HEXO: Offloading HPC Compute-Intensive Workloads on Low-Cost, Low-Power Embedded Systems

Pierre Olivier, A K M Fazla Mehrab
Virginia Tech
{ polivier | mehrab }@vt.edu

Stefan Lankes
RWTH Aachen University
slankes@eonerc.rwth-aachen.de

Mohamed Lamine Karaoui, Rob Lyerly, Binoy Ravindran
Virginia Tech
{ karaoui | rlyerly | binoy }@vt.edu

ABSTRACT

OS-capable embedded systems exhibiting a very low power consumption are available at an extremely low price point. It makes them highly compelling in a datacenter context. In this paper we show that sharing long-running, compute-intensive datacenter HPC workloads between a server machine and one or a few connected embedded boards of negligible cost and power consumption can bring significant benefits in terms of consolidation. Our approach, named Heterogeneous EXecution Offloading (HEXO), selectively offloads Virtual Machines (VMs) from server class machines to embedded boards. Our design tackles several challenges. We address the Instruction Set Architecture (ISA) difference between typical servers (x86) and embedded systems (ARM) through hypervisor and guest OS-level support for heterogeneous-ISA runtime VM migration. We cope with the low amount of resources in embedded systems by using lightweight VMs: unikernels. VMs are offloaded based on an estimation of the slowdown expected from running on a given board. We build a prototype of HEXO and demonstrate significant increase in throughput (up to 67%) and energy efficiency (up to 56%) over a set of macro-benchmarks running datacenter compute-intensive jobs.

CCS CONCEPTS

• **Computer systems organization → Heterogeneous (hybrid) systems**; • **Software and its engineering → Virtual machines**; **Operating systems**.

KEYWORDS

heterogeneous ISAs, unikernels, migration, offloading

ACM Reference Format:
Pierre Olivier, A K M Fazla Mehrab, Stefan Lankes, and Mohamed Lamine Karaoui, Rob Lyerly, Binoy Ravindran. 2019. HEXO: Offloading HPC Compute-Intensive Workloads on Low-Cost, Low-Power Embedded Systems. In *The 28th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '19), June 22–29, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3307681.3325408

1 INTRODUCTION

Costs in datacenters are driven by machine acquisition and equipment power consumption/cooling [4] and operators are constantly seeking to lower such expenditures. Manufacturers are today producing at an extremely low price point OS-capable embedded systems exhibiting a very low power consumption, making them highly attractive in a data-center context as shown by feasibility studies and simulation works [33, 42].

In this paper we demonstrate that in certain scenarios, *some embedded systems may not be as slow as they are cheap*: compared to server execution times, slowdowns on embedded boards are about one order of magnitude while the prices and power consumption of the boards are 2 orders of magnitude lower than those of servers. Based on these observations, we propose a new approach in which HPC datacenter workloads are selectively offloaded at runtime from servers to OS-capable embedded systems for consolidation purposes: the key idea is that by augmenting a server with one or a few embedded boards for a negligible cost (less than 5% of the server price/energy), one can consolidate more jobs and obtain a non-negligible increase in throughput. To achieve optimal price/power consumption/performance characteristics, one needs to consider machines implementing the most efficient Instruction Set Architecture (ISA) in each domain: x86-64 for servers, Arm64 for embedded systems.

Existing works related to the integration of embedded systems in the datacenter present various limitations: managing/operating embedded systems and servers separately [27, 44] does not allow fully exploiting the strengths of each type of machine, and simply assuming homogeneous ISAs [15, 19, 24, 28, 48] forbids benefiting from the combined efficiency of Intel x86-64 servers and ARM embedded systems. Finally, some virtualization techniques/languages [3, 9, 12, 17] help to bridge the ISA gap but at the cost of a non-negligible performance overhead.

We present an approach named Heterogeneous EXecution Offloading (HEXO). With high degrees of consolidation as our objective, we migrate/checkpoint/restart at runtime between ISAs Virtual Machines (VMs) running with hardware virtualization support, i.e. executing native code directly [7] on the CPU for maximized performance. Contrary to existing approaches that relocate part of the execution of mobile applications to servers for performance reasons [9, 12, 17, 32], we propose to migrate or checkpoint/restart entire applications from servers to embedded systems as the objective of consolidation is to free resources.

We address the low resources of embedded systems by choosing unikernels [23, 29, 35, 36] as the unit of execution for jobs on both servers and embedded systems. It provides a virtualized environment that is both lightweight and secure (hardware enforced), suitable for running multi-tenant workloads on embedded systems [34] where traditional VMs cannot run due to large resources requirements. In this paper we port the HermitCore [29] unikernel to Arm64 and redesign it to support cross-ISA migration.

We solve the disjoint ISA challenge by applying state transformation techniques, allowing conversion of the architecture-specific state of an application between ISAs. Existing implementations [2, 14, 50] target process migration with the Linux kernel and need to be adapted to VM migration schemes in a unikernel context. This is no trivial task, in particular because for part of the VM state (e.g., kernel state) there exists no clear mapping between different ISAs. Thus, traditional VM migration implementations cannot be used. We redesign the virtualization layer to implement the concept of *semantic migration* where the guest OS and hypervisor cooperate to extract the entire application state from the migrated VM as well as a minimal architecture-independent subset of the kernel state.

A final challenge consists in selecting the best candidates to offload from a server to an embedded board. Widely present in the datacenter [38], HPC compute-intensive jobs are a primary target for HEXO due to their long-running characteristics. We observe that the slowdown incurred by offloading to an embedded system is correlated to application characteristics that are measurable online. Without assuming any kind of offline profiling, HEXO uses a simple but efficient scheduler to decide which applications should be offloaded to embedded systems based on an estimation of the slowdown incurred on that target.

We build and evaluate a prototype of HEXO over a set of micro- and macro-benchmarks. By enhancing a server-class machine with one or a few embedded boards, HEXO can obtain up to 67% increase in throughput for a negligible increase in price and power consumption. In this paper we make the following contributions:

- The design and implementation of HEXO, to our knowledge the first system to propose offloading for consolidation direct-execution unikernels from servers to embedded boards;
- The multi-ISA semantic VM migration technique that allows a unikernel to migrate between machines of different ISAs;
- The port of a x86-64 unikernel, HermitCore, to Arm64;
- An evaluation of HEXO demonstrating improvements of up to 67% in throughput and 55% in energy efficiency for HPC compute-intensive datacenter workloads.

This paper is organized as follows: Section 2 motivates HEXO's development. Section 3 describes our design choices and assumptions, and gives a general overview of HEXO. Implementation details are given in Section 4. We evaluate HEXO in Section 5, discuss related works in Section 6 and conclude in Section 7.

2 MOTIVATION

In this section we motivate HEXO by showing the cost and power consumption benefits of the embedded systems we consider, as well as addressing concerns about the low processing power of embedded systems compared to server-class machines. In HEXO, the embedded systems we consider are *single board computers*, popularized by the Raspberry Pi. Such embedded systems satisfy the requirements of HEXO in terms of price as they are two orders of magnitude cheaper than traditional servers. In terms of resources, they can run medium-sized HPC workloads as unikernels.

Table 1: Considered server & boards characteristics.

Machine	Xeon	RPi	Potato	Hikey
CPU model	Xeon E5-2637	Broadcom BCM2837	Amlogic S905X	HiSilicon Kirin 620
ISA	x86-64	Arm64	Arm64	Arm64
CPU frequency	3 (turbo 3.5) GHz	1.2 GHz	1.5 GHz	1.2 GHz
Cores	4 (8 HT)	4	4	8
RAM	64 GB	1 GB	2 GB	2 GB
Power (idle)	60 W	1.75 W	1.8 W	2.5 W
Power (1 thread)	83 W	2 W	2.1 W	2.8 W
Power (4 threads)	124 W	4.35 W	2.9 W	4.3 W
Power (8 threads)	127 W	-	-	6.6 W
Price	$ 3049	$ 35	$ 45	$ 119

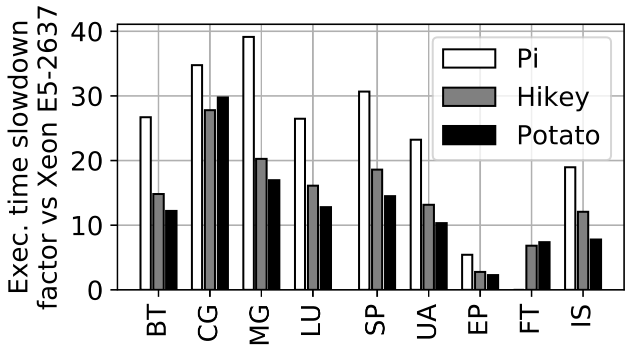

Figure 1: NPB embedded boards slowdown vs Xeon.

We measure the performance and power consumption of a traditional server class machine (Colfax CX1120s-X6, named *Xeon* in the rest of this paper), and a set of single board embedded systems: Raspberry Pi 3 Model B (*RPi*), Libre Computer LePotato (*Potato*), and 96Boards Hikey LeMaker (*Hikey*). The machines' characteristics and prices are given in Table 1. The power consumption is measured at the entire machine level using a Kill-A-Watt P4400, while each machine is idle and running 1/4/8 instances of the `stress` program (i.e. 1/4/8 cores/hardware threads active at 100%) for a sufficiently long time. One important point for HEXO's motivation is that the price and power consumption of these boards is *negligible* compared to those of the server.

For this motivation experiment, we use the NAS Parallel Benchmarks (NPB) [1] which are representative of HPC datacenter compute-intensive workloads. For these tests we use the natively compiled, serial version and the class B (medium data sets). We compute, for each benchmark and each board, the slowdown incurred while running on a board compared to the server. Performance results are presented on Figure 1, where execution times are normalized to the Xeon's performance, i.e. 1 on the Y axis represents the execution time of the server.

For some benchmarks the slowdown is relatively small – for the Potato board, which generally performs better than the other embedded systems, in some situations the slowdown is less than 10x (EP, FT and IS). *This slowdown is less than one order of magnitude and needs to be put into perspective with the fact that the board is two orders of magnitude cheaper than the server, both in terms of dollars and power consumption.* Some other benchmarks show the limits of the boards; for example, CG and MG are more than 30x slower on the Raspberry Pi compared to the server. The RPi was also unable to run FT due to a lack of RAM. A second important observation is that *the slowdown is highly variable depending on the benchmark*. This difference, is in particular due to the memory intensity of the benchmarks. The boards' memory subsystems are slower than the server's (smaller caches, lower DRAM frequency, etc.). A key idea in HEXO is to offload from server to embedded boards the jobs with the lowest expected slowdowns.

We estimate the energy consumption for each machine and benchmark based on the measured power (see Table 1) and execution times. This reveals that independently of the benchmark, *it is always more energy-efficient to run on the Potato*, even when the slowdown is high. CG exhibits the highest slowdown but takes 30% less energy to execute on the board, and the energy reduction goes up to 17x for EP. Multi-threaded tests and tests over different data set sizes (class A and C) confirmed these observations. In conclusion, these experiments show that some boards are not as slow as they are cheap, and that is it always better from the power consumption standpoint to run on these boards.

We conclude that the costs associated with these boards are so low that, assuming acceptable migration overheads, even if augmenting a server with one or a few embedded boards for consolidation only gives a small increase in throughput, it is still worthwhile. Moreover, the relatively small slowdown observed for some benchmarks with some boards shows that in certain situations that throughput increase may actually be significant.

Memory Capacity in Embedded Systems. Considering HPC, a concern is the low RAM amount present on such boards. Workloads requiring tens of GBs of RAM or more will not fit on such systems. Nevertheless, prior work studying cluster traces [8, 18] from companies including Google or Microsoft show a significant amount of long-running jobs requiring less than 1 or 2 GB of memory.

Moreover, given the trends in handheld devices, it is likely that the amount of RAM in embedded systems will increase in the future. To study the impact of that evolution, we built a simple analytical model estimating the price and power consumption of a hypothetical board that would embeds up to 128 GB of RAM. To that aim we used price data from DRAMeXchange [47], and computed the memory power consumption in activity using the potato board's DRAM datasheet [46] and the Micron DDR3 SDRAM system power calculator [37]. In this model we consider both the original DRAM module of the potato board [46] (512MB capacity), but also another one from the same family with a capacity of 1GB [22]. Note that our model is simple and scopes out issues such as the maximum of RAM supported by the CPU, space and thermal constraints, etc.

Results are presented on Figure 2. We wish to decide on a given value in terms of price or power that, when passed, HEXO's motivation looses it strength. However we previously observed that the slowdown on the board varies among applications. Therefore we define 3 break-even points, based on 3 representative slowdowns observed on Figure 1: *low slowdown* (EP, 2.2x), *medium slowdown* (FT, 7.3x) and *high slowdown* (MG, 16.9x). We compute each break-even point by dividing the price and power consumption of the server (see Table 1) by each of these factors. Looking at the model with these points in mind highlights the fact that *for HEXO to be worth it a board that is n times slower should be n times cheaper than a server*, with *n* equals to 2.2, 7.3, and 16.9, according to the considered break-even point. These points are represented by horizontal lines on Figure 2. As one can observe, for low slowdown applications, even with a large increase in RAM capacity HEXO is still generally beneficial up to 64/128 GB. For medium slowdown applications, we expect that limit to be 32 GB (64GB for price alone and 512 MB modules). Regarding high slowdown applications, they are generally a bad fit for HEXO.

Finally, I/O performance in embedded systems is notoriously slow, hence our focus on compute-intensive jobs. However it is probable that in a near future I/O speed will increase in the embedded systems we target, without a serious impact on their price and power consumption. Multiple single-board computers costing tens of dollars now offer gigabit Ethernet, USB 3.1 or PCIe connections.

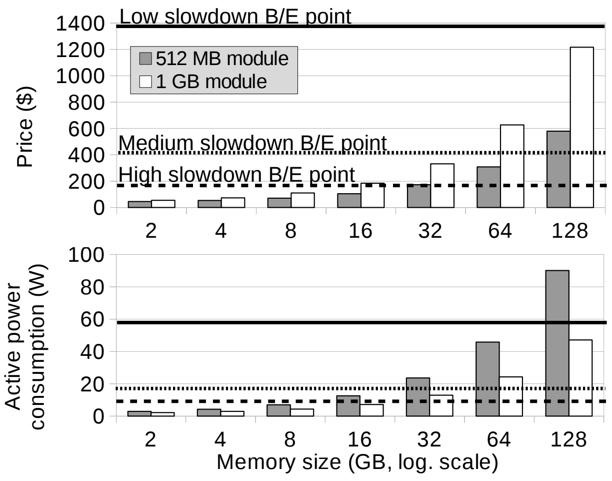

Figure 2: Evolution of the price and power consumption of a hypothetical embedded system with high amounts of RAM.

3 DESIGN

3.1 HEXO: Assumptions and Scope

In HEXO we assume that the server is equipped with an Intel x86-64 CPU and the embedded systems use Arm64 CPUs. It is possible to create setups composed of servers and embedded boards of the same ISA, avoiding the complexity of heterogeneous-ISA translation. However, we show that Arm64 servers and Intel's embedded CPUs are not as compelling as their competitors from the opposite side, concerning metrics critical to the related markets. On the one hand, the first generation of Arm64-based servers has been entering the market in the last few years but is not yet on par with similarly-priced Intel's CPUs for multiple performance and power consumption metrics [2, 13]. Concerning x86-based embedded platforms, we demonstrate that ARM platforms are significantly

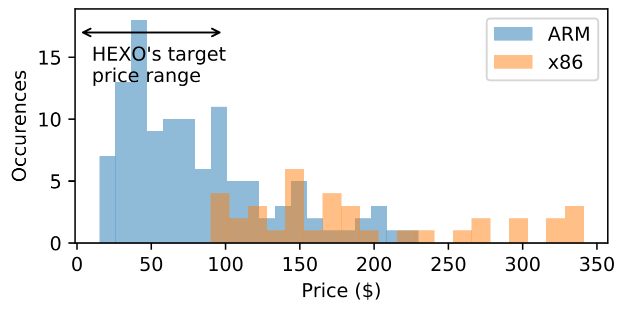

Figure 3: Prices distribution for 38 x86 and 115 ARM boards.

cheaper. To that aim we select from various sources online (including linuxgizmos website [6]) a large list of single board computers (115 ARM, 38 x86). We selected platforms under $350, 1 to 8 GB of RAM, and a CPU frequency of at least 1 GHz. We plot the distribution of these boards' prices on Figure 3. As mentioned above we seek embedded systems in a price range of tens of dollars. While it is hard to find x86 boards below 100 dollars, there is a plethora of ARM boards with similar and higher specifications in that window. Finally, in HEXO translating the architecture specific state between ISAs takes a negligible overhead (max 2 ms), so the overhead of VM state transfer, that would also be present in homogeneous setups, completely dominates the migration latency. Thus, to reap the benefits of the most efficient machines in their respective domains, we consider heterogeneous-ISA setups.

HEXO targets datacenter HPC compute-intensive workloads. These jobs are long-running [38] (from minutes to days) and thus offloading cannot be achieved by killing and restarting regular native binaries [16], as the loss of progress would be unacceptable – such jobs need *runtime* migration or checkpoint/restart.

We assume a cloud provider datacenter scenario, i.e. a multi-tenant environment. A high level of security is then needed and jobs cannot run natively but must rather be virtualized. Due to lack of resources, the embedded systems that we target cannot run full-fledged VMs and have to rely on lightweight virtualization. Moreover, we argue that hardware-assisted virtualization provides a fundamentally stronger isolation [36] than software solutions such as containers, as confirmed by current trends of running containers inside VMs for security (clear containers [11]). Thus, unikernels are suitable for HEXO.

For a minimal performance overhead, we select hardware-assisted directly-executing [7] VMs running native code (C language) as opposed to emulation or managed runtimes [3, 12, 17] that can help to bridge the ISA gap but involve an unavoidable performance overhead. To enable heterogeneous-ISA migration, we assume the same endianness for the server and boards (Arm64 endianness is configurable) as well as the same size and alignments for primitive C types.

In the context of datacenter workloads, the use of embedded systems may raise concerns in terms of reliability. HEXO employs a multi-ISA checkpoint/restart system and when a job is offloaded to a board, a checkpoint can be maintained on the server to save and restore the job's state in the case its execution fails on the board. Moreover, the very low cost of the boards we consider makes it

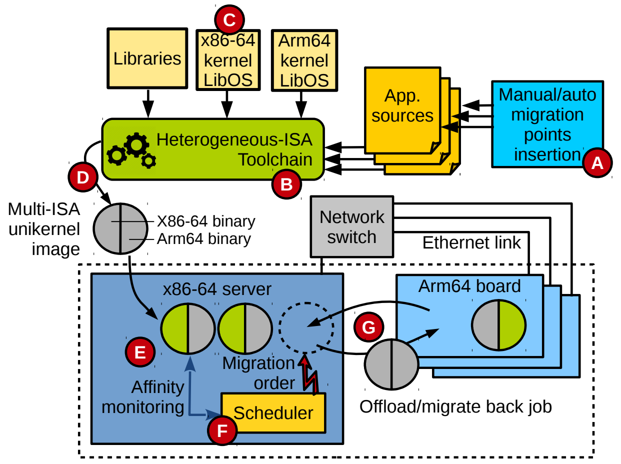

Figure 4: Overview of HEXO's building & execution flow.

possible to use redundancy and offload jobs in parallel on multiple boards for a low additional cost.

3.2 System Overview

Figure 4 represents an overview of HEXO's building and execution flow. The first step to create a multi-ISA unikernel is to instrument the application code by inserting *migration points* (Ⓐ on Figure 4), points in the application execution where migration is possible. Adding a migration point consists of the insertion of a simple call to a library function.

Application sources are fed to HEXO's heterogeneous ISA toolchain Ⓑ. Metadata needed to transform the architecture-specific application state (stack and registers) at runtime is inserted by the compiler in the produced binaries. The toolchain outputs two static binaries, one per ISA, that can be put together into an archive forming a unikernel image ready for migration/checkpoint/restart between servers and embedded boards Ⓓ. For each ISA the code is compiled and linked against multiple libraries: HEXO's kernel library OS Ⓒ, a standard C library (newlib) ported to HEXO's kernel, a library containing the code needed for runtime architectural state transformation, as well as any user-specified library.

At runtime, the x86-64 unikernel binary is first launched on the server Ⓔ and the resulting VM is managed by HEXO's hypervisor, *Uhyve*, using the KVM API. Multiple unikernels are consolidated on the server. A unikernel runs on the server until the scheduler triggers its migration to a board, for example if the server runs out of resources (available cores or RAM). The scheduler runs on the server and monitors resource usage as well as some performance metrics related to each job in order to estimate the slowdown they would exhibit if offloaded to the board. When resource congestion is detected on the server, the scheduler selects the jobs to be offloaded based on multiple criteria, in particular the slowdown expected on the board.

When a job is selected for offloading the scheduler signals the hypervisor, which triggers the heterogeneous-ISA migration process when the guest reaches the next migration point. At that point the guest kernel freezes application execution and rewrites the architecture-specific state for the target ISA, Arm64. The application state is then transferred to the embedded system. Next, the Arm64 binary is bootstrapped on the board, the guest kernel boots and the application state is restored before resuming execution.

Saving, transferring and restoring the VM state is a complex process as part of the VM memory needs not to be transferred and others parts need to be transformed to the target ISA before transfer. Existing VM migration implementations [10, 21] that blindly snapshot the entire guest physical RAM cannot be used. Thus, we design a new VM migration scheme targeting heterogeneous-ISA migration of unikernels, in which the guest OS and hypervisor communicate to correctly extract the part of the VM physical address space that needs to be transferred. We call this method *semantic migration*, in reference to the well-known virtualization concept of the *semantic gap* illustrating the hypervisor's lack of knowledge about a VM's inner workings. By opposition to classical migration schemes that simply transfer a snapshot of the guest physical memory without consideration for the nature of its content, semantic migration requires coordination and information exchange between the guest and the hypervisor. Concerning the transfer method, HEXO offers both checkpoint/restart and post-copy on-demand memory transfer.

In the case where a unikernel needs to be migrated from the board to the server, for example when the scheduler detects some free resources on the server and there is no upcoming jobs, the inverse operation is performed.

3.3 Multi-ISA Unikernel Semantic Migration

Migration Points & Cross-ISA State Translation. Heterogeneous-ISA migration cannot happen at arbitrary points during program execution [51] because there is not always a meaningful mapping of application state across ISAs. Our toolchain instruments the code with migration points ensuring a state of equivalence and making migration possible at these particular points. Equivalence is guaranteed at function boundaries [2, 51], so inserting a migration point corresponds to inserting a function call to a library we developed. It can be placed anywhere in user code, but not in kernel code so that migration does not happen when executing a system call/an interrupt; this greatly simplifies the kernel state to be migrated. This does not alter the flexibility of HEXO as only a negligible amount of time is spent executing the kernel rather than the application in the compute-intensive jobs we target.

The architecture-specific state of an application running as a unikernel is composed of the stack and register content. To be able to transform this state at runtime, we adopt a similar method as in Popcorn [2] (and reuse part of its toolchain) by using a modified version of LLVM/Clang [31]. The compiler records at each migration point the list and location of live values (stack & register slots) on both ISAs. This information is placed in custom ELF sections and loaded in memory at migration time. It is used to perform a rewriting of the stack and register content by placing each live value at the correct location for the target ISA.

Apart from the stack and registers, program data is architecture-independent in HEXO– we assume the same endianness, primitive types sizes/alignments (true between x86-64 and Arm64), and heap management algorithm (we use the same C library on both sides).

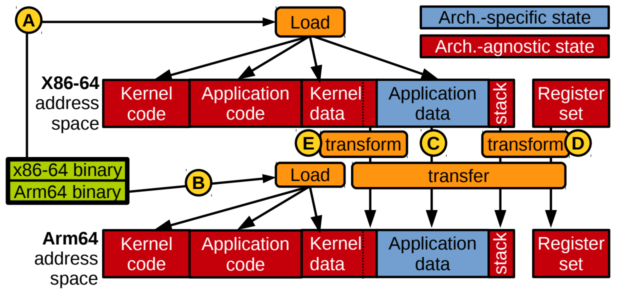

Figure 5: Semantic heterogeneous-ISA migration.

We assume that the program does not make use of non-local gotos (`set/longjmp`). To maintain the validity of functions/data pointers, global variables and functions are located at the same virtual addresses on both ISAs. We use a custom linker script generated by a tool analyzing function and global variable sizes and alignment requirements. The linker, GOLD, is patched to generate a common Thread Local Storage (TLS) layout for both ISAs.

The fact that we do not migrate while processing a system call means that at a migration point the kernel is mostly stateless. Important data structures that are mostly architecture agnostic are extracted on the source and restored on the destination machine: process descriptors, open file descriptors, timer information, etc.

Semantic Migration. We cannot reuse existing VM migration implementations as only some parts of the VM address space need to be transferred. Some areas need to be transformed before transfer and all areas must be identified by the hypervisor or the middleware performing the migration. We define the concept of semantic migration in which the guest OS communicates to the hypervisor information about its address space to bridge this semantic gap.

Figure 5 illustrates semantic heterogeneous ISA migration from x86-64 to Arm64. A unikernel is initially loaded by the hypervisor Ⓐ and segments from the x86-64 binary are written in the guest memory. The kernel initializes and control is passed to the application which starts to execute. When migration is needed, the first step is to perform the same loading process on the target machine Ⓑ with the corresponding Arm64 binary. The kernel initializes on the target machine then the state of the application and kernel are restored, either by transferring and restoring a checkpoint (checkpoint/restart) or in an on-demand fashion (post-copy). Some memory areas are directly transferred Ⓒ and others need first to be transformed to the target ISA Ⓓ Ⓔ. Note that a unikernel boots very fast (25 ms for HEXO's kernel) so booting the guest kernel on the target machine as part of the restoring process is not a concern.

To describe which memory areas should be transferred and maybe transformed, HEXO divides the VM state as the content of the registers plus the content of the memory. The content of the registers is obviously architecture specific and needs to be transformed before transfer. The content of the memory can be further divided between architecture-specific and architecture-agnostic memory areas, with the former describing memory areas in which content would differ considering a program at the same point in its execution on both ISAs, and the latter areas in which content would be identical.

Semantic heterogeneous-ISA migration applies the following rules: (1) directly transfer architecture-agnostic state (Ⓒ on Figure 5); (2) reload architecture-specific read-only state Ⓑ; (3) transform and then transfer architecture-specific read-write state Ⓓ Ⓔ. Architecture-agnostic state is composed of application static memory, i.e., areas where the .bss and .data sections were loaded, and dynamic memory, i.e., the heap and TLS. These can be directly transferred to the target machine during migration. It is in terms of size the largest part of the VM state. Architecture-specific read-only state includes application and kernel code – they are stateless due to their read-only nature so they are simply reloaded alongside other stateless data such as .rodata memory.

Read-write architecture-specific state is composed of the application stack and register set – before transfer these are transformed as previously described. It also includes kernel data – kernel .data and .bss, heap, etc. It is highly architecture-specific as close to 50% of the kernel's LoC is included only for either the x86-64 or Arm64 build. As mentioned earlier HEXO minimizes this state by migrating outside of system call and interrupt processing. The small amount of kernel read-write state left to checkpoint are important data structures needed to correctly resume the application on the target machine: process descriptors, open file descriptors, etc.

State Transfer: Checkpoint/Restart vs Post-Copy. We offer two ways to transfer the state between machines: checkpoint/restart or post-copy [21]. The former consists of dumping the VM state to a file, transferring that file through the network, and restoring the VM state on the target machine. With the latter [21], a minimal checkpoint is transferred (CPU state) and the rest (memory state) is served on-demand from source to target machine. We chose not to implement pre-copy [10] because (A) it generates a lot of network activity which is undesirable on the sometimes slow networks available in embedded boards (e.g. 100Mb/s) and (B) it is highly undeterministic in terms of migration time which does not help the goal of HEXO– freeing resources.

Both techniques have benefits. Checkpoint/restart is useful when the objective is to have a deterministic migration time and to free resources as soon as possible, and post-copy is needed when minimal downtime is required. In the context of HEXO, post-copy also provides significant benefits when, after resuming on the target machine, the unikernel executes then exits without requesting 100% of the memory state. This can considerably reduce the transfer overhead as with a full checkpoint, all the memory state is transferred independently of which amount will be needed on the destination.

3.4 Datacenter Integration and Scheduler

Datacenter Integration. There are multiple ways to integrate HEXO in the datacenter. We believe that pure integration within existing cluster management software [20, 49] would involve a lot of complexity and falls out of the scope of this paper. Indeed, while such software supports managing clusters of heterogeneous nodes in terms of hardware resources, they do not consider the concept of heterogeneous-ISA migration and do not support unikernels. Moreover some have poor or no support for managing nodes of various ISAs and managing embedded systems (due to the amount of resources they require).

Thus we propose a simple integration method allowing the use of existing cluster schedulers with minimal modifications – some servers are augmented with one or a few embedded boards, and each of these machine sets (1 server + boards) is seen as an abstract machine (depicted by the dashed-line rectangle on Figure 4) by the cluster scheduler, with an amount of resources equals to that of the server. The HEXO scheduler runs on the server and takes job migration decisions between the server and the boards. Thus, the cluster scheduler needs only to be updated of the resources usage on the server as jobs move between it and the boards.

Scheduler. The scheduler mainly decides if some job currently running on the server should be offloaded to the board. As we focus on long-running jobs, the goal is to maximize throughput, i.e. how many jobs can be completed in a given period of time by the group of machines consisting of the server and the boards. In order to determine if a job should be offloaded, a central criterion is the slowdown that job would exhibit if run on the board. We do not assume that jobs have been profiled offline: the slowdown is unknown before execution. When scheduling unknown applications on heterogeneous compute units, a central point is to estimate the behavior of an application on one type of unit while observing its behavior on another type [41]. In HEXO's context, we need to compute an estimation of the slowdown a job would incur on the board by observing the job's execution behavior on the server.

We measured on the Xeon server (see Table 1) the instructions per second, last level cache references per second, and last level cache misses per second for each of the NPB [1] benchmarks. These are mostly stable throughout the execution. Using linear regression we found a strong correlation between these three metrics and the slowdowns observed on the Potato board – the correlation r is 0.95 and R^2 is 0.90). These numbers were confirmed by running the same experiment on other boards including the Raspberry Pi. Monitoring these metrics for a job on the servers allows HEXO to estimate with a relatively good accuracy the slowdown that job would incur if ran on the board. It allows the scheduler to offload jobs having the lowest estimated slowdown in order to maximize the overall throughput. Note that currently a job always starts on the server, so the scheduler does not need to monitor performance on the embedded board to determine which job would get the best speedup if migrated back to the server – the speedup is simply the inverse of the slowdown initially estimated on the server. While this assumption does not hold if we consider applications with dynamic behavior, we do not observe such behavior for any of the macro-benchmarks presented in our evaluation.

An additional scheduling criterion is the amount of free resources (cores and memory) on the board and server. We assume jobs are characterized by the number of cores and the amount of memory they require. As the jobs we consider are compute intensive we do not consolidate more than one job per core on both server and boards. The memory available, especially on the board, also sets a hard cap on which and how many jobs can be offloaded.

We assume that the jobs to run are available in a queue. The scheduler considers the next job to execute J and starts by assessing if the amount of RAM and cores needed for the job is available on the server. If it is the case, the job is launched on the server. If not, the scheduler searches for a victim candidate job V to offload on a board among the jobs currently running on the server. That choice is made according to the estimated slowdown of V, which we want to minimize, and of the resources available on the boards. If V is found then it is offloaded and J is launched on the server. Otherwise J is re-queued and the scheduler waits for a job to finish either on the board or on the server. Once this happens the scheduler considers J again and the previous steps are repeated.

Under a steady flow of upcoming jobs, which we believe to be the case in a datacenter, there are no chances for jobs to migrate back to the server from the embedded systems. However in the rare case of an idle period, the scheduler wakes up regularly and, if no upcoming job was detected for a long time, migrates jobs from the boards to the server according to the expected speedup. The scheduling algorithm presented here is relatively naïve and there is a lot of room for improvement. Designing an in-depth scheduler for HEXO is out of the scope of this paper, however we show in the evaluation section that even with a simple scheduler, HEXO can give a significant increase in throughput.

Note that independently of its migration capabilities, a HEXO unikernel image brings portability benefits as it can execute on any of the ISAs it is compiled for. This can be useful in scenarios where particular jobs that are performance insensitive (for example development/debug jobs) may be executed on the embedded system.

4 IMPLEMENTATION

The implementation of HEXO is divided into 3 components: (1) the port of HermitCore's kernel to Arm64 (4,583 LoC added/modified); (2) the support for heterogeneous semantic migration, subdivided into (2.1) kernel and hypervisor support (6,368 LoC) and (2.2) toolchain support (1,806 LoC); (3) the scheduler (400 LoC). The total number of LoC added and modified is 13,157. HEXO is currently implemented on x86-64 and Arm64. Porting to new ISAs would mainly require porting the architecture-specific parts of the kernel and hypervisor as well as porting the state transformation software.

4.1 Porting HermitCore to ARM

HermitCore [30] is a unikernel designed for HPC and Cloud workloads on x86-64 processors. HermitCore's initially focused on HPC in common multicore clusters and combined multi-kernel designs like FusedOS [40], mOS [53], and McKernel [45] with a unikernel design. It was later extended with support for standalone execution without Linux running alongside [30].

General hypervisors such as QEMU [3] exhibit a large overhead in a unikernel context, in particular at initialization time which is critical in HEXO when resuming for migration. HermitCore comes with a lightweight specialized hypervisor *Uhyve*, extended from Solo5 [52] with support for larger guest memory sizes and symmetric multiprocessing (SMP). In contrast to common kernels within QEMU, the boot mechanism starts directly in 64 bit mode and does not rely on inter-processor interrupts to wake up additional cores. Guest boot time can then be reduced from ~2500 to ~25 ms when comparing QEMU with Uhyve. In general, Uhyve realizes a higher abstraction layer than traditional hypervisors, as it does not virtualize on the hardware layer but rather provides an interface to the system software of the host. Due to its swift boot time, we use and heavily modify Uhyve in this paper.

To support multi-ISA migration, we integrated Arm64 support into the Uhyve hypervisor and the HermitCore kernel. For Uhyve, the support for Arm64 is relatively simple. Uhyve uses KVM, the Linux interface to hardware virtualization extensions. KVM supports the hardware virtualization extensions from ARM, Intel and AMD. The differences between ISAs, for example the access method to guest registers upon trap, are small and easy to support.

The kernel configures the Arm64 processor in a comparable mode to the x86-64 configuration: HermitCore runs on both systems in 64 bit mode, uses little-endian, and 4-level page tables. Consequently, on both systems HermitCore uses a page frame size of 4 KB. To avoid Translation Look-aside Buffer (TLB) misses, HermitCore uses 2 MB pages on x86-64 for the code and static data segments. Consequently only dynamically-allocated pages (for example heap pages) have a size of 4 KB. HermitCore supports a feature of Arm64 known as contiguous blocks to efficiently use the TLB – a bit in the page table signals that the page belongs to a 16 KB block which is mapped to physically contiguous blocks of page frames. This hint helps to reduce the number of TLB misses and improves performance. HermitCore runs on Arm64 in *exception level 1*, which is comparable to *ring 0* of x86 and is the typical mode to run a kernel.

4.2 Semantic Heterogeneous Migration

State Transformation. We adapted the LLVM compiler and GNU Gold linker modifications proposed in Popcorn [2], originally developed for Linux, to embed the necessary metadata for heterogeneous migration in our unikernel binaries. This includes modifying the linker and compiler to generate ELF object files and binaries using the HermitCore OSABI identifier. HermitCore was initially compiled with GCC, and a few kernel updates were necessary to enable compilation with LLVM/Clang, such as refactoring nested functions and changing unsupported assembly data types. The Newlib C library used by HermitCore also needed instrumentation. In particular, it was modified to mark the bottom of the stack so that during the stack translation process, when rewriting the stack from top to bottom, the translation runtime knows where to stop. Functions and global variables also needed to be located at the same addresses in both ISAs to preserve the validity of pointers across migration. We developed a python script that analyzes functions and global variable sizes and alignment requirements, and generates custom linker scripts (1 per ISA) to compile an application with aligned functions and global variables across ISAs.

Also adapted from Popcorn [2], the runtime responsible for stack and register translation executes in the unikernel's context. As it originally assumed Linux, modifications were necessary for that software to interface with HEXO's kernel. It also involved a port of Libelf to HEXO for the unikernel to access the binary ELF sections containing the metadata through the host.

Semantic Migration. Triggering migration is a multi-step process as (1) the entity deciding if a unikernel should migrate (the scheduler) is running independently of the unikernels themselves and (2) a unikernel can only migrate when it reaches a migration point.

Inserting a migration point into user code simply corresponds to inserting a function call – `hexo_check_migrate();`. As it is a function call, this creates a state of equivalence between ISAs. Inside this function, our migration library checks if migration should actually happen. This is indicated by a flag in the VM address space that is in a shared memory area between the guest and the hypervisor, accessed with atomic instructions. When the scheduler decides that a unikernel should migrate, it sends a signal to the hypervisor, which immediately sets the flag. Upon reaching the next migration point, the guest will check and then discover that the flag is set, and will begin the translation and migration process. Migration points can be manually inserted at strategic points by the programmer, or fully automatically inserted by the compiler through the use of the `-finstrument-functions` flag.

The stack and registers are transformed in the guest's context. Afterwards, the guest issues a hypercall indicating which areas need to be transferred to the target machine. The hypervisor then checkpoints them to a file for checkpoint/restart, or serves them for post-copy. As mentioned previously, most of the kernel data is architecture-specific and is not transferred as opposed to application data. For the application static data (`.data` and `.bss`) to be efficiently checkpointed or served on demand, it is important for it not to be intertwined with kernel static data. Because HEXO, as a unikernel, is a LibOS, the kernel and the application are compiled together into a single static binary. We must then ensure that application and kernel data are placed on different memory pages. This is achieved by placing kernel static data into separate sections (`.kdata` and `.kbss` from application static data `.data` and `.bss`) and enforcing alignment constraints using `objcopy` and modifications to the linker scripts. A few kernel data structures (process descriptors, open file descriptors, etc.) are transmitted by the guest to the hypervisor which saves them in a file. Currently HEXO does not support the migration of sockets.

When resuming on the target machine, the guest kernel executes a normal boot process. At that point the kernel state is restored and a user task is created. In the case of a full checkpoint restore, the task address space is then restored by the hypervisor by reading the checkpoint file from the host. A stack is allocated for the user task and care is taken for this stack to be at the same location as the stack on the source machine to preserve the validity of pointers to the stack. The transformed stack is restored and the set of transformed registers is put at its top. The task is then marked as runnable and the scheduler is invoked. While scheduling in the task, it pops the register values from the stack into the CPU registers, allowing the task to resume where it was stopped on the source machine.

State Transfer. With checkpoint/restart, once state is ready to be transferred the guest issues a hypercall indicating which memory areas need to be included in the checkpoint. The hypervisor writes this memory into a file along with the registers' content and some metadata including kernel state. At that point guest and hypervisor data structures can be freed on the source machine. The checkpoint is transferred to the destination machine over the network and is restored by the hypervisor after the guest kernel has initialized.

With post-copy, when the state is ready to be transferred, the hypervisor snapshots a minimal checkpoint of the guest. This snapshot contains the transformed CPU register set and stack content, as well as some metadata. The hypervisor next spawns a TCP/IP

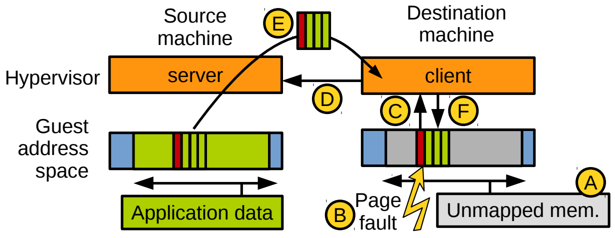

Figure 6: Post-copy batch page fault handling.

server. The minimal checkpoint is transferred to the target machine where it is restored after guest kernel initialization. The next steps are illustrated on Figure 6. When booting, the guest kernel ensures that the areas of the address space corresponding to remote memory (e.g. application data) are unmapped (Ⓐ on Figure 6). The application resumes and accesses to remote memory trigger a page fault Ⓑ. The page fault handler issues a hypercall Ⓒ. The hypervisor is connected to the server, i.e., the other hypervisor on the source machine, and requests the virtual address Ⓓ. The server then transfers the page to the client Ⓔ.

We observed that on-demand paging involves a significant per-page overhead, including a large latency due to the slow networks that may be found in embedded systems. Thus we use batching: the server transfers sets of pages which virtual addresses are contiguous to the page that triggered the fault. Measurements between a typical server and the Potato board show that batching pages by sets of 4 or 8 brings speed improvement of 50% compared to transferring pages one by one. Once the pages are received the client writes them in guest memory Ⓕ and the guest resumes.

In HEXO we migrate for consolidation, i.e. to free resources on the server to accommodate more jobs. With post-copy, as long as the entire memory to be transferred is not transmitted, resources are still occupied on the source machine. This corresponds mostly to RAM as the CPU usage while serving remote memory is minimal. The amount of time to reach the end of a post-copy migration can be undeterministic as it depends on the memory usage of the application. Upon resuming in post-copy mode the guest kernel spawns a kernel thread that has a larger priority than the application and wakes up at regular intervals to proactively pull remote memory. The frequency of this thread is configurable so that a system administrator can set a trade-off between the overhead the thread brings to user code and the length of the post-copy migration.

The server has only a view upon the guest physical memory. Thus, to be able to take a checkpoint or transfer a given virtual page on-demand, a guest virtual-to-physical translation step is needed. That overhead is reduced in HEXO as for the concerned static memory (application data), there is a direct virtual to physical mapping. Thus, only the heap pages require such translation. To obtain a guest physical address from a guest virtual one within the hypervisor, one can use the `KVM_TRANSLATE` command for x86-64. Unfortunately this command is not available for Arm64, so HEXO includes hypervisor code to perform a manual walk of the guest page table to perform such translation, which has the benefit of avoiding a KVM call on the host, i.e. a system call.

Table 2: Checkpoint sizes at half of the execution.

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

4.3 Scheduling Infrastructure

The scheduler is a Python daemon running on the server. It takes as input a list of jobs to run and gathers unikernel images from a dedicated folder. As in a regular datacenter infrastructure, we assume that each job comes with requirements in terms of vCPUs and RAM. Because we target are HPC compute-intensive jobs, we do not consolidate multiple VCPUs on a single physical CPU.

The scheduler monitors performance metrics for the jobs running on the server in order to estimate the slowdown they would incur if offloaded to the board. This monitoring is achieved using the live monitoring function of the `perf-stat` tool with the `kvm` switch. It allows us to obtain at runtime basic performance counter values for a VM running under KVM. In HEXO we do not need to sample the performance counters very frequently – the period is set to 1 second which has no noticeable overhead on performance.

In the case of post-copy migration, the scheduler has to keep track of the unikernel states to avoid (1) killing a unikernel serving remote memory and (2) sending a migration order to a unikernel that did not yet pull its entire address space. To that aim, a file is maintained on the host containing the current state of each unikernel. It is updated by the hypervisor and read by the scheduler to determine if a unikernel is migratable or not.

5 EVALUATION

We evaluate HEXO by showing that a server augmented with one or a few embedded boards of negligible cost (1) provides a better throughput than a single server and (2) is cheaper and more energy-efficient than two servers in consolidated scenarios with macro-benchmarks. We also analyze the migration overhead over a set of compute-intensive macro- and micro-benchmarks.

The server and embedded board are the *Xeon* and *Potato* machines whose characteristics are in Table 1. They are linked with Ethernet, capped at 100Mb/s by to the board's NIC. Both run Ubuntu 16.04 as host, with Linux 4.4 (server) and 4.14 (board). We use a wide variety of serial macro-benchmarks representative of modern HPC compute-intensive datacenter workloads: the shared-memory MapReduce implementation Phoenix [43], PARSEC [5], NPB [1], and the micro-benchmarks Linpack, Dhrystone and Whetstone.

5.1 Migration Overhead

Full Checkpoint/Restart. We use the benchmarks listed in Table 2 and manually trigger migration in full checkpoint mode at half of the execution of each benchmark. We measure the execution times of (A) taking a checkpoint (includes ISA translation) and writing it to a file, (B) transferring it to another machine, and (C) restoring it. Numbers are gathered while migrating at half of the execution of each benchmark, from the server to the board and from the board to the server. Table 2 presents the size of the checkpoint for each benchmark. It is dependent of the size of memory used by the application and is relatively varied among programs, from 1.2 MB (Dhry/Whetstone) to 612 MB (Blackscholes). Application heap, `.data` and `.bss` make the major part of these checkpoints.

Results are presented on Figure 7 in which each data point corresponds to a benchmark run identified by its checkpoint size on the x-axis. As one can observe, all phase times are function of the checkpoint size which is not surprising; the bigger the checkpoint, the more data needs to be written/transferred/read in each phase. An interesting observation is that checkpointing and restoring times differ according to the direction of the migration – checkpointing is slower when going from the board to the server, while restoring is slower the other way. The explanation lies in the difference in terms of storage of the two machines. The SD card used on the board is much slower than the hard disk of the server – checkpointing the 612 MB of Blackscholes is more than 10 times slower on the board (70s) than on the server (6.3s). This is also true for restoring times, even if the slowdown is smaller – about 3x. Restoring is generally faster than checkpointing as it is a read operation and is probably served by the Linux host from the page cache as the checkpoint file was just transferred before restoration.

The board's slow NIC caps the checkpoint transfer speed, independently of the migration direction. For example, it takes close to one minute to transfer Blackschole's checkpoint. These tests point out that a major factor in migration overhead is the slow I/O capabilities of embedded systems, both in terms of network and storage. However, given HEXO's focus on HPC long-running jobs, long migration overheads are not a fundamental limitation. Moreover, post-copy can help to reduce that overhead.

Post-Copy. Post-copy can reduce I/O overhead in two ways. First, it avoids writing/reading a potentially large checkpoint to/from disk, Second, it reduces the network activity compared to full checkpoint/restore in situations where a job finishes on the target machine without requesting the entire data set in memory.

To measure the efficiency of post-copy over full checkpoint transfer, we select jobs with large datasets (see Table 2) and migrate at half of the execution of each. We choose Blackscholes and DC because they do not require the full data set to terminate after resuming from migration. We also select FT and IS as they do require all the data set to terminate. We compare the overheads of full checkpoint transfer vs. post-copy. Concerning the latter, we compute the overhead after resuming by subtracting the time after the restore step in full checkpoint mode from the time after the restore step in post-copy mode.

Results are presented in Figure 8. As one can observe, the overhead reduction brought by post-copy is significant for benchmarks that do not require the entire memory to finish – for example it is reduced by 40% for Blackscholes when migrating from the server to the board, and more than 70% when migrating the other way around. The improvement is due to the lowered network activity

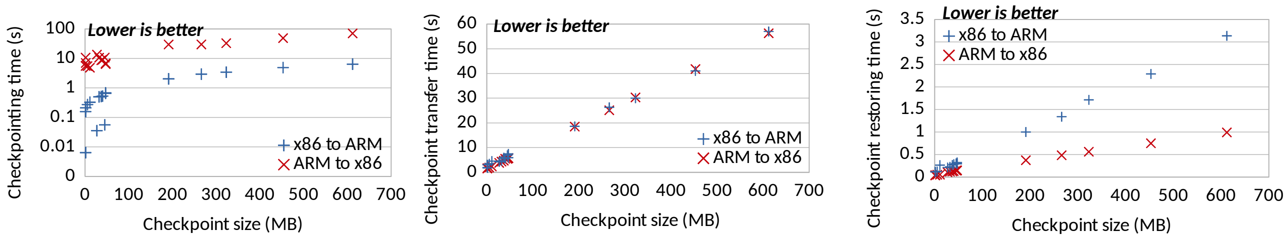

Figure 7: Full checkpoint checkpointing (left), transferring (middle) and restoring (right) times.

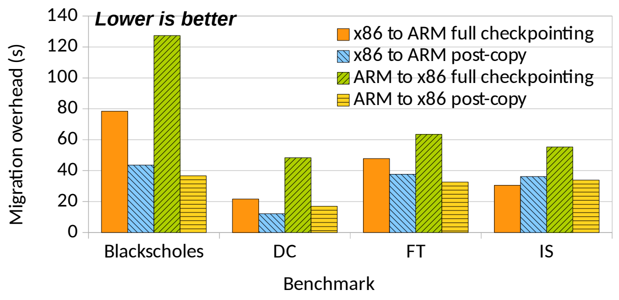

Figure 8: Migration overhead comparison.

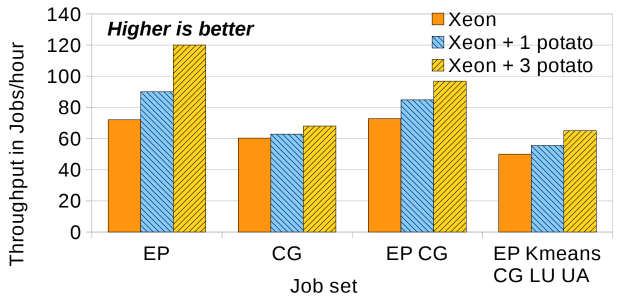

Figure 9: Throughput for HEXO versus 1 Xeon.

brought by post-copy. The difference according to the direction of the migration is because there is no large checkpoint taken and restored with post-copy so the impact of the storage system, especially important when checkpointing on the board, is minimized.

For benchmarks requiring the entire memory to finish after migration, post-copy can still reduce the storage overhead and thus the total migration overhead (48% improvement for FT, 40% for IS) when going from the board to the server. However, when going from the server to the board there is not much storage overhead and in terms of network the entire memory needs to be transferred anyway in both migration modes, thus post-copy does not bring significant benefits. It can even slightly increase the overhead, for example with IS, because of page fault management.

5.2 Consolidated Scenarios

In this experiment we demonstrate the benefits of HEXO in consolidated scenarios, showing that a server augmented with one or a few embedded boards (1) offers a better throughput than a single server and (2) is more energy-efficient than 2 servers. We consider 4 setups: 1 single Xeon server (denoted *X*); 1 server and one Potato board (*XP*); 1 server and 3 boards (*X3P*); 2 servers (*2X*). We arbitrarily choose 3 boards for X3P so that the total price of the boards stays under 5% of the server (see Table 1). We disable hyper-threading on the servers and use only 3 of the 4 cores on each machine (servers and board) as we noticed that running compute-intensive jobs on all cores resulted in congestion and low performance for the scheduler and migration runtimes. It is also common practice to reserve part of the host resources for host software [54].

We use four sets of jobs, managed by HEXO's scheduler: $A = \{EP\}$, $B = \{CG\}$, $C = \{EP, CG\}$, $D = \{EP, Kmeans, UA, LU, CG\}$. We perform one run per set. For each run, an infinite queue of jobs is created by picking jobs from the set one by one in a deterministic order. We choose these particular sets because of the slowdown factor exhibited by the jobs when run on the board compared to the server. For *A*, EP represents the best case for HEXO (slowdown 3X). For *B*, it is the worst case as CG's slowdown is 30X. *C* is a middle-ground and *D* contains a mix of jobs with variable slowdowns in addition to EP and CG: Kmeans (7.5X), UA (10X) and LU (13X). We choose the checkpoint/restart migration method as we want to free resources as soon as possible and all of these jobs require the entire data set to finish after migration.

Throughput. We send each queue to each setup and measure how many jobs are completed after 1 hour, i.e. the throughput. Results are presented on Figure 9. As one can see the only-EP set is the best case for HEXO, which brings a high throughput improvement of 25% (*XP*) and 67% (*X3P*). These good numbers are due to the low slowdown of EP on the board, combined with a small checkpoint size. CG has a high slowdown and is the worst case scenario: the throughput improvement is only of 4.3% (*XP*) and 13% (*X3P*). The other mixes of jobs bring throughput improvement that are far superior to the price increase of *XP* (1.5% more expensive) and *X3*P (4.5% more expensive); the improvement for the CG-EP mix is of 16% (*XP*) and 33% (*X3P*), and for the EP-CG-Kmeans-LU-UA mix it is 11% (*XP*) and 30% (*X3P*). These numbers must be put into perspective with the minimal increase in price: \$45 for *XP* and \$135 for *X3P*. These good results show that the scheduler successfully identifies and offloads the jobs with the lowest slowdowns (EP and Kmeans) in all but the full-CG scenarios.

Energy Efficiency. We measured the power consumed by each system when 3 cores are active and estimated the energy cost of running each queue on each setup for 1 hour and computed how many jobs were completed per kilojoule. Results are presented in

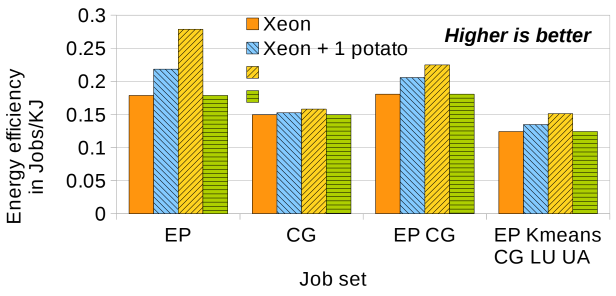

Figure 10: Energy efficiency for HEXO vs 1 and 2 Xeons.

Figure 10. The increases in energy efficiency brought by HEXO are somewhat lower than the increases in throughput as the power consumption ratio between the board and the server is slightly higher than the price ratio. Still, the energy efficiency is always better in HEXO, and significantly increased in some cases. For example for EP-only it is 22% (*XP*) and 56% (*X3P*), and for the EP-CG-Kmeans-LU-UA mix it is 8.5% (*XP*) and 22% (*X3P*). Note that the energy efficiency of *2X* is the same as *X*, because *2X* is twice faster but also consumes twice as much as *X*. We re-ran the consolidation experiments with the post-copy migration method and found no noticeable difference because each of these benchmarks requires the entire data set after migration.

6 RELATED WORKS

Embedded Systems Integration in the Datacenter. Integrating embedded systems in the datacenter has been the subject of past works [12, 15, 17, 19, 24, 27, 28, 33, 42, 44, 48]. Feasibility studies and simulation work [33, 42] have shown the potential benefits but do not propose real implementations. Existing implementations disregard the ISA difference between servers and embedded systems by managing machines of different ISAs separately [27, 44], simply assuming homogeneous ISA [15, 19, 24, 28, 48], or relying on virtualization techniques [3, 9, 12, 17] that negatively impact performance [2]. HEXO proposes a real implementation and focuses on directly executing VMs running native code on the best ISAs in their market domains: x86-64 for servers and Arm64 for embedded systems. Moreover in HEXO the machines of different ISAs are managed together and cooperate to process the same workload.

Embedded Systems and Servers Cooperation. Existing studies propose to offload part of the execution of mobile applications to servers in order to accelerate specific parts of the code [9, 12, 17, 32]. We take the inverse approach where the migration is mostly performed from the server to the embedded system. Our goal is also different as we target data-center workload consolidation. Finally, our design differs significantly as most of these efforts target costly virtualized runtimes [9, 12, 17]. Some consider native code [32], but all are designed to offload only a section of an application, while we migrate/checkpoint/restart an entire virtual machine.

Native ISA Translation Techniques. ISA translation techniques for native code have been discussed in recent years [2, 14, 50]. Most of these work are simulations, assuming hypothetical CPU chips containing multiple cores of different ISAs [14, 50]. Popcorn Linux [2] proposes a real-world implementation. However it strongly differs from HEXO in terms of objective and design, as it targets process migration with the Linux kernel in a server-to-server context, while we focus on VM (unikernel) migration between servers and embedded systems. Helios [39] ships applications in an intermediary format which is recompiled before execution on the target ISA. It requires the developer to write application in a specific language which is hinders programmability and is unacceptable in many situations. In HEXO, we do not require source modification, i.e. there is no effort from the application programmer.

Offloading Execution Flow. Some studies propose using checkpoint/restart to freeze containers [55] and VMs [25, 26] during periods of inactivity to free resources. They share a similar goal to HEXO, consolidation. However the approach and contexts differ: they target mostly-idle/sporadically active applications [55], where it is acceptable not to run at all during idle periods. In HEXO, we consider HPC jobs that are always active, which are offloaded to slower execution units rather than completely frozen.

7 CONCLUSION

We advocate augmenting datacenter servers with one or a few embedded boards of negligible price and power consumption. HEXO offloads at runtime HPC compute-intensive jobs from servers to embedded systems for consolidation purposes. This involves coping with the ISA difference (x86-64 and Arm64), using lightweight VMs suitable for embedded systems (unikernels), and selectively offloading jobs based on an estimation of the slowdown they incur on the board. Evaluation shows a significant increase in throughput and energy efficiency in consolidated scenarios. HEXO's code is available online: http://popcornlinux.org/index.php/hexo.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and our shepherd, David Irwin, for their valuable feedback. We thank Gilles Muller for his precious comments. This work is supported by US ONR under grants N00014-16-1-2104 and N00014-16-1-2711. This research and development was also supported by the German Federal Ministry of Education and Research under Grant 01IH16010C (Project ENVELOPE).

REFERENCES

[1] David H Bailey, Eric Barszcz, John T Barton, David S Browning, Robert L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S Schreiber, et al. 1991. The NAS parallel benchmarks. *The International Journal of Supercomputing Applications* 5, 3 (1991), 63–73.
[2] Antonio Barbalace, Robert Lyerly, Christopher Jelesnianski, Anthony Carno, Ho-Ren Chuang, Vincent Legout, and Binoy Ravindran. 2017. Breaking the boundaries in heterogeneous-ISA datacenters. In *ACM SIGPLAN Notices*, Vol. 52. ACM, 645–659.
[3] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator.. In *USENIX Annual Technical Conference, FREENIX Track*, Vol. 41. 46.
[4] Ricardo Bianchini. 2017. Improving Datacenter Efficiency. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. ACM, New York, NY, USA, 1.
[5] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. ACM, 72–81.
[6] Eric Brown. 2018. Catalog of 116 open-spec hacker boards. http://linuxgizmos.com/catalog-of-116-open-spec-hacker-boards/, Online, accessed 01/10/2019.
[7] Edouard Bugnion, Jason Nieh, and Dan Tsafrir. 2017. Hardware and software support for virtualization. *Synthesis Lectures on Computer Architecture* 12, 1 (2017), 1–206.
[8] Yanpei Chen, Archana Sulochana Ganapathi, Rean Griffith, and Randy H Katz. 2010. Towards understanding cloud performance tradeoffs using statistical

workload analysis and replay. *University of California at Berkeley, Technical Report No. UCB/EECS-2010-81* (2010).

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

[10] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. 2005. Live migration of virtual machines. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation-Volume 2*. USENIX Association, 273–286.

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

[12] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. 2010. MAUI: Making Smartphones Last Longer with Code Offload. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services (MobiSys '10)*. 49–62.

[13] Johan de Gelas. 2016. Investigating Cavium's ThunderX: The First ARM Server SoC With Ambition. Online, accessed 2018/09/15.

[14] Matthew DeVuyst, Ashish Venkat, and Dean M Tullsen. 2012. Execution migration in a heterogeneous-ISA chip multiprocessor. In *ACM SIGARCH Computer Architecture News*, Vol. 40. ACM, 261–272.

[15] Yves Durand, Paul M Carpenter, Stefano Adami, Angelos Bilas, Denis Dutoit, Alexis Farcy, Georgi Gaydadjiev, John Goodacre, Manolis Katevenis, Manolis Marazakis, et al. 2014. Euroserver: Energy efficient node for european micro-servers. In *Digital System Design (DSD), 2014 17th Euromicro Conference on*. IEEE.

[16] Peter Garraghan, PM Townend, and Jie Xu. 2013. An analysis of the server characteristics and resource utilization in google cloud. In *IC2E'13 Proceedings of the 2013 IEEE International Conference on Cloud Engineering*. IEEE, 124–131.

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

[18] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. 2015. Multi-resource packing for cluster schedulers. *ACM SIGCOMM Computer Communication Review* 44, 4 (2015), 455–466.

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

[20] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center.. In *NSDI*, Vol. 11. 22–22.

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

[22] SK Hynix. 2013. H5TQ8G63AMR Datasheet. https://www.skhynix.com/products.view.do?vseq=904&cseq=74.

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

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

[25] Thomas Knauth and Christof Fetzer. 2014. DreamServer: Truly On-Demand Cloud Services. In *Proceedings of International Conference on Systems and Storage (SYSTOR 2014)*. ACM, New York, NY, USA, Article 9, 11 pages.

[26] Thomas Knauth, Pradeep Kiruvale, Matti Hiltunen, and Christof Fetzer. 2014. Sloth: SDN-enabled Activity-based Virtual Machine Deployment. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking (HotSDN '14)*. ACM, New York, NY, USA, 205–206.

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

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

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

[30] S. Lankes, S. Pickartz, and J. Breitbart. 2017. A Low Noise Unikernel for Extrem-Scale Systems. In *30th International Conference on Architecture of Computing Systems (ARCS 2017), Vienna, Austria, April 3–6, 2017*. Springer International Publishing, 73–84.

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

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

[33] Kevin Lim, Parthasarathy Ranganathan, Jichuan Chang, Chandrakant Patel, Trevor Mudge, and Steven Reinhardt. 2008. Understanding and designing new server architectures for emerging warehouse-computing environments. In *ACM SIGARCH Computer Architecture News*, Vol. 36. IEEE Computer Society, 315–326.

[34] Anil Madhavapeddy, Thomas Leonard, Magnus Skjegstad, Thomas Gazagnaire, David Sheets, David J Scott, Richard Mortier, Amir Chaudhry, Balraj Singh, Jon Ludlam, et al. 2015. Jitsu: Just-In-Time Summoning of Unikernels.. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI'15)*. 559–573.

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

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

[37] Micron. 2019. Systems Power Calculators. https://www.micron.com/support/tools-and-utilities/power-calc.

[38] Asit K Mishra, Joseph L Hellerstein, Walfredo Cirne, and Chita R Das. 2010. Towards characterizing cloud backend workloads: insights from Google compute clusters. *ACM SIGMETRICS Performance Evaluation Review* 37, 4 (2010), 34–41.

[39] Edmund B Nightingale, Orion Hodson, Ross McIlroy, Chris Hawblitzel, and Galen Hunt. 2009. Helios: heterogeneous multiprocessing with satellite kernels. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 221–234.

[40] Y. Park, E. Van Hensbergen, M. Hillenbrand, T. Inglett, B. S. Rosenburg, K. D. Ryu, and R. W. Wisniewski. 2012. FusedOS: Fusing LWK Performance with FWK Functionality in a Heterogeneous Environment. *SBAC-PAD* (2012), 211–218.

[41] Vinicius Petrucci, Orlando Loques, and Daniel Mossé. 2012. Lucky Scheduling for Energy-Efficient Heterogeneous Multi-Core Systems.. In *HotPower*.

[42] Nikola Rajovic, Paul M Carpenter, Isaac Gelado, Nikola Puzovic, Alex Ramirez, and Mateo Valero. 2013. Supercomputing with commodity CPUs: Are mobile SoCs ready for HPC?. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM, 40.

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

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

[45] Taku Shimosawa, Balazs Gerofi, Masamichi Takagi, Gou Nakamura, Tomoki Shirasawa, Yuji Saeki, Masaaki Shimizu, Atsushi Hori, and Yutaka Ishikawa. 2014. Interface for Heterogeneous Kernels: A Framework to Enable Hybrid OS Designs targeting High Performance Computing on Manycore Architectures. *2014 21st International Conference on High Performance Computing (HiPC)* (2014), 1–10.

[46] SK Hynix. 2013. H5TQ4G63AFR Datasheet. https://www.skhynix.com/products.view.do?vseq=881&cseq=74.

[47] TrendForce Corp. 2019. DRAM Exchange. https://www.dramexchange.com/.

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

[49] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. 2013. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 5.

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

[51] David G Von Bank, Charles M Shub, and Robert W Sebesta. 1994. A unified model of pointwise equivalence of procedural computations. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16, 6 (1994), 1842–1874.

[52] D. Williams and R. Koller. 2016. Unikernel Monitors: Extending Minimalism Outside of the Box. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*. USENIX Association, Denver, CO, USA. https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/williams

[53] R. Wisniewski, T. Inglett, P. Keppel, R. Murty, and R. Riesen. 2014. mOS: An Architecture for Extreme-Scale Operating Systems. In *Proceedings of the 4th International Workshop on Runtime and Operating Systems for Supercomputers (ROSS '14)*. ACM Request Permissions, New York, New York, USA, 1–8.

[54] Xen Wiki. 2015. Xen Project Best Practices. https://bit.ly/2HpVYQs, Online, accessed 01/10/2019.

[55] Liang Zhang, James Litton, Frank Cangialosi, Theophilus Benson, Dave Levin, and Alan Mislove. 2016. Picocenter: Supporting Long-lived, Mostly-idle Applications in Cloud Environments. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*. ACM, New York, NY, USA, Article 37, 16 pages.