H-Container: Enabling Heterogeneous-ISA Container Migration in Edge Computing

TONG XING and ANTONIO BARBALACE, The University of Edinburgh
PIERRE OLIVIER, University of Manchester
MOHAMED L. KARAOUI, Huawei Research Paris
WEI WANG, Brookhaven National Laboratory
BINOY RAVINDRAN, Virginia Tech

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 should run on edge nodes that are physically as close as possible to their clients. Thus, when the physical location of a client changes, a service should migrate between edge nodes to maintain proximity. Differently from cloud nodes, edge nodes integrate CPUs of different Instruction Set Architectures (ISAs), hence a program natively compiled for a given ISA cannot migrate to a server equipped with a CPU of a different ISA. This hinders migration to the closest node.

We introduce H-Container, a system that 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 user's 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 – it can migrate most Linux software, including server applications and dynamically linked binaries. H-Container targets Linux and its already-compiled executables, adopts LLVM, extends CRIU, and integrates with Docker. Experiments demonstrate that H-Container adds no overheads during program execution, while 10 – 100 ms are added during migration. Furthermore, we show the benefits of H-Container in real-world scenarios, demonstrating, for example, up to 94% increase in Redis throughput when client/server proximity is maintained through heterogeneous container migration.

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

5

T. Xing, A. Barbalace, W. Wang performed part of this work when at Stevens Institute of Technology.

This article is based on a paper published at VEE 2020 [11]. The main delta over this work is the development of H-Container support for Docker and live-migration. Hence, the principal additions in this article include the presentation of the design, implementation, and evaluation of H-Container's Docker and live-migration support. Moreover, the evaluation of H-Container on an additional set of benchmarks (including OCR and RANSAC) and platforms (specifically, AWS) has been added to this version of the paper. This article also includes several other minor additions.

This work is funded in part by the UK's EPSRC grant EP/V012134/1 (UniFaaS) and in part by the US Office of Naval Research under grants N00014-16-1-2104, N00014-16-1-2711, N00014-18-1-2022, and N00014-19-1-2493.

Authors' addresses: T. Xing and A. Barbalace, The University of Edinburgh; emails: {tong.xing, abarbala}@ed.ac.uk; P. Olivier, University of Manchester; email: pierre.olivier@manchester.ac.uk; M. L. Karaoui, Huawei Research Paris; email: mohamed.karaoui@huawei.com; W. Wang, Brookhaven National Laboratory; email: wwang2@bnl.gov; B. Ravindran, Virginia Tech; email: binoy@vt.edu.

0734-2071/2022/07-ART5 $15.00
https://doi.org/10.1145/3524452

Additional Key Words and Phrases: Edge, heterogeneous ISA, containers, migration

ACM Reference format:
Tong Xing, Antonio Barbalace, Pierre Olivier, Mohamed L. Karaoui, Wei Wang, and Binoy Ravindran. 2022. H-Container: Enabling Heterogeneous-ISA Container Migration in Edge Computing. *ACM Trans. Comput. Syst.* 39, 1–4, Article 5 (July 2022), 36 pages.
https://doi.org/10.1145/3524452

1 INTRODUCTION

Edge computing [86, 89] is an emerging computing paradigm that advocates moving computations typically performed by centralized cloud computing services onto distributed nodes physically closer to the end-user or data production source—at the edge. In addition to relieving the data center from computation load, and the network from data traffic, edge servers' physical proximity to clients ensures the lowest latencies [21, 22]. Thus, edge computing's application domains are plentiful: mobile computation offload, failure resilience, IoT privacy [45, 86], real-time analytics [40, 69], cognitive assistance [87], just-in-time instantiation [70], gaming [23, 36], and so on.

In this context, the need for runtime software migration between machines has been identified as a critical feature [40, 45, 69, 74, 101]. 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 the near future, and so on. Although stateless applications can be easily restarted between nodes, stateful ones generally cannot—they must implement persistence (cf. Redis), which requires costly application-specific software support. Generic migration is thus a better option, as it supports any application. Moreover, studies have shown migration to be faster than stateless restarting [60, 81].

Differently from the cloud, computers at the edge are wildly heterogeneous [41, 50, 61, 101], ranging from micro/nano/pico clusters [31, 80, 103] to routing devices [70, 75, 96]. They span from servers to embedded computers with CPUs of diverse **Instruction Set Architectures (ISAs)**: x86 [51], but also ARM [47, 59, 62], with other ISAs announced [46, 91, 106]. This heterogeneity was recognized by major software players, such as Docker, which now support multi-ISA deployments [9, 76, 76, 79]. The ISA-heterogeneity is a colossal barrier to agile migration of software at the edge, as a (stateful) service natively compiled for an ISA is unable to migrate to a machine of a different ISA. The intuition behind this article is that *enabling heterogeneous-ISA migration maximizes the number of potential migration targets and increases the chances of optimal placement for latency-sensitive services.*

Similarly to the cloud, physical machines at the edge are shared between mutually untrusting tenants. Therefore, when deploying services at the edge, a form of virtualization is necessary. Furthermore, computing resources in edge nodes are rather constrained compared to the abundance in the cloud. Thus, in that context, lightweight virtualization technologies are more compelling than traditional heavyweight **Virtual Machines (VMs)** [45]. Containers are a form of lightweight OS-level virtualization, offering near-native performance [18], fast invocation latencies, and low memory footprints. Because of these characteristics, they are increasingly popular at the edge [30, 45, 69, 95, 110].

Containers can migrate at runtime across homogeneous nodes in production environments [37], but not on heterogeneous-ISA ones—a requirement of the edge. Runtime cross-ISA migration has been studied [12, 24, 29, 32, 42, 44, 58, 77, 102] for OS-level processes [12], unikernel virtual machines [77], or Java applications [42]. 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 installation of

either a custom kernel [12], hypervisor [77], or language VM [42]. These are only compatible with a handful of machines and are are unlikely to provide Third, they implement application's state transfer techniques that may not handle all application's residual dependencies [25] such as socket descriptors—thus, they may not support network applications such as servers. Finally, dynamically linked binaries, massively more widespread than static ones [99], are not supported.

H-Container. This article 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 a group of OS-isolated processes, e.g., a Linux container, between machines of different ISAs to enhance the flexibility of edge applications. Focusing on the widespread x86-64 and arm64 ISAs, we enhance the Linux's **Checkpoint and Restore 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 upon which homogeneous-ISA container migration is built, 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, which enables support for server applications in H-Container.

Furthermore, contrary to existing works on heterogeneous-ISA migration [12, 77], H-Container is easily deployable: It does not require access to applications' source code and supports dynamically linked binaries. H-Container introduces a transpiler that takes natively compiled application binaries and converts them into an **Intermediate Representation (IR)** using a disassembler and a lifting tool [34]. Next, it instruments and compiles the IR into natively compiled binaries for multiple ISAs. Such binaries are built with additional metadata that enables cross-ISA migration at runtime. 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/development boards, 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. In summary, the article 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 application for cross-ISA migration without source code;
- The overhead and performance *evaluation* of H-Container.

This article is organized as follows: Section 2 presents our motivations, and Section 3 lays out background information about software migration and executable binary transformation. We describe our system model in Section 4. The design principles of H-Container are presented in Section 5, and its implementation in Section 6. We evaluate H-Container in Section 7, present related works in Section 9, and conclude in Section 10.

2 MOTIVATION

Edge is a rather new distributed computing technology that enables Internet services to run physically as close as possible to users, hence with low service latencies—in most cases latency at the

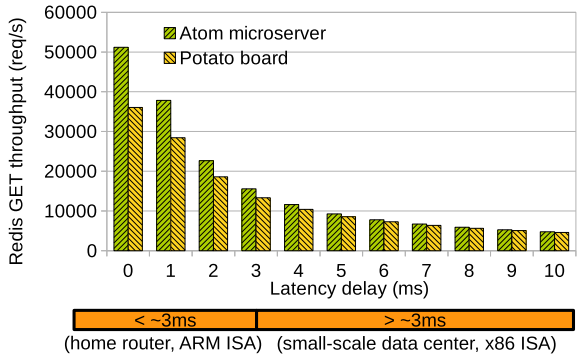

Fig. 1. Redis throughput when varying client-server network latency.

edge is lower than 10 ms [21, 22], with companies already targeting 5 ms [98]. To run services physically as close as possible to users, today's edge deployments may feature limited computing resources—constrained number of machines cased in a smaller volume with reduced energy consumption because space and power are on a budget, encompassing from micro/nano/pico clusters to router devices, including home routers [70, 75, 96]. Thus, edge distinguishes from cloud for its remarked heterogeneity in terms of computing hardware, resources, form factors, processor classes, and ISAs [41, 50, 61, 101]—e.g., if home routers are ARM-based, then enterprise ones are x86-powered. Although current micro/nano clusters [31, 103] are mostly equipped with x86 servers, ARM servers for the edge are emerging [47, 59, 62] and ARM pico clusters exist already [80]. Hence, we foresee ISA heterogeneity among edge data centers as well as in the same data center (something that is already the case in cloud data centers, i.e., AWS [7]).

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). Similarly, due to resource fragmentation into an edge data center, a node of the same ISA of the machine where the application is currently running may be already fully loaded, while nodes of a different ISAs have spare capacity. Sticking to the same ISA for the target machine is thus sub-optimal towards the objective of reducing latency.

Why bring 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 (RedisEdge), a prime example of edge application [83], 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 2.5 GHz Atom C2758 microserver and a librecomputer LePotato arm64 1.5 GHz single board computer. A third machine, the edge client, is connected to both edge nodes on the same Ethernet LAN (Section 7 justifies such setup). The base latency between the client and the edge nodes is less than 500 μs 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 [5, 21–23]. 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 loses 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

Fig. 2. Redis persistence vs. CRIU.

Fig. 3. SQLite persistence vs. CRIU (time).

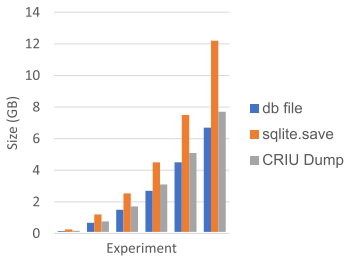

Fig. 4. SQLite persistence vs. CRIU (size).

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 [70].

At the same time, latency-sensitive server applications on the edge migrate between edge nodes to follow an end-user [40, 45, 69, 74, 101]. Thus, it is fundamental to maximize the number of edge nodes target of migration so a server can be as close as possible to the end-user—hence, 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 (application-agnostic) migration vs. ad hoc state transfer. Applications running at the edge are a mix of stateless and stateful ones. The former, which includes data filtering, triggering, and so on, can simply be stopped on one node and restarted on another one, because there is no state to carry over. In these cases, migration is not strictly needed although in some scenarios initialization times can be substantial, e.g., FaaS runtime initialization time [100]—making migration attractive. Migration is also interesting when the restart cost is high; for example, when the application components are not stored locally, at the edge, but have to be downloaded from somewhere else, e.g., the cloud.

Concerning stateful services, stopping a stateful service means losing all of its data, if it does not implement persistency. Only certain services implement persistency, because supporting this feature requires a non-negligible, application-specific, development effort. Thus, an application-agnostic migration mechanism is preferable. Additionally, we observed that application-agnostic migration, especially live migration, can be superior to ad hoc application-specific persistency methods in terms of performance. This is true for at least a certain class of applications and it is due to the additional time required by the application for data marshaling and serialization. Figure 2 compares the time required by Redis persistence (memory-to/from-disk) vs. CRIU checkpoint/restore, without compression, for different **database (DB)** sizes. CRIU is up to 25 times faster, and when enabling compression it is even faster—this is because of its trivial dumping of the memory state instead of Redis state serialization. Figure 3 compares the time required by SQLite persistence (save and read to/from disk) vs. CRIU checkpoint/restore, without compression, for different DB sizes. Once again CRIU is faster, by up to 23× (on restore). Figure 4 shows the size of the generated checkpoint or persistency state, clearly CRIU creates a file of a similar size of the original database, while SQLite persistency state may be double the size of the original database file. Larger sizes means larger migration times. To conclude, we demonstrated on two different database applications that migration can be faster than ad hoc state transfer mechanisms, in addition to being more generic and having shorter downtimes.

3 BACKGROUND

This section provides background information about software migration, container migration with CRIU, and static binary rewriting.

3.1 Software Migration During Execution

The need to migrate software at runtime between different computers dates back to the first multi-computer networks. Software has been migrated at the granularity of threads [12, 14], processes [12, 14, 53], groups of processes [78], or entire **operating systems (OS)** and its running applications [25]. 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-mastered technology. It relies on a virtual model of the hardware, the **Virtual Machine (VM)**, and transfers its state between computers. Among other components, it includes the content of the VM's physical RAM that incorporates the OS and applications state. Process-level migration identifies and transfers only the state related to a specific program. It includes the program's address space, CPU registers, and part of the OS state related to the process itself—e.g., open file descriptors. Container migration applies process-level migration to the group of processes populating a container [37].

Migration mechanisms include checkpoint/restore [37], in which a full dump of the software state is created and then restored, and live migration [25], 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 [64] (pre-copy) or transferred on-demand after the restore phase [49, 63] (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 [25]—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 and Restore In User-space (CRIU)** tool [37].

Figure 5 illustrates the major steps involved in CRIU process migration. The *Checkpoint* operation produces a set of image files that are transferred to the destination machine and used by the *restore* operation to resume the application execution. To produce the image files, CRIU has first to pause the target process in a consistent state. It does so by using the *compel* library, which injects within the target process some code that snapshots the process' resources. CRIU comes as a single binary application, `criu`, which can be used to checkpoint/dump the state of a process, and to restore/reload that state. Moreover, CRIU includes a tool to manipulate the state dump (i.e., the image files), named `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 [15] is adopted for this purpose, but other tools such as Podman [3] and LXD [2] exist. Docker, Podman, and LXD support deployments on multiple architectures, but only LXD natively supports migration. However, LXD focuses on an entire Linux distribution not at the application level.

Heterogeneous-ISA Migration. In the 80's and 90's, multiple projects studied the migration of applications in a network of heterogeneous machines [8, 53, 92]. Such works used to convert the entire migrated state of the application, including data, from one ISA format to another, thus

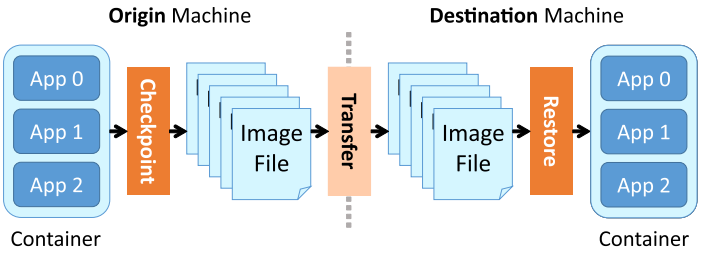

Fig. 5. Homogeneous-ISA container migration process.

involving large overheads. Recent works, such as HSA [85], Venkat et al. [102], or Popcorn Linux [12], 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 special-purpose 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 are able to support the same primitive data types' sizes, alignment constraints, and endianness. Moreover, to preserve the validity of pointers across migration, every function, variable, or other program symbol should be located at the same virtual address irrespective of the 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.

Given that 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 [12]. 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 stack is converted right before reaching a migration point and it is realized via code injected by the Popcorn Linux compiler, which also adds a runtime library for the transformation itself. Venkat et al. [102] seem to manually convert the state that is not in a uniform format either.

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 [57], rejuvenate the topic. In this article, 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 [93, 104], 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 [34] (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 [19]). 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 6. A team of developers implements 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 consist of several servers, but at least one server should run as close as possible to the client part of the application, i.e., on the edge. For clarity, in this article, we focus on applications featuring only a server running on the edge—i.e., none on the cloud, which is a common use case in previous works [45, 66].

The model defines an applications' repository residing in a cloud data center (e.g., Docker Hub), as well as multiple edge segments, each maintaining a local repository of applications (e.g., Docker cache). This article 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 [15]) and an orchestrator (e.g., Kubernetes [48], OpenShift [90]) administer the edge-local repository and are responsible of the deployment of the server on each different edge node. Such nodes can be equipped with different ISA CPUs.

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 article.

Horizontal and Vertical Migration. Within this article, we focus on server applications migration across different edge nodes. Therefore, we refer to this as horizontal migration. This term is defined as opposed to vertical migration, where applications migrate between datacenter, fog/edge, and devices, which has been referred to before as *cloud offload* [44]. It is worth noting that the technology we propose is also suitable to be implemented in vertical migration schemes as well: Indeed, the ISA heterogeneity is also remarkable between the machines present in the datacenter, at the edge/fog, and the end-user devices. However, we scope that out as future work.

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:

- Enabling application software to transparently execute on and migrate across edge nodes of heterogeneous ISAs;
- Offering a high-degree of portability between edge machines and genericity in terms of supported software;
- Being easy to deploy, maintain, and manage;
- Being of minimal performance overhead.

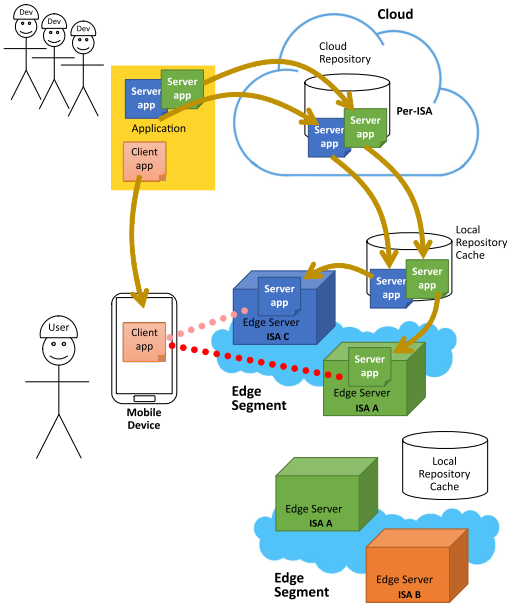

Fig. 6. Classic deployment (e.g., Docker).

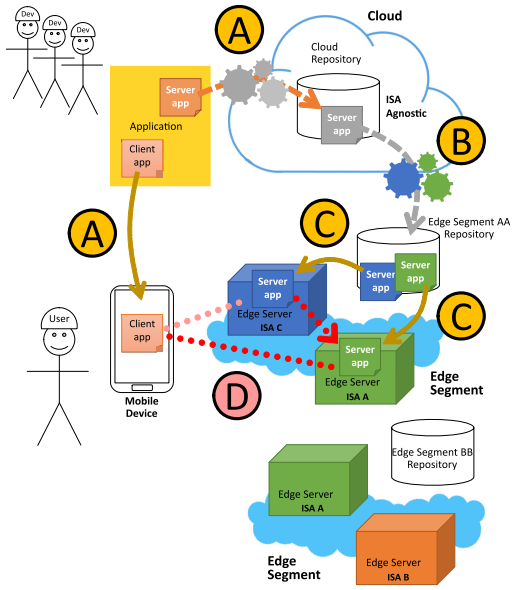

Fig. 7. H-Container system model and architecture.

Therefore, we designed the H-Container architecture that:

- is fully implemented in user-space to maximize portability, ease of deployment and maintenance, and avoids relying on specific kernel versions or patches;
- targets OS-level lightweight virtualization systems—i.e., containers, jails, zones, which are perfect candidates for edge deployments due to their minimal overheads;
- is generic, as it does not require access to the application's source code—binaries are automatically re-purposed for migration among ISA-different nodes when moved between edge segments;
- minimizes application's state transformation when runtime migrates between different ISA processors 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 7:

(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 LLVM IR by H-Container;
(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/restore or live migration), and its state sent to the node that is closer to the user Ⓓ;

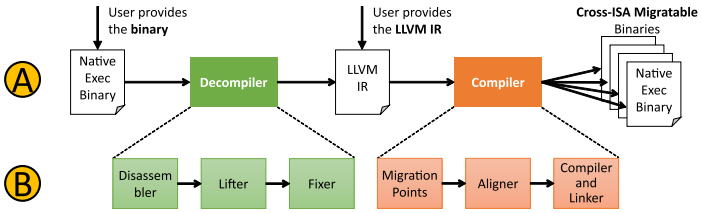

Fig. 8. (A) High-level design of the H-Container *automatic executable transformation* infrastructure; (B) breakdown of the decompiler and compiler blocks.

(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 to migrate, when and where.

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

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. However, in an edge setting, CPU ISAs are more varied and the application developer does not necessarily have the knowledge of all the ISAs implemented by the machines his code may execute upon. 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 [105] questions such solution. Moreover, pre-compiling for all possible ISAs is not practical for legacy applications for which the source code may not exist anymore, or the porting cost to enable recompilation with modern toolchains and libraries may be 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, and so on.

Our architecture (Figure 7) stores applications in an **intermediate representation (IR)** that is ISA-agnostic, specifically LLVM IR, but does not force application developers to compile applications in IR. Therefore, we provide the possibility to upload a natively compiled binary, and H-Container will transform it into IR. This is depicted in Figure 8 Ⓐ. Note that an IR is also a good compromise, because it does not require a company to expose its own code, which is intellectual property, to cloud/edge providers. Moreover, we propose that it is among the responsibilities of the edge/cloud orchestrator to determine what are the binaries that may be needed by a specific application in the near feature and automatically compile them from the IR.

Transforming the code from native to LLVM IR (Decompiler) is the first step of our automatic executable binary transformer. The second step in Figure 8 Ⓐ 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, and so on. 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.

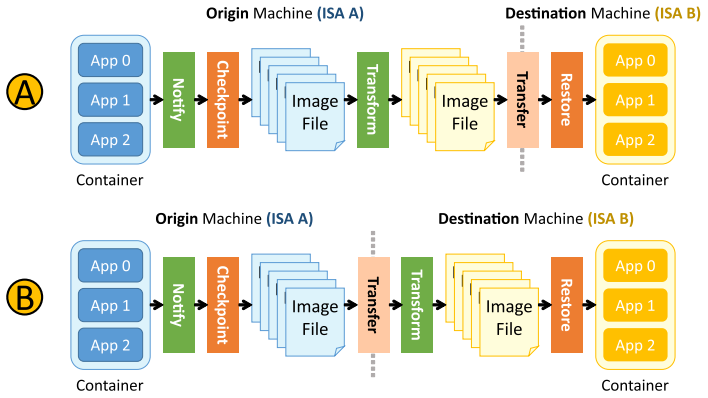

Fig. 9. The main steps in container migration and their outputs. Origin machine is of blue ISA, while destination is yellow. Green boxes are added by H-Container.

The main idea behind heterogeneous-ISA migration is very similar to Popcorn Linux [12]: The entire address space has the same layout on multiple ISAs, all code sections of an ISA A are overlayed with the ones of an ISA B, and code is functionally equivalent (each function produces the same output with the same input); because **all** symbols are at the same virtual address for each binary (functions or variables) and have the same size and padding, any function or variable pointer can be used among different ISAs. Therefore, 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 inserted by the H-Container compiler within custom ELF sections of the binary, thus, always present and correct. 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 infrastructures available in modern OSes, to maximize compatibility and potential adoption of our system. 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.

All steps required by heterogeneous container migration are depicted in Figure 9. The first step in heterogeneous container migration is to *notify* all applications in the container that they have 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 arbitrary points (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 leveraging existent tools. The entire container state is dumped into *image files*. Differently from migration among homogeneous CPUs, heterogeneous migration requires these image files to be transformed from the origin ISA to the destination ISA. In the *transform* step, H-Container rewrites selected part of the checkpoint consisting in the image files so it can be successfully restored on the target machine. To minimize the overhead of this step, although H-Container implements a mostly common address space layout, each thread's stack is kept in the native format. Hence, the *transform* step requires rewriting each thread's stack in the application dump to suit the format corresponding to the target ISA.

The *transform* step produces a new version of the checkpoint (image files). This version is sent to the destination machine (*transfer* step), which after reception can restore the checkpoint and resume the execution of the container. It is worth noting that the transform step does not have to be executed on the origin machine (Figure 9 Ⓐ); in fact, it may run on the destination machine as well (Figure 9 Ⓑ), wherever the transformation step runs faster. In fact, on the edge, where to transform the state can be decided dynamically, e.g., based on each node's transformation performance.

These mechanisms can be leveraged by an 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 article, and many related works exist (see Section 9).

6 IMPLEMENTATION

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

In Linux, containers are based on namespaces [55] and **control groups (cgroups)** [17]. This work extends the CRIU project to migrate an application between computers of diverse ISAs. 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 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 8 Ⓐ, H-Container is composed of two main components: a decompiler and a compiler. Their internals are illustrated in Figure 8 Ⓑ, we describe the implementation details below. These components counts ∼750 LoC for the Decompiler part, mostly bash and Python, and ∼890 LoC of modifications to 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 and include at least support for riscv64. Note that this step is not on the critical path—decompilation and recompilation are not executed at runtime, but before application deployment, hence it can run as a batch job in the data center. Because the execution time of this operation is not on the critical path, we did not measure it.

Decompiler. H-Container exploits McSema for binary disassembling and lifting. 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 [43], RetDec [56], rev.ng [33]. 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 several issues, two of them highly critical. 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 [12, 77] supports dynamically linked binaries even though dynamic ones are much more widespread compared to static—99% of the ELF executable binaries in a modern Linux distribution are dynamically linked [99].

The second problem regards 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 itself 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()`, and so on. 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 does not reinvent the wheel but capitalizes on 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 at function boundaries. A migration point requires the addition of only a couple of assembly instructions to call the `migrate()` Popcorn's runtime function. Thanks to the minimal number of instructions added per migration point, in our experience working with the Popcorn Linux's compiler framework, we never witnessed either binary/memory bloat or performance overhead due to the presence of migration points. Furthermore, this feature can be disabled, and migration points can be inserted manually. In fact, for some multithreaded programs, the automatic insertion of migration points may potentially induce deadlocks when a thread reaches a migration point while holding a lock that prevents other threads waiting on the said lock from ever reaching a migration point. Although this can be solved by patching `unlock()`-like functions with migration points, as we patch all existent locking libraries, we introduced the possibility to manually add migration points anywhere in the code (a simple call to a library functions) to avoid such scenarios. Note that a deadlock can only happen before the checkpoint (cf. Figure 9), therefore, standard debugging tools like GDB can be used to identify if the described situation has happened—i.e., one or more thread migrated out and one or more thread are either busy-waiting or waiting on a blocking syscall.

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 instruct the linker to align global symbols (i.e., functions, variables) at the same address among ISAs. This tool was entirely rewritten in the context of this work and it is generic because 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 to be triggered from the notify tool discussed above. Finally, Popcorn's musl libc was extended to let the C library initialization code enforce the same virtual address space aperture on every architecture. Before, it was enforced by the OS kernel.

6.2 Heterogeneous Migration

H-Container introduces **Heterogeneous Checkpoint and Restore In User-space (HetCRIU)**. HetCRIU extends CRIU to support the design in Figure 9, 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 target process; (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 and (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* (details are discussed below). The version with `crit` accounts for ~1,820 LoC, while the *all-in-one* requires additional ~1,210 LoC.

Notify Step. The "Notify" step (cf. Figure 9) is implemented as an additional CRIU tool called `popcorn-notify` (or notify). Popcorn-notify does not 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 9, 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, avoiding extensive modification to the original source code that may be hard to streamline.

Transform Step. We implemented the *transform* step (cf. Figure 9) as an extension of `crit` by adding a new command line argument, `recode`. 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 CPU architectures. Conversion includes the remapping of arithmetic, floating point, and miscellaneous registers content between CPU architectures, the adjustment of VDSO, `vvar`, and `vsyscall` areas, the fixing of the CPU architecture name and executable name, and so on. Additionally to those, container-related modifications are required. These include the updates of all per-session limits (i.e., what is controllable with `ulimit`), and the modifications to the thread to CPUs mappings, as different machines may have a different number of CPUs.

Unfortunately, `crit` is characterized by a large overhead, due to Python initialization and file copies. We break down that overhead in Section 7.2. Hence, we implemented another version of HetCRIU that integrates all the code we added in `crit recode` into the main CRIU binary itself, called *all-in-one*. In this way, dump files are also opened and written once instead of twice—which improves performance of our tool.

Checkpoint/restore and Live-migration. In addition to its basic checkpoint/restore functionality, the original version of CRIU comes with several implementations of live-migration. Live-migration can reduce the user-perceived downtime to less than a second in our setups. The *all-in-one* version of HetCRIU fully supports CRIU's pre-dump live-migration (pre-copy), including CRIU's page server. HetCRIU live-migration periodically checkpoints, transforms, and transfers the container state from the origin to the destination machine. When the amount of state to be transferred falls below a threshold, `popcorn-notify` is called and a last checkpoint is taken, transformed, and transferred. Finally, on the destination machine, the container is restarted.

6.3 Integration and Managed Deployment

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 requested variant of the *transform* operation. Everything else works exactly in the same way as with the original version of CRIU, which maximizes compatibility.

Similarly to CRIU, HetCRIU is a tool that focuses on checkpoint/restore and eventually live-migration, but not on the deployment or scheduling/mapping of software among machines in a data-center. Therefore, it is the user that has to transfer the image files between the origin and destination machines. We currently implement this step using a distributed file system, like NFS, as well as with an in-house developed utility that automates the image files transfer(s). Note that this applies to checkpoint/restore and live-migration.

To support automated application deployment and scheduling/mapping as sketched in our model in Figure 7, HetCRIU comes with an entire suite of extensions for Docker, which enables the seamless integration of heterogeneous-ISA migration with the existent multi-arch feature [38] of Docker. The development of such extensions was mainly motivated by the fact that the current version of Docker supports checkpoint/restore only as an experimental feature, designed to pause and resume a container on the same machine—thus, lacking features to "transfer" checkpoint image files among different machines. This required the development of several utilities and patches for transferring the image files or share them over a distributed file system. In future works, we plan to extend orchestrator software to handle heterogeneous-ISA machines migration.

Another issue we addressed during the integration with Docker arose by strict security rule checks with Docker. Although security rule checks can be relaxed in Docker (with the `--cap-add=ALL` option for the `docker run` command), that is not acceptable in multi-tenant production environments. Hence, we first identified that such security issues were mainly due to different Linux kernel configurations and then coded a set of "safe" transformation rules to enable container migration among different Linux kernels with different (cgroup) configurations. Obviously, with an identical Linux kernel version and security configuration, which is possible to achieve for several ISAs (cf. AWS experiments in Section 7), no transformation is needed.

7 EVALUATION

We thoroughly evaluate H-Container to answer the following questions:

(1) What are the space and runtime overheads added by H-Container transpiler?
(2) What are the runtime overheads added by HetCRIU (*all-in-one* and `crit recode`)?
(3) What are the benefits in terms of latency and throughput introduced by H-Container on a realistic edge testbed for diverse real-world applications and various request sizes?
(4) What are the advantages introduced by HetCRIU live-migration compared to HetCRIU checkpoint/restore, in particular in terms of service downtime?

(5) Does H-Container's performance behavior vary when, instead of a bare-metal setup, containers are being run into virtual machines in IaaS offer?

Thus, herein, we first introduce our experimental (bare-metal) hardware and software setups in Section 7.1, then we characterize our experimental setup and evaluate H-Container's overheads (focusing on memory/compute-intensive applications to minimize OS-induced delays) in Section 7.2, answering questions (1) and (2). After that, to demonstrate the benefits that H-Container can bring to real-world Internet services at the edge—questions (3) and (4)—we use a set of edge applications from various domains including key-value store, compression, game server, optical character recognition, and computer vision in Section 7.3. We answer question (5) in Section 7.4, where we first introduce a characterization of an IaaS setup and then we report the benefits introduced by H-Container when running on it.

7.1 Experimental Setup

H-Container has been tested on a variety of ARM64 and x86-64 computers to assess its deployability and performance, with machines ranging from embedded platforms to servers, including **Amazon Web Services (AWS)** instances [7]. We report the key results on a handful of platforms[1] whose hardware and software are described below.

Hardware. Other than the system setup described in Section 2, composed by two embedded-class computers, in this section, we will present results on two other system setups to demonstrate the general applicability of H-Container. One is a setup composed of a workstation and an embedded board. The workstation (Dell PowerEdge R430) mounts a single Intel Xeon E5-2620 v4 clocked at 2.1 GHz, 8 dual-threaded cores, 16 GB of RAM, and dual 1 GbE connections. The embedded board (FireFly ROC-RK3328-CC) is equipped with a Rockchip CPU RK3328 with 4 Cortex-A53 cores clocked at 1.5 GHz, 4 GB of RAM, and single 10/100/1,000MbE connection. The other setup consists of two server-grade machines: the first being a Dell PowerEdge R7425 mounting two AMD EPYC 7451 clocked at 2.3 GHz, for a total of 48 cores and 96 threads, 256 GB of RAM, dual 1 GbE and dual 40 GbE; the second being a Gigabyte R150-T62 equipped with two Cavium ThunderX1 clocked at 2.0 GHz, for a total of 96 cores, with 256 MB of RAM, single 1 GbE (over USB 3.0) and quad 40 GbE. For each setup, we use an additional third machine (more details below).

All machines are interconnected via a 1 GbE switch. 1 GbE has been selected as it is representative of the likely connection speed between edge nodes and users, specifically (a) current widely adopted 4G broadband cellular network standard defines 1 Gbit/s as the maximum communication speed (see ITU-R, Report M.2134) while experimental works found speeds are lower [108], (b) the fastest 5 GHz WiFi antennas can do up to 1.73 Gbit/s (802.11ac-2013) on physical link-layer, but the actual measured speeds hardly surpass 1 Gbit [20, 108], (c) most houses, offices, and fabric buildings are wired with 1 GbE, while a modern WiFi router 802.11ac standard can provide 867 Mbit/s with one antenna (160 Mhz). Moreover, following Reference [1], 1GbE is Ericsson's forecasted typical backhaul capacity for a high-capacity cell site in 2022; therefore, and although on the conservative side, we used 1 GbE as edge-node to edge-node speed.

Through the end of this section, we introduce a fourth setup based on AWS. Overall, we believe that the multiple hardware setups we consider in this evaluation cover all the spectrum of machines that can be found at the edge, from embedded to server-class machines, with diverse networking capabilities that represent a real-world range of latencies and throughputs.

[1]Additional results are available online [4, 107].

Software. H-Container is Linux-version neutral and only requires that the kernel supports 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 RK3328 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 uses LLVM/clang 3.7.1 (git commit fd578a9).

To characterize H-Container, we use a wide variety of benchmarks, discussed in each of the following subsections.

7.2 Overheads Evaluation

We characterized H-Container execution time overheads on a set of benchmarks collected from different projects. Based on previous works [6, 45, 52, 67–69, 95, 110], we believe that such a set of benchmarks may cover diverse compute/memory ratios of the workloads that can be found at the edge. The focus on compute/memory workloads is motivated by the necessity of identifying compiler/runtime overheads, not OS ones. Specifically, we used the **NAS Parallel Benchmarks (NPB)** [10], the Phoenix [82] MapReduce applications for shared memory, Linpack [35], and Dhrystone [109]. We run NPB is, ep, ft, cg, and bt, for different dataset sizes (class S, A, B, C). For Phoenix matrix-multiply (mm), pca, and kmeans, we also present numbers with different data input sizes. The behavior of these programs is representative of the other benchmarks in the suite. 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 to identify such overheads on different benchmarks. Figure 10 illustrates the results for Phoenix matrix multiplication, compiled without optimization (-O0) and with maximum optimizations (-O3) using gcc and clang (top graph and bottom graph). We decompiled it with the McSema-based decompiler and recompiled it back with the extended Popcorn compiler, 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 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, then it can be as fast 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 11. 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—to highlight the benefits of the proposed architecture that employs natively compiled binaries for both ISAs, instead of using one single native binary and emulate the others. The results are reported in Figure 12. 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× slower than native to up to 18.9× slower than native—while H-Container is up to 70.7% faster than native, thanks to McSema and

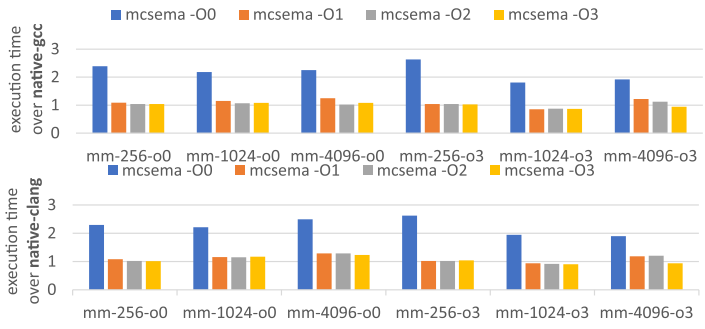

Fig. 10. Execution time ratio of a transformed executable versus the original one for Phoenix mm on the AMD EPYC. When the original is compiled with -O0 (first three clusters of bars in the graph) and with -O3 (latter three clusters of bars in the graph), varying the optimization level when recompiling after decompilation. Top graph gcc, bottom `clang`.

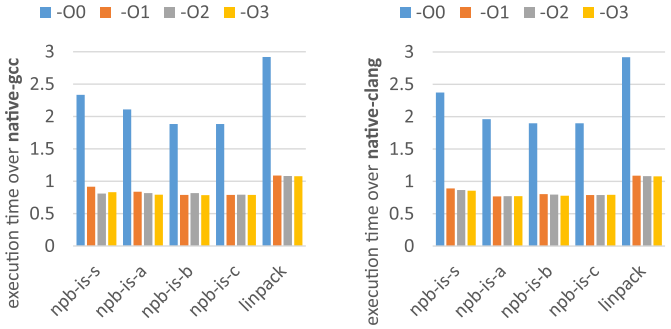

Fig. 11. Execution time ratio of a transformed executable versus the original one for npb-is and Linpack on AMD EPYC, varying the optimization level when recompiling after decompilation. Left graph gcc, right `clang`.

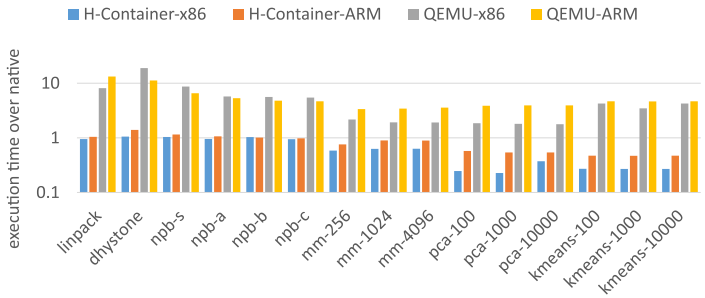

Fig. 12. Execution time ratio of a binary transformed with the decompiler-compiler infrastructure (mcsema) over the original and its execution on QEMU over original. For an ARM binary running on x86 (blue and gray) and an x86 binary running on ARM (orange and yellow). Note the logarithmic scale on the Y-axis.

LLVM's optimizations (O3). Note that the same experiments have been performed for statically and dynamically compiled binaries, and the results (averages) are similar (around 1% of difference).

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

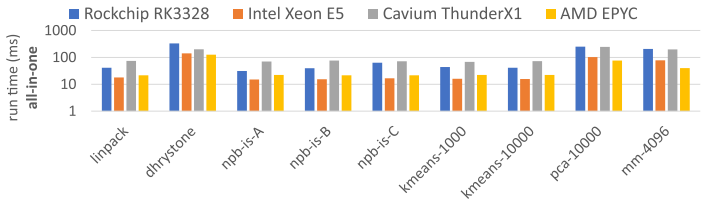

Fig. 13. Popcorn notify cost on ARM (Rockchip RK3328, Cavium ThunderX1) and x86 (Intel Xeon E5, AMD EPYC) for different benchmarks.

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 efficient to highlight computation overheads rather than IO ones. We present the results for Cavium ThunderX1, AMD EPYC, Rockchip RK3328, 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 13. Note that the results are the same independently of the state transformation method (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 RK3328), but x86 machines are equally fast. As stack transformation exhibits the same overheads as reported in Reference [12], 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 cost of two implementations of CRIU's exported state transformation on the same set of four machines and the same set of benchmarks. Figure 14 reports the breakdown of the cost to dump a checkpoint and to convert it to the destination architecture within the same program (CRIU). Most of these numbers are already reported by the `stats` file generated by CRIU; we added a field "dump_transform" that accounts for transforming the state from the origin to the destination architecture. Despite the total dump time being mostly proportional to the total amount of memory to checkpoint (see "dump_memwrite"), the transformation overhead ("dump_transform") is always lower than 1% of the total dump time 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 15. Differently from the previous one, this option is considerably more expensive than using *normal* CRIU: running `crit recode` requires from 2× to 17.5× additional time (respectively, for Phoenix matrix multiplication and Linpack). This is because `crit recode` needs to reload the image files and copy them, while there is no copy with all-in-one. Another issue with this tool is that because it is written in Python, it has a non-negligible 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*),

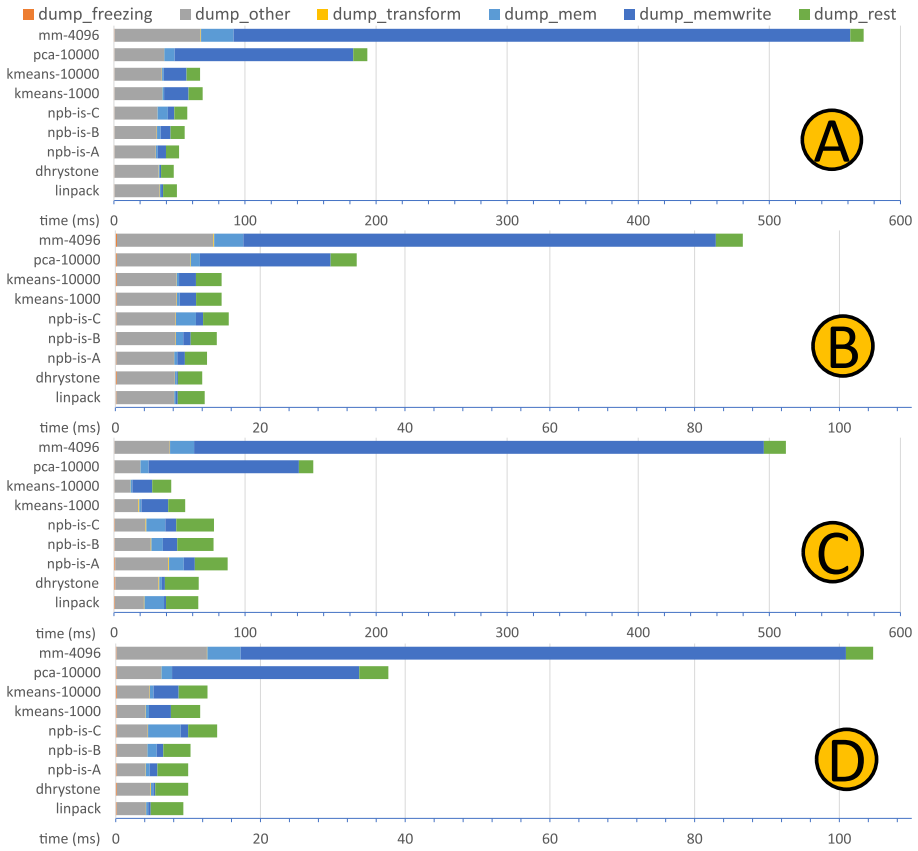

Fig. 14. Checkpoints breakdown on Cavium ThunderX1 (A), AMD EPYC (B), Rockchip RK3328 (C), and Intel Xeon E5 (D) for the *all-in-one* implementation of transform.

transformation time is negligible vs. CRIU time. When patching CRIU is not possible—crit recode, an additional overhead from 2×to 17.5× must be paid.

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 [83], Nginx [84], a compression server (Gzip), and a game server [23], **optical character recognition (OCR)** server based on GOCR [88], and a computer vision server based on the Ransac algorithm. We believe these services are representative of edge applications by virtue of previous works [6, 45, 52, 67–69, 72, 95, 110]. Further, we show that H-Container handles all kinds of applications, including network-serving ones, not supported by previous work [12, 77].

The proposed scenario is illustrated in Figure 16. 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 the *origin* node and closer to the *destination* node. 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 1,000 seconds (ca. 16 mins). Note that latency is the main objective metric in edge context. The impact on latency of the physical distance between

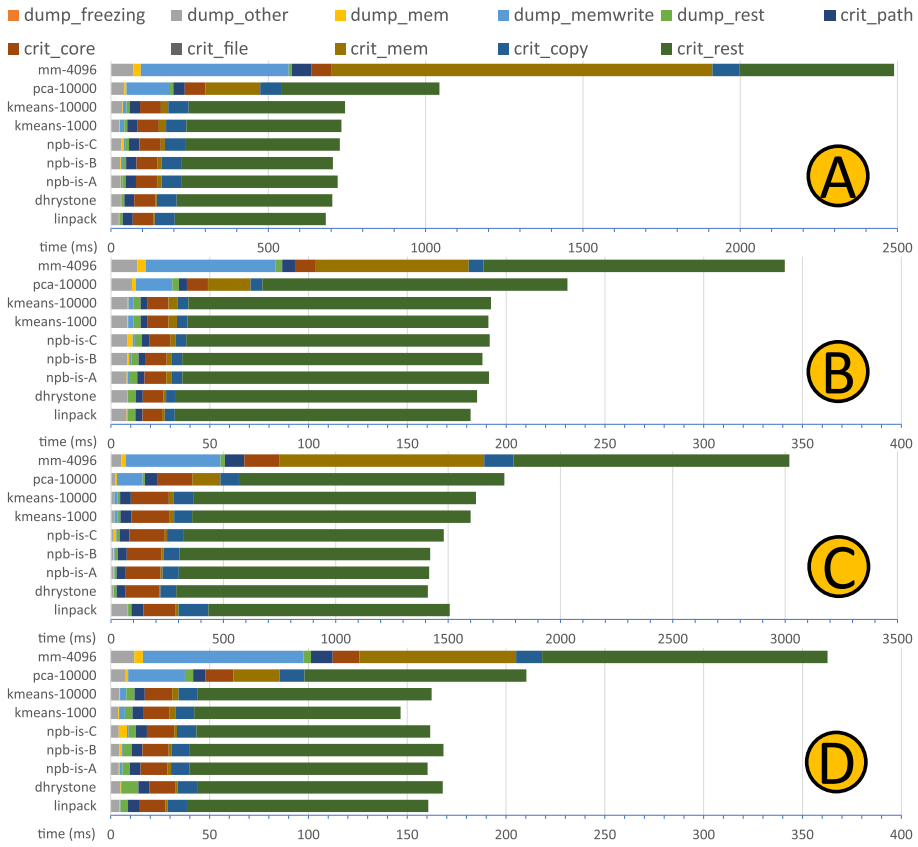

Fig. 15. Checkpoints breakdown on Cavium ThunderX1 (A), AMD EPYC (B), Rockchip RK3328 (C), and Intel Xeon E5 (D) for the `crit recode` implementation of transform.

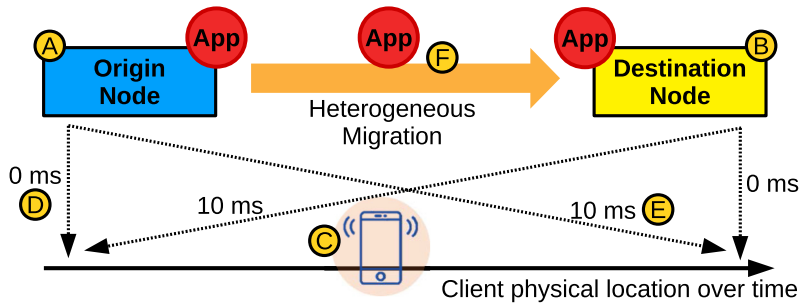

Fig. 16. Migrating latency-sensitive service following a moving end-user (client).

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* is increased by 1 ms and *destination*'s latency is decreased by 1 ms. These latency values are on par with what is expected at the edge [5, 21–23]—i.e., most of the time the latency at the edge is lower than 10 ms. To stress this point, note that in a recent study [5],

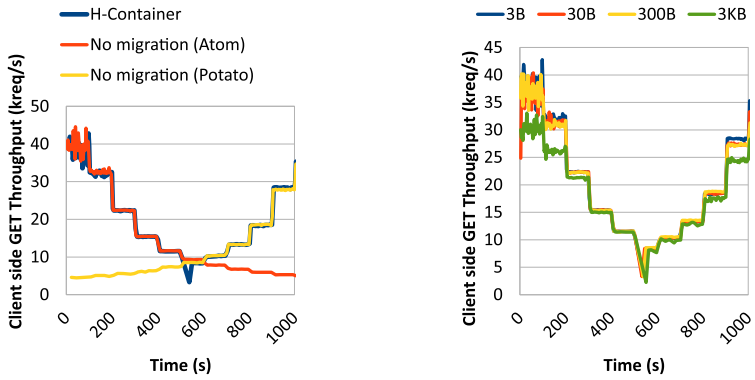

Fig. 17. Redis GET throughput with and without migration (left) and migration with different payloads (right). *origin* node is an Intel Atom, *destination* is the Potato board.

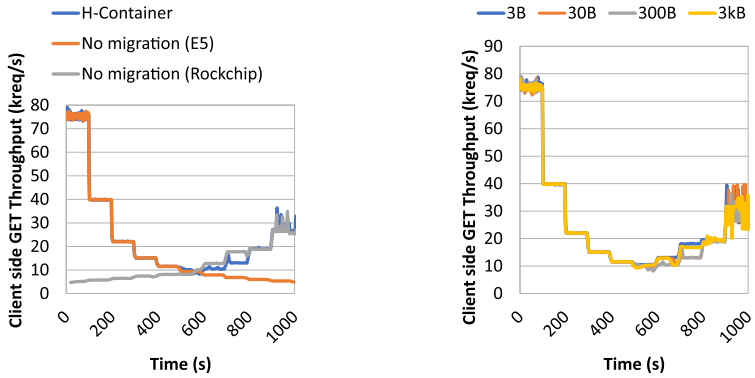

Fig. 18. Redis GET throughput with and without migration (left) and migration with different payloads (right). *origin* node is an Intel E5, *destination* is the Rockchip RK3328.

75% of the responders mentioned they would expect latencies inferior to 5 ms at the edge, and 90% expect latencies inferior to 10 ms. Thus, it is likely that edge datacenters are and will be deployed in such a way, justifying our choices.

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 latency, the actions/s, the recognized images/s, and total number of Ransac iterations/s, respectively.

We experimented with three scenarios. In the first two scenarios, we assume that heterogeneous migration is not possible, thus the server's location is fixed either on the *origin* or *destination*. We refer to these scenarios as "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 [97] that uses health check rules to automatically redirect requests to the node running the server.

Characterization of the Test Setups. In Tables 1 and 2, we report the measured network latencies for the AMD EPYC and Cavium ThunderX1 setup and the Intel E5 and Rockchip RK3328 setup,

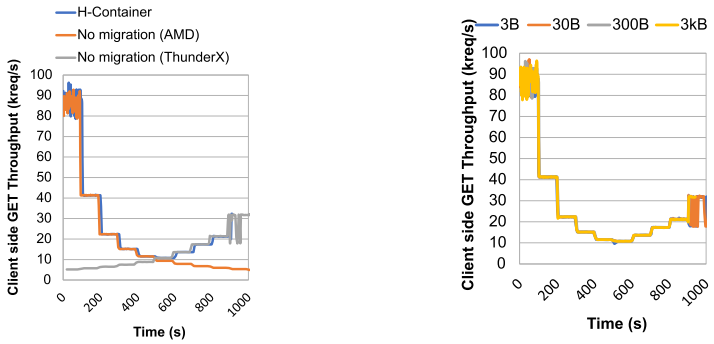

Fig. 19. Redis GET throughput with and without migration (left) and migration with different payloads (right). *origin* node is an AMD, *destination* is the ThunderX1.

Table 1. Measured Network Latencies (`iperf`) in the AMD EPYC and Cavium ThunderX1 Setup, Including the Client

From \ To	AMD	ThunderX1
AMD	x	0.32 ms
ThunderX1	0.32 ms	x
Client	0.14 ms	0.37 ms

Table 2. Measured Network Latencies (`iperf`) in the Intel E5 and Rockchip RK3328 Setup, Including the Client

From \ To	E5	Rockchip
E5	x	0.64 ms
Rockchip	0.65 ms	x
Client	0.13 ms	0.61 ms

Table 3. Binary and Checkpoint Image Sizes of Gzip, game, GOCR, and Ransac Servers

Application	Binary Size	Image Size
Gzip	24.9 MB	0.3 MB
game	24.6 MB	0.4 MB
GOCR	36.5 MB	0.5 MB
Ransac	24.7 MB	0.3 MB

Table 4. Binary and Checkpoint Image Sizes of Nginx with Different Number of Threads

Nginx	Binary Size	Image Size
4 threads	38.2 MB	3.3 MB
8 threads	38.2 MB	6.2 MB
16 threads	38.2 MB	12 MB
32 threads	38.2 MB	24 MB

respectively. For each setup, a third machine, the client node, has been added, and all nodes are connected via a 1 Gb Ethernet switch independently of the network technology used by each machine. For convenience, the Client node is an x86 server identical to the Intel E5 node.

Measurements have been taken with the `iperf` tool, which also reports a consistent bandwidth of 112 MB/s between each node in the first and second setup. Hence, as expected, the numbers show that the network latency is dominated by the hardware and software network latencies of the slowest node, which in the case of Table 1 is ThunderX1 due to the USB to Ethernet converter, while in the case of Table 2 is the Rockchip—which is notably slower than the E5.

For the compression server, game server, OCR server, and vision server, we report the binary and checkpoint image sizes in Table 3. The first is the size of the actual native compiled binary generated by the transpiler, while the latter is the size of the CRIU dump when no requests are served—the size increases by the size of the request during servicing. The sum of these two gives the total size of the data to be migrated between servers, which contributes to the downtime during migration. Note that the same numbers are recorded on all machines ARM or x86 (variation < 1%). Table 4 reports the same numbers for Nginx with a variable number of serving threads. Redis numbers are presented through the end of this section.

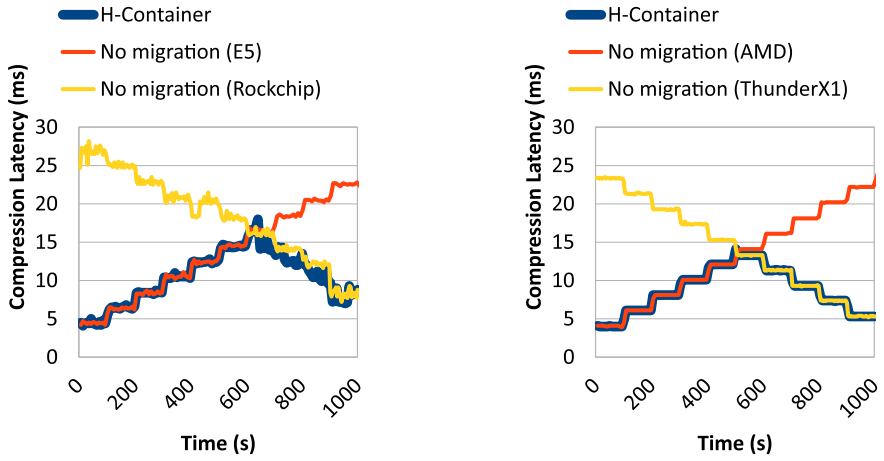

Fig. 20. Gzip server latency for 3 KB packets for Intel E5 plus Rockchip RK3328 (left) and AMD plus Cavium ThunderX1 (right).

Latency and Throughput. As already mentioned in Section 2, even small variations in latency have huge impact on performance. The left graphs in Figures 17, 18, and 19 show for a fixed payload size, Redis' GET throughput for the Intel Atom plus Potato board, for the Intel E5 plus Rockchip RK3308, and for the AMD plus the ThunderX1. Despite the Potato board that connects via a 100 MbE, all other machines communicate via 1GbE.

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, 78,000 req/s, and 86,000 req/s, respectively).

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 start 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 $15,497$ req/s when the server is always on *origin* and $10,907$ req/s when it is always on *destination*. For the same machines using H-Container the average throughput is $19,766 req/s$, showing an improvement of 27.5% and 81.2% vs. no migration scenarios. When using the E5 plus Rockchip, without migration throughput, averages are 19,751 req/s and 12,029 req/s, while H-Container achieves 24,393 req/s, i.e., an improvement of 23.4% and 102.7%, respectively. Finally, in the AMD plus ThunderX1 setup, without migration throughput, averages are 21,036 req/s and 12,105 req/s, while H-Container achieves 25,221 req/s, i.e., an improvement of 19.9% and 108.3%, respectively.

Similar conclusions can be drawn for the other applications and platforms. Figure 20 shows the latencies for the Gzip server when compressing a 3KB packet. The throughput, measured for the exact same experiment is reported in Figure 21. Despite the fact that Gzip is likely to be a relatively stateless server—i.e., with minimal state, thus amenable to stop and restart—these experiments clearly highlight the benefits of using H-Container to maintain a sustained higher throughput and sustained minimal latency.

The graphs show that similarly to the throughput, the latency is definitely affected by the performance of a node on which a server is running, but the impact of the distance to the node is

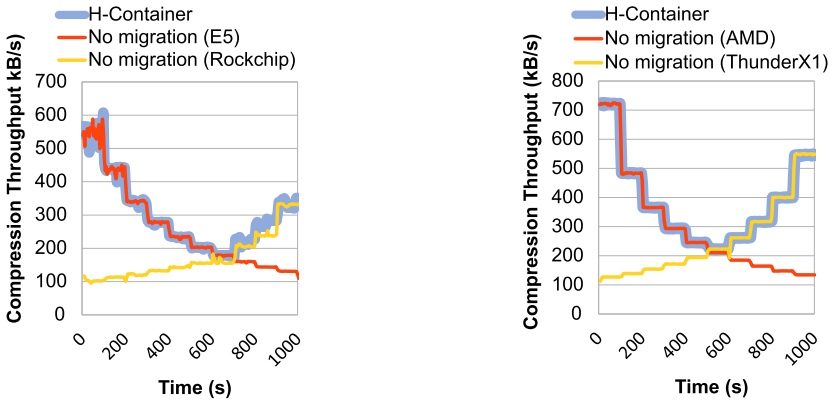

Fig. 21. Gzip server throughput for 3 KB packets for Intel E5 plus Rockchip RK3328 (left) and AMD plus Cavium ThunderX1 (right).

dominating. With no migration, when the server is running on the AMD EPYC the average latency is 12.945 ms, and 14.168 ms on the Cavium ThunderX1, while with H-Container it goes down to 8.646 ms, showing a reduction of 33.2% and 39%, respectively, when adopting H-Container. Changing setup, with no migration, when the server is running on the Intel E5 the average latency is 13.228 ms, and 18.296 ms on the Rockchip RK3328, while with H-Container it goes down to 10.251 ms, showing a reduction of 22.5% and 44.0%, respectively, when adopting H-Container. The same consideration applies to the throughputs, in KB/s, which we report for reference in Figure 21.

Varying Request Size. For Redis, we varied the payload size from 3B to 3KB. The results are shown in the right graphs of Figures 17, 18, and 19. For payloads up to 300 bytes, the behavior is similar because the experiment is bounded by latency. However, on the Atom plus Potato setup, 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. This is due to the network bandwidth capping the performance—in fact, the Potato is equipped with a 100 MbE, while all others connect via a 1 GbE connection. Another observation is that after migration, the request throughput decreases on the *destination* node. As Redis is a single-threaded application, this is due to single-thread compute performance capping the overall performance. Indeed, the Potato is equipped with a 100 MbE NIC, as opposed to all others that use to connect via a 1GbE NIC, but graphs clearly show that even when all nodes are connected to 1 GbE the throughput is lower when running on a weaker node.

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 [39].

Results differ with compute bound benchmarks such as OCR and Ransac algorithm. With those, the ARM machines cannot compete with the x86 machines in terms of performance—on the used hardware, the same applications may run more than twice slower on ARM CPUs than on x86 CPUs given the same input. Therefore, we throttle the CPU usage on the x86 machines to 1/8—i.e., to match the CPU performance of the ARM and the x86 CPUs for each setup. We then run the same experiments as above, changing the input size for both OCR and Ransac. Figures 22 and 23 show the experimental results for OCR on the AMD plus Cavium ThunderX1 and the Intel E5 plus Rockchip

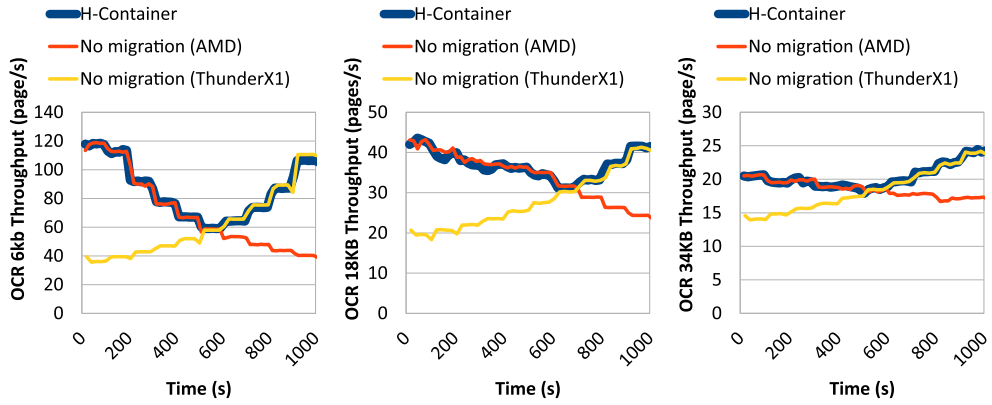

Fig. 22. OCR (GOCR) server throughput varying the picture size (page) for AMD plus Cavium ThunderX1.

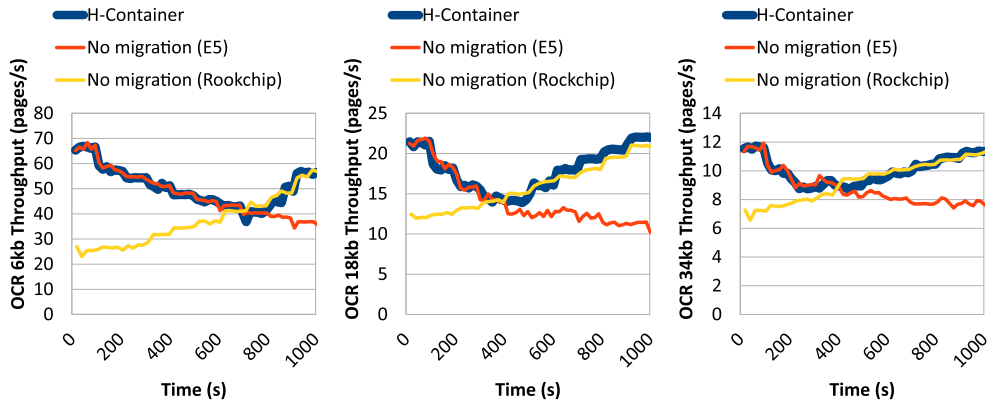

Fig. 23. OCR (GOCR) server throughput varying the picture size (page) for Intel E5 plus Rockchip RK3328.

RK3328, respectively. Figures 24 and 25 show the experimental results for the Ransac algorithm on the AMD plus Cavium ThunderX1 and the Intel E5 plus Rockchip RK3328, respectively.

Differently from the Redis number shown above, any change of the input size evidently changes throughput, and this is independent of the setup used. However, in the case of the OCR, the throughput reduces increasing the input size, while in the case of the Ransac, the throughput increases increasing the input size. From this, we learn that (a) today, for certain compute-intensive workloads, it may not always be advantageous to migrate, and (b) with (future) similar performance x86 and ARM machines it is always advantageous to migrate instead of paying for a long communication latency even when the latency is dominated by the computational time (as it sums up to the latency).

Live-migration. Here, we evaluate the performance of H-Container's live-migration feature and compare it to checkpoint/restore. For a fair comparison, which is also amenable to clear cost breakdowns, we decided not to rely on any distributed file systems in the following experiments. While in the checkpoint/restore experiments above we used a distributed file system among nodes (with support for compression), in the experiments below, checkpoint/restore migration does not adopt a distributed file system. Instead, first a checkpoint is taken, then the image files are transferred from the origin to the destination nodes, and when the transfer is complete, restore is triggered.

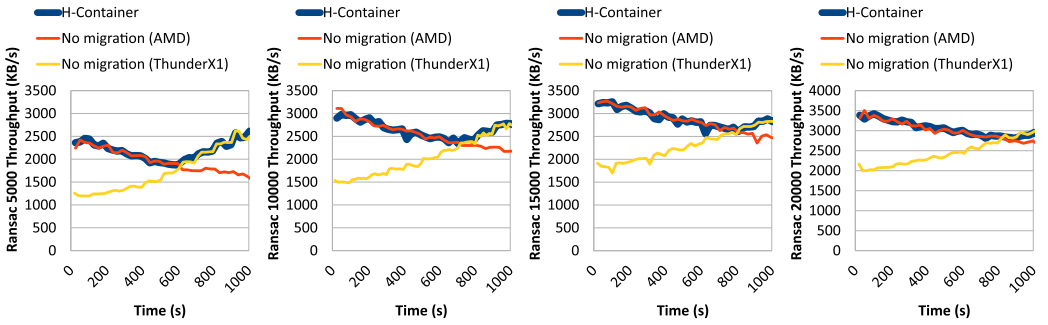

Fig. 24. Ransac server throughput varying the matrix size for AMD plus Cavium ThunderX1.

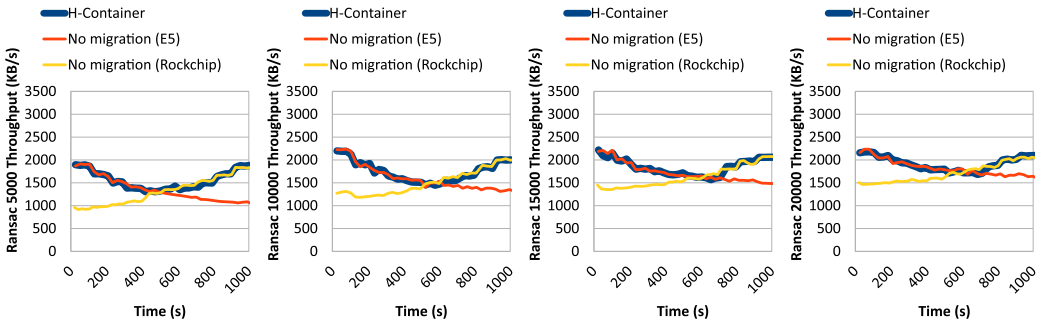

Fig. 25. Ransac server throughput varying the matrix size for Intel E5 plus Rockchip RK3328.

However, live-migration periodically checkpoints and transfers the image files, and after the last checkpoint, the image files are transferred and the container is restored.

We used Redis in the evaluations, and vary the database size to be 638 MB, 350 MB, and 185 MB. Experimental results for the AMD EPYC plus Cavium ThunderX1 setup are reported in Figure 26 and Figure 27 (from x86 to ARM). The first graph shows the perceived service downtime from the client perspective while varying database size. The service downtime is the time during which the service, in this case Redis, cannot be reached by a client because it is migrating [64]. As expected, checkpoint/restore is from 3× to 5× slower. This is due to the large amount of data transferred via the (slow) network. This is because part of the container state has been previously transferred from the origin node to the destination node during pre-copy. However, the restore phase becomes more expensive with live-migration, because several image files have to be merged.

Figure 27 further investigates the cost of the last checkpoint in live-migration vs. checkpoint/restore migration. Compared to the breakdown we discussed above in Figure 15 and Figure 14, here, we highlight a new cost component—"dump_page." It represents the cost to read-in the previously dumped image files and check if a specific memory page has been modified (dirty bit). Interestingly, this cost is also incurred in checkpoint/restore, which checks for modifications. This is the real additional cost that live-migration has to pay, as both "dump_other" and "dump_page" shrunk considerably when switching from checkpoint/restore to live-migration.

7.4 H-Container on IaaS Edge Cloud

As several cloud and edge providers today tend to run containers into virtual machines to increase security, we repeated the experiments in Section 7.3 on Amazon Web Services. AWS was the IaaS

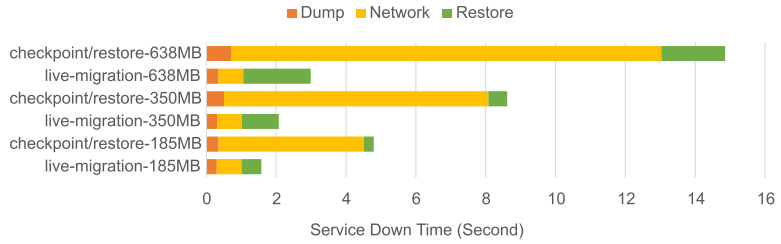

Fig. 26. Checkpoint/restore vs. live-migration Redis perceived service down time varying database size.

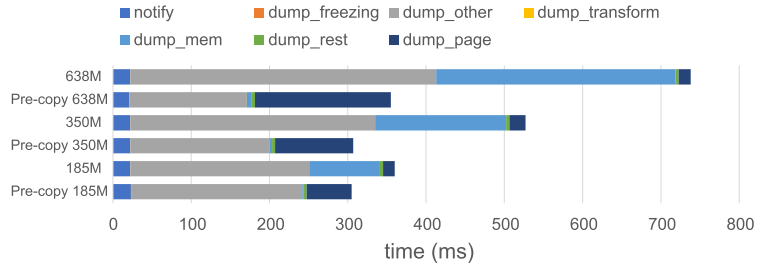

Fig. 27. Checkpoint/restore vs. live-migration Redis time for checkpointing varying database size.

provider of choice, because it is the only one that at the time of writing offered both arm64 and x86-64 virtual machines.

Hardware and Software. The origin node of x86 ISA, referred to as AWS-x86, is an AWS "t2.micro general-purpose" instance, configured with 1 vCPU, 1 GB of RAM, EBS-only storage, and its network performance is reported as "low to moderate." The CPU shows up as an Intel Xeon E5-2676 v3 at 2.40 GHz—very similar to the E5 used in the previous bare-metal experiments. Such instance runs AWS' Ubuntu 18.04 amd64, `ami-07c1207a9d40bc3bd`. The destination node, of ARM ISA, referred to as AWS-ARM, is an AWS "a1.medium general-purpose" instance, configured with 1 vCPU, 2 GB of RAM, EBS-only storage, and network performance reported as "up to 10 GbE." The CPU is an Amazon Graviton processor. Such instance runs AWS' Ubuntu 18.04 arm64, `ami-0606a0d9f566249d3`. A client node, also needed for our experiments, is referred to as AWS-Client and is identical to the AWS-x86 node—the entire experimental setup runs on the same data-center location.

Characterization of the Test Setup. We measured the network latencies among the three nodes with `iperf`. Table 5 summarizes the results. Moreover, throughput among such nodes has been measured to be consistently 125 MB/s. This is because all nodes have been deployed in the same data center. It is interesting to note that the latencies are uniform, differently from what we obtained in our bare-metal setup.

Results. We repeated the Redis experiment in this setup and present the results in Figure 28. The AWS-Client node increments/decrements its network latency to AWS-x86/AWS-ARM using `tc` every 100 ms. The graph on the left reports the Redis throughput (in kReq/s) when there is no migration or when using H-Container. Also in this case, H-Container enables the client to experience the highest throughput. In fact, without migration, the average throughput experienced by the AWS-x86 is of 17.72 kreq/s, while of 16.80 kreq/s on AWS-ARM, instead H-Container allows for 21.09 kreq/s—that is an improvement of 19% and 25%, respectively. Note that as the two virtual

Table 5. Measured Network Latencies (`iperf`) in AWS, Including the Origin, Destination, and Client Nodes, Respectively, AWS-x86, AWS-ARM, and AWS-Client

From \ To	AWS-ARM	AWS-x86	AWS-Client
AWS-ARM	x	0.519 ms	0.517 ms
AWS-x86	0.506 ms	x	0.509 ms
AWS-Client	0.499 ms	0.488 ms	x

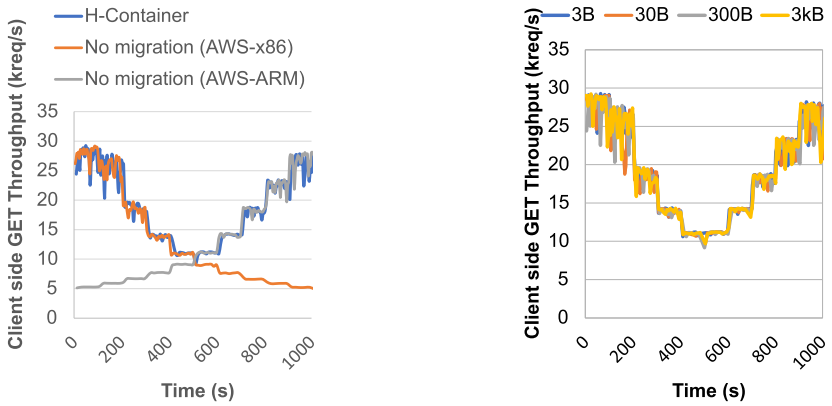

Fig. 28. Redis GET throughput with and without migration (left) and migration with different payloads (right) on AWS.

CPUs offer similar performance, the graph is very symmetric. Finally, the graph on the right of Figure 28 shows H-Container for different payload sizes, 3B, 30B, 300B, and 3KB—due to the similar network and performance of the virtual machines, there is just minimal differences when varying the payload size.

Summary. Results show that H-Container effectively migrates containers among heterogeneous-ISA machines on bare-metal as well as on IaaS offering. Migration proves to be essential to reduce latencies as well as to improve throughput, when CPUs are comparable in terms of performance but also when a node has a weaker CPU or network connection.

8 DISCUSSION

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. Note that none of such limitations is conceptual and that they are rather due to the lack of a comprehensive implementation. First and foremost, at the time of writing, 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 ~1,900 LoC). Second, the Popcorn Linux compiler framework cannot create migratable dynamically linked binaries. Despite clearly a limitation, Slinky [26] demonstrated that statically linked binaries can enjoy the same reduced-memory and ease of updating advantages of dynamically linked binaries. Moreover, Popcorn cannot compile functions with variable number of arguments, hence, we had to patch the application's source code (up to ~1,100 LoC). Further engineering is needed to make

the compiler and migration library support functions with variable number of arguments. Third, Docker live-migration support is incomplete as of today, therefore, we proposed two alternative solutions to cope with that: using either a distributed filesystem or the utility for automated file transfer that we developed.

Supporting Other ISAs. To enable runtime cross-ISA migration, H-Container setups for an application a unique address space layout among between all ISAs one would desire the application to execute on. As explained above, this relies on symbols alignment in virtual memory, but also on a set of assumptions regarding an ISA ABI, padding, and so on. Although new ISAs emerging on the market is a relatively rare phenomenon, it is important to discuss how H-Container can accommodate such a scenario. Should a new ISA emerge, it would be fully compatible with H-Container if it satisfies the basic assumptions we make to enable cross-ISA migration, i.e., same endianness, primitive type sizes, alignment constraints as the other ISA(s) we wish to migrate to/from. In this case, minimal modifications to the LLVM backend and linker of the new ISA have to be done to add the support to current Popcorn compiler. An H-Container application has to be re-transpiled to support the new ISA. However, to support a non fully compatible ISA, it is necessary to rethink how to handle different data types, endianness, and so on, which may require a large number of modifications not only in the LLVM backend and linker.

Cross-ISA Migration Use Cases. Although most of our experiments in this article demonstrate the usefulness of H-Container in scenarios where an application migrates between edge data centers to maintain proximity to the user, it is worth noting that cross-ISA migration as implemented by H-Container can also bring benefits within a single data center, similarly to existing works [12, 77], when such cloud or edge data center deploys machines of different ISAs, like AWS (see Section 7.4). For example, H-Container can be used to address the resource fragmentation [71] problem by migrating jobs between servers of different ISAs—which is something particularly important at the edge given the fixed and limited nature of computing resources in edge data centers.

9 RELATED WORK

Migrating at the Edge. The topic of application migration among edge nodes has been considered before. K. Ha et al. [45] propose VM handoff, a set of techniques for efficient VM live migration on the edge. L. Ma et al. [67] looked at the problem of migrating containerized application between edge nodes with Docker. A. Machen et al. [68, 69] introduced a three-layer framework to support VM and container migrations. None of these works considered that edge nodes are intrinsically built with CPUs of different ISAs, thus H-Container is the first study addressing the problem. In fact, 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 References [6, 52, 95, 110]. H-Container does not 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 software at runtime between different machines. The majority of which have been developed in the context of data centers where computers are homogeneous. Amongst others, the most similar works to H-Container are ZAP [78] and CRIU [37, 94], which implement checkpoint/restore of Linux processes among same ISA processors. CRIU supports live migration, and it is the underlying mechanism enabling container migration in container engines such as Docker [27], LXC [28], and so on, 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 [8, 53, 92]. Recently, Popcorn Linux [12, 13, 16, 54, 65] proposes a compiler and runtime toolset for cross-ISA migration, reconsidering the same problem on emerging heterogeneous platforms. More recently, HEXO [77] 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. Finally, leveraging CRIU allows H-Container to support applications with relatively complex kernel state (such as servers), which are not supported by neither Popcorn nor HEXO. At the same time, H-Container uses and extends Popcorn's compiler framework.

A unified address space among heterogeneous ISA processors has been also proposed by MutekH [73], which code was not practically usable, because it targets an exokernel/libos, as well as A. Venkat et al. [102], whose code is not publicly available.

10 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—this is because an application compiled for an ISA cannot migrate to, nor run, on another.

This article introduces H-Container, which enables containerized applications to migrate across heterogeneous-ISA nodes. H-Container targets Linux and is composed of (1) an LLVM-based decompiler-compiler transforming executable binaries for execution/migration on/between multiple ISAs, as well as a (2) CRIU-based user-space checkpoint/restore framework to pause 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.

SOURCE CODE

H-Container is open-source and publicly available at the following address: http://popcornlinux.org/index.php/hcontainer.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their insightful comments and suggestions, which significantly helped improve this article.

REFERENCES

[1] Jake Saunders and Nick Marshall. 2018. Mobile backhaul options Spectrum analysis and recommendations. GSM Association. Retrieved from https://www.gsma.com/spectrum/wp-content/uploads/2019/04/Mobile-Backhaul-Options.pdf.

[2] 2020. Linux Containers > LXD > Introduction. website. Retrieved from https://linuxcontainers.org/lxd/introduction/.

[3] 2020. Pod Manager tool. website. Retrieved from https://podman.io/.
[4] 2020. The H-Containers Project. website. http://popcornlinux.org/index.php/hcontainer.
[5] Ghassan Abdo and Dave McCarthy. 2021. Edge Computing Solutions Powering the Fourth Industrial Revolution. Retrieved from https://bit.ly/3bHXsTl.
[6] O. I. Abdullaziz, L. Wang, S. B. Chundrigar, and K. Huang. 2018. ARNAB: Transparent service continuity across orchestrated edge networks. In *IEEE Globecom Workshops (GC'Wkshps)*. 1–6.
[7] Amazon. 2018. EC2 Instances (A1) Powered by Arm-based AWS Graviton Processors. Retrieved from https://aws.amazon.com/blogs/aws/new-ec2-instances-a1-powered-by-arm-based-aws-graviton-processors/.
[8] 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 *5th Annual ESPRIT Conference (ESPRIT'88)*, Vol. 1.
[9] Mohamed Awad. 2019. Arm and Docker: Better Together. Retrieved from https://www.arm.com/company/news/2019/04/arm-and-docker-better-together.
[10] 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 *ACM/IEEE Conference on Supercomputing*. 158–165.
[11] Antonio Barbalace, Mohamed L. Karaoui, Wei Wang, Tong Xing, Pierre Olivier, and 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)*. 73–87.
[12] Antonio Barbalace, Rob Lyerly, Christopher Jelesnianski, Anthony Carno, Ho-ren Chuang, and Binoy Ravindran. 2017. Breaking the boundaries in heterogeneous-ISA datacenters. In *22th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*.
[13] Antonio Barbalace, Alastair Murray, Rob Lyerly, and Binoy Ravindran. 2014. Towards operating system support for heterogeneous-ISA platforms. In *4th Workshop on Systems for Future Multicore Architectures (SFMA'14)*.
[14] 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 *10th European Conference on Computer Systems (EuroSys'15)*. 29:1–29:16.
[15] D. Bernstein. 2014. Containers and cloud: From LXC to Docker to Kubernetes. *IEEE Cloud Comput.* 1, 3 (Sep. 2014), 81–84.
[16] Sharath K. Bhat, Ajithchandra Saya, Hemedra K. Rawat, Antonio Barbalace, and Binoy Ravindran. 2016. Harnessing energy efficiency of heterogeneous-ISA platforms. *SIGOPS Oper. Syst. Rev.* 49, 2 (Jan. 2016).
[17] Neil Brown. 2014. Control groups series. *Linux Weekly News*. Retrieved from https://lwn.net/Articles/604609/.
[18] Edouard Bugnion, Jason Nieh, and Dan Tsafrir. 2017. Hardware and software support for virtualization. *Synth. Lect. Comput. Archit.* 12, 1 (2017), 1–206.
[19] Cristian Cadar, Daniel Dunbar, Dawson R. Engler, et al. 2008. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Conference on Operating Systems Design and Implementation*. 209–224.
[20] CenturyLink. 2021. What's the difference between 2.4 GHz and 5 GHz WiFi? Retrieved from https://www.centurylink.com/home/help/internet/wireless/which-frequency-should-you-use.html.
[21] B. Charyyev, E. Arslan, and M. H. Gunes. 2020. Latency comparison of cloud datacenters and edge servers. In *IEEE Global Communications Conference*. 1–6.
[22] Batyr Charyyev and Mehmet Gunes. 2020. Latency characteristics of edge and cloud. In *GLOBECOM 2020 - 2020 IEEE Global Communications Conference*. 1–6. DOI: 10.1109/GLOBECOM42002.2020.9322406
[23] 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 *11th Annual Workshop on Network and Systems Support for Games*. IEEE Press, 2.
[24] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. 2011. CloneCloud: Elastic execution between mobile device and cloud. In *6th Conference on Computer Systems*. ACM, 301–314.
[25] 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 *2nd Conference on Symposium on Networked Systems Design & Implementation*. USENIX Association, 273–286.
[26] Christian S. Collberg, John H. Hartman, Sridivya Babu, and Sharath K. Udupa. 2005. SLINKY: Static linking reloaded. In *USENIX Annual Technical Conference*. 309–322.
[27] CRIU contributors. 2019. CRIU Wiki – Docker page. Retrieved from https://criu.org/Docker.
[28] CRIU contributors. 2019. CRIU Wiki – LXC page. Retrieved from https://criu.org/LXC.
[29] 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 *8th International Conference on Mobile Systems, Applications, and Services*. ACM, 49–62.

[30] Anirban Das, Stacy Patterson, and Mike Wittie. 2018. EdgeBench: Benchmarking edge computing platforms. In *IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC'18)*. IEEE, 175–180.
[31] Dell. 2019. Micro Modular Data Centers: Taking Computing to the Edge. Retrieved from https://blog.dellemc.com/en-us/micro-modular-data-centers-taking-computing-to-edge/.
[32] Matthew DeVuyst, Ashish Venkat, and Dean M. Tullsen. 2012. Execution migration in a heterogeneous-ISA chip multiprocessor. In *17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'12)*. 261–272.
[33] Alessandro Di Federico, Mathias Payer, and Giovanni Agosta. 2017. rev. ng: A unified binary analysis framework to recover CFGs and function boundaries. In *26th International Conference on Compiler Construction*. ACM, 131–141.
[34] Artem Dinaburg and Andrew Ruef. 2014. McSema: Static translation of X86 instructions to LLVM. In *ReCon Conference*.
[35] Jack J. Dongarra, Piotr Luszczek, and Antoine Petitet. 2003. The LINPACK benchmark: Past, present and future. *Concurr. Computat.: Pract. Exper.* 15, 9 (2003), 803–820.
[36] Edgeconnex. 2019. Bringing Gaming Closer to Gamers Worldwide. Retrieved from https://www.edgeconnex.com/wp-content/uploads/2019/07/EDC-19-44-NEW-Gaming-DataSheet-V4.pdf.
[37] P. Emelyanov. 2011. CRIU: Checkpoint/Restore In Userspace. (July 2011). https://criu.org/Main_Page.
[38] Phil Estes. 2017. Multi-arch All the Things. (2017). Retrieved from https://www.docker.com/blog/multi-arch-all-the-things/.
[39] 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.
[40] 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)*.
[41] 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.
[42] Joachim Gehweiler and Michael Thies. 2010. Thread migration and checkpointing in Java. *Heinz Nixdorf Institute, Tech. Rep. tr-ri-10* 315 (2010).
[43] Ghidra Contributors. 2019. Ghidra Website. Retrieved from https://ghidra-sre.org/.
[44] Mark S. Gordon, D. Anoushe Jamshidi, Scott Mahlke, Z. Morley Mao, and Xu Chen. 2012. COMET: Code offload by migrating execution transparently. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI'12)*. 93–106.
[45] 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 *2nd ACM/IEEE Symposium on Edge Computing (SEC'17)*.
[46] Christine Hall. 2019. Companies Pushing Open Source RISC-V Silicon Out to the Edge. Retrieved from https://www.datacenterknowledge.com/hardware/companies-pushing-open-source-risc-v-silicon-out-edge.
[47] Drew Henry. 2018. Announcing ARM Neoverse. Retrieved from https://www.arm.com/company/news/2018/10/announcing-arm-neoverse.
[48] Kelsey Hightower, Brendan Burns, and Joe Beda. 2017. *Kubernetes: Up and Running: Dive Into the Future of Infrastructure*. O'Reilly Media, Inc.
[49] Michael R. Hines, Umesh Deshpande, and Kartik Gopalan. 2009. Post-copy live migration of virtual machines. *ACM SIGOPS Oper. Syst. Rev.* 43, 3 (2009).
[50] 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 (Sept. 2019). Retrieved from DOI:https://doi.org/10.1145/3326066
[51] Intel. 2018. Intelligence at the "Edge": the Intel Xeon D-2100 Processor. Retrieved from https://www.intel.com/content/www/us/en/communications/d-2100-processor-edge-computing-benefits-infographic.html.
[52] M. Jia, J. Cao, and W. Liang. 2017. Optimal cloudlet placement and user to cloudlet allocation in wireless metropolitan area networks. *IEEE Trans. Cloud Comput.* 5, 4 (Oct. 2017), 725–737.
[53] 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).
[54] D. Katz, A. Barbalace, S. Ansary, A. Ravichandran, and B. Ravindran. 2015. Thread migration in a replicated-kernel OS. In *IEEE 35th International Conference on Distributed Computing Systems*. 278–287.
[55] Michael Kerrisk. 2013. Namespaces in operation, part 1: Namespaces overview. *Linux Weekly News*. Retrieved from https://lwn.net/Articles/531114/.
[56] Jakub Křoustek and Peter Matula. 2018. RetDec: An open-source machine-code decompiler. Retrieved from https://2018.pass-the-salt.org/files/talks/04-retdec.pdf.

[57] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*. IEEE Computer Society, 75.
[58] Gwangmu Lee, Hyunjoon Park, Seonyeong Heo, Kyung-Ah Chang, Hyogun Lee, and Hanjun Kim. 2015. Architecture-aware automatic computation offload for native applications. In *48th International Symposium on Microarchitecture*. ACM, 521–532.
[59] George Leopold. 2019. Arm, Docker Partner on Cloud-to-Edge Development. Retrieved from https://www.enterpriseai.news/2019/04/24/arm-docker-partner-on-cloud-to-edge-development/.
[60] A. Lertsinsrubtavee, A. Ali, C. Molina-Jimenez, A. Sathiaseelan, and J. Crowcroft. 2017. PiCasso: A lightweight edge computing platform. In *IEEE 6th International Conference on Cloud Networking (CloudNet)*. 1–7.
[61] 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 (Apr. 2018).
[62] ARM Ltd. 2019. Accelerating the transformation to a scalable cloud to edge infrastructure. Retrieved from https://www.arm.com/-/media/global/products/processors/N1%20Solution%20Overview.pdf.
[63] Peng Lu, Antonio Barbalace, Roberto Palmieri, and Binoy Ravindran. 2014. Adaptive live migration to improve load balancing in virtual machine environment. In *Parallel Processing Workshops*. 116–125.
[64] Peng Lu, Antonio Barbalace, and Binoy Ravindran. 2013. HSG-LM: Hybrid-copy speculative guest OS live migration without hypervisor. In *6th International Systems and Storage Conference*. ACM, 2.
[65] Robert Lyerly, Antonio Barbalace, Christopher Jelesnianski, Vincent Legout, Anthony Carno, and Binoy Ravindran. 2016. Operating system process and thread migration in heterogeneous platforms. In *the 2016 Workshop on Multicore and Rack-scale Systems*. https://www.cs.utexas.edu/~mars2016/workshop-program/MaRS_2016_paper_2.pdf.
[66] Lele Ma, Shanhe Yi, Nancy Carter, and Qun Li. 2019. Efficient live migration of edge services leveraging container layered storage. *IEEE Trans. Mob. Comput.* 18, 9 (2019), 2020–2033.
[67] Lele Ma, Shanhe Yi, and Qun Li. 2017. Efficient service handoff across edge servers via Docker container migration. In *2nd ACM/IEEE Symposium on Edge Computing (SEC'17)*.
[68] Andrew Machen, Shiqiang Wang, Kin K. Leung, Bong Jun Ko, and Theodoros Salonidis. 2016. Migrating running applications across mobile edge clouds: Poster. In *22nd Annual International Conference on Mobile Computing and Networking (MobiCom'16)*.
[69] Andrew Machen, Shiqiang Wang, Kin K. Leung, Bong Jun Ko, and Theodoros Salonidis. 2018. Live service migration in mobile edge clouds. *IEEE Wirel. Commun.* 25, 1 (2018), 140–147.
[70] 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.
[71] D. S. Marcon and M. P. Barcellos. 2017. Packer: Minimizing multi-resource fragmentation and performance interference in datacenters. In *IFIP Networking Conference (IFIP Networking) and Workshops*. 1–9.
[72] Jonathan McChesney, Nan Wang, Ashish Tanwer, Eyal de Lara, and Blesson Varghese. 2019. DeFog: Fog computing benchmarks *(SEC'19)*. Association for Computing Machinery, New York, NY, 47–58.
[73] MutekH Authors. 2016. MutekH reference manual. Retrieved from https://www.mutekh.org/doc/index.html.
[74] S. Nadgowda, S. Suneja, N. Bila, and C. Isci. 2017. Voyager: Complete container state migration. In *IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. 2137–2142.
[75] M. Noreikis, Y. Xiao, and A. Ylä-Jaäiski. 2017. QoS-oriented capacity planning for edge computing. In *IEEE International Conference on Communications (ICC)*. 1–6.
[76] Christy Norman Perez and Chris Jones. 2017. The ARM to z of Multi-Architecture Microservices. Retrieved from https://qconsf.com/sf2017/system/files/presentation-slides/from_arm_to_z.pdf.
[77] 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 *28th International Symposium on High-performance Parallel and Distributed Computing (HPDC'19)*.
[78] 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.
[79] Adam Parco. 2019. Building Multi-Arch Images for Arm and x86 with Docker Desktop. Retrieved from https://engineering.docker.com/2019/04/multi-arch-images/.
[80] PicoCluster. 2019. PicoCenter 48. Retrieved from https://www.picocluster.com/products/picocenter-48.
[81] C. Puliafito, E. Mingozzi, C. Vallati, F. Longo, and G. Merlino. 2018. Virtualization and migration at the network edge: An overview. In *IEEE International Conference on Smart Computing (SMARTCOMP)*. 368–374.
[82] 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. https://doi.org/10.1109/HPCA.2007.346181

[83] Redis Labs. 2019. RedisEdge—The Edge Computing Database for the IoT Edge. Retrieved from https://redislabs.com/solutions/redisedge/.
[84] Will Reese. 2008. Nginx: The high-performance web server and reverse proxy. *Linux J.* 2008, 173 (2008), 2.
[85] Phil Rogers. 2013. Heterogeneous system architecture overview. In *Hot Chips*, Vol. 25.
[86] Mahadev Satyanarayanan. 2017. The emergence of edge computing. *Computer* 50, 1 (2017), 30–39.
[87] 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.
[88] Joerg Schulenburg. 2018. GOCR: open-source character recognition. Retrieved from http://jocr.sourceforge.net/.
[89] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. 2016. Edge computing: Vision and challenges. *IEEE Internet Things J.* 3, 5 (2016), 637–646.
[90] Grant Shipley. 2014. *Learning OpenShift*. Packt Publishing Ltd.
[91] SiFive. 2019. SiFive U74. Retrieved from https://www.sifive.com/cores/u74.
[92] Peter Smith and Norman C. Hutchinson. 1998. Heterogeneous process migration: The Tui system. *Softw.: Pract. Exper.* 28, 6 (1998), 611–639.
[93] 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 *20th Working Conference on Reverse Engineering (WCRE)*. IEEE, 52–61.
[94] Radostin Stoyanov and Martin J. Kollingbaum. 2018. Efficient live migration of Linux containers. In *High Performance Computing*. Springer International Publishing, 184–193.
[95] Kyoungjae Sun and Younghan Kim. 2018. Network-based VM migration architecture in edge computing. In *International Conference on Information Science and System (ICISS'18)*. 169–172.
[96] 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 Commun. Surv. Tutor.* 19, 3 (third quarter 2017).
[97] Willy Tarreau, et al. 2012. HAProxy-the reliable, high-performance TCP/HTTP load balancer. http://www.haproxy.org/.
[98] Lumen Technologies. 2021. Edge Computing. Retrieved from https://www.lumen.com/en-uk/solutions/edge-computing.html.
[99] 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 *11th European Conference on Computer Systems*. ACM, 16.
[100] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. 2021. Benchmarking, analysis, and optimization of serverless function snapshots. *CoRR* abs/2101.09355 (2021).
[101] Blesson Varghese, Nan Wang, Dimitrios S. Nikolopoulos, and Rajkumar Buyya. 2017. Feasibility of Fog Computing. https://arxiv.org/abs/1701.05451.
[102] Ashish Venkat and Dean M. Tullsen. 2014. Harnessing ISA diversity: Design of a heterogeneous-ISA chip multiprocessor. In *41st Annual International Symposium on Computer Architecuture (ISCA'14)*. IEEE Press, Piscataway, NJ, 121–132.
[103] VMware. 2019. nanoEDGE. Retrieved from https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/products/vsan/vmw-nanoedge-sddc-solution.pdf.
[104] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. 2017. Ramblr: Making reassembly great again. In *Network and Distributed System Security Symposium*.
[105] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanovic. 2011. *The RISC-V Instruction Set Manual, Volume I Base User-level ISA*. Technical Report UCB/EECS-2011-62. EECS Department, UC Berkeley.
[106] Wave Computing. 2019. Wave Computing Unveils New Licensable 64-bit AI IP Platform to Enable High-speed Inferencing and Training in Edge Applications. Retrieved from https://wavecomp.ai/wave-computing-unveils-new-licensable-64-bit-ai-ip-platform-to-enable-high-speed-inferencing-and-training-in-edge-applications/.
[107] Tong Xing. 2020. Software deployment methods to support heterogeneous-ISA application migration. Stevens Institute of Technology. https://www.proquest.com/openview/237f0c1eb261a2951d507d0e3497e71b/1.pdf?pq-origsite=gscholar&cbl=18750&diss=y.
[108] Dongzhu Xu, Anfu Zhou, Xinyu Zhang, Guixian Wang, Xi Liu, Congkai An, Yiming Shi, Liang Liu, and Huadong Ma. 2020. Understanding operational 5G: A first measurement study on its coverage, performance and energy consumption. In *Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM'20)*. 479–494.

[109] Richard York. 2002. Benchmarking in context: Dhrystone. *ARM, March* (2002). https://www.docjava.com/courses/cr346/data/papers/DhrystoneMIPS-CriticismbyARM.pdf.

[110] Aleksandr Zavodovski, Nitinder Mohan, Suzan Bayhan, Walter Wong, and Jussi Kangasharju. 2018. ICON: Intelligent container overlays. In *17th ACM Workshop on Hot Topics in Networks (HotNets'18)*. ACM, New York, NY, 15–21.

Received July 2020; revised October 2021; accepted March 2022