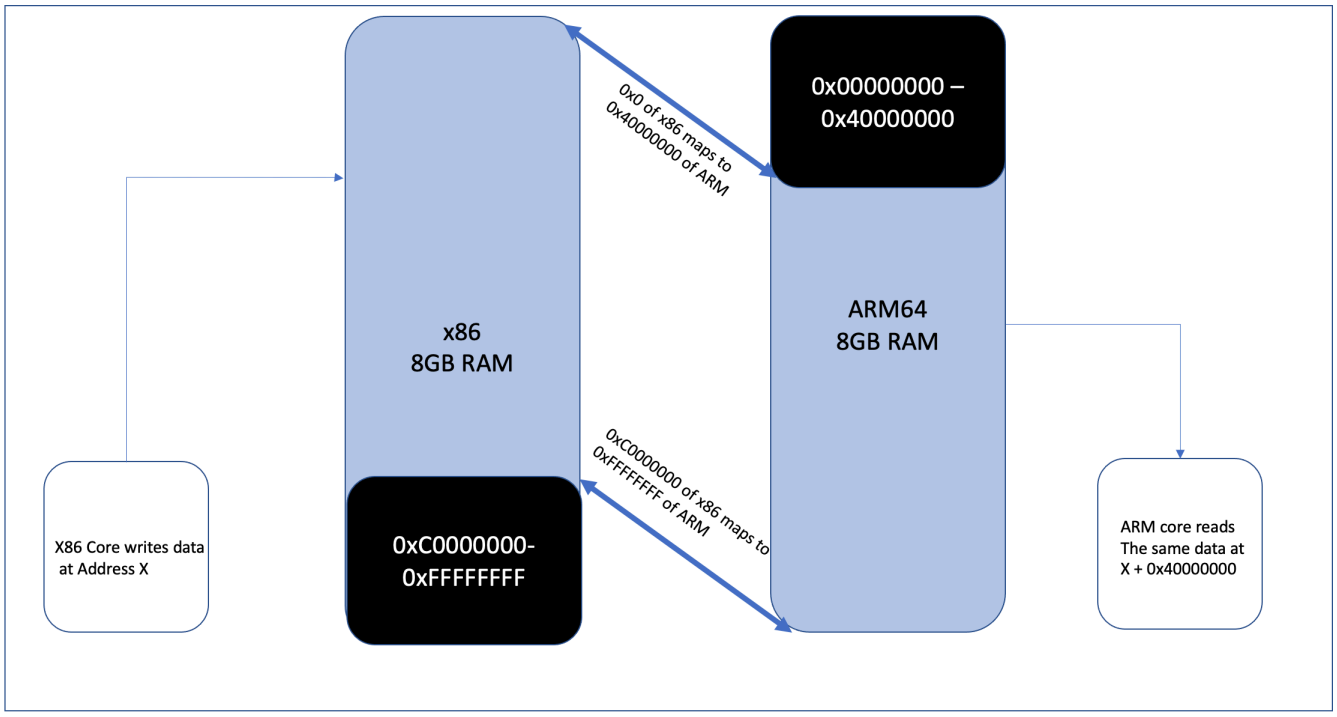

Figure 5.2: Asymmetric access of RAM in hetQEMU

Figure5.2 shows the typical memory access pattern while attempting to access shared memory from kernels. The process induces an overhead when accessing the heap contents in the applications. Attempting to perform conversions for every memory access requiring an offset update can reduce performance heavily. This overhead was solved using page table modifications in the x86 kernel version. Only one of the kernel page tables had to be updated for the guest kernels to share the heap area without the overhead.

The technique employed here is to map the shared physical addresses of the x86 VM to virtual addresses at an offset of 0x40000000. By doing this, the Zephyr kernels do not require any special math to access the same memory. The MMU of the x86 kernel performs the offset during address resolution, and applications require no change for accessing the heap addresses.

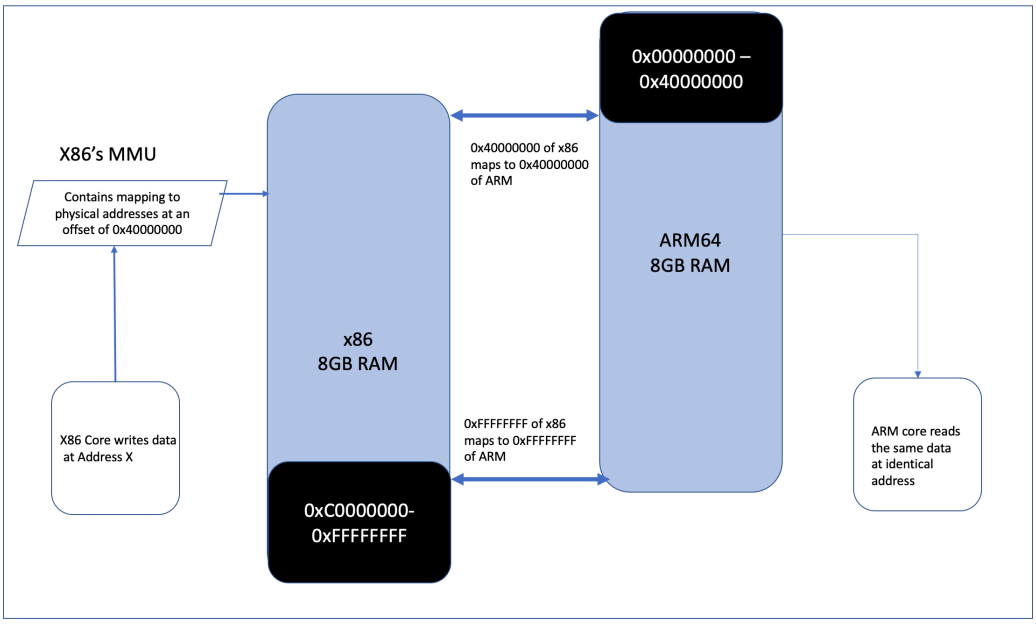

Figure 5.3: Symmetric access of RAM in this Thesis

With the aforementioned manipulations of the page table, the heap memory allocated at shared address space can be used across machines to run applications without any overhead. We have demonstrated some applications that allocate heap memory in the shared region and work simultaneously without race conditions across cores. This technique enables the user to run heterogeneous applications that use heap data seamlessly.

5.3 Global Heap Memory Pool

Modifying the page tables enables the ZxOS OSes to achieve shareable heap addresses without conversions. The Zephyr kernel allocates predefined memory chunks for dynamically allocating heap data. The build system and kernel configurations decide the size of the local heap in vanilla Zephyr. The memory provided is insufficient for the applications that execute on the system. The memory allocated through the inbuilt memory allocation APIs provides the heap memory within the memory hole of the other VMs. ZxOS solves this issue

by allocating a chunk of memory from the shared region to accommodate the heap requirements of large applications. The Black-Scholes benchmark of the PARSEC benchmark suite Christian [24] requires large arrays of input data visible across the threads. The input array alone constitutes several thousand bytes of data. Although the Zephyr kernels can extend memory through the device tree sources, this memory is not appropriate for sharing input data sets with the other cores. A global heap memory pool counters this issue.

The shared memory pool is a predefined memory location used exclusively for heap allocation. Currently, the system supports shared memory up to 1GB. The design requires specific areas for handshaking and other vital information sharing. This shared memory region functions as the memory pool for custom dynamic memory allocation APIs. The new memory allocation APIs provide heap memory from the global shared pool. The memory allocated through these new APIs provides a convenient way to allocate heap memory and share it across cores. The memory allocator follows a block-based split-merge mechanism to handle allocation requests.

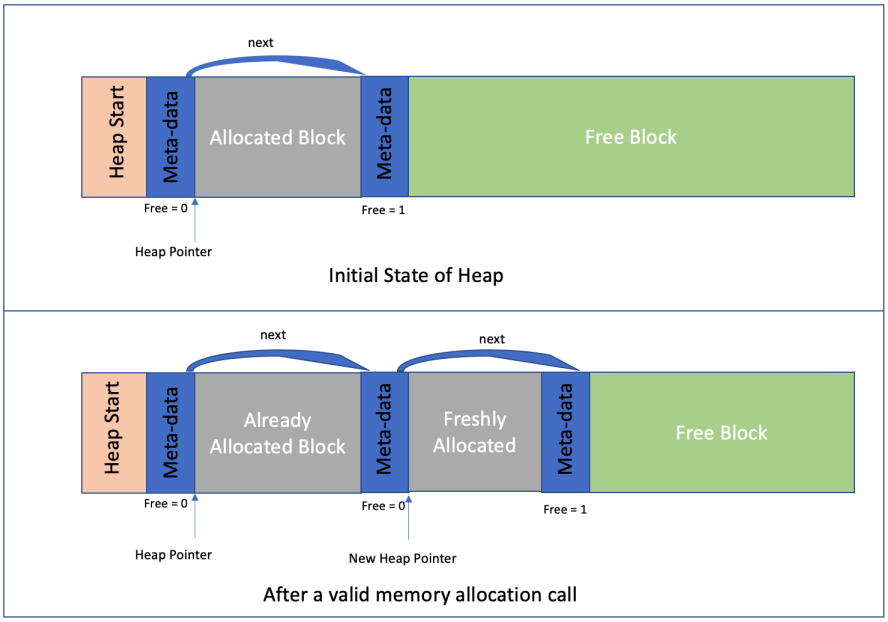

Figure 5.4: Custom Malloc Implementation. Split Logic for Partitioning Heap

Contiguous blocks of data characterize the heap memory pool. These block meta-data are present in the global shared area and help manage the heap memory. When a thread requests the global memory allocator for a chunk of memory, the API scans the linked list of block metadata until it identifies the first free metadata block. Since the memory range is quite extensive for the applications, the last data block holds the size for the rest of the free memory. When the allocator reaches the last block, it creates new block metadata at the offset of the requested memory size and marks the freshly allocated memory used. Similarly, when a memory chunk is released, the memory-releasing logic checks for consecutive blocks with free status and merges them into a single free block. The releasing logic then updates the combined size, which is available for allocation. This merging of freed areas improves the re-usability of freed memory. Refer Figure 5.5 for release and merge logic.

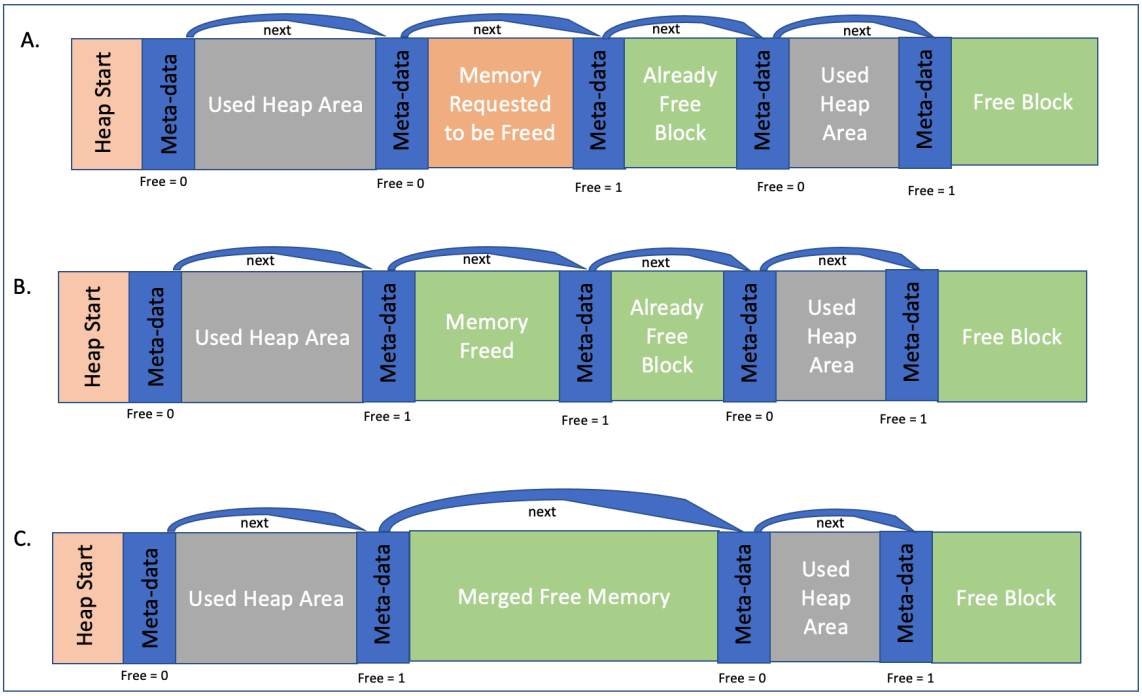

Figure 5.5: Heap Freeing Mechanism.

A. Memory Free request accepted. B. Block requested freed. C. Contiguous free blocks merged

Our experimental evaluations (See Chapter 6) confirms that shared variable accesses across heterogeneous cores work robustly. With the global shared memory pool introduced, both the machines can allocate memory for their respective applications and share the heap area without constraints. In all the applications used for benchmarking the system, we have allocated memory in the global shared region, and both the heterogeneous cores perform a fair share of the processing load, similar to a generic multi-core machine.

5.4 Offloading Threads

The main contribution of this thesis is to design the ZxOS that allows sharing the workload of processes, particularly at the abstraction of the applications running in the VMs, across heterogeneous cores. In ZxOS, cores signify the VMs. Threads created in machines concurrently execute across cores in a homogeneous CMP system. Zephyr OS does not possess multi-threading capabilities comparable to conventional Unix or Unix-like systems. Even the individual stack for the threads is configured at compilation time as specified in the previous chapters. However, by introducing the second core to work in unison, ZxOS demonstrates that heterogeneous cores can share tasks through threads.

POSIX thread functions get the parameters through a generic pointer. Most applications use this parameter for passing in complex data structures packed in as void pointers. Good coding standards enforce not to use local variables for thread parameter inputs. The main reason being if the function creating the thread exits the scope of the application, the threads will point to stack memory with contents irrelevant to the thread.

When an offload of thread occurs, the offloading core bundles all the heap metadata addresses required to execute at the follower core. The metadata offloading is manual. Automating this requires out-of-build system support and needs rewriting the entire build system. These

build modifications, if done, can remove the flexibility in changing kernel parameters on the fly and hinder development. Benchmarks used in this project have followed this convention and did not require any modifications to share the workload.

A similar approach is observed in the Popcorn project. Popcorn Linux uses the Popcorn compiler to align the binaries across architectures. This alignment enables the system to pause and migrate execution at any instruction the application decides. However, the focus of this project is to bring heterogeneity in the process while actively executing in both ISA cores. Once the second core receives an execution command of a specific task, the other core verifies the command and starts working on the shared heap data pointers with the same thread compiled for the core's ISA. ZxOS uses ARM64 and x86_64 ISAs to develop the CMP. The same architecture bus width ensures the memory alignment for sharing structures is similar.

A critical aspect of concurrent execution on ISA-heterogeneous CPU cores is shared-memory synchronization. Even in homogeneous CMP processors, synchronization is a research topic for eternity. For example, the MD5 benchmark from Starbench Parallel suite Andersch et al. [10] utilizes an atomic variable to count the number of iterations performed on the input data to produce the digest. The variable is atomic only within each local VM's context. However, the memory is not guarded by the micro-architectural atomic primitives when distributed. The synchronization of shared variables results from the compiler primitives such as FAA or CAS, which is a synchronization point that will flush x86's in-store processor buffer (where writes are locally buffered) to other cores and memory. These experiments on control variables provide us with confidence in the system's capability to run workloads despite not having shared micro-architectural primitives. This thesis demonstrates a heterogeneous-ISA CMP simulation system in practice through these experiments. Due to the memory synchronization provided by the host Linux system, the ZxOS demonstrates seamless heterogeneous

multi-core execution capabilities.

5.5 Zephyr OS Challenges

Due to its inherent embedded OS nature, the Zephyr project has many restrictions in integrating additional features not part of the build system. The subsystems and libraries provided are not supported across all the architectures within Zephyr. For example, the ARM64 build variant of Zephyr does not support the math library in userspace. For this reason, an externally compiled math library is integrated into the project.

One of the biggest challenges in integrating the Zephyr OSes into hetQEMU is the memory layout of the OSes. The memory layout is designed to place the x86 kernel at the start of the RAM and the ARM64 kernel at address 0xA0000000. However, the default Zephyr kernel places the ARM kernel at the SRAM address 0x40000000. These boot locations would violate hetQEMU's requirements. By modifying the kernel configurations and linker setting, the kernel and all the libraries are placed at the appropriate locations to meet hetQEMU's needs. After the kernel movement to the appropriate locations, an additional patch set from the Zephyr community enabled ZxOS to move the kernels as per requirements freely.

The POSIX thread implementation of a system couples the underlying scheduler and synchronization mechanisms with the threading model. Although the Zephyr system supports running threads, its design is inefficient enough to support a more extensive range of applications. For example, Zephyr's threads do not have the thread contention parameter. The scheduler utilizes the contention of the thread to schedule the tasks based on their resource affinity level and scope. Integrating the Unix-based pthread library is improbable as primitives at the scheduler level to engage the threads are not present. This direct dependency on the core scheduler prevents integrating the external threading libraries.

Chapter 6

Evaluation

This chapter presents experimental evaluation of ZxOS. As discussed before, the Zephyr's kernels have limited support for threads. A careful selection of multi-threaded benchmarks is run on the system to evaluate the heterogeneous scheduling capacity of the system. The core evaluation of the ZxOS is to demonstrate the capability to run applications on heterogeneous cores concurrently. By demonstrating the capability to share data across the cores and functions as a single unit, ZxOS is considered the first of its kind OS design capable of running on heterogeneous ISA CMP.

Section 6.1 details the selection criteria for the benchmarks and lists the benchmarks used to evaluate ZxOS. Section 6.2 discusses the necessity of concurrency for the data structures in a multicore system and provides evidence on the capability of ZxOS for shared memory synchronization across the heterogeneous cores.

6.1 Selection of Benchmarks

The previous chapter details the capabilities of the Zephyr kernel and an idea regarding the class of applications that are executable in the OS. This thesis demonstrates two macro benchmarks towards the evaluation of the ZxOS, i.e., PARSEC suit's Black-Scholes [24] benchmark and Starbench benchmark [10] suit's md5_bench. The benchmarks were chosen because they are self-contained and written in C with no external dependencies. The Zephyr

system has minimal support for C++ and limitations on the usable standard libraries. Since the operating system does not support OpenMP, Zephyr cannot use run-time-dependent parallelization features. The OpenMP is based on the pthread library of the target system. Apart from these benchmarks, the system was tested using the Linux kernel's red-black tree and linked list. These data structures shared across nodes were analyzed for various proportions of read-write cycles to evaluate the system's concurrency.

- PARSEC Black-Scholes Benchmark: The Black-Scholes model performs HPC style computations on financial data to find the theoretical value of an option contract. The calculation uses various input parameters such as current stock prices, expected dividends, and so on. The benchmark runs calculations on a similar set of functionalities in a multi-threaded fashion. The ZxOS splits the threads across the heterogeneous cores and performs these calculations concurrently.

- The Starbench MD5 Benchmark: The MD5 benchmark is an embedded scale benchmark that computes the hash of a large input dataset. The core calculation of the hash is performed in a loop with a global atomic variable as a counter. When the benchmark is executed in multi-threaded mode, the same loop is divided across threads, with the shared variable serving as the loop's counter. The ZxOS shares the hashing thread across heterogeneous cores and calculates. It is interesting to note that the sub-tasks are calculated on heterogeneous-ISA cores.

- Linux's Red-Black Tree and Linked List: The ZxOS tests the Linux RBT that is used to manage the completely fair scheduler. The ZxOS also tests Linux's list data structure. The intent for running these data structures in this system is to provide a path forward for the ZxOS to evolve into managing the processes across heterogeneous cores.

6.2 Concurrency Test

Concurrency tests are vital in evaluating the capabilities of the data structures and their methods used in multicores systems. One of the effective ways to evaluate a concurrent data structure in a multicore chip is to have synchronous locking primitives. Multiple threads should be allowed to access the locks and update the data structure harmoniously. The Linux kernel's RBT and list are time tested in a homogeneous multicore chip. This experiment aims to evaluate the capability of ZxOS to manage the concurrent data structure and maintain its integrity across heterogeneous cores. The experimental evaluation shown in Figure 6.2 displays various proportions of reads and writes on the RBT across the heterogeneous cores.

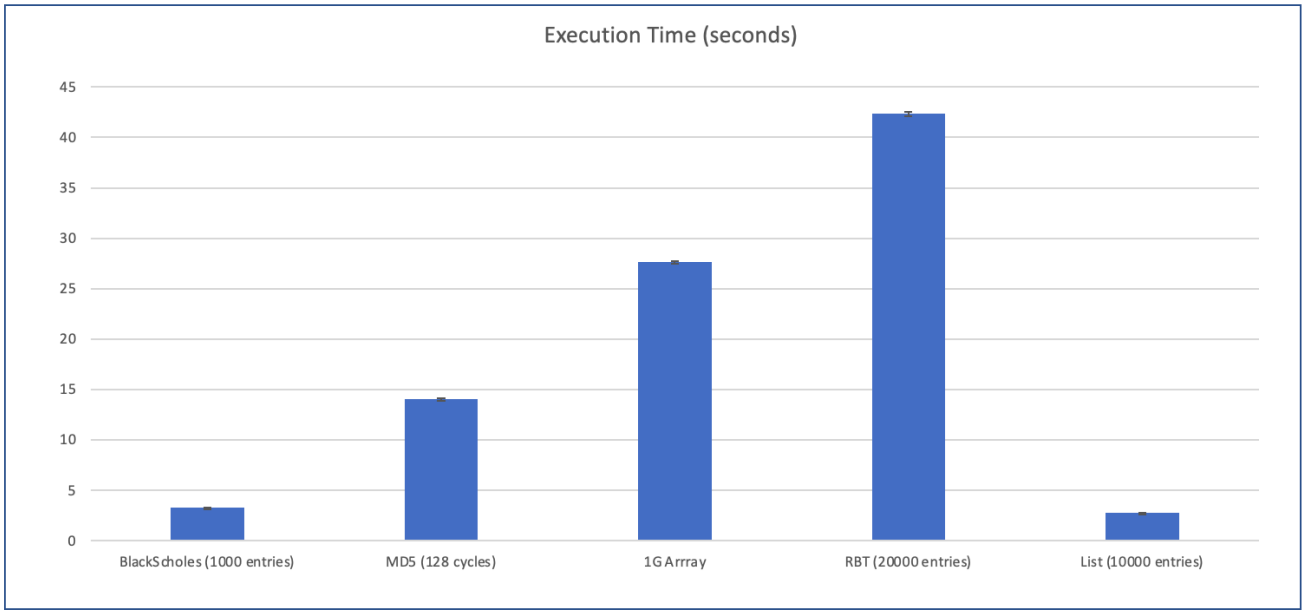

Figure 6.1: Execution time of micro and macro benchmarks on ZxOS.

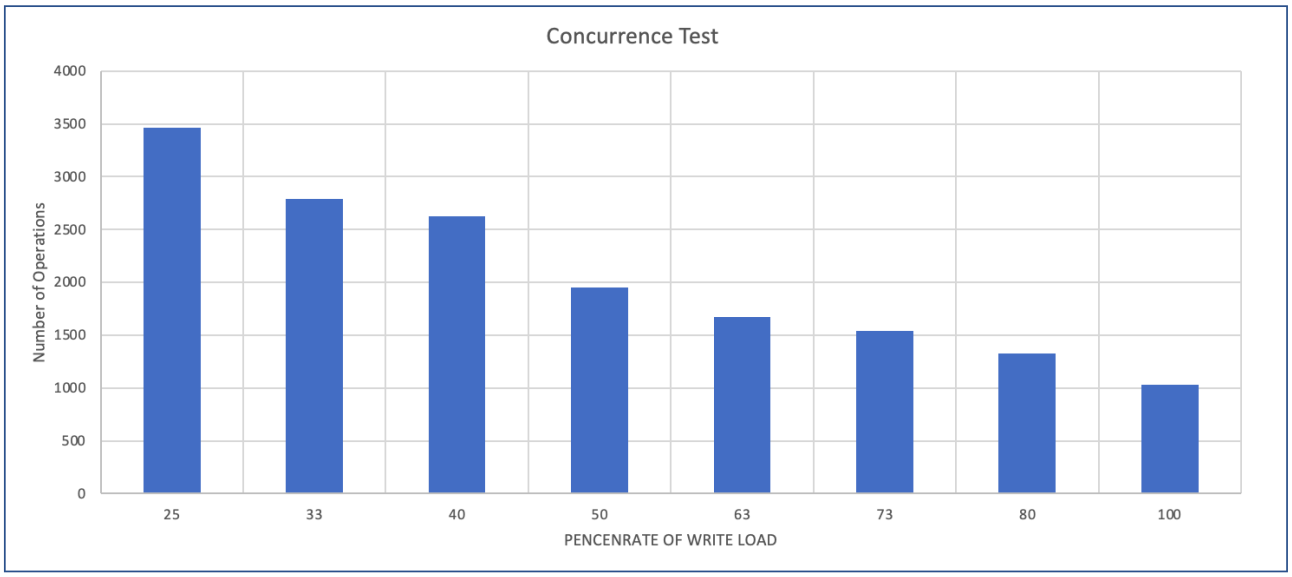

Figure 6.2: Concurrency test

6.3 Evaluation Summary

This chapter details the evaluation metrics used to validate the capabilities of ZxOS. ZxOS has shown reliability in executing synchronous workload across the heterogeneous-IAS cores. The OS has demonstrated its coherent memory properties by sharing atomic variables, barrier variables, and thread-specific data. ZxOS has also demonstrated the ability to share concurrent data structures across the heterogeneous cores and effectively use locking primitives to modify them. The OS built has displayed similar characteristics to a homogeneous multicore chip. The evaluation shows strong evidence to run concurrent multi-threaded applications in the ZxOS platform. The characteristics mentioned above validate the design as the initial prototype for Zephyr based heterogeneous-ISA guest operating system.

Chapter 7

Conclusions and Future Work

Multi-core processors became an important trend in chip design during the early 2000s to workaround the so-called "the end of Moore's law". Multi-core processors provided better power characteristics and the capability to perform concurrent tasks in parallel cores. Further innovations resulted in heterogeneous computing paradigms such as GPUs and FPGAs that enabled performance acceleration of applications or components thereof. Recently, Smart I/O devices such as SmartNICs and SmartSSDs introduced general-purpose cores within them for performing a number of tasks including low-level networking and storage tasks and offloading application workloads from host CPUs, among others. The Intel Agilex series provides configurable and low latency solutions to program application-specific programmable tiles to enhance performance. These trends enable heterogeneous-ISA computing through chip multi-processors having different ISA cores in the same machine or in the same package. However, no cache-coherent shared memory heterogeneous-ISA CMP exists. This impedes research in that space, especially on the design space exploration of efficient OS models for such an architecture.

This thesis presents ZxOS, a guest operating system for heterogeneous-ISA CMPs. ZxOS is an operating system that utilizes coherent shared virtual memory across ISA-heterogeneous cores that is emulated by a software simulator called hetQEMU. The shared memory interface provides the operating system with the perception of executing on the same processor with heterogeneous-ISA cores.

ZxOS uses the open-source Zephyr OS as its base infrastructure. Zephyr OS has supports x86_64 and aarch64 ISAs, the two architectures supported by the hetQEMU system. The Zephyr kernels were modified in their memory and architecture-specific subsystems to convert them into boot-able candidates for the hetQEMU environment. ZxOS manages the heap integrity of the application through shared memory. ZxOS was tested using multi-threaded micro and macro-benchmarks to evaluate its capability to execute threads across heterogeneous-ISA cores. The result of the evaluation confirms ZxOS this, and affirms its potential for exhibiting characteristics similar to a homogeneous multi-core CMP OS.

7.1 Limitations

ZxOS follows the build system of Zephyr OS. Although this thesis achieved a way to integrate external static libraries, the system does not function properly with libraries built out of the build system. This restricts ZxOS from integrating any external dependencies.

The threading capabilities of ZxOS are limited by Zephyr's scheduler and POSIX thread implementations. This restricts the ability to run threads at the user's discretion. The unsophisticated threading infrastructure reduces the scope of multi-threaded applications and does not support OpenMP-based applications. Currently, ZxOS needs modifications on the thread creation APIs in the benchmarks to enable it to be compatible with Zephyr's thread library. Threads cannot be dynamically created, and it requires a static allocation of the stack at compile time to run threads. These limitations are the primary factor in reducing the scope of applications running on the system.

7.2 Future Work

With the demonstration of running multi-threaded applications on a heterogeneous-ISA CMP simulation environment, ZxOS can be developed further into a sophisticated OS capable of thwarting the above restrictions. The ZxOS can be relieved from the Zephyr-based schedulers and threading libraries and be integrated with advanced subsystems. Introducing a new scheduler and supported libraries can enable the system to run OpenMP and sophisticated multi-threaded applications. The POSIX pthread library from UNIX systems can be ported, and dynamic threading capabilities can be provided.

By changing the infrastructure to a good sophistication, the following features can be added. An enhanced performance evaluation can be experimented using OpenMP applications. An user-space thread scheduler can be used to decide which thread runs on what core. A run-time environment can be integrated to decide which core runs what type of applications based on the ISA's inherent characteristics. An enhanced shared memory technique (Instead of file backed memory) can be integrated to improve the shared memory performance. Some other models of OS designs like replicated kernel model to perform run-time migration of tasks to heterogeneous core can be integrated and tested.

Bibliography

[1] Arm little big architecture. http://www.arm.com/products/processors/technologies/biglittleprocessing.php. Accessed: 2022-01-13.

[2] CCI cci arm. https://www.arm.com/products/silicon-ip-system/corelink-interconnect/cci-400. Accessed: 2010-09-30.

[3] Intel Aglilex hardware description. https://www.intel.com/content/www/us/en/products/details/fpga/agilex.html?wapkw=agilex. Accessed: 2010-09-30.

[4] Linked List kernel list. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/include/linux/list.h?id=v4.1-rc8#n1, . Accessed: 2010-09-30.

[5] RBT kernel rbt. https://www.kernel.org/doc/Documentation/rbtree.txt, . Accessed: 2010-09-30.

[6] x86 TSO consistency model. https://ieeexplore.ieee.org/document/546611https://link.springer.com/chapter/10.1007/978-3-642-03359-9_27. Accessed: 2010-09-30.

[7] The Zephyr RTOS [1], released by Linux Foundation, is an open source, scalable, and across multiple architectures of a real-time operating system (RTOS) optimized for resource-constrained devices. zephyr. :https://docs.zephyrproject.org/latest/introduction/index.html. Accessed: 2010-09-30.

[8] Amazon. Amazon Graviton. https://aws.amazon.com/ec2/graviton/, .

[9] Amazon. Amazon Web Services. https://aws.amazon.com/, .

[10] Michael Andersch, Ben Juurlink, and Chi Ching Chi. A benchmark suite for evaluating parallel programming models. In *Proceedings 24th Workshop on Parallel Systems and Algorithms*, 2011. URL http://www.aes.tu-berlin.de/fileadmin/fg196/publication/andersch01.pdf.

[11] Nils Asmussen, Marcus Völp, Benedikt Nöthen, Hermann Härtig, and Gerhard Fettweis. M3: A hardware/operating-system co-design to tame heterogeneous manycores. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, page 189–203, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450340915. doi: 10.1145/2872362.2872371. URL https://doi.org/10.1145/2872362.2872371.

[12] Jonathan Balkind, Katie Lim, Michael Schaffner, Fei Gao, Grigory Chirkov, Ang Li, Alexey Lavrov, Tri M. Nguyen, Yaosheng Fu, Florian Zaruba, Kunal Gulati, Luca Benini, and David Wentzlaff. *BYOC: A "Bring Your Own Core" Framework for Heterogeneous-ISA Research*, page 699–714. Association for Computing Machinery, New York, NY, USA, 2020. ISBN 9781450371025. URL https://doi.org/10.1145/3373376.3378479.

[13] Francisco J. Ballesteros, Noah Paul Evans, C. H. Forsyth, Gorka Guardiola, Jim McKie, Ronald Minnich, and Enrique Soriano. Nix: A case for a manycore system for cloud computing. *Bell Labs Technical Journal*, 17:41–54, 2012.

[14] Antonio Barbalace, Binoy Ravindran, and David Katz. Popcorn: a replicated-kernel os based on linux. pages 123–138, July 2014. Linux Symposium 2014, OLS 2014 ; Conference date: 14-07-2014 Through 16-07-2014.

[15] Antonio Barbalace, Robert Lyerly, Christopher Jelesnianski, Anthony Carno, Ho-Ren Chuang, Vincent Legout, and Binoy Ravindran. Breaking the boundaries in

heterogeneous-isa datacenters. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, page 645–659, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450344654. doi: 10.1145/3037697.3037738. URL https://doi.org/10.1145/3037697.3037738.

[16] Antonio Barbalace, Mohamed L. Karaoui, Wei Wang, Tong Xing, Pierre Olivier, and Binoy Ravindran. Edge computing: The case for heterogeneous-isa container migration. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '20, page 73–87, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450375542. doi: 10.1145/3381052.3381321. URL https://doi.org/10.1145/3381052.3381321.

[17] Antonio Barbalace et al., 2022.

[18] Elena Gabriela Barrantes, David H. Ackley, Stephanie Forrest, Trek S. Palmer, Darko Stefanovic, and Dino Dai Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, CCS '03, page 281–289, New York, NY, USA, 2003. Association for Computing Machinery. ISBN 1581137389. doi: 10.1145/948109.948147. URL https://doi.org/10.1145/948109.948147.

[19] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: A new os architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 29–44, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605587523. doi: 10.1145/1629575.1629579. URL https://doi.org/10.1145/1629575.1629579.

[20] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, page 41, USA, 2005. USENIX Association.

[21] Sharath K. Bhat, Ajithchandra Saya, Hemedra K. Rawat, Antonio Barbalace, and Binoy Ravindran. Harnessing energy efficiency of heterogeneous-isa platforms. In *Proceedings of the Workshop on Power-Aware Computing and Systems*, HotPower '15, page 6–10, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450339469. doi: 10.1145/2818613.2818747. URL https://doi.org/10.1145/2818613.2818747.

[22] Ian Buck. Gpu computing: Programming a massively parallel processor. In *International Symposium on Code Generation and Optimization (CGO'07)*, pages 17–17, 2007. doi: 10.1109/CGO.2007.13.

[23] Shenghsun Cho, Han Chen, Sergey Madaminov, Michael Ferdman, and Peter Milder. Flick: Fast and lightweight isa-crossing call for heterogeneous-isa environments. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 187–198, 2020. doi: 10.1109/ISCA45697.2020.00026.

[24] Christian. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[25] Erik P. DeBenedictis. It's time to redefine moore's law again. *Computer*, 50(2):72–75, 2017. doi: 10.1109/MC.2017.34.

[26] Matthew DeVuyst, Ashish Venkat, and Dean M. Tullsen. Execution migration in a heterogeneous-isa chip multiprocessor. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating*

Systems, ASPLOS XVII, page 261–272, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450307598. doi: 10.1145/2150976.2151004. URL https://doi.org/10.1145/2150976.2151004.

[27] Yaosheng Fu and David Wentzlaff. Prime: A parallel and distributed simulator for thousand-core chips. In *ISPASS 2014 - IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS 2014 - IEEE International Symposium on Performance Analysis of Systems and Software, pages 116–125, United States, 2014. IEEE Computer Society. ISBN 9781479936052. doi: 10.1109/ISPASS.2014.6844467. Copyright: Copyright 2014 Elsevier B.V., All rights reserved.; 2014 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2014 ; Conference date: 23-03-2014 Through 25-03-2014.

[28] Behnam Khaleghi, Sahand Salamat, Mohsen Imani, and Tajana Rosing. Fpga energy efficiency by leveraging thermal margin. In *2019 IEEE 37th International Conference on Computer Design (ICCD)*, pages 376–384, 2019. doi: 10.1109/ICCD46524.2019.00059.

[29] Wooseok Lee, Dam Sunwoo, Christopher D. Emmons, Andreas Gerstlauer, and Lizy K. John. Exploring heterogeneous-isa core architectures for high-performance and energy-efficient mobile socs. In *Proceedings of the on Great Lakes Symposium on VLSI 2017*, GLSVLSI '17, page 419–422, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349727. doi: 10.1145/3060403.3060408. URL https://doi.org/10.1145/3060403.3060408.

[30] Huaicheng Li, Mingzhe Hao, Stanko Novakovic, Vaibhav Gogte, Sriram Govindan, Dan R. K. Ports, Irene Zhang, Ricardo Bianchini, Haryadi S. Gunawi, and Anirudh Badam. *LeapIO: Efficient and Portable Virtual NVMe Storage on ARM SoCs*, page 591–605. As-

sociation for Computing Machinery, New York, NY, USA, 2020. ISBN 9781450371025. URL https://doi.org/10.1145/3373376.3378531.

[31] Katie Lim, Jonathan Balkind, and David Wentzlaff. Juxtapiton: Enabling heterogeneous-isa research with risc-v and sparc fpga soft-cores, 2018.

[32] Felix Xiaozhu Lin, Zhen Wang, and Lin Zhong. K2: A mobile operating system for heterogeneous coherence domains. *ACM Trans. Comput. Syst.*, 33(2), jun 2015. ISSN 0734-2071. doi: 10.1145/2699676. URL https://doi.org/10.1145/2699676.

[33] Robert Lyerly, Antonio Barbalace, Christopher Jelesnianski, Vincent Legout, Anthony Carno, and Binoy Ravindran. Operating system process and thread migration in heterogeneous platforms. April 2016. URL http://www.cs.utexas.edu/~mars2016/workshop-program/. The 2016 Workshop on Multicore and Rack-scale Systems, MaRS 2016 ; Conference date: 18-04-2016 Through 18-04-2016.

[34] N. Manjikian. A framework for simulation and prototype implementation of custom system-on-chip multiprocessors. In *2003 IEEE Pacific Rim Conference on Communications Computers and Signal Processing (PACRIM 2003) (Cat. No.03CH37490)*, volume 2, pages 646–649 vol.2, 2003. doi: 10.1109/PACRIM.2003.1235864.

[35] Edmund B. Nightingale, Orion Hodson, Ross McIlroy, Chris Hawblitzel, and Galen C. Hunt. Helios: heterogeneous multiprocessing with satellite kernels. In *SOSP '09*, 2009.

[36] Nvidia. SmartNIC: Computational Network Drive. https://blogs.nvidia.com/blog/2021/10/29/what-is-a-smartnic.

[37] Pierre Olivier, A. K. M. Fazla Mehrab, Stefan Lankes, Mohamed Lamine Karaoui, Robert Lyerly, and Binoy Ravindran. Hexo: Offloading hpc compute-intensive workloads on low-cost, low-power embedded systems. In *Proceedings of the 28th Interna-*

tional Symposium on High-Performance Parallel and Distributed Computing, HPDC '19, page 85–96, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450366700. doi: 10.1145/3307681.3325408. URL https://doi.org/10.1145/3307681.3325408.

[38] Benjamin Rothenberger, Konstantin Taranov, Adrian Perrig, and Torsten Hoefler. ReDMArk: Bypassing RDMA security mechanisms. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 4277–4292. USENIX Association, August 2021. ISBN 978-1-939133-24-3. URL https://www.usenix.org/conference/usenixsecurity21/presentation/rothenberger.

[39] T. Sakurai and A. El Gamal. Multi-million gate asic's. In *Symposium 1993 on VLSI Circuits*, pages 95–95, 1993. doi: 10.1109/VLSIC.1993.920555.

[40] Samsung. SmartSSD: Computational Storage Drive. https://samsungsemiconductor-us.com/smartssd/.

[41] J. Stuecheli, B. Blaner, C. R. Johns, and M. S. Siegel. Capi: A coherent accelerator processor interface. *IBM J. Res. Dev.*, 59(1):7:1–7:7, jan 2015. ISSN 0018-8646. doi: 10.1147/JRD.2014.2380198. URL https://doi.org/10.1147/JRD.2014.2380198.

[42] Stephen M. Trimberger. Three ages of fpgas: A retrospective on the first thirty years of fpga technology. *Proceedings of the IEEE*, 103(3):318–331, 2015. doi: 10.1109/JPROC.2015.2392104.

[43] Ashish Venkat and Dean M. Tullsen. Harnessing isa diversity: Design of a heterogeneous-isa chip multiprocessor. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, page 121–132. IEEE Press, 2014. ISBN 9781479943944.

[44] Ashish Venkat, Sriskanda Shamasunder, Hovav Shacham, and Dean M. Tullsen. Hipstr: Heterogeneous-isa program state relocation. *SIGARCH Comput. Archit. News*, 44(2): 727–741, mar 2016. ISSN 0163-5964. doi: 10.1145/2980024.2872408. URL https://doi.org/10.1145/2980024.2872408.

[45] K. Virk and J. Madsen. A system-level multiprocessor system-on-chip modeling framework. In *2004 International Symposium on System-on-Chip, 2004. Proceedings.*, pages 81–84, 2004. doi: 10.1109/ISSOC.2004.1411154.