Edge Computing – the Case for Heterogeneous-ISA Container Migration

Antonio Barbalace, Mohamed L. Karaoui, Wei Wang, Tong Xing, Pierre Olivier, Binoy Ravindran*
University of Edinburgh, Virginia Tech, Stevens, Stevens, University of Manchester, Virginia Tech
abarbala@ed.ac.uk, karaoui@vt.edu, wwang88@stevens.edu, txing1@stevens.edu, pierre.olivier@manchester.ac.uk, binoy@vt.edu

Abstract

Edge computing is a recent computing paradigm that brings cloud services closer to the client. Among other features, edge computing offers extremely low client/server latencies. To consistently provide such low latencies, services need to run on edge nodes that are physically as close as possible to their clients. Thus, when a client changes its physical location, a service should migrate between edge nodes to maintain proximity. Differently from cloud nodes, edge nodes are built with CPUs of different Instruction Set Architectures (ISAs), hence a server program natively compiled for one ISA cannot migrate to another. This hinders migration to the closest node.

We introduce H-Container, which migrates natively-compiled containerized applications across compute nodes featuring CPUs of different ISAs. H-Container advances over existing heterogeneous-ISA migration systems by being a) highly compatible – no source code nor compiler toolchain modifications are needed; b) easily deployable – fully implemented in user space, thus without any OS or hypervisor dependency, and c) largely Linux compliant – can migrate most Linux software, including server applications and dynamically linked binaries. H-Container targets Linux, adopts LLVM, extends CRIU, and integrates with Docker. Experiments demonstrate that H-Container adds no overhead on average during program execution, while between 10*ms* and 100*ms* are added during migration. Furthermore, we show the benefits of H-Container in real scenarios, proving for example up to 94% increase in Redis throughput when unlocking heterogeneity.

CCS Concepts: • **Computer systems organization → Heterogeneous (hybrid) systems**; • **Software and its engineering → Operating systems**.

Keywords: Edge, Heterogeneous ISA, Containers, Migration

ACM Reference Format:
Antonio Barbalace, Mohamed L. Karaoui, Wei Wang, Tong Xing, Pierre Olivier, Binoy Ravindran. 2020. Edge Computing – the Case for Heterogeneous-ISA Container Migration. In *16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '20), March 17, 2020, Lausanne, Switzerland*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3381052.3381321

1 Introduction

Edge computing [73, 75] is an emerging computing paradigm that advocates moving computations typically performed by centralized cloud computing services onto distributed compute nodes physically closer to the end user or data production source – at the edge. In addition to relieving the data center from compute load, and the network from data traffic, edge servers' physical proximity to clients ensures the lowest latencies. Thus, edge computing's application domains are plentiful: mobile computation offload, failure resilience, IoT privacy [35, 73], real-time analytics [30, 57], cognitive assistance [74], just-in-time instantiation [59], gaming [14, 27], etc.

In this context, the need for runtime software migration between machines has been identified as a critical feature [30, 35, 57, 61, 85]. Software applications may need to migrate between edge nodes for numerous reasons: following a mobile user to maintain low-latency, offloading congested devices, proactively leaving a node that may fail in a near future, etc. Although stateless applications can be easily restarted between nodes, stateful ones generally cannot – they must implement persistence support (cf. Redis), which requires (costly) application-specific software rewriting, thus migration is a better option as it support any application. Moreover, studies have shown migration to be faster than restarting [50, 68].

Differently from the cloud, computers at the edge are wildly heterogeneous [31, 41, 51, 85], ranging from micro/nano/pico clusters [22, 67, 87] to routing devices [59, 62, 82]. They include from server to embedded computers with CPUs of diverse Instruction Set Architectures (ISAs): not only x86 [42], but also ARM [38, 49, 52], with other ISAs announced [37, 77, 91]. This heterogeneity was recognized by major software players, such as Docker, which now support multi-ISA deployments [4, 63, 63, 66]. The ISA-heterogeneity is a colossal barrier to agile migration of software at the edge: a (stateful) service natively compiled for an ISA is unable to migrate to a machine of a different ISA. The intuition behind this paper is that *enabling heterogeneous-ISA migration maximizes the number of potential migration targets and increases the chances of optimal placement for latency-sensitive services.*

*A. Barbalace, M. L. Karaoui, W. Wang equally contributed to the work. A. Barbalace performed part of this work when at Stevens Institue of Technology.

VEE '20, March 17, 2020, Lausanne, Switzerland

With these prospects, computing resources at an edge node are rather constrained compared to the abundance in the cloud. Furthermore, similarly to the cloud, the edge is also multi-tenant. Therefore, when deploying services at the edge, a form of virtualization is necessary. However, in such context a lightweight virtualization technology is more compelling than traditional Virtual Machines (VM) [35]. Containers are a form of lightweight OS-level virtualization, offering near-native performance [12], fast invocation latencies and low memory footprints. Because of these characteristics, they are increasingly popular at the edge [21, 36, 57, 81, 93].

Containers can migrate at runtime across homogeneous nodes in production environments [28], but not on heterogeneous-ISA ones – a requirement of the edge. In fact, runtime cross-ISA migration has been studied [6, 15, 20, 23, 32, 34, 48, 64, 86] for OS-level processes [6], unikernel virtual machines [64], or Java applications [32]. Unfortunately, these works suffer from fundamental flaws making them unlikely to be used in production. First, they require access to the application's sources which is not acceptable in many scenarios, e.g., when using proprietary software. Second, they rely on complex and experimental systems software demanding the install of either a custom kernel [6], hypervisor [64], or language VM [32]. These are only compatible with a handful of machines and unlikely provide stability and security. Third, they implement application's state transfer techniques that may not handle all application's residual dependencies [16] such as socket descriptors – thus, they may not support applications such as servers. Finally, dynamically linked binaries, more widespread than static ones [84], are not supported.

H-Container. This paper focuses on maximizing the performance of latency-sensitive stateful services at the edge by proposing H-Container, an easily-deployable system enabling the migration of containers between machines of different ISAs to enhance the flexibility of edge applications. Focusing on the popular x86-64 and arm64 ISAs, we enhance the Linux's Checkpoint Restart In User space (CRIU) tool, used as the underlying mechanism behind Docker containers' migration technology, with cross-ISA transformation mechanisms. We integrate H-Container into Docker, allowing the migration of containers between hosts of different ISAs. In addition to being the standard tool container migration is built upon, CRIU is the de facto software used to capture a process state and in particular the kernel part of that state. This includes among other things socket descriptors and network stack state, enabling support for server applications. Finally, H-Container support migrating across ISAs applications that were originally dynamically compiled.

Contrary to existing work on heterogeneous-ISA migration, H-Container is easily deployable: it leverages Intermediate Representation (IR) lifting tools [25] to instrument an application with the required metadata allowing cross-ISA migration, thus we do not require access to the edge application's source code. The modifications to stock CRIU are minimal and most of the migration processing is realized by external user-space tools we developed. We successfully tested H-Container on numerous machines including x86-64 and arm64 servers, embedded systems, and AWS EC2 instances. We evaluate and demonstrate the benefits of H-Container. For example we show that in latency-sensitive scenarios, migrating a RedisEdge instance between edge nodes of different ISAs can yield a 23% to 94% throughput increase compared to scenarios where migration is not possible due to the ISA difference. The paper makes the following contributions:

- The *design* of H-Container, a highly compatible, easily deployable, and largely Linux compliant system for container migration between heterogeneous-ISAs computer nodes. It introduces a new deployment model featuring cloud software repositories storing IR binaries (vs. native);
- The *implementation* of H-Container on top of the CRIU checkpointing tool and IR lifting software to instrument an applications for cross-ISA migration without source code;
- The overhead and performance *evaluation* of H-Container.

This paper is organized as follows. Section 2 presents our motivations, and Section 3 lays out background information about software migration and executable binary transformation. We state our system model in Section 4. The design principles of H-Container are presented in Section 5, the implementation in Section 6. We evaluate H-Container in Section 7 and present related works in Section 8. Section 9 concludes.

2 Motivation

Edge data centers distinguish from cloud ones for the remarked heterogeneity in terms of computing hardware, resources, form factors, processor classes and ISAs [31, 41, 51, 85]. If home routers are ARM-based, enterprise ones are x86-powered. While even if micro/nano clusters [22, 87] are mostly equipped with x86 servers, ARM servers for the edge are emerging [38, 49, 52] – ARM pico clusters exist already [67]. Such heterogeneity makes that in many scenarios, for a given application, the closest node to the end-user (say, an ARM home router) has a different ISA from the machine the application currently runs on (e.g., an x86-64 server). Sticking to the same ISA for the target machine is thus sub-optimal towards the objective of reducing latency.

Why bringing services as-close-as-possible to the end-user? We demonstrate that even small changes in latency can critically impact the performance of applications using edge services. We use the Redis server, a prime example of edge application [70], serving small key/value pairs (a few bytes) as an example of latency-sensitive application. We ran this server on two machines representing edge nodes, an Intel x86-64 2.5GHz Atom C2758 microserver and a librecomputer LePotato arm64 1.5GHz single board computer. A third machine, the edge client, is connected to both edge nodes on the same LAN. The base latency between the client and the edge nodes

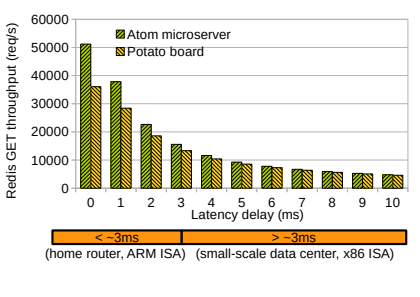

Figure 1. Redis throughput when varying client-server network latency.

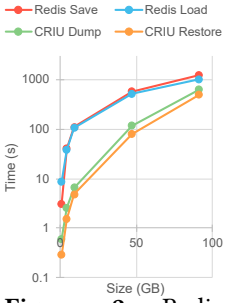

Figure 2. Redis persistence vs CRIU.

is less than 500μs on such a setup. We use Linux's Traffic Control, `tc`, to artificially increase the network latency of the NIC on each edge node by an additional delay from 1 up to 10 ms. Such numbers correspond to the typical latencies expected at the edge [14]. The edge client runs the Redis benchmark.

Results are presented in Figure 1, showing GET throughput as a function of the artificial delay added to the latency, for both machines. Clearly, even a slight increase in latency can significantly bring the performance down. For example, when the Atom goes from 1*ms* to 2*ms* latency it losses more than one third of the throughput. Moreover, the experiment becomes very soon bounded by the latency and the difference in performance between the x86-64 microserver (2.5 GHz) and arm64 board (1.5 GHz) is less than 10% starting at 4*ms* and above. Thus, in latency-sensitive scenarios, when the application does not require a significant computing power, it is worth migrating it to low-specs platforms, such as a home router [59].

At the same time, latency-sensitive server applications on the edge migrate between edge nodes to follow an end-user [30, 35, 57, 61, 85]. Thus, it is fundamental to maximize the number of edge nodes target of migration so that a server can be as close as possible to the end-user – thus, the performance can be maximized. Unfortunately, when dealing with (stateful) natively-compiled applications edge's ISA heterogeneity reduces the number of edge nodes targets for migration; thus, making heterogeneous-ISA migration an appealing feature.

Generic migration vs. ad-hoc state transfer. Applications running at the edge are a mix of stateless and stateful one. The former, which include data filtering, triggering, etc. can simply be stopped on one node and restarted on another one, thus migration is not strictly needed. However, stopping a stateful service means losing all its data if it doesn't implement persistency. Only a few services do, because supporting this feature require a non-negligible, application-specific, effort. Thus, an application-agnostic migration mechanism is preferable. Additionally, we observed that generic migration, especially live migration, can be superior to ad-hoc persistency implementation. Figure 2 compares memory-to/from-disk persistence for Redis vs. CRIU without compression, for different DB sizes. CRIU is up to 25 times faster, and when enabling compression CRIU is still faster – this is because of its trivial serialization. To conclude, migration can be faster than ad-hoc state transfer mechanisms, in addition to being more generic and having shorter downtimes.

3 Background

3.1 Software Migration During Execution

The need to migrate software in execution between different computers dates back to the first multi-computer network. Software has been migrated at the granularity of threads [6, 8], processes [6, 8, 44], groups of processes [65], or entire operating systems (OS) [16]. Independently of the granularity, the main idea beyond migration is that the entire state of the software is moved between computers.

Migrating an OS with its running applications is a well known technology. It relies on a virtual model of the hardware, the Virtual Machine (VM), and transfer its state between computers. Among other components, it includes the content of the VM's physical RAM that incorporates the OS and applications state. On the other hand, process-level migration identifies and transfers only the state related to a specific program. It includes its address space, CPU registers, and part of the OS state related to the process – e.g., open file descriptors. Container migration applies process-level migration to the group of processes populating a container [28].

Migration mechanisms include checkpoint/restore [28], in which a full dump of the software state is created and then restored, and live migration [16], in which a minimal state dump is created, moved to the target machine, and immediately restored. The rest of the state is either proactively transferred before the minimal dump [53] (pre-copy) or transferred on-demand after the restore phase [40] (post-copy).

Homogeneous-ISA Migration. In systems built with machines with processors of the same ISA, both VM and processes/containers migration have been implemented. VM migration is a well-mastered technology, currently implemented by multiple commercial and open-source solutions. Although process migration has been studied for a long time, it was never fully deployed at scale in production due to the difficulty of managing residual dependencies [16] – i.e., the part of the OS kernel state related to the process in question. In recent years, due to the success of containers, process migration re-gained popularity. In Linux, it is implemented by the Checkpoint Restore In User-space (CRIU) tool [28].

Figure 3 illustrates the steps involved in CRIU process migration. *Checkpoint* produces a bunch of image files that are transferred to the destination machine and used by *restore*. In order to produce such files, CRIU has first to stop the process in a consistent state. It does so by using the *compel* library, which "infects" the target process with external code that snapshots its resources. CRIU comes as a single binary application, `criu`, which can be used to checkpoint – dump the state, and to restore – reload the state dump. Moreover, CRIU includes tools to manipulate the state dump, including `crit`. Finally, because CRIU is a user-space tool that just checkpoints and restores

an application, a framework to coordinate applications deployment and migration among computers is usually needed. In many cases today, Docker [9] is adopted for this purpose.

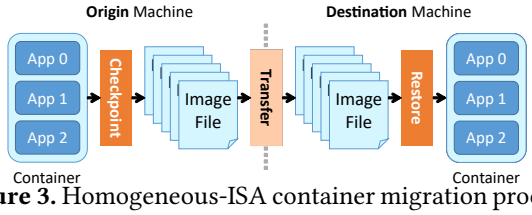

Figure 3. Homogeneous-ISA container migration process.

Heterogeneous-ISA Migration. In the 80/90's, multiple projects studied the migration of applications in a network of heterogeneous machines [3, 44, 78]. Such works used to convert the entire migrated state of the application, including data, from one ISA format to another, thus involving large overheads. Recent works, such as HSA [72], Venkat et al. [86], or Popcorn Linux [6], improved over the state-of-the-art by setting a common data format, therefore reducing the amount of state that has to be converted. Because HSA focuses on platforms with CPU and accelerators we will not discuss it further.

Both Popcorn Linux and the work from Venkat et al. propose the idea of uniform address space layout and common data format among general-purpose CPUs of different ISAs. This implies that the ISAs considered support the same primitive data types' sizes and alignment constraints, and eventually endianess. Moreover, to preserve the validity of pointers across migration, every function, variable, or other program symbol, should be at the same virtual address for each ISA. Because the same machine code cannot execute on processors of different ISAs, every program function or procedure is compiled into the machine code of every ISA. The same function/procedure compiled for different ISAs lives at the same virtual address, therefore there is a `.text` section per ISA and those are overlapping in the virtual address space. Because of the uniform address space layout and data format, migrating a thread or process between ISAs becomes nearly as easy as migrating a thread among different CPUs on a SMP machine – where there is no state transformation.

Because CPUs of different ISAs have different register sets, state transformation cannot be completely avoided. Thus, previous works convert the registers state between architectures. Popcorn Linux achieves this in kernel space [6]. Moreover, migrating between CPUs of different ISAs may only happen at so called migration points. Migration points are machine instructions at which it is valid to migrate between different ISAs: the architecture-specific state (e.g., registers) can be transformed between ISAs because a transformation function exists. Finally, other than the registers state, Popcorn Linux keeps each thread's stack in the ISA's native format, which is architecture dependent. Upon migration the stack's state is converted in addition to the registers' one. The code to convert the stack is injected by the Popcorn Linux compiler.

3.2 Static Executable Binary Transformation

Earlier works on static executable binary transformation (or transpilers) between ISAs date back to the 90's. Latest works on static binary analysis in the security community, together with innovations in compilers [47] rejuvenate the topic. In this paper, executable binaries are programs running atop an OS.

The first step in executable binary transformation is the decompilation. Decompilation takes the executable binary and outputs its assembly code – this is far from being trivial [79, 89] because code and data can be intermixed. After the code has been decompiled, assuming the new executable binary will run on the same OS as the original one, it is necessary to map equivalent machine code instructions, or blocks of them, between the two ISAs. The two ISAs may have different register sets or instructions for which there is no equivalence. For those, software helper functions have to be provided. With all these in place a new executable binary can be produced.

Among others, McSema/Remill [25] (simply, McSema) is a recent software framework for binary analysis and transformation. This tool advances the state-of-the-art by decompiling the executable binary into its native assembly language, and then "lifting" the assembly language into LLVM IR, which is more feature rich and allows for reasoning about what the program is actually doing (cf. symbolic execution [13]). When the application is translated into LLVM IR, by virtue of the fact that LLVM is a cross compiler, the application can be transpiled into any ISA supported by LLVM.

4 System Model

We consider a cloud and edge reference system model as depicted in Figure 4. A team of developers implement a client-server application. The server part of the application is deployed on the cloud or edge, while the client part is installed on user's (mobile) computing device(s). The client-server application may execute on multiple servers, but at least one server should run as close as possible to the client part of the application, i.e., on the edge. In this paper we focus on applications featuring only a server running on the edge – not on the cloud.

The model includes a cloud repository of applications residing in a cloud data center (e.g., Docker Hub), and the edge built by multiple edge segments, each maintaining a local applications repository (e.g., Docker cache). This paper focuses on a single edge segment. We assume that the server part of the application is deployed as a container and it is stored in the cloud repository. We also assume that when a client appears on the edge, the server application is copied into an edge-local repository; a software manager (e.g., Docker [9]) and an orchestrator (e.g., Kubernetes [39], OpenShift [76]) administer the edge-local repository and are responsible of the deployment of the server on each different edge node.

Client applications may be shipped from the developers to the users in any form, e.g., downloaded from a website, or a marketplace, this is out of the scope of this paper.

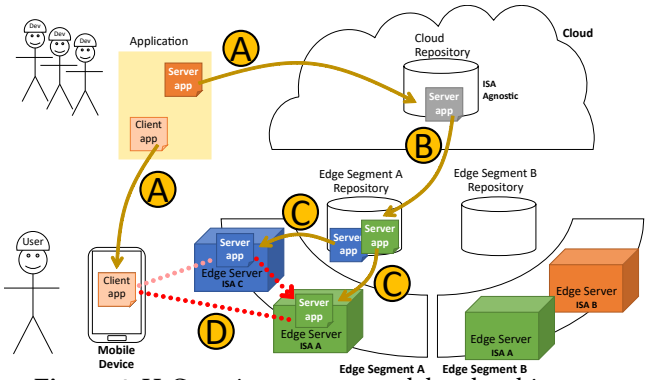

Figure 4. H-Container system model and architecture

5 Design Principles and Architecture

Design Principles. To fulfill the pressing demands for flexibility and agility at the edge, this work is based on the following design principles: 1) enabling application software to transparently execute and migrate across edge nodes of heterogeneous ISAs; 2) offering a high-degree of portability between edge machines, and genericity in terms of supported software; 3) being easy to deploy, maintain, and manage; 4) being of minimal performance overhead.

Therefore, we designed the H-Container architecture that: a) is fully implemented in user-space to maximize portability, ease of maintenance and management, and avoids relying on specific kernel versions or patches; b) targets operating system-level virtualization – i.e., containers, jails, zones, which is acclaimed to be more lightweight than virtual machines, hence with minimal overhead; c) does not require access to the application's source code – binaries are automatically re-purposed for migration among ISA different nodes when moved to each edge segment, thus generic and easy to deploy; d) minimizes application's state transformation when runtime migrate between different ISA processors in order to provide a quick and efficient migration mechanism.

Architecture. The proposed architecture builds atop the system model presented above and enables (server-like) applications migration across heterogeneous ISA processor nodes on the edge. Specifically, our architecture enables the following deployment scenario, with reference to Figure 4:

1. developers initially upload their edge applications on a cloud repository Ⓐ, which stores them in LLVM IR. Developers may upload executable binaries natively compiled or in LLVM IR. Native binaries are transformed into IR;
2. when the application has to be deployed, the local-edge segment repository pulls the IRs from the repository Ⓑ and compiles them into native executable binaries able to migrate across all edge nodes of diverse ISAs in its edge segment;
3. once a server application is running on one node of an edge segment Ⓒ and the user is moving towards another node of the same segment, the server application, running in a container, is frozen, dumped for migration (based on checkpoint/restart or live migration), and its state sent to the node that is closer to the user Ⓓ;
4. when the server application state is received on the destination node, it is eventually converted to the ISA of the receiving machine, if not converted before, and the container is restored to continue execution.

The architecture supports both checkpoint/restore-migration and live-migration, it is the edge segment orchestrator that decides what migration, when and where to migrate.

H-Container introduces two key techniques to implement such architecture: *automatic executable binary transformation into an executable binary that can migrate among different ISA processors at runtime*, and *container migration across diverse ISA processors in user-space.*

5.1 Automatic Executable Binary Transformation

Previous approaches to software migration among heterogeneous ISA CPUs require the source code and the knowledge of what CPU ISAs it will run on. Knowing what ISAs exist on a single computer is trivial, but it may become a problem on the edge, which is not under the control of the developer. Although today it is feasible to compile an application for all existent ISAs, because the number of ISAs is limited, the fact that open-source processors with customizable-ISA are having enormous success [90] questions such solution. Moreover, this is not practical for legacy applications for which the source code may not exist anymore, or the investment to recompile with modern toolchains and libraries is too high. Finally, compiling from sources is not always an easy process, as it requires the usage of a specific toolchain and the availability of all libraries of the right version, etc.

Our architecture (Figure 4) stores applications in ISA agnostic form, LLVM IR, but does not force application developers to compile applications in LLVM IR. Therefore, we provide the possibility to upload a natively compiled binary, and H-Container will transform it into LLVM IR. This is depicted in Figure 5.a. Note that LLVM IR is also a good compromise because it doesn't require a company to expose its own code, which is intellectual property, to cloud/edge providers.

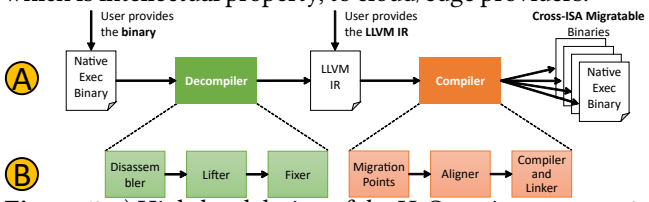

Figure 5. a) High-level design of the H-Container *automatic executable transformation* infrastructure; b) breakdown of the decompiler and compiler blocks.

Transforming the code from native to LLVM IR (Decompiler) is the first step of our automatic executable binary transformation. The second step in Figure 5.a is to compile (Compiler) the code from LLVM IR into a set of different native executable binaries for different ISA processors. For a single program, all executable binaries comply to a unique address space layout, thread local storage, same data format, padding, etc.

Only the stack layout is different. Additionally, the compiler automatically inserts migration points at function boundaries, and potentially outlines part of large functions to add more migration points. This is very similar to Popcorn Linux [6]: the entire address space has the same layout on multiple ISAs, all code sections of ISA A are overlayed with the ones from ISA B and code are functionally equivalent (produce the same output with the same input); because **all** symbols are at the same address (independently if functions or variables), and have the same size and padding executing code from ISA A or ISA B produces the same result. However, because the stack format is kept different for performance, it has to be rewritten at runtime. H-Container user-space runtime rewrites the stack right after a migration request. This rewriting process leverages metadata that were inserted by the compiler within custom ELF sections of the binary. These metadata help to produce a mapping of the stack state between ISAs of the live values residing on the stack at each migration point.

5.2 Heterogeneous Migration of Containers

The heterogeneous migration of containers is designed to capitalize on current container migration infrastructure available in modern OSes, to avoid building just another research toy. Applications running in a container to be migrated across heterogeneous ISA nodes have to be migratable, i.e., compiled as explained in the previous Section.

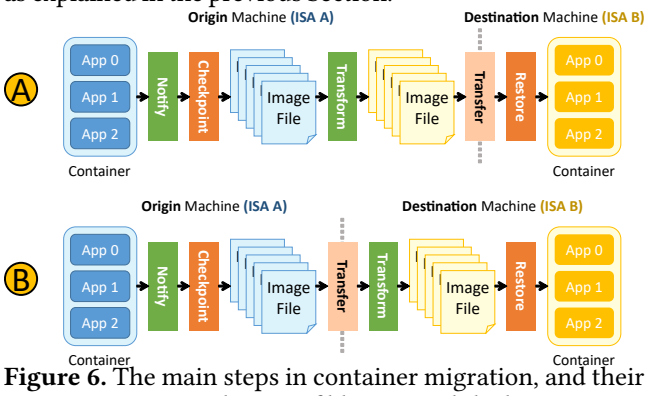

Figure 6. The main steps in container migration, and their outputs. Origin machine is of blue ISA, while destination is yellow. Green boxes are added by H-Container.

All steps required by heterogeneous container migration are depicted in Figure 6. The first step in heterogeneous container migration is to "Notify" any application in the container that it has to stop – this is alike classic container migration on homogeneous CPUs. However, heterogeneous-ISA migration requires that each thread of every application stops *at a valid migration point*, while migration among homogeneous CPUs can happen at any point (cf. Section 3).

When all threads of every application reach a valid migration point, they are frozen, and H-Container takes a checkpoint of the container state – possibly using existent tools. The entire container state is dumped into "Image Files". Differently from migration among homogeneous CPUs, heterogeneous migration requires the Image Files to be transformed from the origin ISA to the destination ISA. In the "Transform" step H-Container rewrites each application dump so that the destination machine thinks that the machine that was executing the code before is of the exact same ISA. To minimize the overhead of "Transform", this work implements a single address space layout, however each thread's stack is kept in the machine native format. Hence, "Transform" also requires rewriting each thread's stack in the application dump.

"Transform" produces a new version of the "Image Files". This version is sent to the destination machine ("Transfer" step) which after reception can use existent tools to restore the "Image Files" and continue execution of the container. It is worth noting that the transform step doesn't have to be executed on the origin machine (Figure 6 Ⓐ), in fact it may run on the destination machine as well (Figure 6 Ⓑ), wherever the transformation step runs faster. In fact, on the edge where to transform the state can be decided dynamically, based on each node's transformation performance.

These mechanisms can be leveraged by a heterogeneous orchestration framework. The amount of resources on nodes, locating the best target for migration, SLA vs. pure speed tradeoffs, would be managed by such orchestration framework. Developing orchestration policies is out of the scope of this paper, and many related works exist (see Section 8).

6 Implementation

Our implementation of H-Container targets Linux in order to foster adoption and benefit from the large software ecosystem built around it, which includes support for containers. Moreover, Linux has been ported to numerous ISA CPUs, which certainly include processors that will be deployed on the edge.

In Linux, containers are based on namespaces [45] and control groups (cgroups) [11]. Hence, this work extends the CRIU project to migrate an application between computers of diverse ISA. H-Container does not need any modification of the OS kernel. Because we target a distributed environment in which automatic deployment is fundamental, we exploit Docker for deployment and orchestrate migration.

The proposed automatic executable binary transformation infrastructure is based on the McSema project to convert a natively compiled executable binary to LLVM IR. We also leverage components from the open-source Popcorn Linux compiler infrastructure to compile the LLVM IR into multiple binaries ready for cross-ISA migration, one for each processor ISA the application will run on.

6.1 Executable Transformation Infrastructure

As depicted in Figure 5.a H-Container is built by two main components: a decompiler and a compiler. Their internals are illustrated in Figure 5.b, we describe implementation details below. These components counts ~750 LoC for the Decompiler part, mostly bash and Python, and ~890 LoC of modifications on the Popcorn Linux compiler framework, including libraries. Because H-Container builds upon McSema and Popcorn Linux compiler, it mainly supports x86-64 and arm64 –

we plan to remove such limitation in the near future.

Decompiler. H-Container exploits McSema for binary disassembling and lifting. In order to transform a natively compiled executable binary into LLVM IR, McSema first disassembles the code by using IDA Pro and then it lifts the obtained native assembly code into LLVM IR. The produced LLVM IR can be directly recompiled into a dynamically linked binary for x86-64 and arm64 if the original executable binary was dynamically linked. It is worth noting that there are multiple decompilers publicly available other than McSema, including Ghidra [33], RetDec [46], rev.ng [24]. At the time of writing, McSema is the only one that outputs LLVM IR that can be recompiled into fully-functional x86-64 and arm64 binaries.

The third block in the decompiler is the *fixer*. The fixer was developed to modify the LLVM IR generated by McSema to address at least two issues. The first is that the current Popcorn compiler requires applications to be statically linked – more about this below. Thus, we decided to decompile dynamically linked binaries and recompile them as statically linked binaries. A dynamically linked binary includes data structures that enable library functions and symbols to be loaded at runtime, including Global Offset Table (GOT) and Procedure Linkage Table (PLT). The *fixer* substitutes calls to the GOT and PLT with calls to external symbols (in statically linked libraries) in the lifted LLVM IR. Because the format of such tables are compiler/linker dependent we provide support for clang, GCC, Glibc, and musl. Using this technique, we effectively enabled cross-ISA migration in programs that were originally dynamically linked. In fact, none of the available cross-ISA migration systems [6, 64] supports dynamically linked binaries even though dynamic ones are more widespread compared to static – 99% of the ELF executable binaries in a modern Linux distribution are dynamically linked [84].

The second problem is about replicated symbols. The LLVM IR produced by McSema reflects all assembly code included in the executable binary, which comprises other than the code of the program also library code to start the program and terminate it. This corresponds for example to the initialization routines that are called before `main()`, e.g., `_start()`, `_start_c()`, etc. Prior to recompilation, such routines, and relative global variables, have to be removed because the linker automatically re-adds them. The *fixer* takes care of this.

Compiler. H-Container doesn't re-invent the wheel but capitalizes and extends the Popcorn Linux's compiler framework to produce multiple binaries with the same address space layout, overlapping text sections, transformable stack frames, and migration points. The key innovation in H-Container is the possibility to create such multi-ISA binaries directly from the LLVM IR. Hence, the H-Container compiler takes the LLVM IR as input, and in a first stage it automatically adds migration points. For some multithreaded programs, the automatic insertion of migration points may induce a program in deadlock when a thread reaches a migration point while holding a lock while prevent another thread from ever reaching a migration point because the latter thread waits on the lock. Thus, we introduced the possibility to manually add migration points anywhere in the code (a simple call to a library functions) to avoid such scenarios.

In a second stage the compiler compiles and links the LLVM IR into executable binaries for multiple ISAs, a list of the symbols per binary is produced by the compiler, and the "Aligner" tool creates custom linker scripts to enforce the linker to align global symbols (function, global variables) at the same address amongst ISAs. This tool was entirely rewritten in the context of this work and is able to align symbols among any number of ISAs – the original Popcorn Linux's toolchain was limited to two ISAs. In a final stage the LLVM IR is compiled and linked again by using the produced linker scripts.

Additionally, H-Container reimplements the original migration library of Popcorn Linux in order to react to the notify tool discussed above. Finally, Popcorn's musl Libc was extended to let the C library loading code, which runs before `main()`, to enforce the same virtual address space aperture on every architecture. (Before enforced by the OS.)

6.2 Heterogeneous Migration

H-Container introduces Heterogeneous Checkpoint Restart In User-space (HetCRIU). HetCRIU extends CRIU in order to support the design in Figure 6, where the orange boxes are implemented by CRIU while the green ones are added by HetCRIU (*Notify* and *Transform*). With such modifications, migration of a process works as follows: 1) a notification is sent to the process that it has to stop; 2) every thread of the process stops at a migration point, *after* executing stack and registers transformation; 3) CRIU takes a snapshot of the process and writes the files to storage; 4) the extended CRIU Image Tool (crit) converts the dump files between architectures; 5) the snapshot files are transferred between machines; 6) the process is restored by CRIU and it continues execution from the migration point. The same procedure applies also when a container is composed by multiple processes. Operations 1) and 2) are implemented by the *Notify* step, while 4) by the *Transform* step. We provide an additional implementation of HetCRIU in which the functionality of 4) is integrated in 3), therefore there is no call to the external `crit` tool, we call such version *all-in-one*. The version with `crit` accounts for ~1820 LoC, while the *all-in-one* requires additional ~1200 LoC.

***Notify* Step.** The "Notify" step (cf. Figure 6) is implemented as an additional CRIU tool called `popcorn-notify` (or notify). Popcorn-notify doesn't infect the process as CRIU's *compel* does, because the process's binary is already compiled with migration points in place. Instead, it signals to the process that it has to freeze at the nearest migration point by writing a global variable in the process address space using `ptrace`.

Right after a thread of a process receives the notification, it traps into the closest migration point, it executes stack and register transformation, and freezes. This slightly changes our

design in Figure 6 because part of the transformation happens before the "Transform" block itself. However, this choice reduced the modifications to the Popcorn Linux compiler framework, thus facilitating a future upgrade to a newer version.

Note that the cost of "Notify" is dominated by the process freezing time; hence, implementing it as an external tool adds no overheads. Therefore, the CRIU's "Checkpoint" step was not modified at all – thus, reducing patches to the original source code which may have a long road to be accepted.

***Transform* Step.** We implemented the "Transform" step (cf. Figure 6) as an extension of `crit` by adding the `recode` option. This enables "Transform" to be called either on the origin or destination machine. `recode` opens multiple dump files, including *pages*, *pagemap*, and *core*; and converts these between architectures. Conversion includes the remapping of arithmetic, floating point and miscellaneous registers content between architectures, the adjustment of VDSO, vvar, and vsyscall areas, the fixing of the architecture name and executable name, etc. Additionally to those, container related modifications are required. These includes the updates of all per-session limits (i.e., what is controllable with `ulimit`), and the modifications to the thread to CPUs mappings.

Unfortunately, `crit` is characterized by a large overhead due to Python initialization and file copies (see Section 7). Hence, we implemented another version of HetCRIU that integrates anything done in `crit recode` into the main CRIU binary itself, called *all-in-one*.

Integration and Docker. HetCRIU introduces the `criu-het` executable that extends `criu` with new command options. `criu-het` invokes `popcorn-notify` first, then CRIU, and eventually `crit recode` – depending on the version of "Transform". HetCRIU fully supports CRIU's pre-dump live-migration by calling `popcorn-notify` only on the last checkpoint. Additionally, the *all-in-one* version supports CRIU's page server.

HetCRIU comes with an entire suite of extensions for Docker, which enables Docker container deployment and migration.

7 Evaluation

7.1 Experimental Setup

H-Container has been tested on a variety of ARM 64bit and x86 64bit computers in order to assess its deployability, from embedded platforms to servers, including Amazon Web Services instances [2]. We report the key results on an handful of platforms[1] whose hardware and software are described below.

Hardware. Other than the system described Section 2, composed by two embedded-class computers, in this section we will present results on two other systems. One is a setup with a workstation and an embedded board. The workstation mounts a single Xeon E5-2620 v4 at 2.1GHz, 8 dual-threaded cores, 16GB of RAM, and dual 1GbE connections. The embedded board (FireFly) mounts a Rockchip RK3308 with 4 Cortex-A53 cores at 1.3GHz, 4GB of RAM, and single 100MbE connection. The other system consists of two server grade machines: the first being a dual AMD EPYC 7451 at 2.3GHz, for a total of 48 cores and 96 threads, 256GB of RAM, dual 1GbE and dual 40GbE; the second being a dual Cavium ThunderX1 at 2.0GHz, for a total of 96 cores, with 256MB or RAM, single 1GbE (over USB 3.0) and quad 40GbE. We believe these three options cover all the spectrum of machines that can be found at the edge.

Software. Despite H-Container is Linux version neutral, just requiring that the kernel support CRIU, we used Linux Ubuntu Xenial (16.04.5 and 16.04.6) on all ARM and x86 machines. The Cavium ThunderX1 and the AMD EPYC run Linux kernel 4.15.0-45-generic; while the Rockchip RK3308 and the Intel E5 run Linux kernel 4.4.178. H-Container is built on CRIU version 3.11, and extends Docker version 18.09.06. Finally, we extended the Popcorn's runtime git commit fd578a9.

H-Container compiler's framework has been developed using McSema/Remill (git commit 101940f and c0c0847, respectively) that requires IDA Pro 7.2. Popcorn's compiler version commit fd578a9, which forced us to use LLVM/clang 3.7.1.

Applications' suites we used to characterize H-Container are discussed in each of the following sections.

7.2 Overheads Evaluation

We characterized H-Container costs on a set of benchmarks collected from different projects. Based on previous works [1, 36, 43, 55, 56, 58, 81, 93] we believe such set of benchmarks well represents compute/memory workloads that can be found at the edge. The focus on compute/memory workloads is motivated by the necessity of spotting compiler/runtime overheads, not OS ones. Specifically, we used NAS Parallel Benchmarks [5] (NPB), Map-Reduce's applications for shared memory from Phoenix [69], Linpack [26], and Dhrystone [92]. We run the NPB benchmarks, is, ep, ft, cg, and bt, for different data sizes (class S, A, B, C). We run Phoenix benchmarks mm, pca, and kmeans, with different data input sizes. In the following we first present the decompiler-compiler tool overheads and then the overheads introduced by our implementation(s) of HetCRIU. Values are averages of 10 samples.

Decompiler-compiler Overheads. All experiments herein run on the Cavium ThunderX1 and AMD EPYC. We first investigate how much does the decompilation and recompilation processes cost. Hence, we run a set of experiments on both ARM and x86 in order to identify such overheads on different benchmarks. Figure 7 illustrates the results for Phoenix matrix multiplication, for which we compiled such application without optimization (-O0) and with max optimizations (-O3) with gcc and clang (top graph and bottom graph), we decompiled it with the McSema-based decompiler and we recompiled it back with the extended Popcorn compiler, by using different optimization levels (-O0, -O1, -O2, -O3). Graphs show the execution time of the newly produced binaries over the execution time of the original binaries, a value higher than one means that the new binary is slower, while lower than one means

[1] Additional results are available online [88].

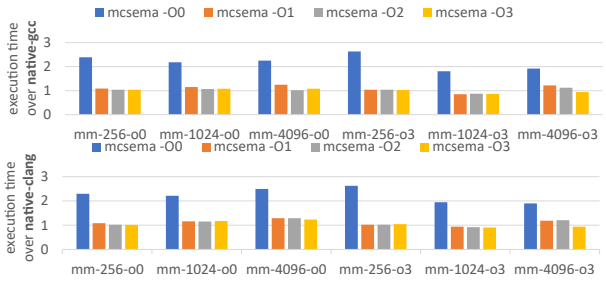

Figure 7. Execution time ratio of a transformed executable versus the original one for Phoenix mm. When the original is compiled with -O0 (first 3 clusters) and with -O3 (latter 3 clusters), varying the optimization level when recompiling after decompilation. Top graph gcc, bottom clang.

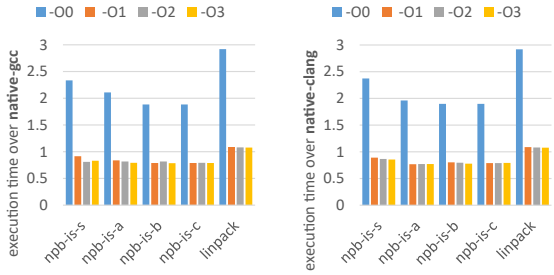

Figure 8. Execution time ratio of a transformed executable versus the original one for npb-is and Linpack, varying the optimization level when recompiling after decompilation. Left graph gcc, right clang.

the new binary is faster. From the graph it is clear that independently of the way the original binary is created, if the new binary is produced with maximum optimization it can be as faster as the original one, up to 6% slower, and up to 9% faster.

We then repeated the same experiment for all other micro-benchmarks, and some of the results are reported in Figure 8. These results confirm what we learn for Phoenix matrix multiplication: the decompiler-compiler tool produces executable binaries that are as fast as the original, in this case up to 20% faster than the original, and up to 9% slower. With and without migration points the observed results are the same.

Finally, we repeated all such experiments on x86 as well as on ARM and compared to the overhead of using emulation (QEMU 2.5) instead of H-Container– in order to highlight the benefits of the proposed architecture that employs natively-compiled binaries for both ISAs versus using one binary and emulate the others. The results are reported in Figure 9. Generally, ARM execution has higher overheads than x86. However, and more importantly, emulation is always slower than H-Container static binary transformation, from 2.2*x* than native to up to 18.9 times than native – while H-Container is up to 70.7% faster than native, thanks to McSema and LLVM's optimizations. Note that the same experiments have been performed for statically and dynamically compiled binaries, and the results (averages) are similar (around 1% of difference).

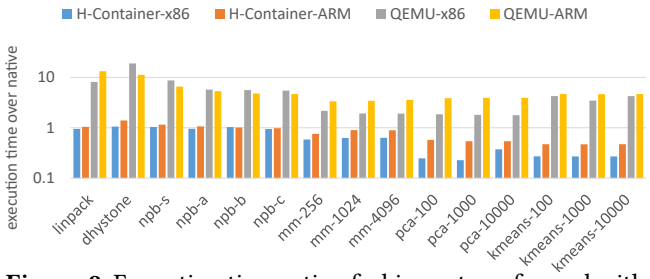

Figure 9. Execution time ratio of a binary transformed with the decompiler-compiler infrastructure (mcsema) over the original, and its execution on QEMU over original. For an ARM binary running on x86 (blue and gray), and an x86 binary running on ARM (orange and yellow).

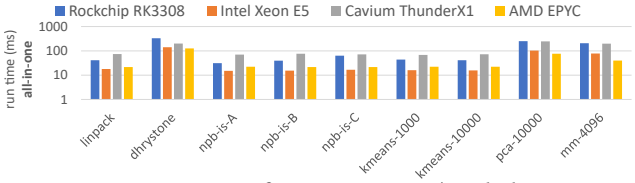

Figure 10. Popcorn notify cost on ARM (Rockchip RK3308, Cavium ThunderX1) and x86 (Intel Xeon E5, AMD EPYC) for different benchmarks.

Summary. The decompiler-compiler either adds trivial overheads (up to 9%) or makes the executable faster (up to 70.7%). Therefore, H-Container is better than emulation, which may slow down execution up to 18.9 times.

HetCRIU Overheads. As edge workloads are likely to be latency-sensitive, migrations must be as fast as possible. Therefore, below we analyze the overheads introduced by HetCRIU. Specifically, we characterize the overhead of notifying (popcorn-notify), and transforming the state (all-in-one, or crit). We use the same set of benchmarks because they are more keen in highlighting latencies than IO bound ones. We present the results for Cavium ThunderX1, AMD EPYC, Rockchip RK3308, and Intel Xeon E5.

A first set of experiments analyzes the cost of popcorn-notify, i.e., the time required to stop the Popcorn binary and transform its stack for the destination architecture. Results on ARM and on x86 platforms are depicted in Figure 10. Note that the results are the same independently of the level of the integration of the CRIU extracted state's transformation (all-in-one, crit). The graphs show that stopping the executable may require between tens and hundreds of milliseconds. Moreover, this process is slower on the slowest platform (Rockchip RK3308), but x86 machines are equally fast. As stack transformation exhibit the same overheads as reported in [6], the reasons for these overheads are rooted in the compiler that keeps migration points only at the existent function boundaries. We believe that with better outlining, or with automatic placement of additional migration points, the overhead of popcorn-notify can be further reduced, also on the ARM machines.

A second set of experiments breaks down the two implementations of CRIU's exported state transformation, on the

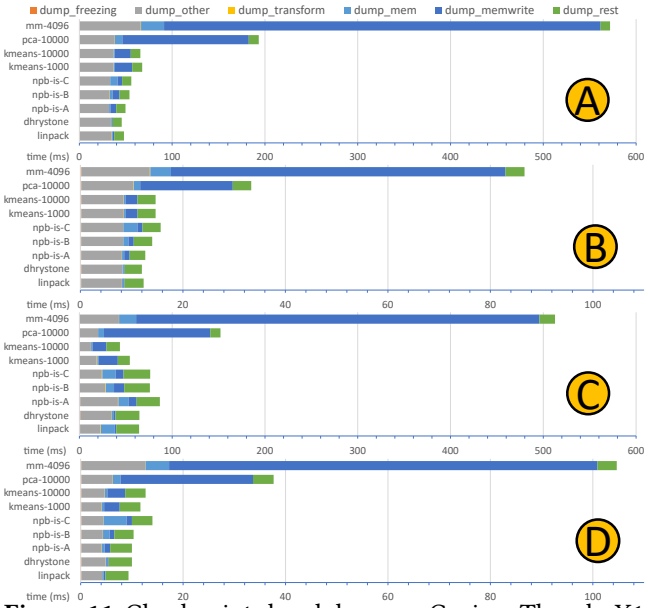

Figure 11. Checkpoints breakdown on Cavium ThunderX1 (A), AMD EPYC (B), Rockchip RK3308 (C), and Intel Xeon E5 (D) for the *all-in-one* implementation of transform.

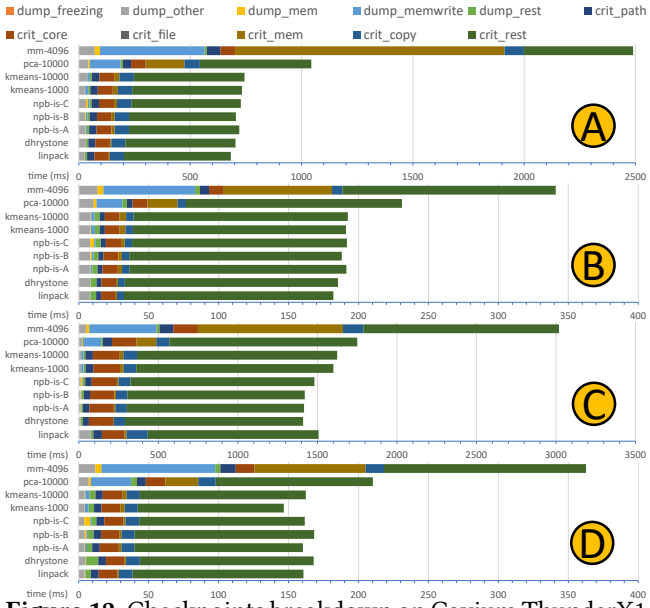

Figure 12. Checkpoints breakdown on Cavium ThunderX1 (A), AMD EPYC (B), Rockchip RK3308 (C), and Intel Xeon E5 (D) for the `crit recode` implementation of transform.

same set of 4 machines and the same set of benchmarks. Figure 11 reports the breakdown of the cost to do run a checkpoint (dump) and converting it to the destination architecture within the same program (CRIU). Most of these numbers are already reported by `stats` file generated by CRIU, we added "dump_transform" that accounts for transforming the state from the origin to the destination architecture. Despite the total dump time is mostly proportional to the total memory to checkpoint (see "dump_memwrite"), what our code adds, "dump_transform", is always lower than 1% of the total dump – for both ARM and x86. Thus, the HetCRIU all-in-one overhead is negligible. Please note that the "dump_freezing" time, which is the time to stop the application in *vanilla* CRIU, is always lower than 0.1% because popcorn-notify stops the task(s).

When modifying CRIU is not an option, our `crit recode` should be used. The overheads of checkpointing with this tool are reported in Figure 12. Differently from the previous one, this option is way more expensive than using *normal* CRIU: running `crit recode` requires from twice to 17.5 times additional time (respectively for Phoenix matrix multiplication, and Linpack). This is because `crit recode` needs to reload the image files as well as copying them (there is no copy with all-in-one). Another issue with this tool, is that because it is written in python it has a fixed cost for loading all the imports and termination, this cost is summarized in the graphs in "crit_rest".

Summary. Stopping a Popcorn binary (notify) requires between tens and hundreds of milliseconds. However, we shown that by including the "Transform" step in CRIU itself (*all-in-one*), transformation time is negligible vs. CRIU time. When patching CRIU is not possible – crit recode, an additional overhead from 2x to 17.5x must be paid.

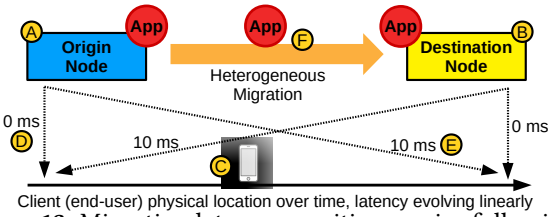

Figure 13. Migrating latency-sensitive service following a moving user.

7.3 Migration of Latency-Sensitive Services

We demonstrate the usefulness of H-Container in a scenario where a latency-sensitive service migrates between edge nodes to stay as close as possible to a mobile end user. We experiment with multiple services, including Redis [70], Nginx [71], a compression server (Gzip), and a game server [14]. We believe these services are representative of edge applications by virtue of previous works [1, 36, 43, 55, 56, 58, 81, 93]. Further, we show that H-Container handles server applications, something not supported by previous work [6, 64].

The proposed scenario is illustrated in Figure 13. We assume that along the path of the end user (the client), two close by edge nodes are present – the *origin* Ⓐ and the *destination* Ⓑ, and they have different ISA. The client Ⓒ is mobile, and moves further away from *origin* and closer to *destination*. We arbitrarily define the length of the experiment, i.e., the time for the client to go from one node to the other, to be 1000 seconds (c.a., 16 mins). The impact on latency of the physical distance between the client and the server is represented by having each node artificially increase the latency of its NIC using `tc` from 0*ms* (Ⓓ) when the client is close by, up to 10*ms* (Ⓔ) when it is the farthest: every 100*s*, the latency of *origin*

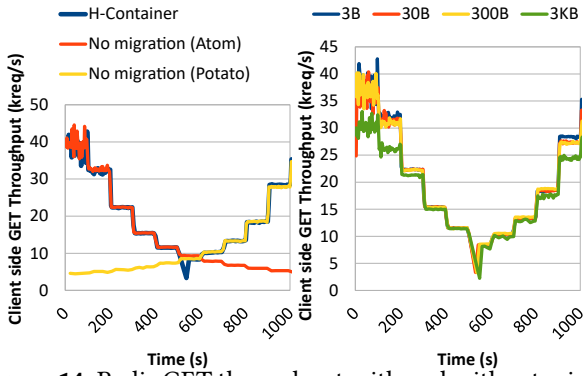

Figure 14. Redis GET throughput with and without migration (left), and migration with different payloads (right). *origin* node is an Intel Atom, *destination* is the Potato board.

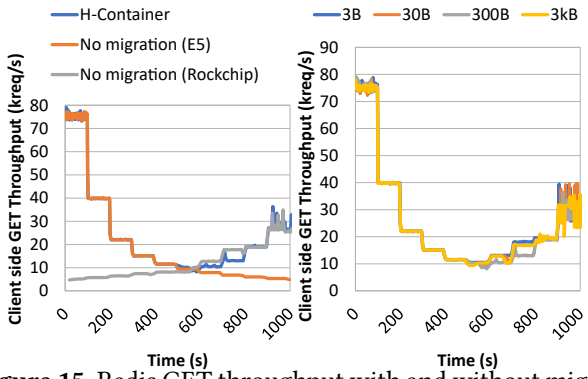

Figure 15. Redis GET throughput with and without migration (left), and migration with different payloads (right). *origin* node is an Intel E5, *destination* is the Rockchip RK3308.

is increased by 1*ms* and *destination*'s latency is decreased by 1*ms*. These latency values are on par with what expected at the edge [14]. The client uses a different benchmark to sample how many operations it can run on the server it is currently connected to. We used `redis-benchmark` for GET throughput, `apachebench` for latency, to get the total compressed B/s, and actions/s, respectively.

In fact, we experimented with three scenarios. In the first two scenarios we assume that heterogeneous migration is not possible, thus the server is stuck either on the *origin* or *destination* nodes, both called "No migration". In the third scenario we enable H-Container, which allows service migration from *origin* to *destination* Ⓕ when the throughput (Redis), latency (Nginx, Gzip), or operations (game server) fall under a certain threshold. In such scenario the client is able to redirect its traffic to the right node by the use of a local instance of HAProxy [83] that uses health check rules to automatically redirect requests to the node running the server.

Latency and Throughput. As already mentioned in Section 2, even small variations in latency have huge impact on performance. The left graphs in Figure 14 and 15 show for a fixed payload size, Redis' GET throughput for the Intel Atom plus Potato board, and for the Intel E5 plus Rockchip RK3308. The Intels are connected via 1GbE while the others

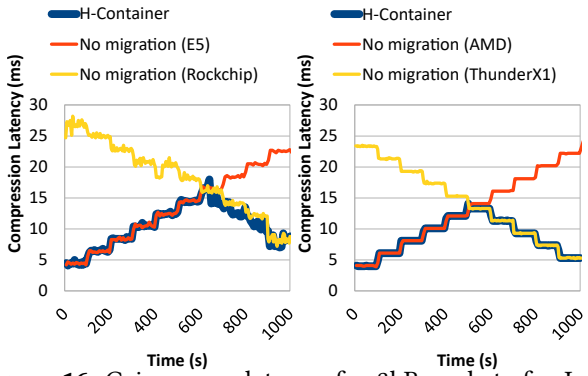

Figure 16. Gzip server latency for 3kB packets for Intel E5 plus Rockchip RK3308 (left), and AMD plus Cavium ThunderX1 (right).

via 100MbE.

In scenarios where migration is not possible, the server is stuck on one node and the throughput either gradually decreases or gradually increases while the client gets further or closer to the node in question: in these scenarios, about one half of the experiment execution is spent under one fourth of the maximum achievable throughput (40 000 req/s).

With H-Container, the server can migrate between nodes and follow the client. There is a slight drop in throughput in the middle of the experiment – i.e., the downtime caused by the migration itself. While this throughput decreases as the client gets further away from *origin*, the migration enabled by H-Container makes that performance starts to increase again past the migration point as the client gets closer to *destination*. We computed the average throughput over the entire experiment for the three scenarios. For the Atom plus Potato it is 15,497req/s when the server is always on *origin* and 10, 907req/s when it is always on *destination*. For the same machines using H-Container the average throughput is 19,766req/s showing an improvement of 27.5% and 81.2% vs. no migration scenarios. When using the E5 plus Rockchip, without migration throughput averages are 19,751req/s and 12,029req/s, while H-Container achieves 24,393req/s, i.e., an improvement of 23.4% and 94.5% respectively.

Similar conclusions can be drawn for the other applications and platforms, Figure 16 shows the latencies of the Gzip server.

Varying Request Size. For Redis, we varied the payload size from 3B to 3kB. The results are shown in the right graphs of Figure 14 and 15. For payloads up to 300 bytes the behavior is similar because the experiment is bounded by latency. However, performance starts to differ when the request size increases to 3kB. On the *origin* node we observe a throughput decrease of about 15% compared to smaller request sizes in the case the latency is low (first 200 seconds) and the same happens in the opposite direction. Another observation is that after migration, the request throughput lowers on the *destination* node. This is due to the network bandwidth capping the performance. Indeed, both the Potato and Rockchip are equipped with a 100MbE NIC as opposed to the Atom and E5 that use

GbE. This asymmetry is not visible when both machines have similar compute power (e.g., AMD plus Cavium ThunderX1).

That being said, even with low-cost and slow NICs, nodes such as the Potato board are competitive in latency-sensitive scenarios where the request size is relatively small, which is a quite common case on the edge, e.g., game servers [29].

7.4 Limitations

H-Container strictly depends on its software components. Thus, the current version of H-Container is also affected by their limitations, which are mainly three. First and foremost, McSema does not fully support FPU instructions, thus applications such as NPB FT cannot be transformed into a migratable binary by our decompiler-compiler. Additionally, library calls that pass arguments by reference (e.g., `fstat`) do not work, thus we needed to patch the source code of the applications and libraries (up to ~1900 LoC). Secondly, the Popcorn Linux compiler framework cannot create migratable dynamically linked binaries. Despite clearly a limitation, Slinky [17] has demonstrated that statically linked binary can be treated such as dynamically linked one, thus manageable via Docker. Moreover, Popcorn cannot compile functions with variable arguments, we needed to patch the sources (up to ~1100 LoC). Third, Docker live-migration support is incomplete today.

8 Related Work

Migrating at the Edge. The topic of application migration among edge nodes has been considered before. K. Ha et al. [36] proposes VM handoff, a set of techniques for efficient VM live migration on the edge, while L. Ma et al. [55] looked at the problem of migration containerized application between edge nodes with Docker. A. Machen et al. [56, 58] introduced a three layer framework to support virtual machine and container migrations. None of such works considered that edge nodes are intrinsically built with heterogeneous ISA CPUs – thus, letting H-Container being the first in addressing the problem. De facto, H-Container is orthogonal to such works: any optimization developed by previous work can be used by it.

Finally, several other papers considered migration as a scheduling, mapping, and orchestration problem, including [1, 43, 81, 93]. H-Container doesn't address these problems, but it can exploit, or be leveraged by, such works.

Runtime Software Migration. A long list of previous works addressed the problem of how to migrate running software between different machines. The majority of which has been developed in the context of data centers where computers are homogeneous. Amongst others, the most similar works to H-Container are ZAP [65] and CRIU [28, 80], which implement checkpoint/restart of Linux processes among same ISA processors. CRIU supports live migration, and it is the underlying mechanism enabling container migration via deployment software such as Docker [18], LXC [19], etc. H-Container extends CRIU and integrates with Docker to migrate a container across heterogeneous ISA processors – additionally, H-Container includes a binary executable transformation infrastructure to support such migration.

In the past, different works have been published on the topic of process migration among heterogeneous ISA processors [3, 44, 78]. Recently, Popcorn Linux [6, 7, 10, 54] proposes a compiler and runtime toolset for cross-ISA migration, reconsidering the same problem on emerging heterogeneous platforms. More recently, HEXO [64] leverages the Popcorn compiler to migrate lightweight VMs (unikernels) between machines of different ISAs. H-Container differs from these works by implementing migration completely in user space, without any dependence on a custom OS kernel (Popcorn) or on a custom hypervisor (HEXO) – because such solutions are unlikely to be easily deployable in production, H-Container is also more flexible as it requires no access to the application sources.

A unified address space among heterogeneous ISA processors has been also proposed by MutekH [60], which code was not practically usable because targets an exokernel/libos, and A. Venkat et al. [86], whose code is not publicly available.

9 Conclusion

Migrating server applications between edge nodes to maintain physical proximity to a moving client application running on a mobile device has been demonstrated to guarantee minimal client-server latencies for edge computing scenarios on homogeneous-ISA nodes. However, the edge is populated by computers with CPUs of different ISA, which hinders server migration to the closest node to the client – an application compiled for an ISA cannot migrate to, nor run, on another.

This paper introduces H-Container, which enables containerized applications to migrate across heterogeneous-ISA nodes. H-Container targets Linux and is composed of (1) a LLVM-based decompiler-compiler transforming executable binaries for multiple ISA execution, as well as a (2) CRIU-based user-space checkpoint/restart framework to stop an application on one ISA and resume it on another. H-Container is based on a new deployment model where cloud software repositories store IR binaries. It also improves upon state-of-the-art cross-ISA migration frameworks by being highly compatible, easily deployable, and largely Linux compliant. Experiments show that the executable binary transformation does not add overhead on average, and that the overhead for heterogeneous migration is between 10*ms* and 100*ms* compared to stock CRIU. Overall, we show that heterogeneous-ISA migration at the edge unlocks higher performance for latency-sensitive applications, e.g., 94% better throughput on average on Redis.

H-Container is open-source and publicly available [88].

Acknowledgments

We thank the anonymous reviewers and our shepherd, Mark Silberstein, for their invaluable feedback. This work was supported in part by US Office of Naval Research under grants N00014-16-1-2104, N00014-16-1-2711, and N00014-19-1-2493.

References

[1] O. I. Abdullaziz, L. Wang, S. B. Chundrigar, and K. Huang. 2018. ARNAB: Transparent Service Continuity Across Orchestrated Edge Networks. In *2018 IEEE Globecom Workshops (GC Wkshps)*. 1–6. https://doi.org/10.1109/GLOCOMW.2018.8644091

[2] Amazon. 2018. EC2 Instances (A1) Powered by Arm-Based AWS Graviton Processors. https://aws.amazon.com/blogs/aws/new-ec2-instances-a1-powered-by-arm-based-aws-graviton-processors/.

[3] Giuseppe Attardi, A Baldi, U Boni, F Carignani, G Cozzi, A Pelligrini, E Durocher, I Filotti, Wang Qing, M Hunter, et al. 1988. Techniques for dynamic software migration. In *Proceedings of the 5th Annual ESPRIT Conference (ESPRIT'88)*, Vol. 1.

[4] MOhamed Awad. 2019. Arm and Docker: Better Together. https://www.arm.com/company/news/2019/04/arm-and-docker-better-together.

[5] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. 1991. The NAS parallel benchmarks summary and preliminary results. In *Supercomputing '91:Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*. 158–165. https://doi.org/10.1145/125826.125925

[6] Antonio Barbalace, Rob Lyerly, Christopher Jelesnianski, Anthony Carno, Ho-ren Chuang, and Binoy Ravindran. 2017. Breaking the Boundaries in Heterogeneous-ISA Datacenters. In *Proceedings of the 22th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*.

[7] Antonio Barbalace, Alastair Murray, Rob Lyerly, and Binoy Ravindran. 2014. Towards Operating System Support for Heterogeneous-ISA Platforms. In *In Proceedings of The 4th Workshop on Systems for Future Multicore Architectures (4th SFMA*.

[8] Antonio Barbalace, Marina Sadini, Saif Ansary, Christopher Jelesnianski, Akshay 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 (EuroSys '15)*. 29:1–29:16.

[9] D. Bernstein. 2014. Containers and Cloud: From LXC to Docker to Kubernetes. *IEEE Cloud Computing* 1, 3 (Sep. 2014), 81–84. https://doi.org/10.1109/MCC.2014.51

[10] Sharath K. Bhat, Ajithchandra Saya, Hemedra K. Rawat, Antonio Barbalace, and Binoy Ravindran. [n.d.]. Harnessing Energy Efficiency of Heterogeneous-ISA Platforms. *SIGOPS Oper. Syst. Rev.* 49, 2 ([n. d.]).

[11] Neil Brown. 2014. Control groups series. Linux Weekly News. https://lwn.net/Articles/604609/.

[12] Edouard Bugnion, Jason Nieh, and Dan Tsafrir. 2017. Hardware and software support for virtualization. *Synthesis Lectures on Computer Architecture* 12, 1 (2017), 1–206.

[13] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.. In *OSDI*, Vol. 8. 209–224.

[14] Sharon Choy, Bernard Wong, Gwendal Simon, and Catherine Rosenberg. 2012. The brewing storm in cloud gaming: A measurement study on cloud to end-user latency. In *Proceedings of the 11th annual workshop on network and systems support for games*. IEEE Press, 2.

[15] 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.

[16] 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*. USENIX Association, 273–286.

[17] Christian S Collberg, John H Hartman, Sridivya Babu, and Sharath K Udupa. 2005. SLINKY: Static Linking Reloaded.. In *USENIX Annual Technical Conference, General Track*. 309–322.

[18] CRIU contributors. 2019. CRIU Wiki – Docker page. https://criu.org/Docker.

[19] CRIU contributors. 2019. CRIU Wiki – LXC page. https://criu.org/LXC.

[20] 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*. ACM, 49–62.

[21] Anirban Das, Stacy Patterson, and Mike Wittie. 2018. EdgeBench: Benchmarking Edge Computing Platforms. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*. IEEE, 175–180.

[22] Dell. 2019. Micro Modular Data Centers: Taking Computing to the Edge. https://blog.dellemc.com/en-us/micro-modular-data-centers-taking-computing-to-edge/.

[23] Matthew DeVuyst, Ashish Venkat, and Dean M. Tullsen. 2012. Execution Migration in a heterogeneous-ISA Chip Multiprocessor. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII)*. 261–272.

[24] Alessandro Di Federico, Mathias Payer, and Giovanni Agosta. 2017. rev. ng: a unified binary analysis framework to recover CFGs and function boundaries. In *Proceedings of the 26th International Conference on Compiler Construction*. ACM, 131–141.

[25] Artem Dinaburg and Andrew Ruef. 2014. Mcsema: Static translation of x86 instructions to llvm. In *ReCon 2014 Conference, Montreal, Canada*.

[26] 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.

[27] edgeconnex. 2019. Bringing Gaming Closer to Gamers Worldwide. https://www.edgeconnex.com/wp-content/uploads/2019/07/EDC-19-44-NEW-Gaming-DataSheet-V4.pdf.

[28] P EMELYANOV. [n.d.]. CRIU: Checkpoint/Restore In Userspace, July 2011.

[29] Wu-chang Feng, Francis Chang, Wu-chi Feng, and Jonathan Walpole. 2005. A Traffic Characterization of Popular On-line Games. *IEEE/ACM Trans. Netw.* 13, 3 (June 2005), 488–500. https://doi.org/10.1109/TNET.2005.850221

[30] Matthew Furlong, Andrew Quinn, and Jason Flinn. 2019. The Case for Determinism on the Edge. In *2nd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 19)*. USENIX Association, Renton, WA. https://www.usenix.org/conference/hotedge19/presentation/furlong

[31] J. Gedeon, F. Brandherm, R. Egert, T. Grube, and M. Mühlhäuser. 2019. What the Fog? Edge Computing Revisited: Promises, Applications and Future Challenges. *IEEE Access* 7 (2019), 152847–152878. https://doi.org/10.1109/ACCESS.2019.2948399

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

[33] Ghidra Contributors. 2019. Ghidra Website. https://ghidra-sre.org/.

[34] Mark S Gordon, D Anoushe Jamshidi, Scott Mahlke, Z Morley Mao, and Xu Chen. 2012. {COMET}: Code Offload by Migrating Execution Transparently. In *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*. 93–106.

[35] Kiryong Ha, Yoshihisa Abe, Thomas Eiszler, Zhuo Chen, Wenlu Hu, Brandon Amos, Rohit Upadhyaya, Padmanabhan Pillai, and Mahadev Satyanarayanan. 2017. You can teach elephants to dance: agile vm handoff for edge computing. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*. ACM, 12.

[36] Kiryong Ha, Yoshihisa Abe, Thomas Eiszler, Zhuo Chen, Wenlu Hu, Brandon Amos, Rohit Upadhyaya, Padmanabhan Pillai, and Mahadev Satyanarayanan. 2017. You Can Teach Elephants to Dance: Agile VM Handoff for Edge Computing. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing* (San Jose,

California) *(SEC '17)*. ACM, New York, NY, USA, Article 12, 14 pages. https://doi.org/10.1145/3132211.3134453

[37] Christine Hall. 2019. Companies Pushing Open Source RISC-V Silicon Out to the Edge. https://www.datacenterknowledge.com/hardware/companies-pushing-open-source-risc-v-silicon-out-edge.

[38] Drew Henry. 2018. Announcing ARM Neoverse. https://www.arm.com/company/news/2018/10/announcing-arm-neoverse.

[39] Kelsey Hightower, Brendan Burns, and Joe Beda. 2017. *Kubernetes: Up and Running: Dive Into the Future of Infrastructure.* " O'Reilly Media, Inc.".

[40] Michael R Hines, Umesh Deshpande, and Kartik Gopalan. 2009. Post-copy live migration of virtual machines. *ACM SIGOPS operating systems review* 43, 3 (2009).

[41] Cheol-Ho Hong and Blesson Varghese. 2019. Resource Management in Fog/Edge Computing: A Survey on Architectures, Infrastructure, and Algorithms. *ACM Comput. Surv.* 52, 5, Article 97 (Sept. 2019), 37 pages. https://doi.org/10.1145/3326066

[42] Intel. 2018. Intelligence at the 'Edge': the Intel Xeon D-2100 Processor. https://www.intel.com/content/www/us/en/communications/d-2100-processor-edge-computing-benefits-infographic.html.

[43] M. Jia, J. Cao, and W. Liang. 2017. Optimal Cloudlet Placement and User to Cloudlet Allocation in Wireless Metropolitan Area Networks. *IEEE Transactions on Cloud Computing* 5, 4 (Oct 2017), 725–737. https://doi.org/10.1109/TCC.2015.2449834

[44] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. 1988. Fine-grained Mobility in the Emerald System. *ACM Trans. Comput. Syst.* 6, 1 (Feb. 1988), 109–133. https://doi.org/10.1145/35037.42182

[45] Michael Kerrisk. 2013. Namespaces in operation, part 1: namespaces overview. Linux Weekly News. https://lwn.net/Articles/531114/.

[46] Jakub Křoustek and Peter Matula. 2018. Retdec: An open-source machine-code decompiler. (2018). https://2018.pass-the-salt.org/files/talks/04-retdec.pdf.

[47] 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.

[48] 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.

[49] George Leopold. 2019. Arm, Docker Partner on Cloud-to-Edge Development. https://www.enterpriseai.news/2019/04/24/arm-docker-partner-on-cloud-to-edge-development/.

[50] A. Lertsinsrubtavee, A. Ali, C. Molina-Jimenez, A. Sathiaseelan, and J. Crowcroft. 2017. PiCasso: A lightweight edge computing platform. In *2017 IEEE 6th International Conference on Cloud Networking (CloudNet)*. 1–7. https://doi.org/10.1109/CloudNet.2017.8071529

[51] Chao Li, Yushu Xue, Jing Wang, Weigong Zhang, and Tao Li. 2018. Edge-Oriented Computing Paradigms: A Survey on Architecture Design and System Management. *ACM Comput. Surv.* 51, 2, Article 39 (April 2018), 34 pages. https://doi.org/10.1145/3154815

[52] ARM Ltd. 2019. Accelerating the transformation to a scalable cloud to edge infrastructure. https://www.arm.com/-/media/global/products/processors/N1%20Solution%20Overview.pdf.

[53] Peng Lu, Antonio Barbalace, and Binoy Ravindran. 2013. HSG-LM: hybrid-copy speculative guest OS live migration without hypervisor. In *Proceedings of the 6th International Systems and Storage Conference*. ACM, 2.

[54] Robert Lyerly, Antonio Barbalace, Christopher Jelesnianski, Vincent Legout, Anthony Carno, and Binoy Ravindran. 2016. Operating System Process and Thread Migration in Heterogeneous Platforms. (2016).

[55] Lele Ma, Shanhe Yi, and Qun Li. 2017. Efficient Service Handoff Across Edge Servers via Docker Container Migration. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing* (San Jose, California) *(SEC '17)*. ACM, New York, NY, USA, Article 11, 13 pages. https://doi.org/10.1145/3132211.3134460

[56] Andrew Machen, Shiqiang Wang, Kin K. Leung, Bong Jun Ko, and Theodoros Salonidis. 2016. Migrating Running Applications Across Mobile Edge Clouds: Poster. In *Proceedings of the 22Nd Annual International Conference on Mobile Computing and Networking* (New York City, New York) *(MobiCom '16)*. ACM, New York, NY, USA, 435–436. https://doi.org/10.1145/2973750.2985265

[57] Andrew Machen, Shiqiang Wang, Kin K Leung, Bong Jun Ko, and Theodoros Salonidis. 2018. Live service migration in mobile edge clouds. *IEEE Wireless Communications* 25, 1 (2018), 140–147.

[58] Andrew Machen, Shiqiang Wang, Kin K. Leung, Bong Jun Ko, and Theodoros Salonidis. 2018. Live Service Migration in Mobile Edge Clouds. *Wireless Commun.* 25, 1 (Feb. 2018), 140–147. https://doi.org/10.1109/MWC.2017.1700011

[59] Anil Madhavapeddy, Thomas Leonard, Magnus Skjegstad, Thomas Gazagnaire, David Sheets, Dave Scott, Richard Mortier, Amir Chaudhry, Balraj Singh, Jon Ludlam, et al. 2015. Jitsu: Just-in-time summoning of unikernels. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*. 559–573.

[60] MutekH Authors. 2016. MutekH reference manual. https://www.mutekh.org/doc/index.html.

[61] S. Nadgowda, S. Suneja, N. Bila, and C. Isci. 2017. Voyager: Complete Container State Migration. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. 2137–2142. https://doi.org/10.1109/ICDCS.2017.91

[62] M. Noreikis, Y. Xiao, and A. Ylä-Jaäiski. 2017. QoS-oriented capacity planning for edge computing. In *2017 IEEE International Conference on Communications (ICC)*. 1–6. https://doi.org/10.1109/ICC.2017.7997387

[63] Christy Norman Perez and Chris Jones. 2017. The ARM to z of Multi-Architecture Microservices. https://qconsf.com/sf2017/system/files/presentation-slides/from_arm_to_z.pdf.

[64] Pierre Olivier, Mehrab Fazla, Stefan Lankes, Mohamed Lamine Karaoui, Rob Lyerly, and Binoy Ravindran. 2019. HEXO: Offloading HPC Compute-Intensive Workloads on Low-Cost, Low-Power Embedded Systems. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing (HPDC'19)*.

[65] Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh. 2002. The Design and Implementation of Zap: A System for Migrating Computing Environments. *SIGOPS Oper. Syst. Rev.* 36, SI (Dec. 2002), 361–376. https://doi.org/10.1145/844128.844162

[66] Adam Parco. 2019. Building Multi-Arch Images for Arm and x86 with Docker Desktop. https://engineering.docker.com/2019/04/multi-arch-images/.

[67] PicoCluster. 2019. PicoCenter 48. https://www.picocluster.com/products/picocenter-48.

[68] C. Puliafito, E. Mingozzi, C. Vallati, F. Longo, and G. Merlino. 2018. Virtualization and Migration at the Network Edge: An Overview. In *2018 IEEE International Conference on Smart Computing (SMARTCOMP)*. 368–374. https://doi.org/10.1109/SMARTCOMP.2018.00031

[69] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary R Bradski, and Christos Kozyrakis. 2007. Evaluating MapReduce for multi-core and multiprocessor systems.. In *hpca*, Vol. 7. 19.

[70] Redis Labs. 2019. RedisEdge - The Edge Computing Database for the IoT Edge. https://redislabs.com/solutions/redisedge/.

[71] Will Reese. 2008. Nginx: the high-performance web server and reverse proxy. *Linux Journal* 2008, 173 (2008), 2.

[72] Phil Rogers. 2013. Heterogeneous system architecture overview. In *Hot Chips*, Vol. 25.

[73] Mahadev Satyanarayanan. 2017. The emergence of edge computing. *Computer* 50, 1 (2017), 30–39.

[74] Mahadev Satyanarayanan, Zhuo Chen, Kiryong Ha, Wenlu Hu, Wolfgang Richter, and Padmanabhan Pillai. 2014. Cloudlets: at the leading edge of mobile-cloud convergence. In *6th International Conference on Mobile Computing, Applications and Services*. IEEE, 1–9.

[75] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. 2016. Edge computing: Vision and challenges. *IEEE Internet of Things Journal* 3, 5 (2016), 637–646.

[76] Grant Shipley. 2014. *Learning OpenShift.* Packt Publishing Ltd.

[77] SiFive. 2019. SiFive U74. https://www.sifive.com/cores/u74.

[78] Peter Smith and Norman C Hutchinson. 1998. Heterogeneous process migration: The Tui system. *Software: Practice and Experience* 28, 6 (1998), 611–639.

[79] Matthew Smithson, Khaled ElWazeer, Kapil Anand, Aparna Kotha, and Rajeev Barua. 2013. Static binary rewriting without supplemental information: Overcoming the tradeoff between coverage and correctness. In *2013 20th Working Conference on Reverse Engineering (WCRE)*. IEEE, 52–61.

[80] Radostin Stoyanov and Martin J. Kollingbaum. 2018. Efficient Live Migration of Linux Containers. In *High Performance Computing*. Springer International Publishing, 184–193.

[81] Kyoungjae Sun and Younghan Kim. 2018. Network-based VM Migration Architecture in Edge Computing. In *Proceedings of the 2018 International Conference on Information Science and System* (Jeju, Republic of Korea) *(ICISS '18)*. ACM, New York, NY, USA, 169–172. https://doi.org/10.1145/3209914.3209930

[82] T. Taleb, K. Samdanis, B. Mada, H. Flinck, S. Dutta, and D. Sabella. 2017. On Multi-Access Edge Computing: A Survey of the Emerging 5G Network Edge Cloud Architecture and Orchestration. *IEEE Communications Surveys Tutorials* 19, 3 (thirdquarter 2017), 1657–1681. https://doi.org/10.1109/COMST.2017.2705720

[83] Willy Tarreau et al. 2012. HAProxy-the reliable, high-performance TCP/HTTP load balancer.

[84] Chia-Che Tsai, Bhushan Jain, Nafees Ahmed Abdul, and Donald E Porter. 2016. A study of modern Linux API usage and compatibility: what to support when you're supporting. In *Proceedings of the Eleventh European Conference on Computer Systems*. ACM, 16.

[85] Blesson Varghese, Nan Wang, Dimitrios S. Nikolopoulos, and Rajkumar Buyya. 2017. Feasibility of Fog Computing. arXiv:cs.DC/1701.05451

[86] Ashish Venkat and Dean M. Tullsen. 2014. Harnessing ISA Diversity: Design of a heterogeneous-ISA Chip Multiprocessor. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture* (Minneapolis, Minnesota, USA) *(ISCA '14)*. IEEE Press, Piscataway, NJ, USA, 121–132. http://dl.acm.org/citation.cfm?id=2665671.2665692

[87] VMware. 2019. nanoEDGE. https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/products/vsan/vmw-nanoedge-sddc-solution.pdf.

[88] SSRG VT. 2020. Popcorn Linux Project Website. http://www.popcornlinux.org.

[89] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. 2017. Ramblr: Making Reassembly Great Again.. In *NDSS*.

[90] Andrew Waterman, Yunsup Lee, David A Patterson, and Krste Asanovic. 2011. The risc-v instruction set manual, volume i: Base user-level isa. *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2011-62* (2011).

[91] Wave Computing. 2019. Wave Computing Unveils New Licensable 64-Bit AI IP Platform to Enable High-Speed Inferencing and Training in Edge Applications. https://wavecomp.ai/wave-computing-unveils-new-licensable-64-bit-ai-ip-platform-to-enable-high-speed-inferencing-and-training-in-edge-applications/.

[92] Richard York. 2002. Benchmarking in context: Dhrystone. *ARM, March* (2002).

[93] Aleksandr Zavodovski, Nitinder Mohan, Suzan Bayhan, Walter Wong, and Jussi Kangasharju. 2018. ICON: Intelligent Container Overlays. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks* (Redmond, WA, USA) *(HotNets '18)*. ACM, New York, NY, USA, 15–21. https://doi.org/10.1145/3286062.3286065