Cross-ISA Execution Migration of Unikernels: Build Toolchain, Memory Alignment, and VM State Transfer Techniques

A K M Fazla Mehrab

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
Haibo Zeng
Changwoo Min

November 29, 2018
Blacksburg, Virginia

Keywords: Operating Systems, Unikernels, Virtualization, Heterogeneous Systems

Cross-ISA Execution Migration of Unikernels: Build Toolchain, Memory Alignment, and VM State Transfer Techniques

A K M Fazla Mehrab

(ABSTRACT)

The data centers are composed of resource-rich expensive server machines. A server, overloadeded with workloads, offloads some jobs to other servers; otherwise, its throughput becomes low. On the other hand, low-end embedded computers are low-power, and cheap OS-capable devices. We propose a system to use these embedded devices besides the servers and migrate some jobs from the server to the boards to increase the throughput when overloaded. The datacenters usually run workloads inside virtual machines (VM), but these embedded boards are not capable of running full-fledged VMs. In this thesis, we propose to use lightweight VMs, called unikernel, which can run on these low-end embedded devices. Another problem is that the most efficient versions of these boards have different instruction set architectures than the servers have. The ISA-difference between the servers and the embedded boards and the migration of the entire unikernel between them makes the migration a non-trivial problem. This thesis proposes a way to provide the unikernels with migration capabilities so that it becomes possible to offload workloads from the server to the embedded boards. This thesis describes a toolchain development process for building migratable unikernel for the native applications. This thesis also describes the alignment of the memory components between unikernels for different ISAs, so that the memory referencing remains valid and consistent after migration. Moreover, this thesis represents an efficient VM state transfer method so that the workloads experience higher execution time and minimum downtime due to the migration.

Cross-ISA Execution Migration of Unikernels: Build Toolchain, Memory Alignment, and VM State Transfer Techniques

A K M Fazla Mehrab

(GENERAL AUDIENCE ABSTRACT)

Cloud computing providers run data centers which are composed of thousands of server machines. Servers are robust, scalable, and thus capable of executing many jobs efficiently. At the same time, they are expensive to purchase and maintain. However, these servers may become overloaded by the jobs and take more time to finish their execution. In this situation, we propose a system which runs low-cost, low-power single-board computers in the data centers to help the servers, in considered scenarios, reduce execution time by transferring jobs from the server to the boards. Cloud providers run services inside virtual machines (VM) which provides isolation from other services. As these boards are not capable of running traditional VMs due to the low resources, we run lightweight VMs, called unikernel, in them. So if the servers are overloaded, some jobs running inside unikernels are offloaded to the boards. Later when the server gets some of its resources freed, these jobs are migrated back to the server. This back and forth migration system development for a unikernel is composed of several modules. This thesis discuss detail design and implementation of a few of these modules such as unikernel build environment implementation, and unikernel's execution state transfer during the migration.

Dedication

This thesis is dedicated to my family who sacrificed their happiness for my wellbeing.

Acknowledgements

I want to thank my honorable advisor Dr. Binoy Ravindran for giving me the chance to work with him, for being very supportive from the very beginning, and for providing the valuable guideline. I want to thank my committee members for managing the time from their busy schedule to review and provide valuable feedback on this thesis. Nevertheless, I would like to thank Dr. Pierre Olivier for his immense support, valuable suggestions, knowledge sharing, precious time and, patience on me. This thesis wouldn't be possible without him. I want to thank all of my brilliant groupmates from the System Software Research Group. At the same time, I would like to thank my friends, in Blacksburg and back in Bangladesh, for being at my side spiritually during my journey. Last but not least, I would like to thank my parents, my brothers and other family members for all of their sacrifices and believing in me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Data centers are computer warehouses composed of different servers and network infrastructures [50]. These servers are not only expensive but also consumes a tremendous amount of energy to be run and maintained. Overall, they incur a massive cost for the data centers which are increasing day by day due to technological advances such as cloud computing and the rapid growth of the use of Internet services [3]. This reason makes the data center operators seek economic alternatives to the traditional server computers. On the other hand, manufacturers are producing tremendously low-price OS-capable embedded devices which consume very low power. It draws the attention of many researchers toward the usage of these devices in the data centers. This thesis discusses providing support for exploring the opportunities of using the servers and OS-capable embedded boards side by side and taking most advantages from the heterogeneity in the context of data centers.

1.1 Motivations

OS-capable small embedded devices, categorized as single-board computers, such as Raspberry Pi (RPi) [51], Libre Computer LePotato [39] (Potato), 96 Boards Hikey LeMaker [1]

(Hikey), and BBC Micro Bit [7], posed a great impact on the world of electronic design [23]. These single-board computers are two order of magnitude cheaper, as cheap as $25, compared to the computers used as servers. Moreover, the power required to drive these boards is two orders of magnitude lower than the servers.

On the other hand, embedded boards are equipped with lesser resources and thus are slower than the servers as a result takes longer execution time on the board compared to the servers for the same program. For some compute-intensive application the board is 30% slower than the server, and for some others, the slowdown is less than 10x. This cost vs. slowdown analysis for an embedded board and a server, discussed in 2.1.1, shows that for some applications the slowdown is not as significant as the lower cost of the embedded boards.

Based on this observations, we propose a new system, named CSWAP, where if the server becomes overloaded, some workloads will be selectively offloaded to these boards. As a result, the server can free resources and thus the data center can deal with more jobs for a cost of less than 5% of server price and power consumption. Concerning price, power, and performance to get optimal performance form both the board and the server, we use the Instruction Set Architectures(ISA) that best suits them: x86_64 for the server and AArch64 for the board.

1.2 Challenges

CSWAP utilizes the advantages of both high-performance servers and low-power, cheap embedded devices where both types have different ISA. CSWAP bridges this disjoint ISA gap by translating the states of the application across ISAs which was not done before by other approaches [11, 28, 31, 33, 34, 53, 56]. CSWAP is made flexible regarding migrating the server applications back and forth (server to the board and vice versa) as a whole based on the overload scenario. This is done by inserting migration points (points where an application is allowed to be migrated) into the source at function level granularity. Some

previous approaches migrate part of the application [14, 19, 38] for speeding up the execution time which keeps both the source and the destination machine occupied. Few approaches migrate the entire application using ISA emulation [8] or language-based Virtual Machines (VM) [24, 25, 27] which incur a lot of overhead. CSWAP migrates the whole application to free the resources from the source machine without using emulator or language based VM so that resulted overhead remains minimal. CSWAP addresses the challenge of running VM, to provide security in a multi-tenant environment, by choosing unikernel [13, 30, 32, 36, 40, 41, 42, 48, 60], a lightweight minimalist single application VM, over full-fledged VM because embedded boards are not capable of running them due to low resources.

Unikernel applications are run in a different environment from the native applications. On top of that, no existing unikernel has the migration capability. So a migratable unikernel requires a special toolchain which can build an application as a migratable unikernel. This thesis shows an approach to building such a toolchain.

CSWAP migrates unikernels, libOS VM running a single application, between heterogeneous ISAs as opposed to existing approaches [6, 21, 57] designed for process migration. Moreover, a VM migration between heterogenous ISAs is far from the trivial migration in a homogeneous setup because there is no explicit mapping of VM states between different ISAs. To solve this challenge we extract the entire application state and some ISA independent kernel state, translate them and map them on the destination machine.

Moreover, the static memories of the unikernel binaries, like global data section in Executable Linkable Format (ELF), can be located at different addresses in different ISAs for the same application. As a result, anything pointing to the static memory, like a pointer variable, may not lead to the same location on different machines. Thus memory referencing may become inconsistent before and after migration. In order to solve this problem, we align all the static and dynamic memory of the unikernel so that they can have a common address layout across different ISAs.

How states of the unikernel and the corresponding application will be transferred from the

source machine to the destination machine is another challenge. There are different approaches [15, 29, 47] to solve this problem in homogeneous setups for VM migration. As we are migrating between heterogeneous ISAs, it requires a different approach to solve this problem. CSWAP proof-of-concept (POC) implementation already has a checkpoint/restart [47] based technique to solve this problem. This thesis proposes a post-copy [29] based approach called on-demand memory transfer for faster execution time and lesser downtime.

1.3 Thesis Contribution

This thesis partially contributes for implementing a POC of CSWAP. This POC implementation is composed of enabling a unikernel with migration support and providing toolchain necessary for building this migratable unikernel. This thesis describes the design and development of the toolchain, static and dynamic memory alignment of the unikernels across different ISAs, and a unikernel state transfer method. Following is a discussion of these contributions in brief:

1.3.1 Toolchain Development

There are several techniques for developing unikernels. On the other hand, different process migration approaches have own method to enable application with migration capability. However, in our knowledge, no existing approach allows a unikernel to be migrated between heterogeneous ISAs, and thus there is no tool for developing migration-capable unikernel.

We develop a toolchain which builds multi-ISA unikernel binaries enabled for migration across heterogeneous-ISA boundaries. This toolchain cross-compiles the unikernel and the application written in C using LLVM [37] for both x86_64 and AArch64 architecture and inserts migration points into the source code at function call sites. The compiler also generates metadata that describes live data and code location. This toolchain provides with several libraries including the C library and the state transformation library. The state

transformation library uses these metadata and translates the state of the application into destination ISA-specific format for migration. This toolchain modifies GNU Gold [26] which links all the generated object files and libraries and produces ISA-specific ELF files.

1.3.2 Alignment

When the unikernel receives a migration request, it stops the normal execution of the application at the very next migration point. Next, the state transformation runtime translates the stack contents for the target machines ISA. However, the dynamic and static contents of the application across ISAs needs to be aligned in the same address location so that any pointer pointing to a static or a dynamic memory location before migration, can point to a valid location after migration. A previous approach [5] uses a tool, called pyalign, for aligning static memory locations for process migration instead of unikernel. We extend their work for aligning static memory locations in unikernel. Moreover, we align dynamic memory locations. We align these memory locations by relocating the virtual addresses of different sections and symbols using a tool and produce aligned unikernel binaries. We refer to this process as alignment.

1.3.3 On-demand Memory Transfer

There are several approaches to application state migration where there is no OS migration involved. At time same time, there are multiple VM live migration techniques which only works for homogeneous migration. We develop an on-demand state transfer method for unikernel which is different from the state-of-art techniques for application and VM migration between homogeneous ISAs. Using this technique, we migrate and resume an unikernel application without transferring the static and dynamic memory, which helps to reduce the downtime. When the migrated job requires access to any application data that is not available in the current machine, it pulls the particular data from the source machine which reduces storage overhead associated with the technique that requires to store these memories

in files. Overall, this technique provides efficient migration and less overhead.

1.4 Scope of the work

This work of unikernel migration between heterogeneous-ISAs, presented in this thesis, considers x86_64 and AArch64 as target ISAs. The reason behind this is most of the datacenters run servers, which are based on x86_64 ISA. At the same time, most popular OS-capable embedded devices use AArch64 processors. Here, we describe how we develop toolchain necessary for migration, align memory for getting a common address space, and transfer VM memory during migration across x86_64 and AArch64 ISA. We assume the same endianness for both x86_64 and AArch64 ISA, and both processors use 64-bit addressing for the memory. Moreover, the datapath widths, integer size is also of 64 bits and have the same size and alignments for all primitive C types.

For the work presented in this thesis, we choose a unikernel called HermitCore, which is capable of running applications developed using different languages like C, C++, Fortran, and Rust. HermitCore is developed using C language. For this thesis, we target HermitCore unikernels running native applications, to migrate them between heterogeneous ISAs. That means the contributions, presented in this thesis, support C application migration, in the form of unikernel, between x86_64 and AArch64 ISA. It helps the application to take most advantage from the underlying OS.

1.5 Thesis Organization

This thesis is organized as follows. Chapter 2 discusses necessary background information to understand this thesis and summarizes related work in the area of execution and VM migration. Chapter 3 discusses toolchain development for unikernels suitable for heterogeneous migration. Chapter 4 describes dynamic and static memory alignment. Chapter 5 describes on-demand memory transfer technique for semantic migration and performance evaluation

regarding migration overhead. Finally, chapter 6 concludes and outlines future work.

Chapter 2

Background and Related Work

This chapter presents the background information necessary for understanding the rest of the thesis. This thesis provides several essential parts of the CSWAP that offers seamless job offloading between servers and embedded boards of heterogeneous-ISAs for high throughput. Section 2.1 describes the motivation behind CSWAP, its design, and relation to this thesis. Section 2.3 describes the Popcorn Linux, a replicated-kernel operating system designed to provide OS support across diverse architectures, from which some components are adopted in CSWAP (and this thesis). Finally, section 2.4 describes a lightweight operating system model called unikernel and HermitCore, a multi-core unikernel used for prototyping CSWAP.

2.1 CSWAP

CSWAP stands for Compute Swap, refers to swapping workloads from server to low-cost, low-power, and slow-storage computers to consolidate data from fast but expansive memory. It swaps the workloads between heterogeneous ISAs, x86_64 and AArch64, by the checkpoint/migrate/restarting of lightweight VMs running applications. These VMs run under hardware virtualization which enables them to run native applications directly on the CPU

for maximized performance.

Previously proposed approaches [14, 19, 27, 38] offload part of mobile phone applications to the server based on the availability of the server to increase the performance. Instead of being capable of offloading a specific part of an application, CSWAP offloads the entire application. Its checkpoint/restart approach allows to halt the execution of the application and checkpoint the VM states, migrate, and resume the VM in the embedded board in the same states it was stopped in to free the corresponding resources in the server. The following subsections discuss the motivation behind CSWAP and its design.

2.1.1 Motivation of CSWAP

As a low-cost, low-power, and low-storage platform compared to the servers, CSWAP considers single-board computers such as Raspberry Pi [51]. Regarding price, these servers are two order of magnitude cheaper than the traditional servers. These boards are capable of running different distributions of Linux for the operating system (OS), support hardware virtualization and their specifications make them able to run medium-sized compute-intensive workloads such as unikernels.

An example of a server class machine for CSWAP prototype is a Colfax CX1120s-x6 [17] referred to as Xeon in the rest of the thesis. The considered embedded board is Libre Computer LePotato [39] (Potato). Along with different configurations, the Table 2.1 shows the power consumption and prices of these machines. From this table, one can see that the potato board is 67 times cheaper than the Xeon. The entire machine's power consumption is measured using Kill-A-Watt while each machine was idle and running one, four, and eight instances of stress [1] program which keeps corresponding (1, 4 or 8) hardware threads/cores active at 100%. These measurements show that the Xeon's power consumption is roughly 40 times higher than the Potato board. One critical motivating factor for CSWAP is to get benefits from the board's negligible power consumption and cost compared to the server.

[1] http://www.p3international.com/products/p4400.html

Table 2.1: Considered server & boards characteristics.

Machine	Xeon	Potato
CPU model	Xeon E5-2637	Amlogic S905X
ISA	x86-64	Arm64
CPU frequency	3 (turbo 3.5) GHz	1.5 GHz
Cores	4 (8 HT)	4
RAM	64 GB	2 GB
Power (idle)	60 W	**1.8 W**
Power (1 thread)	83 W	**2.1 W**
Power (4 threads)	124 W	**2.9 W**
Price	$ 3049	**$ 45**

The lower configuration of the single-board computers makes them cheaper and less power consuming. At the same time, the same reason naturally hurts the performance. So it's important to know if using this type of single-board computer is worthwhile at all. Thus, it is necessary to study their performance with respect to the servers. In order to do that, NAS Parallel Benchmarks' (NPB) [4, 54] serial version, and class B is run. The slowdown incurred on the board compared to the server is computed and presented in Figure 2.1. The execution times are normalized to the Xeon's performance, i.e., 1 on the Y-axis represents the execution time of the server.

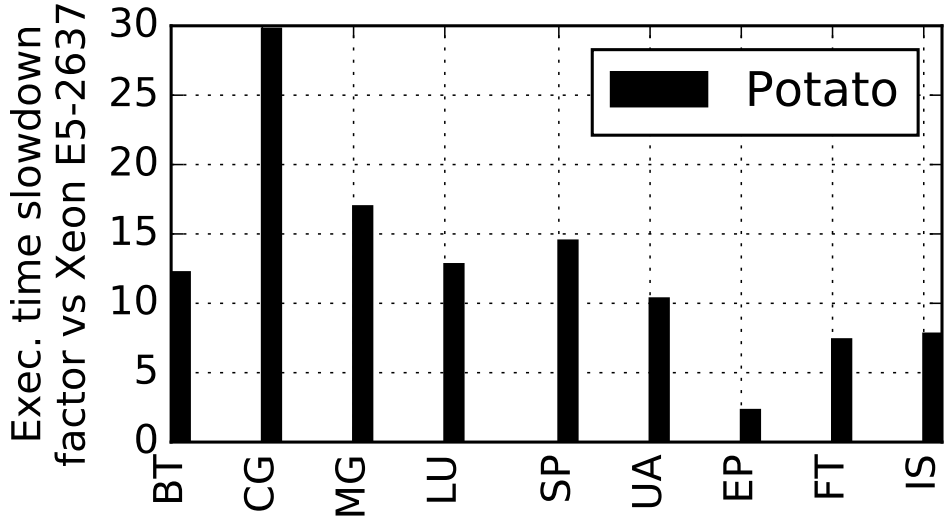


Figure 2.1: NPB embedded boards slowdown vs Xeon.

We can see that for some benchmark the slowdown is relatively small. For example, for the Potato board, the slowdown is less than 10x for EP, FT, and IS. This slowdown is less than one order of magnitude where the board is two orders of magnitude of cheaper than the server regarding both price and power consumption. Expectedly, the Opposite scenario is also available. For example, for Potato, CG and MG are more than 30x slower compared to the server. A key idea in CSWAP is to swap out the jobs with the lowest slowdown from the server to the boards in order to free the resources in the server when necessary.

Concerning the power consumption, based on the Table 2.1, even for the execution of CG, which exhibits the highest slowdown, on the Potato board it requires 30% less energy than the server. For EP the energy reduction goes up to 17x.

From these observations, we can conclude that the usage of these boards coupling with the server costs negligible price and for some applications, the consolidation, from the server to the boards, may contribute a little to the increase of throughput. However, in exchange for that, CSWAP can free valuable resources in the server and house more jobs.

2.2 System design of CSWAP

The considered embedded boards, discussed in section 2.1.1 cannot run full-fledged VMs due to the lack of resources and thus CSWAP relies on lightweight virtualization. On the other hand, as hardware-based virtualization provides with strong isolation compared to the software-based virtualization, such as containers, CSWAP uses unikernel which is discussed in section 2.4.

Figure 2.2 depicts a high-level overview of CSWAP's system design. To utilize the most efficient ISA in each considered domain, CSWAP assumes that the architecture of the server CPU is x86_64 and AArch64 for the embedded boards. So the first step to creating an ISA-specific unikernel is to insert migration points (Ⓐ on Figure 2.2 in the application code where the migration is possible.

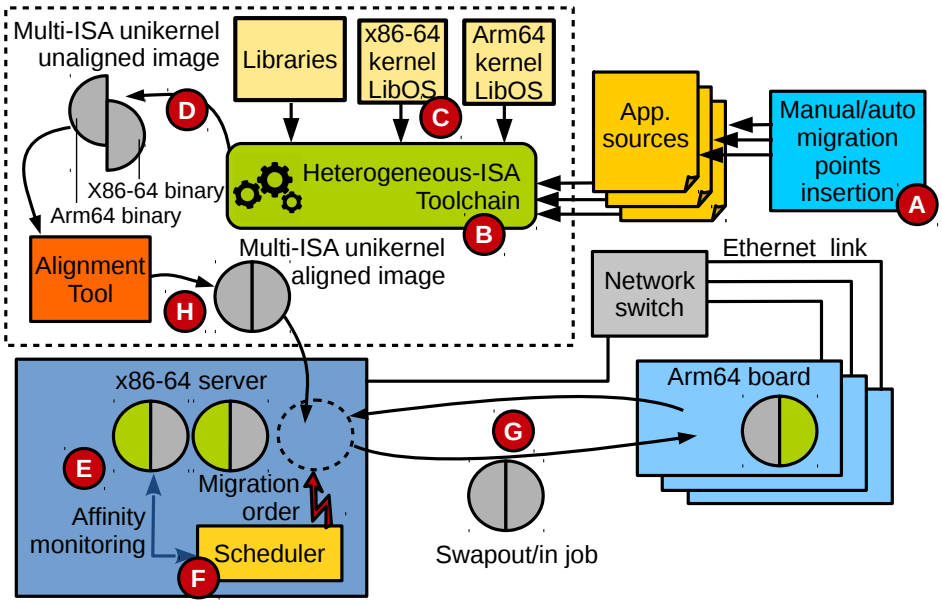


Figure 2.2: Overview of CSWAP's building and execution flow.

In the next step, the application sources are provided to CSWAP's heterogeneous ISA toolchain Ⓑ. The compiler in the toolchain inserts the metadata needed to transform the architecture-specific application state at runtime. For each ISA, the code is compiled and linked against multiple libraries: CSWAP's kernel library OS Ⓒ and other libraries containing standard C library, runtime ISA state transformation library, and any user-specified libraries. Finally, the toolchain generates two ISA-specific unikernel binaries Ⓓ. But the symbols in these binaries are unaligned with each other. The alignment tool Ⓗ aligns these symbols and generates aligned unikernel binaries.

x86_64 unikernel binary is first launched in the server, and there can be multiple unikernel instances running on the server at the same time. CSWAP uses a scheduler to decide which application should be swapped out to embedded systems based on various criteria, in particular, an estimation of the slowdown on the board we discussed in section 2.1.1. When the scheduler detects resource congestion, it selects the job to be offloaded and triggers the corresponding hypervisor, which triggers the heterogeneous migration process once the application reaches the next migration point. At this point, the guest OS stops the application execution and translates the states for the target ISA, AArch64. The transformed application

state is transferred to the board. Finally, the unikernel binary for AArch64 is bootstrapped on the board, and the application state is restored before resuming the execution.

2.2.1 Thesis correlation to CSWAP

Building the prototype of CSWAP includes several challenges. One of the challenges is the ISA difference between the server and the board. To solve this disjoint ISA problem, CSWAP allows the conversion of the architecture-specific state of an application between ISAs using state transformation. The state transformation is adapted from [5] which targets process migration with Linux kernel as opposed to the VM migration scheme in a context of unikernel. This adaptation requires a new toolchain which can produce heterogeneous-ISA unikernel binaries equipped with state transformation information. One vital scope of this thesis is to develop this toolchain which we discuss in the chapter 3. To preserve the validity of the pointers across migration, the functions and the global variables need to be aligned at the same addresses in both ISAs. Meeting this alignment challenge is another objective of this thesis which we discuss in the chapter 4. These contributions are marked by the dotted box in Figure 2.2.

CSWAP offloads the application running inside a VM. The part of the VM, kernel state, is very different across different ISAs because there is no explicit mapping between heterogeneous ISAs. So the heterogeneous VM migration is very different from traditional VM migration implementation. To solve this challenge, CSWAP extracts the entire application state from the VM and minimal architecture independent subset of the kernel state, transfers them, and resumes the application on the destination VM after restoring the states. This thesis proposes an alternative to the checkpoint/restart approach called on-demand migration which we discuss in chapter 5.

2.3 Popcorn Linux

Unlike CSWAP, instead of migrating a VM in the data center context, Popcorn Linux [5] builds a prototype for demonstrating the concept of application migration across heterogeneous-ISA. This prototype includes an operating system, compiler, and a runtime which enables the application to be migrated between an ARM and x86 processors. The following sections discuss some of these components in brief.

2.3.1 Popcorn compiler Toolchain

The Popcorn compiler toolchain builds application binaries for seamless migration across heterogeneous ISAs. This toolchain uses modified LLVM [37] for compiling, modified GNU Gold [26] for linking, and several custom-built tool for preparing binaries for migration. The application binary generation is done in the following several steps:

1. **Inserting migration point:** As threads cannot be migrated in middle of a function, migration points are inserted at the beginning and the end of every function. This insertion is done by modifying Intermediate Representation (IR) generated by Clang.
2. **Backend Analysis:** Multiple backend analyses are done which include marking function return addresses, gathering live value locations and generating metadata.
3. **Linking:** Generated object files are linked with the libraries and generate executable binaries.
4. **Alignment:** An alignment tool gathers the size and location of all symbols from the binaries and relocates them in the same address across different ISAs.
5. **Inserting Metadata:** Aligned binaries are processed to gather state transformation metadata necessary for transforming thread's states at runtime.

CSWAP also follows these steps by adapting the Popcorn Linux toolchain for enabling application with migration capabilities. Along with that, as this toolchain generates only

application binaries for heterogeneous ISAs, not for VM, CSWAP adapts necessary steps for building a particular VM to be migrated. This thesis discusses the combined approach to construct a separate toolchain that can build a VM with migration capability.

2.3.2 State Transformation

Because of the heterogeneity of the ISA, the state of a running application at any migration point at runtime in one ISA cannot be directly restored in another ISA. To bridge this disjoint ISA gap, Popcorn Linux does state transformation on the application state which generates an equivalent application state for the target ISA. This state transformation is done in runtime. Threads execute normally and check for migration requests by the scheduler at every migration point. Once it receives a migration request, it pauses the application execution and takes the snapshot of its registers and calls the state transformation runtime. Next, the state transformation runtime efficiently reconstructs the thread's current live function activations in the format of the target ISA as well as the register set, call frame of the thread, and the pointers to the other stack objects.

Once the transformation is done by the runtime, it signals the Popcorn Linux kernel. Using the thread migration service, the kernel migrates the thread to the target architecture.

Unlike Popcorn Linux, CSWAP targets VMs to migrate between heterogenous-ISAs. It requires the VM to transform its states before migrating to another machine with a different ISA. So CSWAP adapts state transformation runtime from Popcorn Linux. This thesis shows how to port state transformation runtime for VM migration.

2.4 Unikernel

Unikernels [13, 30, 32, 36, 40, 41, 42, 48, 60] are lightweight VM where most of them are benefited from hardware-assisted virtualization support, as opposed to software solutions

such as clear containers [18]. Unlike full-fledged VMs, unikernels are capable of hosting a single application, which is statically compiled with a kernel and other libraries, e.g. C library, are only necessary to run the particular application. This minimalistic characteristic of the unikernel makes it very lightweight, which has some significant benefit over the regular VMs.

2.4.1 Benefits of using Unikernel

In the data center scenario, where the clients are provided resources as a form of VM based on how much they pay for the service, the VMs run regular operating system equipped with various default features such text editor, package manage manager, etc. These features may not be very relevant to the kind of program the user wants to run, instead consumes resources. The usage of unikernels can benefit clients from utilizing the maximum from the resources.

Due to the high resource requirement of full-fledged VMs, it is not possible to run them on the embedded boards as discussed in section 2.1.1. On the other hand, unikernels are lightweight, less resource and power hungry. To be benefited from the virtualization on the embedded boards, the usage of unikernels can be a viable solution.

Finally, cloud provider data centers provide different services which require stronger isolation from each other concerning the security. So instead of natively running these services, they are run inside VMs. The minimal amount of code presented within a unikernel induces a minimal attack surface and less RAM/disk access, thus less footprint, compared to the regular VMs. As opposed to the hardware-assisted virtualization, software-based alternatives of unikernel like containers [43] are found to be less secure [41] so that they are sometimes run inside VMs.

Considering all of these benefits of unikernel, CSWAP chooses unikernel over regular VM for migrating workloads from the server to the embedded boards.

2.4.2 HermitCore

The unikernel we choose for CSWAP POC implementation is called HermitCore. HermitCore [36] is a unikernel operating system designed for High-Performance Computing (HPC). It runs a very lightweight 64-bit kernel, sharing the same address space with the application, which benefits from low overhead, stability, and low system noise.

HermitCore uses Newlib [46] for C library which requires few system call implementations for the support for full ANSI C compliance. These system calls are implemented as regular function calls in HermitCore kernel, as opposed to Linux's interrupt-based system calls, which reduces the overhead of libC layer. However, some requests, which are not implemented by the kernel, e.g., I/O operations, are forwarded to the host operating system through the hypervisor. Apart from that, HermitCore provides networking support by lwip [2].

HermitCore can run on two hypervisors: QEMU/KVM and uhyve. QEMU is a very versatile virtualization software which can be used to do many things including virtualizing a full system. On the other hand, uhyve, the abbreviation of Unikernel hypervisor, is a lightweight hypervisor developed as a part of the HermitCore project based on UkVM [59]. Like HermitCore, uhyve has a small code base, so that it has a low attack surface, and to utilize this advantage this thesis chooses uhyve over QEMU. HermitCore provides with a Linux native application, called proxy, for loading the hypervisor and starts the HermitCore application by providing a predefined set of cores.

The build process of HermitCore is realized by the GNU Compiler Collection (GCC) modified for cross-compilation of new target OS. HermitCore has its linker script which determines the location of the kernel symbols and names. HermitCore provides openMP support by porting a minimal pthread library, pthread embedded, and GCC's openMP runtime library, libgomp. HermitCore toolchain produces ELF binaries which can be distinguished by a magic ABI number assigned to HermitCore.

CSWAP, as its name implies compute-swap, targets compute intensive native applications to migrate between the server and the embedded boards. As HermitCore is designed for HPC, capable of running native applications, and runs on a lightweight hypervisor, its properties align with CSWAPs goals. So we choose HermitCore as the target unikernel. This thesis describes how we adopted HermitCore and enabled it for heterogeneous-ISA migration.

2.5 Related Work

2.5.1 Migration Between Heterogeneous ISAs

Most of the process migrations techniques are developed targeting homogeneous setup. But there exists several approaches which explore the potential of heterogeneity by migrating executions between ISAs. Few of them [12, 52, 55, 61] take advantage of Java Virtual Machine (JVM), as it is idiosyncratic of the operating system and underlying hardware platform, for the process migration in a heterogeneous environment. One common problem with these approaches is that they only work for JVM and Java language, i.e., they are lacking genericity. However, this thesis is particularly interested in migrating native applications to take full advantage of the OS. Geoffray et al. [25] describe a method to extend the heterogeneous migration for languages like C and C++. They introduce a micro-virtual machine layer beneath other VMs, like JVM and .NET VM, to help with thread migration where the application source needs to be available in bytecode. This multi-layered approach incurs a lot of overhead for native application migration.

DeVuyst [21] et al. introduce a simulation of execution migration between heterogeneous-ISA cores built inside a single chip by performing binary translation and state transformation. To extend this work for different computing platforms, Bhat et al. [9] adapts replicated kernel OS [6]. This runs ISA-specific kernels on the platforms but provides an illusion to the upper layer (application) of single OS running on the heterogeneous platforms. As a result, standard applications can run on a heterogeneous platform without any modification. Both

of these approaches are run and evaluated in simulated environments. Popcorn Linux [5] proposes a real-world implementation of process migration which is different from the VM (unikernel) migration technique described in this thesis.

Few other real-world implementations of heterogeneous approaches [14, 19, 38] offload a specific part of the application to the target machine instead of whole. It aligns with their particular purpose of decreasing the execution time of the application or making the smartphone last longer. But this thesis aims to free resources in the server machine by offloading the entire application where these approaches keep both the source and destination machine occupied until the execution ends. So these approaches incur more energy consumption and less throughput than possible with complete migration.

2.5.2 Virtual Machine Live Migration

A very-practiced approach in VM migration is the virtual machine checkpoint/restart [47]. In this method, When there is a migration request, the entire state of the VM is saved by the virtual machine monitor (VMM). These states include the content of the VMs memory, the CPU states, and states of the virtualized devices which are stored in the disk. Then the VM is stopped, and all the saved states are transferred to the destination machine. Finally, the VM is resumed, and the states are restored from the disk. CSWAP follows similar checkpoint/restart technique for VM migration and implements in the context of heterogeneous ISA.

Pre-copy techniques [15, 45] require pages of memory to be iteratively copied from the source machine to the destination machine while the VM continues to run live services in the source machine. In the first iteration, all pages are transferred and in the following iterations, updated pages are sent. At the final iteration, the VM is stopped and any remaining dirty pages are copied. So this method results in sending more data than the VM has in total. As a result, it incurs a lot of network activity which is a burden for the embedded devices which are equipped with a relatively slow Network Interface Card (NIC). On the other hand,

the migration time for this approach is quiet indeterministic, which delays freeing resources from the source machine.

In contrast to the pre-copy method, the post-copy [29] live migration method first transfers only the CPU states to the destination machine, and starts the VM. Once the VM is up and running in the target machine, memory pages in the source machine are actively pushed to the destination machine. At the same time, if any page fault occurs because of any memory page that is not retrieved yet from the source machine, then that particular page is transferred over the network from the source machine. This method retains from transferring the same page twice where pre-copy may transfer a lot of duplicate pages because of dirtied pages.

This thesis adopts and implements the idea of post-copy, which is called on-demand memory transfer and replaces CSWAP's checkpoint/restart implementation. But it is much different from migrating an entire OS, and all of its applications, as one unit like post-copy and pre-copy does. Because we migrate between heterogeneous ISAs, unlike pre-copy and post-copy, there is no clear mapping for VM states between the source and the destination machine. So we perform VM state migration by extracting and transferring only the application states and part of the kernel state instead of copying the entire VM from one machine to another.

Chapter 3

Heterogeneous ISA Toolchain for CSWAP

CSWAP runs the applications in a VM instead of a native environment. The development of the VM and process of embedding them with the offloading capability is different from trivial application development, which requires special VM-specific toolchain. Section 3.1 provides the overview of the cross-compilation toolchain for CSWAP. Section 3.2, 3.3, 3.4, 3.5, and 3.7 describes how we adapted compiler, C library, kernel, OpenMP library, and state transformation library respectively, which are vital components of the toolchain for CSWAP. Section 3.6 describes how to link a HermitCore application with the adapted toolchain. Finally, section 3.8 discusses the performance of the toolchain for CSWAP.

3.1 Toolchain for unikernel migration

3.1.1 Concept of Cross-compilation Toolchain

A toolchain is a set of different tools working together to support software development for a target system. These tools may include compiler, linker, libraries, and other utilities. It is possible to come up with different types of toolchain building processes such as -

1. A native toolchain, has usually been compiled on x86, runs on x86 and generates code for x86;
2. A cross-compilation toolchain is typically compiled on x86, runs on x86 and generates code for the target architecture (be it ARM, MIPS, PowerPC or any other architecture supported by the different toolchain components).

So there could be two different machines involved when someone plans to use a toolchain for cross-compilation.

1. The host machine, where the toolchain is executed;
2. The target machine, where the generated code is executed.

Most of the x86 applications are developed using native toolchains which can be built using popular development environments including Visual C (Windows) and GCC (Linux and macOS). As we use Linux for the operating system on an x86_64 build machine, for rest of the document we are going to assume Linux on x86_64 is our host.

On the other hand, embedded applications are usually developed using cross-compilation toolchain. For example, an Android application is usually built in an x86 machine using IDE, like Eclipse, and run on mobile devices supported by ARM architecture.

3.1.2 Hermit-popcorn Toolchain

Our target unikernel, HermitCore, runs on two different Instruction Set Architectures (ISA): x86_64 and AArch64. It is very intuitive that a unikernel application for AArch64 is built using cross-compilation toolchain. At the same time, unlike a native application, a unikernel application runs in a very different environment than the native environment, because unikernel itself is an operating system. So even for x86_64, we need the unikernel application to be built using cross-compilation toolchain. As a result, some tools in the toolchain for HermitCore have two different versions: cross-compilation toolchain for x86_64 and cross-compilation toolchain for AArch64.

The default HermitCore toolchain [35] consists of the following tools:

- GCC: Compiler;
- Newlib: Embedded C library;
- Pthread Embedded: Pthread library;
- Binutils: Binary utilities like ld, objcopy, objdump, readelf etc.;
- libhermit: HermitCore unikernel;
- hermit-caves: Tool to start and handle HermitCore applications;
- lwIP: Light weight networking library.

On the other hand, CSWAP project shares the same interest of creating multi-ISA binaries suitable for migration across heterogeneous-ISA boundaries with Popcorn Linux. Popcorn Linux itself has its own toolchain which consists of the following tools:

- LLVM/Clang: Compiler;
- musl libc: C library;
- Libelf: Library for manipulating ELF files;
- Stack transformation library: State transformation runtime;

Tool	HermitCore Toolchain		Hermit-popcorn Toolchain		Popcorn Linux Toolchain
Compiler	GCC		LLVM	←	LLVM
C library	Newlib	→	Newlib		musl libc
Utils	Binutils	→	Binutils	←	Binutils
Pthread library	PTE	→	PTE		POSIX Threads
State transformation library			Stack Transformation Runtime	←	Stack Transformation Runtime
Kernel	libhermit	→	libhermit		Linux Kernel
Migration library			libmigrate	←	libmigrate
Hypervisor	Hermit-caves	→	Hermit-caves		
ELF manipulation library			LibElf	←	LibElf
Stack metadata generating tool			gen-stackinfo	←	gen-stackinfo
TCP/IP library	lwip	→	lwip		
Alignment tool			pyalign	←	pyalign

Figure 3.1: Hermit-popcorn toolchain's adaptation from different toolchains.

- Migration library: Migrate threads between architectures;
- Binutils: Binary utilities like Gold, objcopy, objdump, readelf etc.;
- Stack metadata: inserts stack transformation metadata sections to each binary;
- Alignment tool: tool for aligning static memory.

To provide HermitCore with migration capability across heterogeneous ISAs, we adopt different tools from both toolchains and bring other necessary changes to it. The Popcorn Linux compiler is capable of inserting migration points into the program source and the linker is capable of generating alignment information. So we adapt its compiler and linker, from binutils, to work with HermitCore. We choose Newlib for C library over musl libc for portability across different operating systems because the latter assumes that Linux is the underlying OS. For the same reason, we choose PTE over POSIX Threads for the pthread library, and lwip for networking using the TCP/IP stack. As we aim to migrate the whole unikernel rather than just the Linux application, we choose libhermit — the kernel for HermitCore and hermit-caves which contains the hypervisor. We also adapt Popcorn Linux's stack transformation library for the application's state transformation, the migration library for migrating the unikernel, libelf, and gen-stack info (generates stack info) necessary for

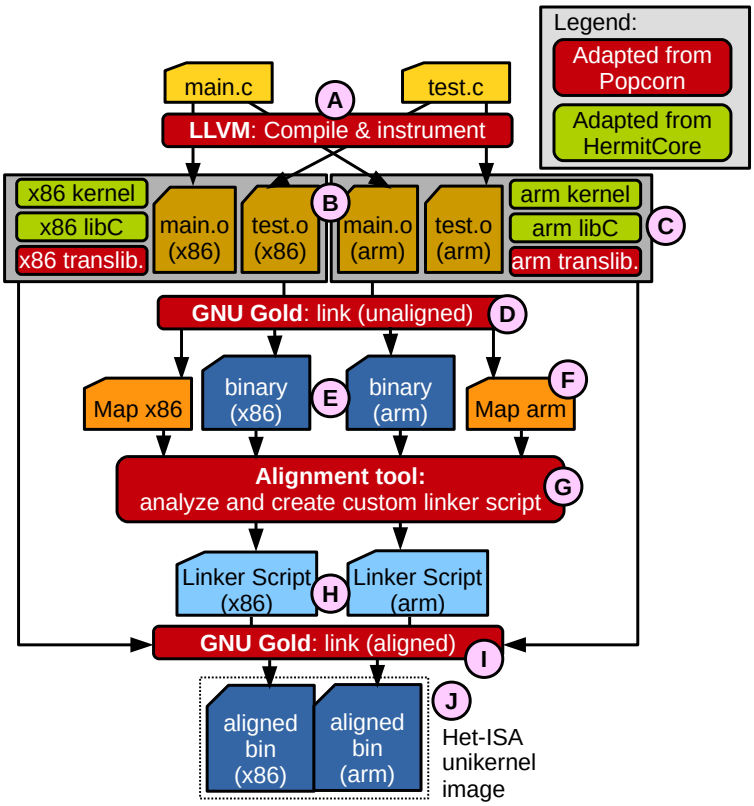


Figure 3.2: Different steps of aligned executable generation.

stack transformation library, which are not available in the HermitCore toolchain. We also adapt the static memory alignment tool called pyalign from Popcorn Linux and other utilities like objdump, readelf, and ar from binutils for HermitCore.

This newly transformed toolchain is named hermit-popcorn toolchain. In the rest of this chapter, our work on hermit-popcorn toolchain development process is described in detail.

3.1.3 Migratable Unikernel building using Hermit-popcorn Toolchain

Building a migration-capable HermitCore unikernel comprises several steps. As shown in Figure 3.2 fist the LLVM cross-compiler compiles and instruments Ⓐ the application source code and generates the architecture-specific object files Ⓑ. Then the GNU Gold linker links Ⓑ the generated object files with libraries such as the C library, the kernel, and the

stack transformation runtime library Ⓓ and generates architecture-specific binaries Ⓔ and corresponding Map files Ⓕ. Next, the alignment tool Ⓖ creates aligned linker scripts Ⓗ to align all the symbols in each binary with one another. Finally, using these aligned linker scripts, the GNU Gold linker Ⓘ generates architecture-specific aligned unikernel images Ⓙ.

3.2 Adapting LLVM/Clang for CSWAP

LLVM [37] is a compiler infrastructure which can provide a powerful intermediate representation for efficient compiler transformations and analysis. It can produce an optimized intermediate representation (IR) by taking IR code from a compiler. This new IR can then be converted and linked into machine-dependent assembly language code for a target platform. Barbalace et al. [5] run analysis over the intermediate representation on the application to gather live data that must be transformed between ISA-specific formats by modifying LLVM. Additionally, they modified the compiler so that it inserts migration points into the source code at function call sites. We adopted their IR analysis approach and compiler modifications without any changes.

On the other hand, the HermitCore toolchain uses modified version of GCC that supports HermitCore cross-compilation. Neither stock nor popcorn LLVM comes with any HermitCore support, so we add support for a new OS to both LLVM and Clang so that we can cross-compile applications for HermitCore. The following sections describe the process in detail.

3.2.1 Modifying LLVM for HermitCore cross-compilation

LLVM uses CMake [16] to configure source code packages. This process requires the configuration names of each platform that the sources are built for. These names are called triples because they have three fields:

ARCHITECTURE - VENDOR - OPERATING_SYSTEM

Because HermitCore was not introduced in LLVM by default, it is a new platform whose configuration names, i.e. a new triples, we need to define. HermitCore runs on x86_64 and AArch64 architecture. Since these architecture entries are already available in the enum called ArchType in `llvm/include/llvm/ADT/Triple.h`, we don't need to add architecture entries separately. Additionally, we also don't define a vendor for HermitCore, instead leaving it untouched. We add a new entry, Hermit, in the enum called OSType in the file `llvm/lib/Support/Triple.cpp`, which represents the HermitCore operating system.

Configuration names are actually strings so we need to define a unique name for HermitCore, which we do by defining the name `'hermit'`. When someone queries for OS name, by calling the function `Triple::getOSTypeName()` defined in `llvm/lib/Support/Triple.cpp` with the enum Hermit, we return the string 'hermit'.

To let others get a triple by the corresponding OS name there is a function called `parseOS()` in the same file. Here, we add an entry for HermitCore as follows.

```
.StartsWith("hermit", Triple::Hermit)
```

All the libraries and sources required to build a HermitCore application are compiled separately. To allow the linker to combine these separately compiled modules into one executable without recompilation, we need to define a unique Operating System Application Binary Interface (OSABI) for HermitCore. This OSABI can be used as an identification. Thus, we declare an OSABI for HermitCore in LLVM with a unique number, 0x42, defined by HermitCore kernel, in `include/llvm/Support/ELF.h` as follows.

```
ELFOSABI_HERMIT = 0x42
```

To let others get the OSABI for HermitCore by the Hermit triple, from the function called `getOSABI()` in `include/llvm/MC/MCELFObjectWriter.h` we return this OSABI.

3.2.2 Modifying Clang for HermitCore cross-compilation

Clang needs to know about the target OS it is cross-compiling for. To let Clang know about the new target HermitCore, we create a class called HermitTargetInfo inheriting OSTargetInfo in `lib/Basic/Targets.cpp`. This class can be used for HermitCore specific internals related to the compiler. For example, different applications or libraries may have some HermitCore specific implementation in the source. To distinguish those implementations from others a macro named `__hermit__` is used, which used to be defined in GCC of hermit toolchain. In order to the same for LLVM, in `getOSDefines()` method of `HermitTargetInfo`, we define this macro.

This HermitTargetInfo is instantiated in the method called AllocateTarget. This method takes a triple as the parameter and creates a new instance of it, specific to the architecture, and return the pointer. As HermitCore is run on two architectures, x86_64 and AArch64, if the triple is for x86_64 then a new instance is created and returned as follows.

```
return new HermitTargetInfo<X86_64TargetInfo>(Triple);
```

And if the triple is for AArch64 then the new instance is created and returned as follows.

```
return new HermitTargetInfo<AArch64leTargetInfo>(Triple, Opts);
```

Clang is the C compiler of the hermit-popcorn toolchain. Moreover, we can let Clang know about some other tools of this toolchain. For example, to enable Clang to figure out how to link and assemble for HermitCore, we define two methods, `buildLinker()` and `buildAssember()`, in a new class called Hermit, which inherits the class called `Generic_ELF` in `lib/Driver/ToolChains.h`. We implement corresponding constructor and these methods in `lib/Driver/ToolChains.cpp`.

In `lib/Driver/Tools.h`, we define a new class called `Link`, inheriting `GnuTool`, which corresponds to the default linker for the hermit-popcorn toolchain. As we do not use LLVM for linking, instead we have a script for doing this, we do not put our linker path in this class.

`buildLinker()` method returns an instance of the Link class.

Similarly, in `lib/Driver/Tools.h` we define another new class called Assemble which corresponds to the default assembler, `x86_64-hermit-as`, for hermit-popcorn toolchain. In the method called `ConstructJob` of this class, we indicate the path of `x86_64-hermit-as`. `buildAssember()` method, in class Hermit, returns an instance of the Assemble class. With these two classes, Link and Assemble, we construct a new namespace for HermitCore called hermit.

As we have this class, Hermit, for representing HermitCore tools, if there is a request for HermitCore toolchain we return an instance of this class. For example, in `getToolChain()` method of the Driver class in `lib/Driver/Driver.cpp`, if the parameter triple belongs to Hermit, then we create an instance of Hermit as follows and later return it.

```
TC = new toolchains::Hermit(*this, Target, Args);
```

3.2.3 Cross-compiling using modified LLVM and Clang

For a sample C source test.c, we can cross-compile for HermitCore issuing following commands.

For x86_64:

```
clang -target x86_64-hermit -c -o test.o test.c
```

For AArch64:

```
clang -target aarch64-hermit -c -o test.o test.c
```

3.3 Adapting Newlib for CSWAP

The C library used for Popcorn Linux is called musl [44]. As musl is designed for the operating systems based on the Linux kernel, it is highly dependent on Linux APIs. On the other hand, HermitCore is a lightweight libOS which is very different from full-fledged Linux. As a result, it is not possible to use musl as the C library for HermitCore. Instead, we stick to the default C library for HermitCore toolchain, which is Newlib [46].

Newlib is a standard C library which is easily portable to new operating systems and intended for use on embedded systems. There are two challenges associated with using Newlib for hermit-popcorn toolchain. One is porting popcorn Linux specific changes from musl. Another is cross-compiling Newlib for both x86_64 and AArch64 with LLVM as it used to be compiled with GCC using the HermitCore toolchain. This section describes different changes we bring to Newlib.

3.3.1 Changes in Newlib source code

The function called frexp in Newlib breaks a floating point value into a number and an exponent. This function is used by compute-intensive parallel applications like NPB OMP IS. When we link this type of application with the C library it results in a linking error because a function called numtest, called inside frexp for checking special value, is not defined in Newlib or other libraries we use. One simple way to avoid this linking problem is to disable numtest uses in frexp.

Cross-compilation of Newlib for AArch64 architecture using LLVM may result in some errors due to the fact that some AArch64 assembly directives are unknown to LLVM or they have alternates. For example, usage of dword produces 64bit values which are recognized well by GCC. However, when we compile with LLVM, it doesn't recognize dword. In this case, we find alternate of dword, which is xword. Thus, we replace all of the occurrences of dword with xword.

```
- HeapBase:  .dword  0
+ HeapBase:  .xword  0
```

Here we replaced the dword directive for HeapBase in libgloss/aarch64/crt0.S file using xword.

Newlib for AArch64 uses the several register names for implementing of several C library functions, e.g. memchr, strchr, strchrnul, and strrchr, such as : vend.2d, vend1.2d, vhas_nul1.2d, and vhas_chr1.2d.

But LLVM does not recognize these register names though these are compiled without any error by GCC. In this case, we used the alternatives such as vend.d, vend1.d, vhas_nul1.d, and vhas_chr1.dwhich which work fine for LLVM.

When an assert function call fails in any program it eventually ends up calling ___assert_fail, which is not available in Newlib. As a result, we get linking error during link time. To handle this situation, we implement our own ___assert_fail as follows. The ___builtin_trap() function causes the program to exit abnormally. As a result, instead of giving linking error, if there is an assert fail, the program exists instead of executing the rest of the instructions.

```
void __assert_fail() {
    __builtin_trap();
}
```

Several system call wrappers, e.g. chown, open, close, execve, open etc., contain a call to the function named stub_warning, which eventually produces nothing but a linking error because it's not defined by Newlib or in other libraries. If we compile Newlib with LLVM this stub_warning function gets a call and thus ends up into linking failure. To prevent this issue from happening we simply disable all stub_warning calls by commenting them.

3.3.2 Compiling Newlib with LLVM

Compiling Newlib with LLVM is different from GCC in various aspects. For compiling with LLVM, first we set environment variables CFLAGS and CXXFLAGS with multiple flags depending on the target architecture we are going to build for. For the x86_64 the target flag is set to x86_64-hermit and for AArch64 to aarch64-hermit. Next, we set the environment variable AS_FOR_TARGET, AS_FOR_TARGET, RANLIB_FOR_TARGET, and CC_FOR_TARGET by the path of the assembler, achiever, ranlib, and clang, respectively, for hermit-popcorn toolchain.

Next, we need to configure the Newlib before compilation. Following shows how to configure Newlib for AArch64.

```
configure --target=aarch64-hermit --prefix=installation_path
        --disable-shared --disable-multilib --enable-lto
        --enable-Newlib-hw-fp --enable-Newlib-io-c99-formats
        --enable-Newlib-multithread target_alias=aarch64-hermit
```

The `target` flag indicates for which platform the Newlib is being cross-compiled. The `prefix` denotes the installation path for Newlib binaries. The `disable-shared` flag indicates that the Newlib to be compiled as a static library. The usage of `disable-multilib` flag prevents the Newlib from being compiled to support different target variants, calling conventions, etc. The flag `enable-lto` enables support for link-time optimization (LTO). To let Newlib use hardware floating points `enable-Newlib-hw-fp` is used. The flag `enable-Newlib-io-c99-formats` indicates Newlib to enable C99 support in IO functions and the flag `enable-Newlib-multithread` enables the multi-threading support in Newlib.

Once we configure Newlib, we compile and install it.

For AArch64 we need to copy `crt0.o` manually from `aarch64-hermit/Newlib/libc/sys/hermit` to installation library directory.

3.4 Adapting libhermit for CSWAP

HermitCore have same address space for the kernel and user application. As a result, the kernel is used as a normal library. The libhermit library is the core kernel part of HermitCore unikernel. libhermit build-infrastructure is built and sources are written assuming GCC as the compiler and LD as the linker. However, as we are using LLVM as the compiler of the hermit-popcorn toolchain, it requires some modification in libhermit to successfully compile it. I am going to talk about these modifications in two parts: source code and build-infrastructure.

3.4.1 libhermit source Code modifications

Unlike GCC, LLVM does not support nested functions. libhermit contains several nested functions. We define these nested functions as normal functions outside the function it was defined as before. For example, inner_fun is defined inside outer_fun as follows.

```
void outer_fun(void) {
    int i, j;
    void inner_fun(int *p) {
        while (p) {
            ...
            i += *p + 2;
            j = *p / 2;
            ...
        }
    }
}
```

To make this work in LLVM, we refactor both `inner_fun` and `outer_fun`. We define `inner_fun` outside of `outer_fun` as a normal function instead of nesting. We return some values from `inner_fun` if they are declared by `outer_fun` and changed by `inner_fun` as shown for `j` below. If there are multiple variables changed by `inner_fun`, then we pass

those local variables of `outer_fun` as the pointer parameters as we pass the reference of the variable `i` here.

```
static int inner_fun(int *p, int *i) {
    int j;
    while (p) {
        ...
        *i += *p + 2;
        j = *p / 2;
        ...
    }
    return j;
}

void outer_fun(void) {
    int i, j;
    ...
    j = inner_fun(p, &i);
    ...
}
```

When we compile libhermit for AArch64 LLVM doesn't execute the bitwise shift operation if we declare any constant using bitwise shift operation. Instead, it uses the expression as it was declared. For example, if we declare a constant `PAGE_SIZE` as follows,

```
#define PAGE_SHIFT  12
#define PAGE_SIZE (1 << PAGE_SHIFT)
```

and later use it for an assembly instruction like this,

```
add x0, x0, PAGE_SIZE
```

then LLVM cannot handle the intended expression and results in an error.

```
add x0, x0, (1<<12)
```

In this case, we simply use the numeric value for defining `PAGE_SIZE` which resolves the error.

```
#define PAGE_SIZE   4096
```

The method we described before how we replace all the usages of `dword` using `xword` in section 3.3 also applies to libhermit. In file `arch/aarch64/kernel/entry.S` we replace all the occurrences of `dword`.

Due to a bug in the Gold linker, we cannot read the Thread Local Storage (tls) size properly. As an alternate solution, we read the tls size from the unikernel application's ELF file. We iterate through all the program headers and when we find the program header for the TLS segment, we read the number of bytes in the memory image of the segment into a variable called `tls_size` as follows.

```
size_t memsz = phdr[ph_i].p_memsz;

if (phdr[ph_i].p_type == PT_TLS)
        tls_size = memsz;
```

This `tls_size` variable is actually the size of the TLS segment. Then, we write the value of this variable into a shared memory as follows.

```
*((uint64_t*) (mem_start + 0xD4)) = tls_size;
```

Here, `mem_start` is the physical location of where the shared memory starts and `(mem + 0xD4)` is the exact location where we are putting the value of `tls_size`. At kernel side, we read this value into a variable called `forward_tls_size` during the boot as follows.

```
.global forwarded_tls_size
forwarded_tls_size: .quad 0
```

In TLS initialization function, called `init_tls()`, we update current task's TLS size using `forwarded_tls_size`.

```
curr_task->tls_size = forwarded_tls_size;
```

We follow the same technique for both x86_64 and AArch64 versions of libhermit for forwarding the TLS size.

3.4.2 CMake changes for LLVM support

HermitCore uses CMake to build libhermit build-infrastructure, which compiles libhermit and other sample applications, produce libhermit package and link sample applications. This CMake infrastructure uses GCC for compiling libhermit. We make this infrastructure compatible with LLVM.

First of all, we change the compiler location from GCC to Clang.

```
- set(CMAKE_C_COMPILER ${TOOLCHAIN_BIN_DIR}/${TARGET_ARCH}-gcc)
+ set(CMAKE_C_COMPILER ${COMPILER_BIN_DIR}/clang)
```

Along with C, HermitCore supports other languages like C++, Fortran and Go. For CSWAP project we are only interested in running C applications. Thus, we disable support for other languages.

The CMake variable `HERMIT_KERNEL_FLAGS`, defined in `cmake/HermitCore-Toolchain-x86_64.cmake` for x86_64 ISA and `cmake/HermitCore-Toolchain-AArch64.cmake` for AArch64 ISA, contains compiler flags/options needed to compile libhermit. LLVM expects the target platform name for it is going to compile for as an option. Thus, we pass the option `` `-target x86_64-hermit' `` and `` `-target aarch64-hermit' `` for x86_64 and AArch64 respectively. We do the same for the cmake variable `HERMIT_APP_FLAGS` which is used for compiling HermitCore applications.

We also modify cmake variable called `HERMIT_APP_LINKER`, which is used for linking a HermitCore application with necessary libraries. We introduce a new cmake variable called

HERMIT_PREFIX, which is used for representing the installation locations. It helps us generalizing location of different libraries and object files pointed by HERMIT_APP_LINKER.

3.5 Adaptation of libomp for LLVM

Intel's OpenMP library comes as a submodule with libhermit, which enables us to build OpenMP applications. Like libhermit, it needs to be cross-compiled with LLVM. Thus, in CMakelist, we add -`DLIBOMP_CFLAGS=--target=x86_64-hermit' as a compilation flag so that the library gets cross-compiled with LLVM.

It is important to maintain the sequence of the libraries and the object file so that the linking process satisfies the dependencies between different libraries and object file. We create a new cmake variable called NPB_APP_LINKER, which holds the location of different libraries and object files as the following.

```
set(NPB_APP_LINKER
                     --target=x86_64-hermit
                    ${HERMIT_PREFIX}/x86_64-hermit/lib/crt0.o
                    ${HERMIT_PREFIX}/x86_64-hermit/lib/crti.o
                    ${HERMIT_PREFIX}/x86_64-hermit/lib/crtbegin.o
                    ${HERMIT_PREFIX}/x86_64-hermit/lib/crtend.o
                    ${HERMIT_PREFIX}/x86_64-hermit/lib/crtn.o
                    ${HERMIT_PREFIX}/x86_64-hermit/lib/libhermit.a
                    ${HERMIT_PREFIX}/x86_64-hermit/lib/libm.a
                    ${HERMIT_PREFIX}/x86_64-hermit/lib/libiomp.a
                    ${HERMIT_PREFIX}/x86_64-hermit/lib/libpthread.a
                    ${HERMIT_PREFIX}/x86_64-hermit/lib/libc.a
                    ${HERMIT_PREFIX}/x86_64-hermit/lib/libgcc.a)
```

crt0.o contains several execution startup routines for the C program, where crti.o, crtn.o, crtbegin.o, and crtend.o defines function prologs, defines the function epilogs, indicates the start of the constructors, indicates the end of the destructors, respectively. libhermit.a,

libm.a, libiomp.a, libpthread.a and libc.a is the kernel, math library, OpenMP library, and C library, respectively. The libgcc.a library contains the shared code, that would be inefficient to duplicate every time, as well as auxiliary helper routines and runtime support.

CMake build-infrastructure requires CMakelists.txt file for each application, which contains application name, source file list, compilation options, linking options, and the installation path. For a HermitCore application, which usages libomp, the CMakelists should be as follows.

```
file(GLOB srcs *.c)

add_executable(executable_name ${srcs})

target_compile_options(executable_name PRIVATE
            -fopenmp=libiomp5 -O3 -mcmodel=medium -isystem
                    ${HERMIT_PREFIX}/x86_64-hermit/include)
target_link_libraries(executable_name -fopenmp=libiomp5
                                        ${NPB_APP_LINKER})

install_local_targets(installation_path)
```

Here libiomp5 represents the openMP library we are using.

3.6 Linking a HermitCore application

For linking, HermitCore toolchain uses LD built for x86_64 and AArch64 named `x86_64-hermit-ld` and `aarch64-hermit-ld`, respectively. These executables contain default linker scripts written for HermitCore applications. On the other hand, Popcorn Linux toolchain modifies and uses Gold for linking. These modifications include generating some extra information regarding alignment and stack transformation.

HermitCore toolchain builds LD along with other tools in binutils, but not Gold. Moreover,

Gold requires bfd and libiberty to be built as a prerequisite. Thus, after building bfd and libiberty, we build Gold.

For hermit-popcorn toolchain, we decided to stick to Gold, instead of LD. The reason behind this is discussed in Chapter 4. As a result, unlike LD, we need to provide linker scripts for both x86_64 and AArch64 HermitCore applications. One way to extract the linker script, internally used in LD, to a file, e.g. output.txt, is running LD as follows.

```
x86_64-hermit-ld --verbose >> output.txt
```

Once the linker script is extracted, we can use it with -T option in Gold as follows.

```
x86_64-hermit-ld.gold -T output.txt
```

HermitCore expects it's application to have only three ELF segments: LOAD segment, TLS segment, and GNU_STACK segment. The LOAD segment describes areas of the new program's running memory like code, initialized and uninitialized data. The TLS segment, which is also a part of the LOAD segment, descirbes the areas specific to the initialized and uninitialized thread local data. GNU_STACK segment is not a part of the LOAD segment, and it is empty. GNU_STACK segment is used for holding different kinds of flags like the executable flag. These segments hold, in together, all the ELF sections of the application. However, when we link with Gold, it creates multiple LOAD segments and no TLS segment. As a result, HermitCore crashes when we run the application. To overcome this problem, we use PHDRS command in linker script as follows.

```
PHDRS {
        load_segment PT_LOAD ;
        tls_segment PT_TLS ;
        gnu_segment PT_GNU_STACK FLAGS (7) ;
}
```

It gives LOAD, TLS, and GNU_STACK segments separate names. We use these names to indicate which section should belong to which segment. For example, mboot section belongs to LOAD segment. So we use the name load_segment as follows.

```
.mboot phys : AT(ADDR(.mboot)) {
    *(.mboot)
    . = ALIGN((1 << 12));
    *(.kmsg)
  } :load_segment
```

Again, tdata section belongs to both LOAD and TLS segments. So load_segment and tls_segment are used as follows.

```
.tdata : {
     tls_start = .;
     *(.tdata .tdata.* .gnu.linkonce.td.*)
        tdata_end = .;
  } :load_segment :tls_segment
```

If we place a section in one or more segments using a name, for example `:load_segment`, then the linker will place all subsequent allocatable sections, which do not specify `:load_segment`, in the same segments. That is why we don't need to use `:load_segment` for every sections belong to the LOAD segment.

The usages of 'FLAGS (7)' for gnu_segment allows us to specify GNU_STACK segment as a read, write and executable (RWE) section.

When we link a normal, not heterogeneous or migratable, HermitCore application with GOLD, we need to link libraries like libhermit, libm, libc, and libgcc. We also need to link object files like crt0.o, crtbegin.o, crtend.o, crti.o, and crtn.o.

3.7 Stack transformation library

Popcorn Linux toolchain comes with state transformation runtime library, which translates execution state, i.e. stack and registers, of threads between ISA-specific formats when there is a migration request. The state transformation runtime takes the snapshot of thread's

register state, call frames and pointer to other stack objects. Once these snapshots are taken, currents thread's live function activations are reconstructed in the format expected by the destination ISA, using metadata generated by the compiler. Finally, OS's thread migration mechanism is invoked by the runtime, and the application is bootstrapped on the destination machine and resume execution.

CSWAP is different from Popcorn Linux in the sense that the host OS is a libOS instead of full-fledged Linux, and in the destination machine, a new unikernel is bootstrapped before the application part is bootstrapped and resumed. However, still, we use stack transformation library for reconstructing the application state. We need few changes in Newlib, libhermit, and the stack transformation library itself to make it work for CSWAP.

3.7.1 HermitCore changes

Popcorn Linux is built on top of Linux which lets stack transformation library use Linux kernel APIs. But CSWAP uses HermitCore, which is a lightweight unikernel and provides limited number of APIs. As a result, we implement some system calls required by stack transformation library, but not implemented in HermitCore.

One of them is sys_stackaddr, which returns the address at the beginning of the area reserved for the stack of the calling thread. It is defined as follows.

```
void* sys_stackaddr(void) {
        task_t* task = per_core(current_task);
        return task->stack;
}
```

Another is sys_stacksize, which basically returns the calling thread's stack size. It is defined as follows.

```
size_t sys_stacksize(void) {
        return DEFAULT_STACK_SIZE;
}
```

Here DEFAULT_STACK_SIZE is the default size of the stack, as the name implies. This size is defined in cmake/HermitCore-Configuration.cmake and is kept similar to the size of MAX_STACK_SIZE, defined in stack transformation library.

Other than these, we implement sys_gettimeofday_init and sys_gettimeofday for providing the current time.

3.7.2 Newlib modifications

__popcorn_stack_base pointer in Newlib is used for indicating the starting location of the stack so that the stack transformation runtime knows upto which point the stack frames need to be transformed for the destination ISA. Thus, it should be the location of the main function in the stack. In Newlib, we setup the stack base location as follows.

```
__popcorn_stack_base = &argv;
if (env) {
    environ = env;
    __popcorn_stack_base = &env;
}
```

3.7.3 Stack transformation library modifications

Stack transformation library uses clock_gettime() to get the current time which is supported neither by Newlib nor by HermitCore. Instead, HermitCore provides function gettimeofday() for getting the current time. Thus, we replace all occurrences of clock_gettime by gettimeofday().

CSWAP targets single threaded applications for offloading from the to the board and vice versa. As a result, there will be only one stack. Thus, we simplify stack bound values, the start and the end of the stack, as follows in both functions called get_thread_stack() and get_main_stack() in src/userspace.c.

```
bounds->low = 0x0;
bounds->low = sys_stackaddr();
bounds->high = (void *)((uint64_t)bounds->low
                    + (uint64_t)sys_stacksize()) - 1;
```

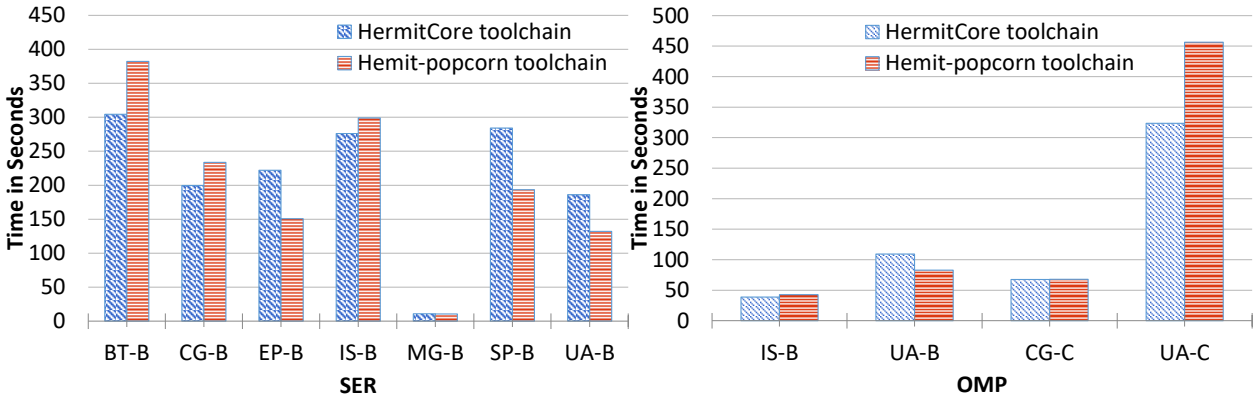
Like other libraries, we cross-compile stack transformation library for both x86_64 and AArch64, respectively, using the option –target=x86_64-hermit and –target=aarch64-hermit.

3.8 Validation of hermit-popcorn toolchain

To validate the correct behavior of the hermit-popcorn toolchain we run compute-intensive datacenter workloads from NPB [4, 54] benchmark suits. We compare the execution time of NPB applications, both serial and parallel, built as unikernel using HermitCore toolchain and hermit-toolchain. The HermitCore uses GCC 6.3.0, where the hermit-toolchain uses LLVM 3.7.1 for the compiler. We do not compare the performance with popcorn Linux toolchain because popcorn Linux toolchain does not build unikernels.

We built and ran these benchmarks on an Intel Xeon machine, running on 3.5GHz clock frequency and 8GB RAM. The host operating system, running on this machine, is Ubuntu 16.04 LTE with kernel Linux 4.4. Each benchmark is run at least 3 times, and the average execution time is taken. As IS execution time is very small, in order to increase the visibility, the execution time of IS, for both serial and parallel, is magnified 100 times.

The results for NPB serial benchmarks are shown in Figure 3.3a. As one can see, for some benchmarks, BT, CG, and IS class B, HermitCore toolchain performs better than the hermit-popcorn toolchain. For other presented benchmarks, EP, MG, SP, and UA class B, hermit-popcorn toolchain performs better than the HermitCore toolchain. From the presented result, we can say that usage of hermit-popcorn toolchain may increase the execution time as high as 25% (BT class B) and reduce the execution time as low as 47% (EP, SP class B).



(a) NPB serial benchmark.

(b) NPB parallel benchmark.

Figure 3.3: NPB benchmark performance comparison between HermitCore toolchain and hermit-popcorn toolchain.

The results for NPB parallel benchmarks are shown in Figure 3.3b. It shows that for most of the cases HermitCore toolchain outperforms hermit-popcorn toolchain. However, the performance difference is very small for most of the cases. The results show that the usage of hermit-popcorn toolchain may increase the execution time as high as 41% (UA class C) and decrease the execution as low as 32%.

Chapter 4

Unikernel Memory Alignment for Migration across Heterogeneous-ISA

To make sure that the application's memory is consistent after a unikernel is migrated from one machine to another, we need to align the corresponding memory. Section 4.1 describes the problem space. In general, we consider for alignment from the perspective of memory location and corresponding pointers. Next, section 4.2 and 4.3 discuss our approach to solve this problem for static memory, and section 4.4 describes how we do the same thing for dynamic memory.

One of the core objectives of CSWAP is to migrate applications from one ISA to another. Two ISAs, the source and the destination, can be different in many aspects, such as their definition of the supported data types, state of the main memory and registers, their semantics such as memory consistency and addressing mode, their instruction sets, and their input/output model.

Each application runs inside a unikernel. As ISAs are very different from each other, one unikernel built for one ISA cannot be run on another ISA. It leads us to building ISA-specific

unikernels for the same application: one for source ISA and another for destination ISA, if the source and destination ISAs are different.

When there is a migration request during the execution, a snapshot of the register states is taken, and then stack transformation runtime is invoked. The runtime uses the stack pointer from register state to attach to the thread's stack and convert all live function activations from the source ISA format to the destination ISA format. Once the transformed stack is transferred to the destination ISA, the corresponding unikernel is booted, and the kernel sets the transformed register state and resumes the execution of the application.

The state transformation runtime, discussed in section 3.7, does not transform the pointers pointing to global data, function, or heap location, assuming these pointers remain valid before and after migration. As the source and the destination ISA's are different, and so are corresponding generated unikernels, without any explicit relocation or alignment of pointer targets there is no guarantee that the symbols in both unikernel binaries are located in the same virtual address. As a result, the corresponding pointer may end up pointing to two different locations in the source and destination ISAs. To prevent this scenario from happening we need to align all the ELF sections, corresponding symbols, and heap memory areas. We assume both ISAs have the same size for the primitive types.

This chapter discusses different types of necessary alignments and describe how we implement these using the hermit-popcorn toolchain.

4.1 Alignment from the pointer perspective

Typically, a C program memory layout consists of two major parts: dynamic and static memory. There are two types of dynamic memory: stack and heap. The stack contains stack frames where each of them stores a function's return address and the caller function's environment. The heap, like the stack, provides runtime memory allocation meant for data that must outlive the function doing the allocation. On the other hand, static memory usually

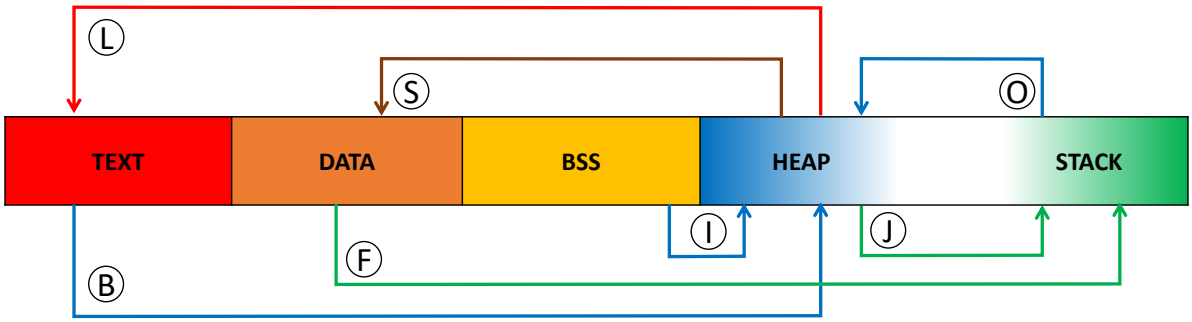


Figure 4.1: Memory referencing scenarios between different parts of the memory layout.

consists of several sections like TEXT, DATA, and BSS. TEXT contains local variables and instructions. DATA and BSS contain initialized and uninitialized global data respectively.

As shown in Figure 4.1, C language allows any part of the memory layout of a program to be pointed at by a pointer variable residing in another part. For example, a memory location in the heap can be pointed by TEXT Ⓑ, DATA Ⓕ, BSS Ⓘ, and stack Ⓞ. Similarly, a pointer residing in the heap can point to any location in TEXT Ⓛ, DATA Ⓢ, BSS Ⓟ, and stack Ⓙ. If any of these pointed locations get displaced for any reason, such as migration, then the pointer will end up pointing to an invalid location. So when the program dereferences the pointer, it will not get the correct value and thus the program will not act as expected.

4.2 Section alignment

Sections comprise all information needed for linking and relocating a target object file in order to build a working executable. There are different types of sections in an ELF object like DATA, TEXT, BSS, RODATA, etc., which basically store static information of the program. These sections can have different alignment across different ISAs which can result in unexpected program behavior once it migrates.

Let's consider the following example application. It has a global variable `i` and `i` is pointed by a pointer `p`. As `i` is a global variable, it belongs to DATA section. Let's assume the address of the variable `i` is 0x67. So `p` will be pointing to the value at address 0x67, which

is 37 as indicated by Ⓒ in Figure 4.2.

Listing 4.1: Example C program shows simple pointer usage

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int i = 37;
5  void (*fun_ptr)(int);
6
7  void fun(int a) {
8          printf("%d", a);
9  }
10
11 int main() {
12         int *p = &i;
13         fun_ptr = &fun;
14         int *q = malloc(sizeof(int));
15         *q = 45;
16
17         printf("%d", *p);
18         (*fun_ptr)(10);
19         printf("%d", *q);
20
21         return 0;
22 }
```

As two ISAs are very different from each other, for this same application the address space may have a different layout, i.e., sections and corresponding symbols may not be aligned across architectures, as shown in figure 4.2. As a result, the same content in a section may have different address across architectures. So when our example program gets migrated and resumed after the execution of line 15 from ISA A to ISA B, it may print some value different from 37 at line 16, because address 0x67 may be pointing Ⓩ to a garbage value

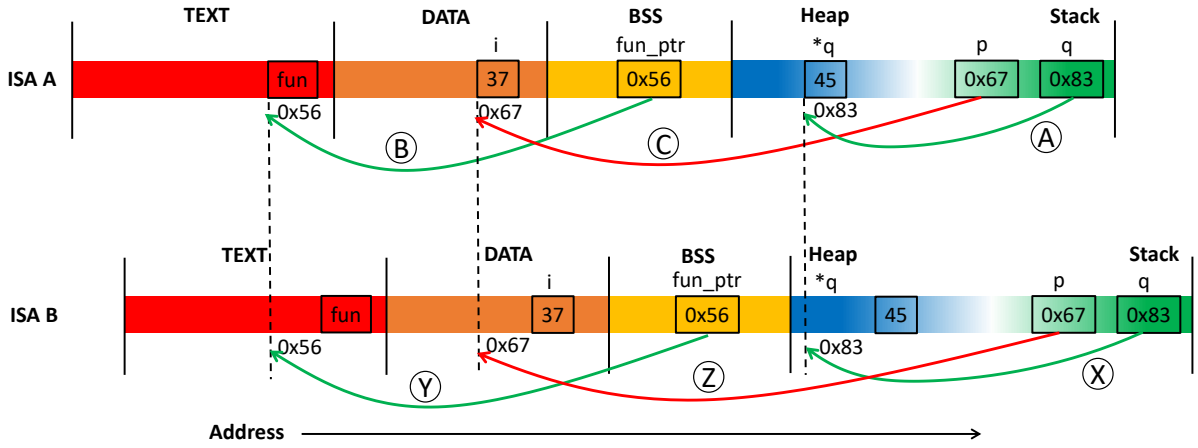


Figure 4.2: Different alignment of memory layout across different ISAs for the same program.

instead of `i`, because the DATA section starts from a different location in the destination ISA.

If `i` was an uninitialized global variable, it would be located at BSS and if `i` was a local variable it would be located at stack instead of DATA. But the above-discussed scenario would remain the same as section and symbols are unaligned across architectures. The same scenario holds for pointers pointing some value at TLS sections.

Now let's consider the function pointer `fun_ptr` in the presented program. As it is declared as an uninitialized global variable, it resides in the BSS section. `fun_ptr` points to the function called fun()(Ⓑ in Figure 4.2). Any program's source code belongs to its TEXT section and, like other sections, a program is not guaranteed to have the TEXT section starting from the same location across different architectures. So if the program gets migrated and resumed after the execution of line 13, it may not be able to successfully call fun() because the TEXT, as well as the function location, got shifted. As a result `fun_ptr`, in BSS, may happen to point to a wrong location Ⓨ instead of fun().

To prevent these scenarios from happening, first thing we need to do is align section boundaries. It simply means making sure that the same section starts from the same virtual address across different ISAs.

In order to do this, we modify the HermitCore linker script for both x86_64 and AArch64. The syntax for aligning a section in linker script is as follows:

```
.section_name : ALIGN(align_value) {
    ...
    ...
}
```

Here `section_name` is the name of the section, e.g. text, data etc. and `align_value` is a numeric value. The ALIGN directive makes the corresponding section start from a virtual address which is multiple of `align_value`.

When we migrate an unikernel between ISAs, we do not migrate the core kernel part, only the application part. The kernel code is so ISA-specific that it will not work on different ISAs. So we launch a new unikernel specific to the target ISA in the destination machine. It means we do not need to align kernel specific sections like mboot, kdata, ktext, and kbss which contain data regarding kernel booting, initialized data, code, and uninitialized data for kernel respectively.

The rodata section in libhermit contains kernel specific symbols which we are not interested in because we only migrate some states of the kernel and all application states. To distinguish application's rodata from kernel rodata we rename kernel's rodata as krodata. We do this using objcopy as follows:

```
objcopy --rename-section .rodata=.krodata libhermit
```

We only align sections presented in Table 4.1 with corresponding alignment values. These alignment values are set based on several experiments analyzing different applications like NPB and Parsec benchmarks so that the values are sufficiently big to have the same alignment across ISAs for any application.

Section Name	Alignment Value
text	0x100000
data	0x100000
tdata	0x200000
tbss	0x100000
bss	0x2000000
rodata	0x100000

Table 4.1: Aligned sections and corresponding alignment values.

4.3 Symbol alignment

Section alignment only takes care of the starting addresses of the sections across different ISAs. But a section consists of different symbols, which represent data and functions in the section. Symbols have their sizes and alignment constraints, and these can vary from ISA to ISA for the same symbol. Moreover, a symbol in one ISA can be absent in another because of the ISA-specific implementation of some part of the program. So we need to align these symbols across different architecture-specific binaries. Figure 3.2 in page 25 shows different components of the whole aligned binary generation process.

We align symbols across architectures in three distinct steps. In the first step, we compile and instrument the sources, kernels and libraries(Ⓐ,Ⓑ, and Ⓒ) and link them Ⓓ to generate unaligned binaries Ⓔ for both x86_64 and AArch64 and their corresponding map files Ⓕ. Using `-map` option with a file name in the linking command produces the map file for the executable. A map file contains all the symbols in a section and their corresponding information, which includes the name of the section it belongs to, virtual address, size, the name of the package it belongs to, and the object file name. But it doesn't contain symbols' alignment size which is necessary for aligning symbols across ISAs.

In order to get the alignment size along with other existing information from the map file, we modify Gold. In `print_input_section` function in the file `gold/mapfile.cc` we print the alignment size to the map file.

Creating section for each symbol

To have the necessary information about each symbol from the map file and to manipulate their locations, we need each of them to have their own section. The way to do this for all functions and data is to use `-ffunction-section` and `-fdata-section` options respectively in the linker command.

Even after using these options there can be some symbols left which do not have their own section in the ELF file, because they are defined in assembly instead of C language. For example, memcpy, memset symbols in libc.a for x86_64, defined respectively in `newlib/libc/machine/x86_64/memc` and `newlib/libc/machine/x86_64/memset.S`, do not have their own section. We force these symbols to have their own section by using following instructions in their respective sources:

```
.section .text.memcpy, "ax"
.section .text.memset, "ax"
```

we set the name for the resulting section based on the section name it used to belong to, followed by a dot and the symbol name to mimic what `--ffuncion_section` does for C files. For the same problem from libc.a for AArch64 the instruction is different, which is as follows:

```
.section .text.\f
```

This way we modify 11 functions in their corresponding assembly files.

Generating aligned linker scripts

Once we have all the symbols in their own sections, our next step is to generate aligned linker scripts Ⓗ. We adapted a tool from [5], developed in python, called pyalign Ⓖ to generate aligned linker scripts. How does this pyalign tool work and how we ported this for unikernel alignment is discussed next.

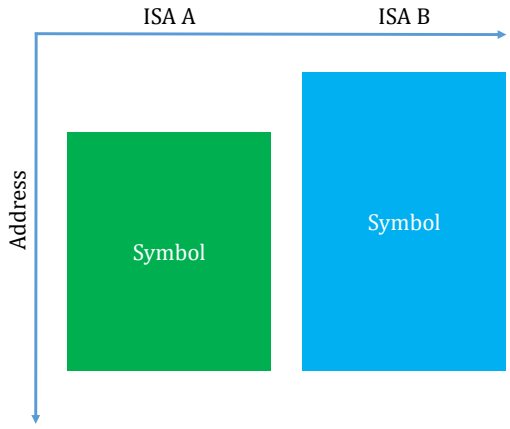
First of all, pyalign reads all the available symbol information from the map file. This information includes the symbol's name, address, size, alignment size, and the corresponding object file name. Then it categorizes these symbols according to their relevant section. For determining the section, it reads the corresponding unaligned ELF executable and extracts section's name, address, and size. Note that for generating migratable unikernel binaries, we are only interested in a subset of sections as we discussed in section 4.2. If a symbol's address falls into any of these section's address range, then it puts that symbol in a dedicated list representing the corresponding section. It does not place any symbol in the list if it is already there. All of these lists constitute a dictionary, say SymbolsDictionary. pyalign repeats this same procedure for all architectures and updates SymbolsDictionary. As a result, it gets all of the symbols gathered in SymbolsDictionary that belong to our considered sections from all ISAs.

Next, pyalign sorts the list of symbols for each section individually. The sorting is done in a descending order based on the number of ISAs referencing the symbol.

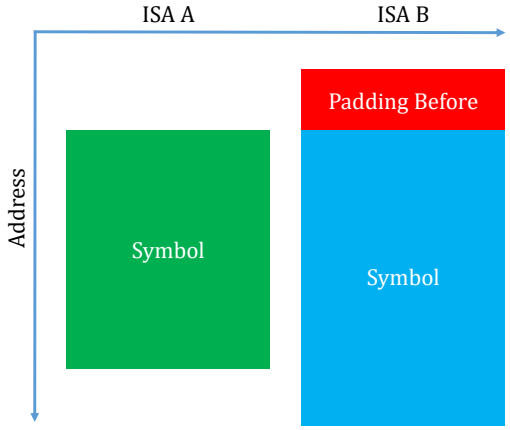
Once pyalign sorts the lists, it aligns each symbol in a list across different ISAs by adding necessary padding. In some cases, we need to add padding before the symbol. For example, in 4.3a the symbol in ISA A has bigger alignment constraint than the same symbol in ISA B. In this case, pyalign picks the bigger alignment constraint and applies it to both symbols which makes it aligned at the symbol's starting location across ISA A and B as shown in 4.3b.

Due to the usage of before padding or the different size of a symbol across different architecture, two symbols may not end at the same address. For example, in 4.4a symbol in ISA is smaller in size than the symbol in ISA B. As a result, the next symbol to be processed won't start from the same location. In this case, we use padding after the symbols as much as necessary as shown in 4.4b.

In some cases, there can be some symbols which are present in one ISA but not in other. For example, ISA A has a symbol which is absent in ISA B. For this symbol we fill the alignment

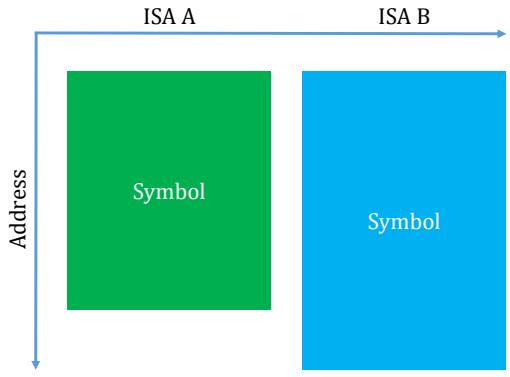


(a) Symbols are unaligned across ISAs.

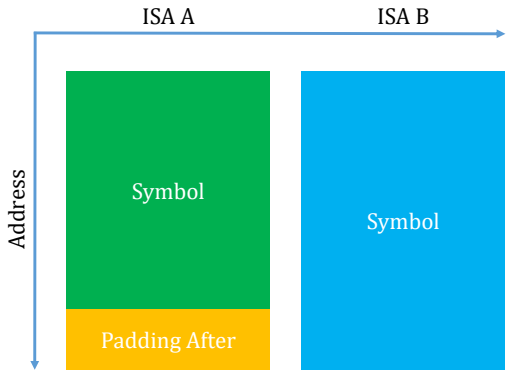


(b) Starting location is aligned using padding.

Figure 4.3: Padding before helps aligning the starting location of a symbol across ISAs.



(a) Symbols end at different location across ISAs.



(b) Ending location aligned using padding.

Figure 4.4: Padding after helps next symbol to start at the same location across ISAs.

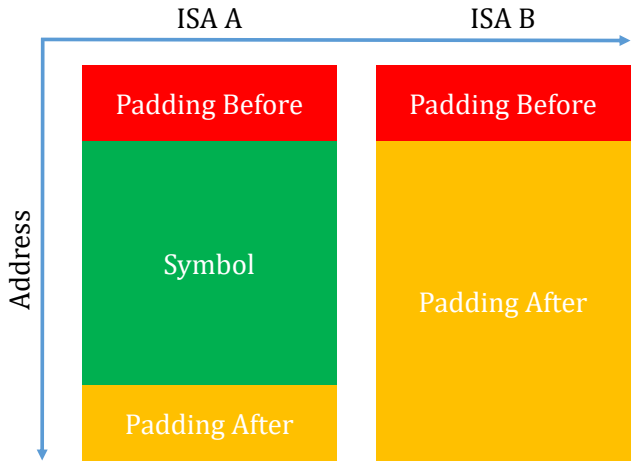


Figure 4.5: Padding for a symbol that doesn't exits in some architecture.

gap using before and after padding, so that the next symbol to be processed can start from the same location as shown in 4.5. Before padding and after padding information are stored as the symbol's attributes.

Once we have all the padding information for all the symbols, we produce aligned linker scripts for each ISA. As we are not going to change all the sections in the linker scripts, we use template linker script. This template contains all the sections and their linking information except for our considered sections. In the place of our considered sections, we place a marker so that we know where to put our section and symbol linking information. For example, for TEXT section we place the marker `__TEXT__` as follows.

```
.text : ALIGN(0x100000) {
__TEXT__                /* marker for text section insertion */
}
```

We put the linking information of the symbols belonging to the TEXT section in the linker script by replacing `__TEXT__`. The format for putting a symbol's linking information is as follows.

```
. = ALIGN("Before Padding Value");
"Object File Path"("Symbol Name");
. = . + "After Padding Value";
```

The third and final step for symbol alignment is to generate Ⓘ aligned binaries Ⓙ for each ISA using corresponding aligned linker scriptsⒽ.

4.4 Heap alignment

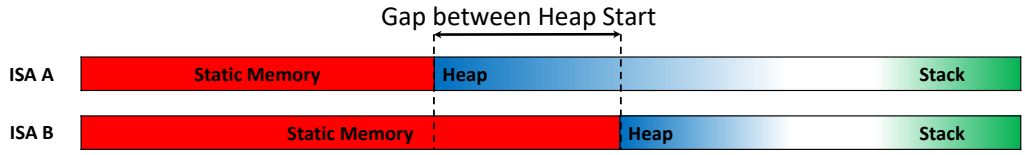
Along with the sections and the symbols, we need to align heap, because heaps can have different starting addresses in different architectures. If any pointer is pointing to a heap location and the heap start location is different in the destination ISA, then after migration and resume it may point to a wrong location.

Let's consider 4.1 once again, where `q` is an integer pointer that is allocated a memory location in the heap by `malloc()`, which holds the value 45. Let's assume the pointer q points to the address 0x83 in the heap as shown as Ⓐ in Figure 4.1.

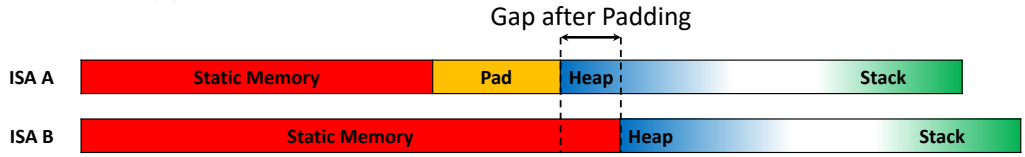
There is no guarantee that the same application will have its heap started from the same location across different ISAs. So it is possible that, if this program gets migrated and resumed after the execution of line 15, it will print some value different than 45 because of a change in the heap's starting location, and thus addresses of contents in the heap will also change Ⓧ.

To prevent this scenario from happening the heap start location of the unikernel should be the same for both source and destination. In order to do this, we first determine the possible maximum gap, as shown in Figure 4.6a, between heap start location of x86_64 and AArch64 unikernels for different applications. We ran all NPB serial benchmarks and found that the maximum gap for them is no more than 0x1600000 and the start location of x86_64 is always ahead of AArch64. Thus, we pad AArch64 heap start-location by 0x1600000.

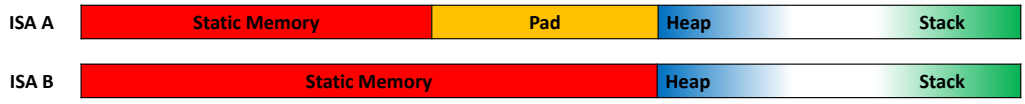
However, this padding doesn't guarantee that the heap start location will be aligned across x86_64 and AArch64 for all applications. There still can be some gap even after padding, as shown in Figure 4.6b, as it just minimizes the initial difference. Thus, we round them up to



(a) Gap between the heap start of two unaligned binaries.



(b) Gap between the starting address after padding the heap start location.



(c) Aligned start location of heap after padding and rounding up.

Figure 4.6: Aligning heap start location of unikernel binary across different ISAs.

the closest boundary of 0x1000000 so that they have the same heap start as shown in Figure 4.6c. We edit corresponding file, `include/hermit/stddef.h`, in the HermitCore source for these changes as follows.

For x86_64:

```
#define ALIGN 0x1000000
#define BEFORE_ALIGN    (PAGE_2M_CEIL(STACK_SLOTS_START +
                    STACK_SLOTS_NUM * DEFAULT_STACK_SIZE ))
#define HEAP_START      BEFORE_ALIGN%ALIGN == 0 ? BEFORE_ALIGN :
                    (BEFORE_ALIGN + (ALIGN - BEFORE_ALIGN%ALIGN))
```

For AArch64:

```
#define PADDING 0x1600000
#define ALIGN 0x1000000
#define BEFORE_ALIGN    (PAGE_2M_CEIL(((size_t)&kernel_start +
                    image_size + (16ULL << 10) + PADDING)))
#define HEAP_START BEFORE_ALIGN%ALIGN == 0 ? BEFORE_ALIGN :
                    (BEFORE_ALIGN + (ALIGN - BEFORE_ALIGN%ALIGN))
```

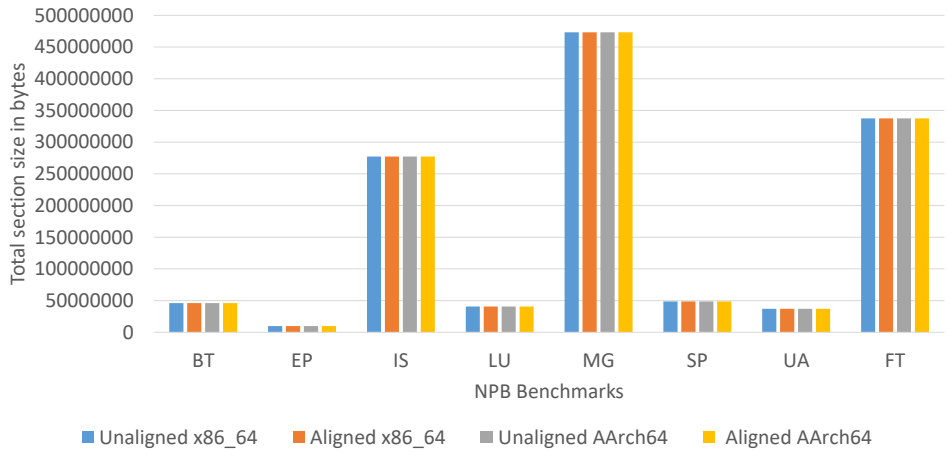


Figure 4.7: Considered section's total size comparison.

4.5 Aligned unikernel binary size

The previous sections describe how we apply paddings and alignment constraints to get a common address space for both static and dynamic memories across heterogeneous ISAs. These memory alignment techniques increase the size of the produced unikernel binaries. As a result, it is normal for one to think that the resulted bigger unikernels require more physical memory to finish their execution. However, the increase in the size of the unikernel binary is not directly related to the increase in the necessary physical memory.

We apply paddings to get same starting addresses for the considered ELF sections, as described in 4.2, across different ISAs. These paddings increase the size of the unikernel binary. However, when a section is mapped to the physical memory, the mapping starts from the starting address of that particular section. As a result, the corresponding padding does not increase the size of the mapped physical memory. The same thing is true for the dynamic memory, i.e. heap.

On the other hand, the paddings and alignment constraints, used to align the symbols inside a section, increases the corresponding section size, which contributes to the increase in the binary size as well. As the whole section is mapped to the physical memory, the section containing aligned symbols requires more physical memory than the unaligned one.

To get an idea about how much bigger the section's size get due to memory alignment, we built different NPB benchmarks as unikernels and aligned them across x86_64 and AArch64 ISA. For each unikernel, we calculated the total size of the considered sections, see Table 4.1, before and after the alignment for both ISAs. These sizes are presented in Figure 4.7. In this figure, one can see that the section's total size doesn't change significantly due to the static memory alignment. As a result, from the above discussion and the presented results, we can say that the memory alignment increases the binary size, but this does not necessarily require a lot more physical memory.

Chapter 5

On-demand memory transfer

We introduce a post-copy method, called on-demand memory transfer, replacing CSWAP's full checkpoint/restart method for faster and efficient migration of unikernel in a heterogenous-ISA setup. Section 5.1 discusses the potential of on-demand memory transfer method over full checkpoint/restart. Section 5.2 describes on-demand memory transfer design and benefit. Section 5.3 discusses the implementation detail of the on-demand memory transfer method. Finally, section 5.4 discusses the performance evaluation of the implemented on-demand memory transfer method over existing full checkpoint/transfer method.

5.1 CSWAP's full checkpoint/restart

Along with the stack transformation, alignment, and other necessary steps, a unikernel migration includes the transfer of the VM states from the source to the destination machine. These states may include dynamic memory, like heap and stack contents, and static memory contents, like metadata and different section's data. These states can be transferred in different ways in a different point of time based on the algorithm used, like [47], pre-copy [15], and post-copy [29]. CSWAP implements full checkpoint/restart method to transfer

VM memory from source machine to destination machine.

5.1.1 Design of full checkpoint/restart

During the execution, once we reach a migration point, for the checkpoint/restart technique, the execution of the application part is stopped, and the preparation for the migration is started. Next, different contents belonging to different parts of the application and the kernel are dumped in the files which we call checkpoint. For example, after doing stack transformation on the stack contents, they are stored in a file called stack.bin. Similarly, heap and other sections' contents are stored in the individual files. Once everything necessary is dumped to the files, these files are transferred to the destination machine using secure copy protocol (SCP). Next, during the resume process on the destination machine, these files are read and the stack, heap, and different sections' memory are updated with the contents of the files. As a result, the program can resume exactly from the state it was stopped at in the source machine.

5.1.2 Problem of full checkpoint/restart

There are a few drawbacks associated with this approach. In order to explain these problems, We divide the memory transfer time for checkpoint/restart into three parts: checkpointing time, transferring time, and restoring time. Checkpointing time is the time required to dump the VM states into the files. With the increase of the VM state's size, the checkpointing time increases. This contributes to the downtime, the time the application will not be in execution. Next, the transferring time refers to the time necessary to transfer the dumped VM states. Full checkpoint/restart method does not boot the unikernel in the destination machine until all the states are transferred. The longer the transfer time, the longer the unikernel has to wait to resume in the destination machine, which contributes to the downtime. Finally, the restoring time corresponds to the necessary time to restore the VM states from the checkpoint. The bigger size of the checkpoint requires a longer time to restore the VM states,

and only after the restoring is done can the application resume to its normal execution. So the restoring time also contributes to the downtime. From the user point of view, the downtime corresponds to the amount of time the service is unavailable to them. This is due to there being no currently executing instance of the application, so it is undesirable to have an increase in the downtime. In order to decrease the downtime, we replace the checkpoint/restart with the on-demand memory transfer.

5.2 On-demand memory transfer overview

On-demand memory transfer adapts the concept of the post-copy [29] live migration approach. Like the post-copy approach, on-demand memory transfer sends the CPU states and starts the VM in the target machine. Later the other VM states are pulled, while the application in the target VM is running. However, unlike post-copy, we cannot transfer the entire OS, as due to heterogeneous ISA setup, it is not possible to map source kernel states with the destination kernel. Instead, we extract the application state and some ISA independent kernel states and transfer them.

5.2.1 Benefit of on-demand memory transfer

Unlike the checkpoint/restart, on-demand memory transfer boots the unikernel on the destination machine and resumes the application as soon as possible, without transferring all the contents of the VM states. This minimal checkpoint transfer reduces the checkpointing time and the restoration time. The downtime is reduced because the application doesn't have to wait for resuming until the entire memory is checkpointed and restored.

At the same time, an application may not need all the VM states, generated in the source machine, to finish the execution in the destination VM. It may require only a subset of the VM states to be restored to finish the execution. As a result, using on-demand memory transfer, we may need to send less data than is required for checkpoint/restart. It helps to

reduce the total network traffic and associated latencies. Thus, required memory transfer time is less compared to full checkpoint/restart.

5.2.2 Design of on-demand memory transfer

When there is a migration request, like checkpoint/restart, the kernel stops the application's execution and starts checkpointing VM states and registers. Instead of checkpointing the entire VM states and registers, in on-demand memory transfer, we checkpoint just the transformed stack and some metadata, i.e., heap and different section's contents are not dumped into the files. Then we transfer this minimal checkpoint and boot the unikernel in the destination machine. Next, the hypervisor in the source machine starts a server to respond with non-checkpointed memory contents each time there is any request.

After restoring the minimal checkpoint, the application resumes the normal execution. During the execution of the application, if it accesses a memory content which is not retrieved from the source VM, it results in a page fault. This page fault is forwarded to the associated hypervisor. The hypervisor requests and retrieves the corresponding memory from the server in the source machine. For the on-demand basis memory retrieval design, we name it on-demand memory transfer.

Apart from restoring VM states only when the application requires the memory, we proactively pull remote memory from the source machine. As a result, the application doesn't page fault for every first access to memory in the destination VM. Additionally, we retrieve the pages in batches which helps to reduce network latency.

5.3 Implementation of on-demand memory transfer

To implement the on-demand memory transfer technique, we adapted the traditional client-server model. The uhyve hypervisor running the unikernel at the source machine works as

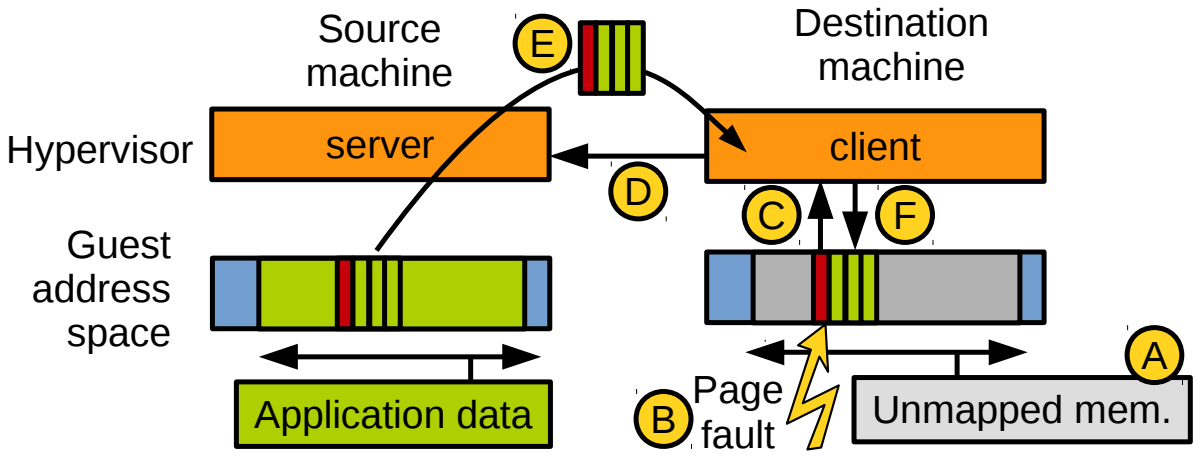


Figure 5.1: On-demand page fault handling.

the page server as shown in Figure 5.1. The uhyve hypervisor, running at the source machine, works as the page client. This client-server model is implemented using TCP/IP where we maintain a persistent connection. That means instead of establishing the connection for each time there is a request, we establish the connection once at the beginning of the resume process.

Once the minimal checkpoint is dumped into the files, the unikernel at the source machine sends a hypercall to the corresponding uhyve hypervisor requesting to initiate the on-demand memory server. Having the request received from the unikernel, the hypervisor starts the server. The server keeps listening for a connection request at the port provided by the agent who launched the unikernel at the beginning.

On the other hand, during the resume process, the unikernel at the destination machine makes hypercall to its corresponding hypervisor, requesting to establish a connection. Having the request received, the hypervisor starts the client and sends a connection request to destination machine at the port the server is listening to. The port number is set by the agent who launched the unikernel at the destination machine. Once the server at the destination machine receives the connection request, it establishes the connection.

Once the connection is established, the application is resumed at the destination machine.

Figure 5.2: Request packet contents.

As shown in Figure 5.1, during booting, the unikernel in the destination machine makes sure that the areas of the address space corresponds to the application memory are unmapped Ⓐ. As a result, if the application accesses a memory location which is not mapped, it results in a page fault Ⓑ. Each time there is a page fault, it is forwarded to the page fault handler in the unikernel. If the page fault location belongs to a memory which is not retrieved yet from the source machine, then a new page is allocated in the memory, and a hypercall Ⓓ is made to the hypervisor, which includes the following information.

- VAddr: Virtual address of the requested memory location;
- PAddr: Physical address of the allocated page in the memory;
- HType: Type of the hypercall: Page Fault;
- MType: Type of the memory requested: heap, DATA, BSS, and etc;
- PSize: Each page size of the requested memory;
- NPages: Number of pages requested.

Once the hypervisor receives a hypercall, it first analyzes the type (HType) of the call. If it is for the page fault, then the hypervisor sends a packet requesting for the memory contents to the server running in the destination machine. A request packet contents look like Figure 5.2.

When the server receives a request, it extracts the packet and reads the information corresponding to the request, which are MType, VAddr, NPages, and PSize. For simplicity's sake, let's assume the number of pages requested, NPages, is 1 for now. For each request received for a page, the server translates the requested virtual address (VAddr) into a physical

address and reads the page of size PSize from the physical memory into a memory buffer. Then the buffer is sent to the client at the destination machine (Ⓔ on Figure 5.1).

After receiving the buffer, filled with the requested page, the hypervisor at the destination writes it into the guest's (unikernel's) physical memory (PAddr). Next time, if the application accesses the same page, the page fault is not triggered anymore because the requested page is already there. Both the source and destination hypervisor keep track of what type of memory (MType) and how much memory is transferred. Once the whole memory is transferred, the server and the client closes the TCP connection.

5.3.1 Batching the memory transfer

So far, it is described how a page request is triggered by the application, and how a page is transferred using on-demand memory transfer. However, we observed that this approach incurs a lot of overheads. One reason is for the each packet transfer, request or response, there is an associated latency. The more packets we transfer over the network, the more latencies are incurred. It is especially critical for the low-grade slow network interfaces of some embedded systems.

At the same time, each page fault management requires a significant amount of time which contributes to the overhead. As a result, he more page faults occur, the bigger the overhead.

For these above reasons, we decided to batch the page requests. When there is a page fault, instead of requesting that one page, we request in batch. Lets say `n` number of consecutive pages, including the faulted one, to the server. As a result, `n` page requests are replaced by only one page request. If the latency for one request in `l` seconds, then we are reducing the transfer time to be `(n-1)l` seconds. Upon receiving the request, the server translates the corresponding virtual address into a physical address using the function `guest_virt_to_phys()` as follows.

```
uint64_t physical_address = guest_virt_to_phys(VAddr + i*PSize);
```

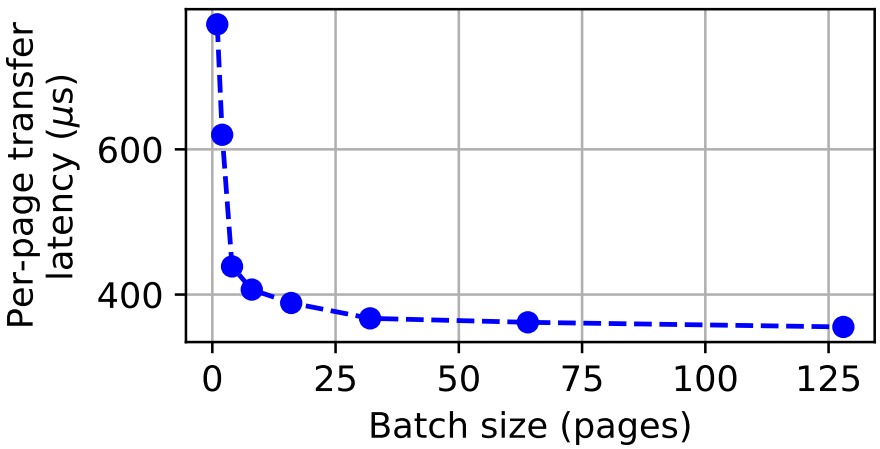


Figure 5.3: Batching impact on per-page transfer latency.

Here, i is the index of the page which starts from 0. Next, the hypervisor reads the memory that corresponds to the page into a buffer. This process is repeated until all the requested pages are read into the buffer. One may think about reading all the pages at once, i.e. reading the memory of size equals NPgaes*PSize. However, it is not guaranteed that the virtual memory is continuously spread in the physical memory. Thus reading the whole memory at once may result in a page fault.

This bulk response from the server to the client reduces the overall overhead due to the latency and page fault management. We measured the amount of time required to transfer different sizes of batch transfers between the Potato board and a typical server. In the Figure 5.3 we present cost of different batch sizes. We can see that the speed can be improved by about 50% for a batch of size 4 or 8 pages, compared to transferring the pages one by one. Once the response packet is received, the client writes the pages into the guest's physical memory.

5.3.2 Proactive memory transfer

In case of on-demand memory transfer, the server and the client keep track of how much memory is served and how much memory is left to be pulled remotely respectively. Either

one of them, the server or the client, can terminate the TCP connection between them. The server terminates the connection if all the memory that is left behind is served by it. The client terminates the connection if either the application finishes the execution in the destination machine, or all of the remote memory is pulled by it. This means, as long as the connection is not terminated for any particular reason, the server keeps running in the source machine. As a result, the corresponding resources in the source machine are kept occupied, which are mostly RAM, as the CPU usage while serving remote memory is minimal.

Algorithm 1 Pro-active memory transfer

```
1: time ← 300
2: for heap or each section do
3:     start ← start_address
4:     end ← page_ceil(end_address)
5:     for i ← start to end − 1 do
6:         if check_pagetables(i) then
7:             continue
8:         end if
9:         j ← ∗ ∗ i
10:        msleep(time)
11:        i ← i + PSize
12:    end for
13: end for
```

One core objective of CSWAP is to migrate workloads from one machine to another, usually from the server to the low-end devices, and free resources as soon as possible. Thus, the source machine can accommodate more jobs. However, with on-demand migration, the state transfer end time is quiet indeterministic as it depends on the memory usage of the application after being resumed. As a result, when the resources will be freed in the source machine is also quite indeterministic, which hinders us from fulfilling the objective.

In order to terminate the TCP connection and thus give on-demand migration a more deterministic time, upon resuming in on-demand migration mode, the unikernel at the destination machine periodically spawns a kernel thread that has bigger priority than the application. This thread wakes up in regular intervals and proactively pulls remote memory as shown in Algorithm 1.

To pull remote memory, this thread serially accesses the unmapped memory locations, of each section or heap, pages by pages starting from the start address (*start*) of the section/heap to the end address (*end*), which is rounded to the page boundary by using *page_ceil()* function. To access the page we simply read the first byte of each of them (line 9). These accesses to unmapped memory location results in page faults, and as a consequence pages are pulled from the source machine as described for on-demand migration mechanism. Pages are pulled in batches as described in section 5.3.1. After each remote memory pulling, the thread goes to sleep (line 10) for a certain amount of time (line 1). As this thread has more priority than the application, too-frequent remote memory pulling may result in starvation for the application and degrade the performance. Thus, how often the volunteer thread will do the pulling is made configurable. In order that a system administrator can set a trade-off between the the length of on-demand memory transfer and overhead, the volunteer thread brings it to the application,

To prevent the thread from accessing already mapped memory location and going to sleep afterward, we first check the page table entry to see if the location is already mapped (line 6). If it's already mapped, then we check for the next memory location and continue this until an unmapped memory location is found or all pages are mapped.

5.3.3 Dynamic memory transfer

We can divide the program memory into two parts: static and dynamic. The stack and the heap belong to the dynamic memory, and different sections belong to the static memory. As the transformed stack is dumped into a file and transferred via SCP, only heap contents are left behind in the source machine. Data in the heap are pulled from the source machine on-demand basis by the destination VM. In this section we are going to use the terms heap and dynamic memory interchangeably.

Initially, any access to the heap will result in a page fault because in the destination machine the dynamic memory usage is zero as we boot a fresh unikernel. Here a dynamic memory

page fault can happen for two different reasons. One is for a new memory allocation; the memory allocation function, for example *malloc()*, results a page fault. Another is due to the access to a heap location previously allocated by the source unikernel but not present in destination unikernel because the corresponding remote memory is not pulled yet from the source machine.

A pseudocode for the page fault handler for both dynamic memory and static memory is presented Algorithm 2. To distinguish a dynamic memory page fault from static, we first check the context of the current *task* when there is a page fault. If it is the *heap* and the corresponding virtual address *virAddr* falls between the heap start location *heapStart* and the end location *heapEnd*, then we can make sure the fault is due to a heap access (line 2). Next, we check if the page fault is due to the migration by checking the *migratedHeap* variable which represents how much memory was allocated before the migration. If the access is located between the *heapStart* and *migratedHeap*, we assess how many pages can be batched together with the requested page based on the *virAddr*, page size *pageSize* and *heapEnd* location (line 4). For both x86_64 and AArch64, the page size for heap is 4KB. We usually batch up to 16 pages for heap but it is configurable by changing *MAX_BATCH_SIZE*. If any of the pages in the batch is found already allocated by checking the page table we reduce the batch size to 1.

Next, we allocate memory for the batched pages (line 5), and it gives us the physical address *phyAddr* which indicates the starting location of the batched pages. Then we map the allocated physical memory to the virtual address space (line 6). Finally, we pull the pages from the remote memory (line 7) as described at the beginning of this section. In this case certainly, the type of the memory requested field (MType) for the hypercall is *heap*.

5.3.4 Static memory transfer

Once the unikernel in the source machine transfers the minimal checkpoint (transformed stack and metadata) after receiving migration request, a new unikernel in the destination

Algorithm 2 Page fault handler

```
1: batchSize ← MAX_BATCH_SIZE
2: if task == heap AND (virAddr < heapStart AND virAddr < heapEnd) then
3:    if migratedHeap AND (VirAddr < heapStart + migratedHeap) then
4:       batchSize ← adjust_batchSize(batchSize, heapStart, heapEnd, virAddr)
5:       phyAddr ← allocate_pages(batchSize)
6:       map_phyAddr_to_virAddr(phyAddr, virAddr)
7:       hypercall_for_rem_mem(heap, phyAddr, virAddr, batchSize, pageSize)
8:       return
9:    else
10:      phyAddr ← allocate_pages(1)
11:      map_phyAddr_to_virAddr(phyAddr, virAddr)
12:      return
13:   end if
14: else if (virAddr < sectionStart AND virAddr < sectionEnd) then
15:   batchSize ← adjust_batchSize(batchSize, heapStart, heapEnd, virAddr)
16:   phyAddr ← allocate_pages(batchSize)
17:   map_phyAddr_to_virAddr(phyAddr, virAddr)
18:   hypercall_for_rem_mem(sectionName, phyAddr, virAddr, batchSize, pageSize)
19:   return
20: end if
```

machine is booted for the same application. To prevent the application from executing from the beginning we update the stack with the transformed stack. At the same time, we also need to update the static memory, which is the part of the ELF executable along with the heap.

By using the term static memory, we refer to the application data which belongs to the sections like DATA, BSS, TDATA, TBSS, and RODATA which only belong to the application. If a virtual address of a faulted page falls in to these section's range then we treat it as a page fault due to the static memory.

Unlike dynamic memory, static memory is a part of the executable and already mapped by the unikernel in the destination machine. For example, the BSS section is mapped and by default filled with zeroed values because the binary is loaded before the application is resumed on the destination machine. However, it doesn't represent the actual values of the application's BSS contents as corresponding pages are not restored yet. As a result, for on-demand memory transfer of the static memory, we first need to unmap the virtual addresses so that it results in a page fault when there is an access to the static memory.

HermitCore kernel for x86_64 supports superpages of size 2MB along with traditional pages of 4KB. Superpages help to avoid translation overhead for memory-intensive workloads, and traditional pages support more granular sized page access. In x86_64 the static memory is aligned with superpage boundary and the dynamic memory is aligned with 4KB pages. On the other hand, in HermitCore kernel for AArch64 everything is aligned with 4KB boundary because it only supports traditional 4KB pages. For the static memory in HermitCore, x86_64 and AArch64 use different alignments. For x86_64, its memory is aligned with superpages where for AArch64, it is aligned with 4KB pages. To unmap static memory corresponds to BSS, upon the resume process of on-demand migration we unmap the pages starting from BSS start location to the end. If the target machine is AArch64, then we use the function called page_unmap() to unmap, which takes starting address and number of the pages to be unmapped as input. The number of pages is calculated dividing BSS

section size by page size. For x86_64 we use page_unmap_2m() which only unmaps one page taking the virtual address as input. Thus we use this function iteratively until all BSS pages are unmapped.

We do the same for all other sections, except TDATA and TBSS, where we use the corresponding section's starting location and size. TDATA and TBSS contain data for initialized TLS variables and uninitialized TLS variables, respectively, which are necessary for multithreaded applications. On the other hand, multithreaded applications are out of the scope of this thesis as we only target migrating unikernels running single threaded applications.

As a result, when there is a page fault for an access in the static memory, we batch the pages as described for dynamic memory as shown in line 15 of Alogrithm 2. However, as the static memory page size for x86_64 is 2MB, which is already large enough, we do not do batching when pulling form x86_64 machine. Next, we allocate physical memory for the pages we are expecting to pull from the server (line 16). As we are pulling memory from a different machine, referencing the physical address could be misleading. Thus, we map this memory from physical to virtual address space (line 17). Finally, we make a hypercall regarding the page fault as described in the beginning of this section (line 18). For the requested memory type we use BSS or DATA based on the page fault location.

5.4 Evaluation of on-demand memory transfer

In order to see how checkpoint/restart and on-demand memory transfer perform for the different applications, we run a wide variety of serial benchmarks which represent modern compute-intensive datacenter workloads. These includes macro-benchmarks like the shared-memory MapReduce implementation Phoenix [49], PARSEC [10], and NPB [4, 54] and micro-benchmarks like Linpack [22], Dhrystone [58], and Whetstone [20]. The benchmarks are listed in Table 5.1. We manually trigger migration in full checkpoint mode at half of the execution of each benchmark. Table 5.1 presents the size of the checkpoint for each benchmark.

Table 5.1: Checkpoint sizes at half of each benchmark execution.

Benchmark	Chkpt. size	Benchmark	Chkpt. size
NPB BT (A)	45 MB	NPB UA (A)	37 MB
NPB CG (A)	27 MB	Phoenix Kmeans	5.7 MB
NPB DC (A)	191 MB	Phoenix Matrix Mult.	47 MB
NPB FT (A)	323 MB	Phoenix PCA	32 MB
NPB IS (B)	266 MB	PARSEC Blackscholes (native)	612 MB
NPB LU (A)	40 MB		
NPB MG (A)	453 MB	Linpack	1.5 MB
NPB SP (A)	47 MB	Dhrystone	1.2 MB
NPB EP (A)	11 MB	Whetstone	1.2 MB

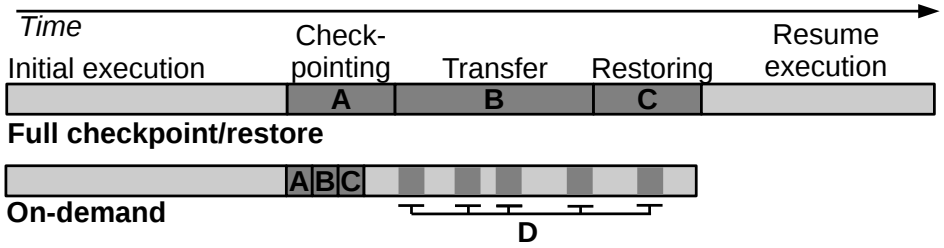
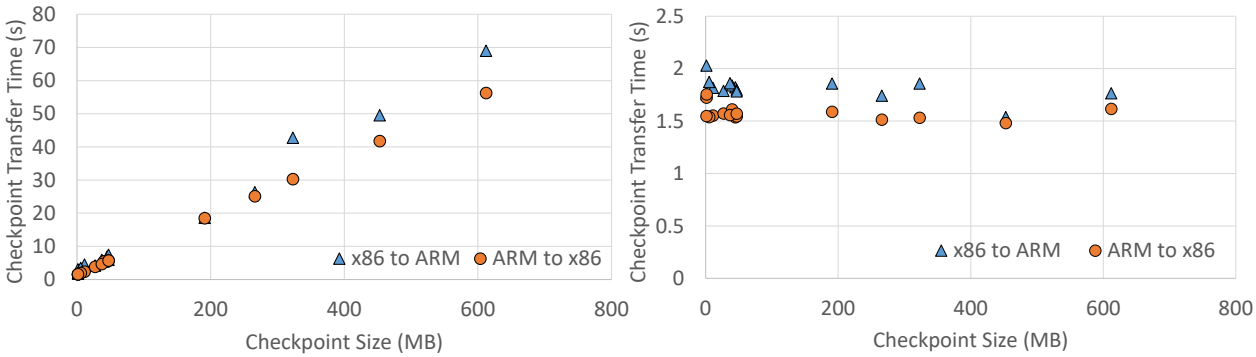


Figure 5.4: Migration overhead illustration.

The migration overhead is constituted of multiple factors, illustrated on Figure5.4. For checkpoint/restart technique the overhead is composed of three factors. The first one is transforming the stack and dumping the checkpoint to the files on the source machine Ⓐ. The second one is transferring these files to the destination machine Ⓑ. The third one is restoring the checkpoint Ⓒ before resuming the execution which includes kernel initialization. These three overheads are also applicable for on-demand migration except for the fact that they are faster as the size of the checkpoint is minimal and thus requires less transfer. However, along with these three overheads, on-demand migration has an additional overhead for serving remote memory page-faults Ⓓ. We compute this time by subtracting from the execution time after the restoring step in on-demand mode the time after the restore step in checkpoint/restart mode.

Our experimental setup is composed of a server and an embedded board connected by Ethernet. The server used for this experiment is an Intel Xeon E5-2637 running Ubuntu 16.04, and the host kernel is Linux 4.4. On the other hand, the embedded board paired with the



(a) Full checkpoint/restart.

(b) On-demand memory transfer.

Figure 5.5: Checkpoint transfer time for different sizes of executables.

server is a credit card sized board called Potato which is also running Ubuntu 16.04, but the host kernel is Linux 4.14. Both of these kernels are compatible with the version of the KVM needed by our hypervisor. The details of the machines used and their characteristics can be found in Table 2.1. The Ethernet speed of the server is 1000Mb/s where for the board it is 100Mb/s. As a result, the communication speed through the Ethernet is limited to 100Mb/s.

Figure 5.5 shows the checkpoint transfer time of different sizes of checkpoints. One can observe that, as Figure 5.5a shows, with the increase of checkpoint size of the benchmarks, for full checkpoint/restart, the checkpoint transfer time keeps linearly increasing. On the other hand, as Figure 5.5b shows, the checkpoint transfer time is almost the same for each benchmark because we transfer minimal checkpoint using on-demand memory transfer method.

To show the experimental results, in terms of overall migration overhead, we selected jobs from the Table 5.1 based on two factors: larger checkpoint size and remote memory usage in the destination machine. We select Blackscholes and DC because they do not require the full remote memory to be pulled in order to finish the execution after resuming from migration. Blackscholes requires 44% of and DC requires 73% of the remote memory. We also select FT and IS as they require the whole remote memory to be pulled for finishing the execution. We migrate the jobs at half of their execution and in both directions: from the server to the board and from the board to the server.

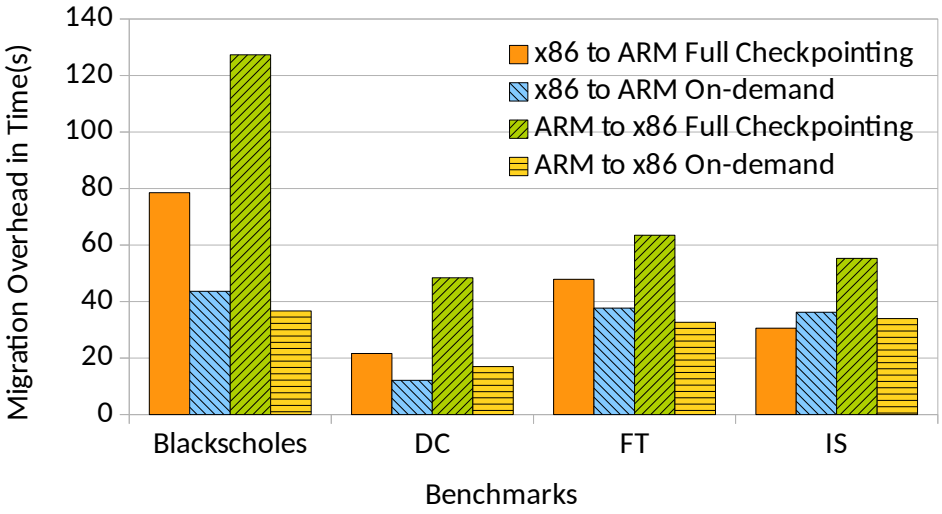


Figure 5.6: Migration overhead comparison between post-copy and checkpoint/restart.

The total migration overhead comparisons are shown in Figure 5.6. There are applications which do not require full remote memory to be transferred to finish the execution after resuming on the destination machine. For these type of applications on-demand memory transfer has significantly less overhead than for the checkpoint/restart. For example, the overhead is reduced by 40% for Blackscholes when migrating from the server to the board and 70% when the other way around. In on-demand memory transfer method, we dump minimal checkpoint to the files, so the storage overhead is reduced significantly. At them same time, as Blackscholes doesn't need the whole remote memory to be pulled, it reduces the network activity. This reduced storage overhead, and the lowered network activity is responsible for the total overhead reduction.

The secondary memory used, an SD card, in the board is slower than the one used in the server. The impact of the slow operation on the SD card becomes visible when the checkpoint size is bigger. For example, checkpointing the 612MB of Blackscholes memory is more than 10 times slower on the board (70s) than on the server (6.3s). When migrating from the board to the server using checkpoint/restart, the checkpoint is large. As a result, when we migrate from the board to the server, checkpoint/restart requires more time than the other way around. This storage overhead is less in case of on-demand mode as here no large checkpoint is taken or restored. This explains why on-demand memory transfer is better

(70% vs. 40%) when migrating from the board to the server.

On-demand memory transfer has less storage overhead even for the applications that require most of the remote memory to finish the execution after migration. For example, IS has 40% improvement and FT has 48% improvement when migrating from the board to the server. When migrating from the server to the board there is no storage overhead. As a result, in this case, if an application requires the whole remote memory to finish the execution after migration, due to the network traffic and page fault management overhead, the on-demand memory transfer doesn't bring significant benefit over checkpoint/restart. For some applications, for example with IS, it can even increase the overhead by a little.

Chapter 6

Conclusions and Future Work

ARM-based single board computers are the lightweight, cheap and low-powered embedded devices compared to other embedded computers of other architectures. On the other hand, x86 based server machines have been dominating the data centers because of their high throughput. But there are difficulties for both types of machines such as servers that are expensive and power-hungry where the embedded boards are low resource platforms. CSWAP proposes a system that aims to benefit the data centers from both conventional x86 servers and lightweight ARM boards by running them side by side.

In a multi-tenant environment like data center, workloads are run in an isolated environment inside VM because of security, resource distribution, load balancing, etc. reasons. Being low resource platform said embedded single board computers are incapable of running full-fledged VM. To solve this problem, we propose to run each workload as a unikernel application which is a lightweight VM that can run inside embedded computers. Moreover, for prototyping, we choose HermitCore for the unikernel which is capable of running native applications utilizing hardware virtualization support.

When the server is overloaded with jobs, we offload some workloads to the embedded boards

by migrating the corresponding VM. Migrating VM from the server to the board is a non-trivial task because of their ISA difference. Moreover, our VM, HermitCore, is not migration-capable. This thesis contributes to this approach by providing support to HermitCore for migration between heterogeneous-ISAs. As a result, it has been possible for HermitCore applications to be entirely offloaded back and forth between the server and the board.

This thesis presents a toolchain capable of building migratable HermitCore applications, for heterogeneous ISAs, from native application sources. This toolchain is composed of a compiler, linker, kernel, C library, state transformation runtime, other different libraries, and tools. The compiler takes an application source, inserts migration points at the function boundaries and create object files. The linker generates ELF executables by linking the object files with the libraries.

This thesis also presents an alignment mechanism so that every memory access before and after migration remains consistent and valid. For aligning the static memory, this thesis discusses about the adaptation of an alignment tool to relocate the symbols present in the generated executables so that each symbol starts from the same address across different ISAs. In chapter 4 this thesis also shows how to modify the OS to align dynamically allocated memories, by the program, across different ISAs.

Transferring VM states from the source machine to the destination machine is an integral part of the VM migration process. VM states are transferred so that the application can resume from the state it was migrated instead of starting from the beginning. In chapter 5 this thesis presents a post-copy live VM migration technique, named on-demand memory transfer, for the transferring of the states of the unikernel between heterogeneous ISAs. For some application on-demand migration technique performs up 70% better compared to another approach called full checkpoint/restart.

In general this thesis discusses necessary steps regarding how to migrate application between the server and the single board computers to utilize the opportunity of heterogeneity in low-cost. This thesis takes a unikernel, HermitCore, and enhances it with migration capabilities

so that applications can be migrated back and forth between different ISAs providing the advantages expected form VMs as well as increasing the throughput while spending less money.

6.1 Limitations

In this thesis, we assume server machines uses x86_64 processors and embedded boards uses AArch64 processors. To make the unikernel migration work between other ISAs, one may need to extend the provided toolchain for those ISAs. For example, HermitCore only supports x86_64 and AArch64 ISAs. To make the HermitCore migration work between other ISAs, first, the corresponding compatible version of HermitCore development is necessary. Moreover, for memory alignment, section alignment constrains needs to be set in the linker script according to the newly considered ISAs. The alignment tool also need slight changes to adapt new ISA.

We also assume that both target and the destination ISAs have the same endianness, have 64-bit addressing, and same size and alignments for primitive C types. To make the migration work between two machines, where these factors are different from one machine to another, along with the presented build infrastructure and the memory alignment technique, one may need to extend the presented VM state transfer method.

6.2 Future Work

The toolchain presented in this thesis for migration capable unikernel binary generation creates two binaries for each application: one for source ISA and another for the destination ISA. Both unikernel binaries are necessary for the stack transformation library to transform the stack from the source ISA to the destination ISA. As our toolchain allows back and forth migration, from the server to the board and vice versa, we need to transfer both unikernel binaries during the migration process. As a result, the stack transformation library can use

them when there is another migration request.

In the future, we aim to create a single binary unikernel for migration between multiple ISAs instead of building one unikernel for each ISA. As a result, we will not need to transfer separate unikernel binaries from the source machine to the destination machine. It will help us reducing multiple binary transfer time and resume the migration process as soon as possible.

For this thesis, we target the best ISA to the date which is x86 for the servers and ARM for the embedded boards. However, there are several other ISAs like the Power architecture, SPARK, Alpha, and RISC-V for server and single computer board. The hermit-popcorn toolchain is capable of building migratable unikernels only for x86 and ARM. In the future, we want to extend this toolchain support for other ISAs so that we can explore the benefit of different architectures and take most advantage from heterogeneity.

Bibliography

[1] 96 Boards. 2018. Hikey (LeMaker). (2018). https://www.96boards.org/product/hikey/ Online, accessed 2018/09/15.

[2] Adam Dunkels. 2018. lwip overview. (2018). https://www.nongnu.org/lwip Online, accessed 2018/10/25.

[3] Maria Avgerinou, Paolo Bertoldi, and Luca Castellazzi. 2017. Trends in Data Centre Energy Consumption under the European Code of Conduct for Data Centre Energy Efficiency. *JRC Technical Reports* (2017).

[4] David H Bailey, Eric Barszcz, John T Barton, David S Browning, Robert L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S Schreiber, and others. 1991. The NAS parallel benchmarks. *The International Journal of Supercomputing Applications* 5, 3 (1991), 63–73.

[5] Antonio Barbalace, Robert Lyerly, Christopher Jelesnianski, Anthony Carno, Ho-Ren Chuang, Vincent Legout, and Binoy Ravindran. 2017. Breaking the boundaries in heterogeneous-ISA datacenters. In *ACM SIGPLAN Notices*. ACM, 645–659.

[6] Antonio Barbalace, Marina Sadini, Saif Ansary, Christopher Jelesnianski, Ashsay Ravichandran, Cagil Kendir, Alastair Murray, and Binoy Ravindran. 2015. Popcorn: bridging the programmability gap in heterogeneous-isa platforms. In *Proceedings of the Tenth European Conference on Computer Systems*. ACM, 29.

[7] BBC. 2015. BBC MicroBit. (2015). http://microbit.org/ Online, accessed 2018/10/25.

[8] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator.. In *USENIX Annual Technical Conference, FREENIX Track*, Vol. 41. 46.

[9] Sharath K Bhat, Ajithchandra Saya, Hemedra K Rawat, Antonio Barbalace, and Binoy Ravindran. 2016. Harnessing energy efficiency of heterogeneous-ISA platforms. *ACM SIGOPS Operating Systems Review* 49, 2 (2016), 65–69.

[10] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. ACM, 72–81.

[11] Boston Limited. 2016. Boston Viridis: presenting the world's first hyperscale server — based on ARM processors. (2016). https://www.boston.co.uk/info/whitepapers/viridiswhitepaper.aspx Online, accessed 2018/09/15.

[12] Sara Bouchenak and Daniel Hagimont. 2002. *Zero Overhead Java Thread Migration*. Research Report RT-0261. INRIA. 33 pages. https://hal.inria.fr/inria-00069913

[13] Alfred Bratterud, Alf-Andre Walla, Hårek Haugerud, Paal E Engelstad, and Kyrre Begnum. 2015. IncludeOS: A minimal, resource efficient unikernel for cloud services. In *Proceedings of the 7th IEEE International Conference on Cloud Computing Technology and Science (CloudCom 2015)*. IEEE, 250–257.

[14] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. 2011. Clonecloud: elastic execution between mobile device and cloud. In *Proceedings of the sixth conference on Computer systems*. ACM, 301–314.

[15] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. 2005. Live Migration of Virtual Machines. In *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design*

& Implementation - Volume 2 (NSDI'05). USENIX Association, Berkeley, CA, USA, 273–286. http://dl.acm.org/citation.cfm?id=1251203.1251223

[16] CMake 2018. CMake Website. (2018). https://cmake.org/. Online, accessed 10/21/2017.

[17] Colfax. 2018. Colfax CX1120s-X6 1U Rackmount Server. (2018). http://www.colfax-intl.com/nd/Servers/CX1120s-X6.aspx Online, accessed 09/15/2018.

[18] Intel Corp. 2018. Intel Clear Containers. (2018). https://clearlinux.org/documentation/clear-containers. Online, accessed 08/04/2018.

[19] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. 2010. MAUI: Making Smartphones Last Longer with Code Offload. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services (MobiSys '10)*. ACM, New York, NY, USA, 49–62. DOI:http://dx.doi.org/10.1145/1814433.1814441

[20] Harold J Curnow and Brian A. Wichmann. 1976. A synthetic benchmark. *Comput. J.* 19, 1 (1976), 43–49.

[21] Matthew DeVuyst, Ashish Venkat, and Dean M Tullsen. 2012. Execution migration in a heterogeneous-ISA chip multiprocessor. In *ACM SIGARCH Computer Architecture News*. ACM, 261–272.

[22] Jack J Dongarra, Piotr Luszczek, and Antoine Petitet. 2003. The LINPACK benchmark: past, present and future. *Concurrency and Computation: practice and experience* 15, 9 (2003), 803–820.

[23] Chris Edwards. 2013. Not-so-humble raspberry pi gets big ideas. *Engineering & Technology* 8 (April 2013), 30–33. Issue 3.

[24] Joachim Gehweiler and Michael Thies. 2010. Thread migration and checkpointing in java. *Heinz Nixdorf Institute, Tech. Rep. tr-ri-10* 315 (2010).

[25] Nicolas Geoffray, Gaël Thomas, and Bertil Folliot. 2006. Live and heterogeneous migration of execution environments. In *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*. Springer, 1254–1263.

[26] GNU. 2018. GNU binutils. (2018). http://sourceware.org/binutils/ Online, accessed 2018/10/24.

[27] Mark S Gordon, Davoud Anoushe Jamshidi, Scott A Mahlke, Zhuoqing Morley Mao, and Xu Chen. 2012. COMET: Code Offload by Migrating Execution Transparently.. In *OSDI*, Vol. 12. 93–106.

[28] Hewlett Packard. 2018. HPE Moonshot System. (2018). https://www.hpe.com/us/en/servers/moonshot.html Online, accessed 2018/09/20.

[29] Michael R. Hines, Umesh Deshpande, and Kartik Gopalan. 2009. Post-copy Live Migration of Virtual Machines. *SIGOPS Oper. Syst. Rev.* 43, 3 (July 2009), 14–26. DOI: http://dx.doi.org/10.1145/1618525.1618528

[30] Antti Kantee and Justin Cormack. 2014. Rump Kernels No OS? No Problem! *USENIX; login: magazine* (2014).

[31] Kimberly Keeton. 2015. The machine: An architecture for memory-centric computing. In *Workshop on Runtime and Operating Systems for Supercomputers (ROSS)*.

[32] Avi Kivity, Dor Laor Glauber Costa, and Pekka Enberg. 2014. OS v - Optimizing the Operating System for Virtual Machines. In *Proceedings of the 2014 USENIX Annual Technical Conference (ATC'14)*. 61.

[33] Andrew Krioukov, Prashanth Mohan, Sara Alspaugh, Laura Keys, David Culler, and Randy Katz. 2011. Napsac: Design and implementation of a power-proportional web cluster. *ACM SIGCOMM computer communication review* 41, 1 (2011), 102–108.

[34] Willis Lang, Jignesh M. Patel, and Srinath Shankar. 2010. Wimpy Node Clusters: What About Non-wimpy Workloads?. In *Proceedings of the Sixth International Workshop on Data Management on New Hardware (DaMoN '10)*. ACM, New York, NY, USA, 47–55. DOI:http://dx.doi.org/10.1145/1869389.1869396

[35] Stefan Lankes. 2018. HermitCore Toolchain. (2018). https://github.com/hermitcore Online, accessed 10/04/2018.

[36] Stefan Lankes, Simon Pickartz, and Jens Breitbart. 2016. HermitCore: a unikernel for extreme scale computing. In *Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers (ROSS 2016)*. ACM.

[37] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, 75.

[38] Gwangmu Lee, Hyunjoon Park, Seonyeong Heo, Kyung-Ah Chang, Hyogun Lee, and Hanjun Kim. 2015. Architecture-aware automatic computation offload for native applications. In *Proceedings of the 48th international symposium on microarchitecture*. ACM, 521–532.

[39] Libre Computer. 2018. AML-S905X-CC (Le Potato). (2018). https://libre.computer/products/boards/aml-s905x-cc/ Online, accessed 2018/09/15.

[40] A Madhavapeddy, R Mortier, C Rotsos, DJ Scott, B Singh, T Gazagnaire, S Smith, S Hand, and J Crowcroft. 2013. Unikernels: library operating systems for the cloud.. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'13)*. ACM, 461–472.

[41] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. 2017. My VM is Lighter (and Safer) Than Your Container. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. ACM, New York, NY, USA, 218–233. DOI:http://dx.doi.org/10.1145/3132747.3132763

[42] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. 2014. ClickOS and the Art of Network Function Virtualization. In *Proceedings of the 11th USENIX Conference on Networked Systems*

Design and Implementation (NSDI'14). USENIX Association, Berkeley, CA, USA, 459–473. http://dl.acm.org/citation.cfm?id=2616448.2616491

[43] Dirk Merkel. 2014. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal* 2014, 239 (2014), 2.

[44] muls 2017. musl Website. (2017). https://www.musl-libc.org/. Online, accessed 10/05/2018.

[45] Michael Nelson, Beng-Hong Lim, and Greg Hutchins. 2005. Fast Transparent Migration for Virtual Machines. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC '05)*. USENIX Association, Berkeley, CA, USA, 25–25. http://dl.acm.org/citation.cfm?id=1247360.1247385

[46] Newlib 2017. Newlib Website. (2017). https://sourceware.org/newlib/. Online, accessed 12/12/2017.

[47] Eunbyung Park, Bernhard Egger, and Jaejin Lee. 2011. Fast and Space-efficient Virtual Machine Checkpointing. *SIGPLAN Not.* 46, 7 (March 2011), 75–86. DOI:http://dx.doi.org/10.1145/2007477.1952694

[48] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. 2011. Rethinking the Library OS from the Top Down. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. ACM, New York, NY, USA, 291–304. DOI: http://dx.doi.org/10.1145/1950365.1950399

[49] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. 2007. Evaluating mapreduce for multi-core and multiprocessor systems. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*. Ieee, 13–24.

[50] Huigui Rong, Haomin Zhang, Sheng Xiao, Canbing Li, and Chunhua Hu. 2016. Optimizing energy consumption for data centers. *Renewable and Sustainable Energy Reviews* 58 (May 2016), 674–691.

[51] RPi Foundataion. 2018. Raspberry Pi 3 Model B. (2018). https://www.raspberrypi.org/products/raspberry-pi-3-model-b/ Online, accessed 09/15/2018.

[52] Takahiro Sakamoto, Tatsurou Sekiguchi, and Akinori Yonezawa. 2000. Bytecode Transformation for Portable Thread Migration in Java. In *Agent Systems, Mobile Agents, and Applications*, David Kotz and Friedemann Mattern (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 16–28.

[53] Scaleway. 2018. Scaleway Cloud Pricing. (2018). https://www.scaleway.com/pricing/ Online, accessed 2018/09/15.

[54] Sangmin Seo, Gangwon Jo, and Jaejin Lee. 2011. Performance characterization of the NAS Parallel Benchmarks in OpenCL. In *IEEE International Symposium on Workload Characterization (IISWC 2011)*. IEEE, 137–148.

[55] Eddy Truyen, Bert Robben, Bart Vanhaute, Tim Coninx, Wouter Joosen, and Pierre Verbaeten. 2000. Portable Support for Transparent Thread Migration in Java. In *Proceedings of the Second International Symposium on Agent Systems and Applications and Fourth International Symposium on Mobile Agents (ASA/MA 2000)*. Springer-Verlag, Berlin, Heidelberg, 29–43. http://dl.acm.org/citation.cfm?id=647629.732581

[56] Vijay Vasudevan, David Andersen, Michael Kaminsky, Lawrence Tan, Jason Franklin, and Iulian Moraru. 2010. Energy-efficient cluster computing with FAWN: Workloads and implications. In *Proceedings of the 1st International Conference on Energy-Efficient Computing and Networking*. ACM, 195–204.

[57] Ashish Venkat and Dean M Tullsen. 2014. Harnessing ISA diversity: Design of a heterogeneous-ISA chip multiprocessor. *ACM SIGARCH Computer Architecture News* 42, 3 (2014), 121–132.

[58] Reinhold P Weicker. 1984. Dhrystone: a synthetic systems programming benchmark. *Commun. ACM* 27, 10 (1984), 1013–1030.

[59] Dan Williams and Ricardo Koller. 2016. Unikernel Monitors: Extending Minimalism Outside of the Box. *8th USENIX Workshop on Hot Topics in Cloud Comput-*

ing (2016). https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/williams

[60] Yiming Zhang, Jon Crowcroft, Dongsheng Li, Chengfen Zhang, Huiba Li, Yaozheng Wang, Kai Yu, Yongqiang Xiong, and Guihai Chen. 2018. KylinX: A Dynamic Library Operating System for Simplified and Efficient Cloud Virtualization. In *Proceedings of the 2018 USENIX Annual Technical Conference*.

[61] Wenzhang Zhu, Cho-Li Wang, and F. C. M. Lau. 2002. JESSICA2: a distributed Java Virtual Machine with transparent thread migration support. In *Proceedings. IEEE International Conference on Cluster Computing*. 381–388.