



THE UNIVERSITY *of* EDINBURGH

## Edinburgh Research Explorer

### HEXO

Offloading long-running compute- and memory-intensive workloads on low-cost, low-power embedded systems

**Citation for published version:**

Olivier, P, Mehrab, AKMF, Errabelly, S, Lankes, S, Karaoui, ML, Lyerly, R, Kim, S-H, Barbalace, A & Ravindran, B 2024, 'HEXO: Offloading long-running compute- and memory-intensive workloads on low-cost, low-power embedded systems', *IEEE Transactions on Cloud Computing*, vol. 12, no. 4, pp. 1415-1432. <https://doi.org/10.1109/TCC.2024.3482178>

**Digital Object Identifier (DOI):**

[10.1109/TCC.2024.3482178](https://doi.org/10.1109/TCC.2024.3482178)

**Link:**

[Link to publication record in Edinburgh Research Explorer](#)

**Document Version:**

Peer reviewed version

**Published In:**

IEEE Transactions on Cloud Computing

**General rights**

Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

**Take down policy**

The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact [openaccess@ed.ac.uk](mailto:openaccess@ed.ac.uk) providing details, and we will remove access to the work immediately and investigate your claim.



# HEXO: Offloading Long-Running Compute- and Memory-Intensive Workloads on Low-Cost, Low-Power Embedded Systems

Pierre Olivier, A K M Fazla Mehrab, Sandeep Errabelly, Stefan Lankes, Mohamed Lamine Karaoui, Robert Lyerly, Sang-Hoon Kim, Antonio Barbalace, Binoy Ravindran

**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. We show that sharing long-running, compute-intensive datacenter workloads between a server machine and one or a few connected embedded boards of negligible cost and power consumption can yield significant performance and energy benefits. 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 – and by using the server’s free RAM as remote memory for embedded boards through a transparent lightweight memory disaggregation mechanism for heterogeneous server-embedded clusters, called Netswap. 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 increases in throughput (up to 67%) and energy efficiency (up to 56%) using benchmarks representative of compute-intensive long-running workloads.

**Index Terms**—heterogeneous ISAs, unikernels, migration, offloading.

## I. INTRODUCTION

Datacenters costs are driven by machine acquisition and power consumption/cooling costs [1] and operators are constantly seeking to lower such expenditures. Manufacturers are today producing OS-capable embedded systems at extremely low price points and low power consumption, making them highly attractive for datacenters [2], [3].

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 two orders of magnitude lower than those of servers. Based on these observations, we propose a new approach in which long-running 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 on integrating embedded systems in datacenters suffer from various limitations: managing embedded systems and servers separately [4], [5] does not allow fully exploiting the strengths of each machine type, and simply assuming homogeneous ISAs [6]–[9] forbids benefiting from the combined efficiency of Intel x86-64 servers and ARM-based embedded systems. Virtualization techniques and languages [10]–[13] do help bridge the ISA gap, but they incur non-negligible performance costs.

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 [14] on the CPU for maximized performance. Contrary to existing approaches that relocate part of the execution of mobile applications to servers for performance reasons [10], [12], [13], [15], 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 [16]–[21] 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 [22] where traditional VMs cannot run due to large resources requirements. In this paper we port the HermitCore [17] 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 ISA-

A preliminary version of this paper appears as "HEXO: Offloading HPC Compute-Intensive Workloads on Low-Cost, Low-Power Embedded Systems", HPDC'19. P. Olivier (pierre.olivier@manchester.ac.uk) is with the University of Manchester, UK, S. Lankes (slankes@eonerc.rwth-aachen.de) is with RWTH Aachen University, Germany, Sang-Hoon Kim (sanghoonkim@ajou.ac.kr) is with Ajou University, South Korea, Antonio Barbalace (antonio.barbalace@ed.ac.uk) is with the University of Edinburgh, UK, M. L. Karaoui (mohamed.karaoui@huawei.com) is with Huawei France, and A. K. M. Fazla Mehrab (mehrab@vt.edu), S. Errabelly (sandeep96@vt.edu), R. Lyerly (rlyerly@vt.edu), and B. Ravindran (binoy@vt.edu) are with Virginia Tech, USA.

specific state of an application between ISAs. Existing implementations [23]–[25] 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 ISA-independent subset of the kernel state.

Since embedded boards are typically equipped with memory in the range of 100 MB to a few GBs, high memory-demand applications cannot be migrated to embedded devices. To overcome this limitation, we propose a memory disaggregation technique that allows utilizing the server’s free memory as remote memory for the embedded board. The mechanism, called Netswap, is proposed as an extension to HermitCore’s Virtual Memory Manager (VMM) and a daemon on the server. Netswap addresses the unique challenge of disaggregating the memory between heterogeneous server-embedded board clusters which runs on lightweight unikernel while being transparent to the applications and not requiring any hardware change. It is achieved by modifying the unikernel’s VMM by proactively evicting pages from the board to the server before all physical pages on the board are depleted (using a swap-out mechanism), and fetching remote pages from the server when an application on the board tries to access them (using a swap-in mechanism). To serve the remote pages from the server side, we introduced a standalone daemon called Netswap Remote Daemon (NRD) which manages connections to multiple embedded boards and maintains the remote page to local page address translations. We also introduced a bitmap-based Free Memory Manager (FMM), which overcomes the scalability challenges in HermitCore’s existing FMM. Netswap supports both random and LRU eviction policies.

A final challenge consists in selecting the best candidates to offload from a server to an embedded board. Widely present in the datacenter [26], long-running compute- and memory-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:

- HEXO’s design and implementation. To our knowledge, HEXO is the first system that offloads natively executing unikernels from servers to embedded boards;
- The multi-ISA semantic VM migration technique that allows a unikernel to migrate between diverse ISA CPUs;

TABLE I  
CONSIDERED SERVER & BOARD CHARACTERISTICS.

Machine	Xeon	Potato
ISA	x86-64	Arm64
CPU	Xeon E5-2637v3 , 4 cores (8 HT) @ 3.5 (turbo 3.7) GHz	Amlogic S905X, 4 cores @1.5 GHz
CPU ISA	x86-64	Arm64
RAM	64 GB	2 GB
NIC speed	10 Gb/s	100 Mb/s
Power (idle)	60 W	1.8 W
Power (active)	1 thread: 83 W, 8 threads : 127 W	1 thread: 2.1 W, 4 threads: 2.9 W
Price	\$ 3049	\$ 45

- 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 long-running compute-intensive datacenter workloads. For workloads that exceed embedded boards’ memory, HEXO with Netswap improves throughput and energy efficiency by as much as 40% and 20%, respectively.

## II. MOTIVATION

We motivate HEXO by showing that the cost and power consumption benefits of the embedded systems we consider outweigh in certain cases the low processing power typical of this class of machines. In HEXO, we consider *single-board computers*, popularized by the Raspberry Pi. Such embedded systems satisfy our requirements in terms of price as they are two orders of magnitude cheaper than traditional servers. In terms of resources, they can run medium-sized long-running workloads as unikernels.

We measure the performance and power consumption of a server class machine (Colfax CX1120s-X6, named *Xeon* in the rest of this paper), and a single-board embedded system: Libre Computer LePotato (*Potato*). The machines’ characteristics and prices are given in Table I. The power consumption is measured at the entire machine level using a Kill-A-Watt P4400, while each machine is idle and running 1 and 4 instances of the `stress`<sup>1</sup> program (i.e. 1 and 4 cores/hardware threads active at 100%) for a sufficiently long time.

For this motivation experiment, we use the NAS Parallel Benchmarks (NPB) [27] which are representative of long-running datacenter compute-intensive workloads. The NPB suite includes a set of computation kernels: integer sort (IS), embarrassingly parallel (EP), discrete 3D fast Fourier transform (FT), unstructured adaptive mesh (UA), conjugate gradient (CG), and multi-grid (MG), as well as a series of pseudo-applications: block tri-diagonal solver (BT), scalar pentadiagonal solver (SP) and lower-upper Gauss-Seidel solver (LU). For these tests we use the natively compiled, serial version and the class B (medium data sets). We compute the slowdown incurred for each benchmark while running on the embedded system vs. on the server. Performance results are presented on Figure 1, where execution times on the board are

<sup>1</sup><https://linux.die.net/man/1/stress>.

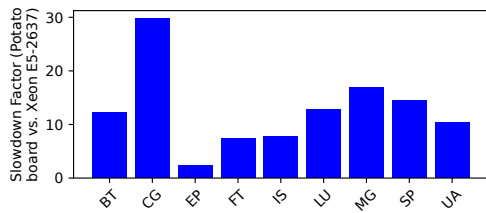


Fig. 1. NPB slowdown when running on the Potato board vs. Xeon server.

normalized to the server’s performance, i.e. 1 on the Y axis represents the execution time on the server.

For some benchmarks, the slowdown is relatively small: in some situations it 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 board; for example, CG is almost 30x slower on the board compared to the server. 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 board’s memory subsystem is slower than the server’s (smaller caches, lower DRAM frequency, etc.). A key idea in HEXO is to offload from the 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 I) and execution times. This reveals that, regardless of the benchmark, it is always more energy-efficient to run on the board, 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. Multithreaded tests and tests over different data set sizes (class A and C) confirmed these observations. In conclusion, these experiments show that such embedded systems are not as slow as they are cheap, and that it is always better from the power consumption standpoint to run on these boards.

We conclude that the costs associated with such 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.

### III. RELATED WORK

**Integration of Embedded Systems in Datacenters.** Integrating embedded systems in the datacenter has been extensively studied in the past [2]–[10], [12]. Feasibility studies and simulation work [2], [3] have shown 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 [4], [5], simply assuming homogeneous ISA [6]–[9], or relying on virtualization techniques [10]–[13] that degrade

performance [23]. 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 machines of different ISAs are managed together and cooperate to process the same workload.

**Native ISA Translation Techniques.** ISA translation for native code have been discussed in recent years [23]–[25]. Most of these work are simulations, assuming hypothetical CPU chips containing multiple cores of different ISAs [24], [25]. Popcorn Linux [23], [28] proposes a real-world implementation. However, it strongly differs from HEXO in terms of objective and design, as it targets process (or containers [29], [30]) migration with the Linux kernel in a server-to-server context, while we focus on VM (unikernel) migration between servers and embedded systems. Helios [31] 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 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 works use checkpoint/restart to freeze containers [32] and VMs [33], [34] 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 [32], where it is acceptable not to run at all during idle periods. In HEXO, we consider long-running jobs that are always active, which are offloaded to slower execution units rather than completely frozen.

**Memory Disaggregation.** HEXO targets heterogeneous server-embedded board clusters and unikernel OSES, and none of the existing memory disaggregation techniques is fit for these goals as they either require the use of Linux, specialized hardware, and/or are non-transparent to applications.

OS-based designs [35]–[38] target high-end server clusters (homogeneous-ISAs) and thus use RDMA for remote memory access, have a Linux OS implementation, and explore design choices such as for prefetching, caching, data sharing, and synchronization. Netswap stands on the shoulders of these works in the way the VMM is modified to utilize remote memory. In contrast, however, Netswap targets diverse ISAs, unikernel OSES, and Ethernet instead of RDMA due to its ubiquitousness for embedded boards. Netswap’s design is also lightweight as it targets unikernels with simple memory management techniques, and the remote daemon is a standalone application. This makes Netswap easily adaptable to other unikernels and different ISAs.

Some techniques [39]–[42] modify the hardware (and OS) to disaggregate memory such as using local DRAMs as virtually indexed caches and memory nodes for address translation instead of the MMU. Netswap does not require specialized hardware and uses Ethernet, widely available among servers/embedded devices. Userspace run-times for accessing remote memory [43], [44] leverage language features for accessing remote memory. Netswap is orthogonal to the language used (although HEXO’s compiler is currently limited to C), and

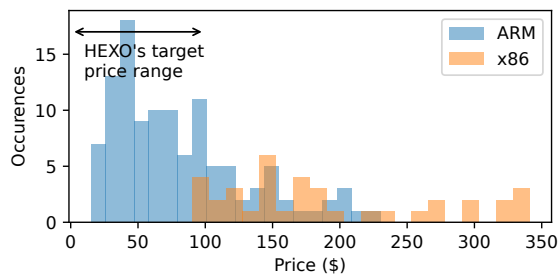


Fig. 2. Prices distribution for 38 x86 and 115 ARM boards.

does not require application modification.

#### IV. DESIGN

##### A. 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 [23], [45]. Concerning x86-based embedded platforms, we demonstrate that ARM platforms are significantly cheaper. To that aim, we select from various sources online (including linuxgizmos website [46]) 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 2. 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 ISA-specific state between ISAs takes a negligible overhead (max 2 ms), so the overhead of VM state transfer, which 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 long-running compute-intensive workloads. These jobs are long-running [26] (from minutes to days) and thus offloading cannot be achieved by killing and restarting regular native binaries [47], 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 [19] than

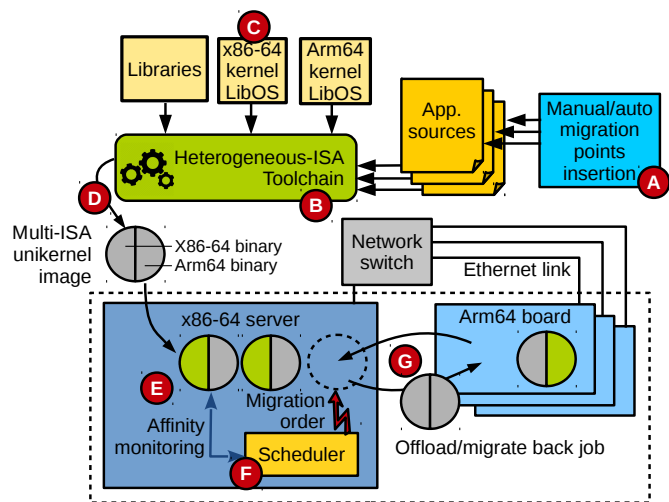


Fig. 3. Overview of HEXO’s building & execution flow.

software solutions such as containers, as confirmed by current trends of running containers inside VMs for security (clear containers [48]). Thus, unikernels are suitable for HEXO.

To reduce performance overhead, we select hardware-assisted directly-executing [14] VMs running native code (C language) as opposed to emulation or managed runtimes [10]–[12] which can help bridge the ISA gap but has 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 makes it possible to use redundancy and offload jobs in parallel on multiple boards for a low additional cost.

##### B. System Overview

Figure 3 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* (A on Figure 3), 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 (B). Metadata needed to transform the ISA-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 (D). For each ISA the code is compiled and linked against multiple libraries: HEXO’s kernel library OS (C), 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 **(E)** 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 ISA-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 other parts need to be transformed to the target ISA before transfer. Existing VM migration implementations [49], [50] 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. In contrast 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 are no upcoming jobs, the inverse operation is performed.

### C. 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 [23], [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

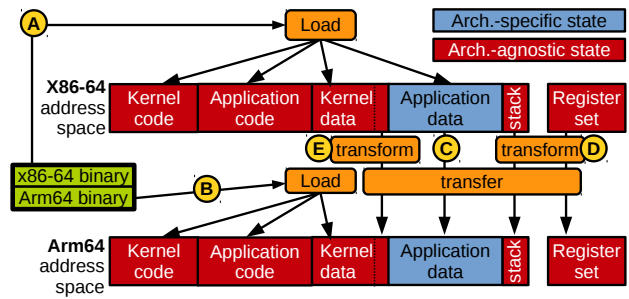


Fig. 4. Semantic heterogeneous-ISA migration.

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 ISA-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 [23] (and reuse part of its toolchain) by using a modified version of LLVM/Clang [52]. 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 stack and registers, program data is ISA-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 ISAs). We assume that the program does not use 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, GNU 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 4 illustrates semantic heterogeneous ISA migration from x86-64 to Arm64. A unikernel is initially loaded by

the hypervisor (A) 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 (B) 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 (C) and others need first to be transformed to the target ISA (D) (E). 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 may be transformed, HEXO divides the VM state into the content of the registers and the content of the memory. The content of the registers is obviously ISA-specific and needs to be transformed before transfer. The content of the memory can be further divided between ISA-specific and ISA-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 ISA-agnostic state (C) on Figure 4); (2) reload ISA-specific read-only state (B); (3) transform and then transfer ISA-specific read-write state (D) (E). The ISA-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 the largest part of the VM state in terms of size. ISA-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 ISA-specific state is composed of the application stack and register set – before the transfer, these are transformed as previously described. It also includes kernel data – kernel `.data` and `.bss`, heap, etc. It is highly ISA-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 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 [50]. 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 [50], a minimal checkpoint is transferred (CPU state) and the rest (memory state) is served on-demand from the source to the target machine. We chose not to implement pre-copy [49] because (A) it generates a lot of network

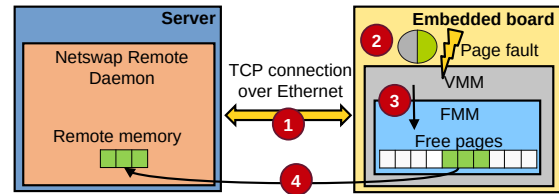


Fig. 5. Netswap's design overview.

activity which is undesirable on the slow networks available in embedded boards (e.g. 100Mb/s) and (B) it is highly nondeterministic 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 HEXO, post-copy also provides significant benefits when, after resuming on the target machine, the unikernel executes and 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 what will be needed on the destination.

#### D. Netswap: Using Server's Free Memory as Board's Remote Memory

Since the embedded boards we target generally have low amounts of onboard memory (typically in the range of 100 MB to a few GBs), applications with high memory needs cannot be migrated to embedded devices. To overcome this limitation, we propose a virtual memory manager (VMM) mechanism that allows utilizing the server's free memory as remote memory for the embedded board. The mechanism, called Netswap, is proposed as an extension to HermitCore's VMM through a modification to its page fault handler and free memory manager (FMM).

Figure 5 presents a high-level overview of Netswap. The server and the embedded boards are connected through TCP/IP-over-Ethernet. The Netswap Remote Daemon (NRD) runs on the server and acts as the remote memory manager for all the embedded nodes. When applications are launched on the embedded board, they establish a connection with the daemon (1). We modify the page fault handler of the unikernel running on the board to implement Netswap. The applications trigger a page fault for allocating memory (2), and the page fault handler requests the FMM to get free pages (3). Netswap modifies this flow to swap-out pages when free pages are depleted and swap in the pages as and when required (4).

The FMM on the embedded board maintains the available physical pages, allocates pages, and reclaims pages during an eviction. HermitCore's FMM is based on a linked list of free pages. For Netswap, the linked list-based free list is not optimal, and we design a bit-map-based free list, discussed later in this section.

**Eviction Candidate.** As previously discussed, when the embedded board's free memory is depleted, Netswap selects a page to evict. The design choices include selecting either a

virtual page or a physical page. We now discuss the pros and cons of both choices and the rationale for our choice of a virtual page-based design. We use the term “local” to refer to the embedded board (e.g. local virtual address and local physical address), and the term “remote” refers to the server (e.g. remote virtual address and remote physical address).

*Physical address-based eviction.* Selecting a physical address for eviction could be seen as an intuitive design choice as there is no need for an address translation before eviction. However, this can lead to complex design and implementation, and degrade scalability. This is because the correspondence between a local physical address and the local virtual address mapping to it must be maintained in the embedded board (which is then used when swapping in the page). Multiple virtual pages can map to the same physical page. Thus, all the virtual addresses associated with a physical address must be saved. In addition, it is necessary to store the local virtual address along with the local physical address in the daemon because the same physical address may be swapped out multiple times and can be allocated to a different local virtual address once it is freed. As the data structure grows in size, this can increase traversal costs. Thus, utilizing a physical address-based eviction candidate can increase the design and implementation complexity and cause scalability challenges.

Physical address-based eviction has advantages. For example, fewer address translations are needed while swapping out since the physical address is already known. Batching does not fragment the physical address space as physically contiguous pages can be batched and evicted.

*Virtual address-based eviction.* If we select a virtual address for eviction, first, it needs to be translated into a physical address, which is followed by evicting the page to the remote server. While this may seem like a slightly more elaborate process during swap-out, it is simple to implement and maintain. This is especially true for a VMM such as that of the HermitCore unikernel which is not as complex as that of the Linux kernel. In such VMMs, using virtual address-based eviction reduces the complexity of swapping. NRD can save the local Virtual Address (VA) to remote VA mappings. Some of the advantages of using a virtual address over a physical address include the following: While swapping in a virtual address, we do not need to know the physical-to-virtual address mappings as the virtual address is already known. Thus, no complex data structures and reverse mappings are needed. In addition, we can avoid concerns regarding multiple virtual addresses being mapped to a single physical address. No additional metadata needs to be sent to and saved on the remote daemon’s end.

Although using virtual address-based eviction reduces the complexity of implementation, it involves additional steps and introduces some new issues. During batch eviction, each local VA needs to be translated to its Physical Address (PA) while preparing the eviction buffer, which adds overhead. Batches of evicted physical pages can get pseudo-fragmented (we discuss batching later in this section). In addition, the virtual address range can change dynamically as memory is allocated on the heap. The eviction policy must accommodate these variations. These disadvantages can be fixed with less

complex solutions compared to problems associated with a physical address-based design. Concerns regarding superfluous network transfers/memory usage on the server may be raised by the fact that several virtual pages can map to the same physical page may raise, but in fact this does not happen in our case: HermitCore does not support shared libraries, shared file mappings, or other constructs such as the Linux physmap [53]. **Eviction Policy.** An eviction policy defines which page(s) is/are evicted when the embedded board runs low on memory. As most embedded boards lack high-speed networks like RDMA and CXL, the number of remote page accesses significantly impacts performance. Netswap provides two eviction policies: random and Least Recently Used (LRU). In the random policy, when a swap-out request is triggered by the page fault handler, a random virtual page from the current heap’s range is selected to be evicted. The major drawback of this design is that, without analyzing which pages are recently or frequently used, hot pages might get evicted. This increases the swap-in requests, as hot pages might get swapped out frequently. Despite its drawbacks, the random policy has the advantage of being simple and not requiring additional data structures for tracking memory accesses.

LRU [54] evicts the least recently used page, which typically reduces the number of swap-in requests, which, in turn, improves the application performance. LRU tracks the page accesses and calculates the least recently used pages. This can be done by the OS during page fault handling and can be accelerated with the help of hardware. Because of caching, prefetching, and TLB hits, the page fault handler is not triggered for most memory accesses. This makes it challenging to determine the exact least recently used page. Hence, an LRU approximation technique must be used. Techniques such as second-chance [55] and clock [56] try to achieve the closest approximation to LRU. Netswap uses the clock algorithm due to its greater efficiency [55]. The algorithm cycles through each virtual page in a circular queue-like pattern.

Prior works [57] show that an eviction policy’s efficiency highly depends on the application’s memory access pattern. If the memory access has a sequential looping pattern, then LRU or other access pattern-based policies do not perform better as each page is accessed at the same rate and pattern [57]. Tracking the page access does not help in this case. The random policy would perform better for such access patterns. This is the case with compute-intensive applications – HEXO’s focus – which generally have iterative, linear memory access patterns. Software-based access tracking adds more overhead when LRU is used. Our experiments (Section VI) confirm this. Since different applications may have different access patterns, Netswap’s design is made configurable to select random or LRU policies.

**Page Access Tracking in ARM.** Most architectures have a bit in the page table to aid page access tracking. In x86 architectures, the CPU sets this bit when a page is accessed, and it can be reset by the OS [58]. However, this hardware feature is introduced in ARM only from version ARMv8.1 [59]. Most embedded boards currently use older architectures. The Raspberry Pi 4, which we used for evaluating Netswap, is based on ARMv8.0. Therefore, we had to implement a

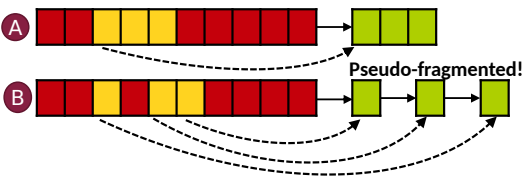


Fig. 6. Linked-list based free list implementation. On top, physically contiguous pages are evicted with physical address, and on the bottom physically non-contiguous pages are evicted with virtual address

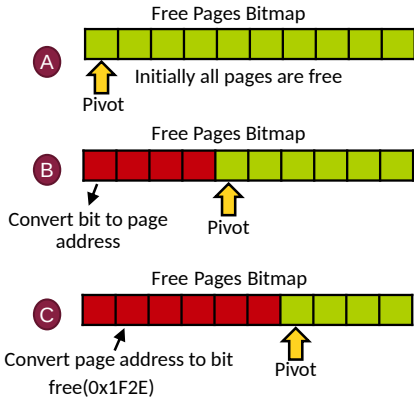


Fig. 7. Free page bit-map implementation.

page access tracing mechanism in software during page fault handling. Our experiments (Section VI) show that this design can significantly increase page faults and offset some of the performance gains obtained by the LRU policy.

For batching, we tried two approaches: batching contiguous pages regardless of the access bit and stopping batching if at least a certain minimum number of contiguous pages are not accessed. The second approach was found to be more effective and was used for batching.

**Free Memory Manager.** The FMM, which is part of Hermit-Core’s VMM, is implemented as a linked list of free pages. Initially, the entire physical address space is a single node in the list, and a pivot is used to allocate pages linearly within a list node. New nodes are created and attached to the list as pages are freed.

When we select the eviction candidate based on the physical address, contiguous physical pages get evicted. It can be added as a single node (Figure 6’s A). However, as we use the virtual address-based eviction candidate, non-contiguous physical pages get evicted. Each page is added as a node (Figure 6’s B). After a few freeing cycles, the memory space can become pseudo-fragmented, meaning that even if there are contiguous free physical pages, combining them for allocation purposes can become difficult.

A linked list-based FMM can be replaced by a bit-map-based FMM design. This design uses a pre-allocated bit map where each bit represents a physical page. The bits are initialized to 1, indicating that the corresponding page is free. The FMM uses a pivot pointer to point to the last allocated page/bit (Figure 7’s A). When pages are requested, the FMM checks if the page pointed to by the pivot is free and returns the address. If not, the pivot pointer is moved until a free page

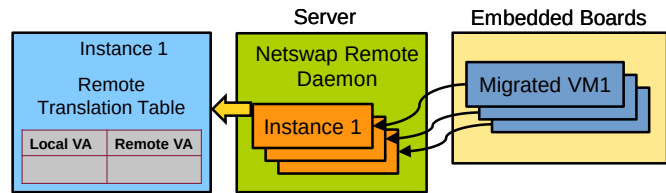


Fig. 8. Netswap Remote Daemon.

is encountered (Figure 7’s B). When pages are evicted and freed, the FMM converts the address to bits and sets the bits to 1 (Figure 7’s C).

**Netswap Remote Daemon.** The Netswap Remote Daemon (NRD) is the only Netswap component running on the server and acts as the gateway for the embedded board to connect to the server. Figure 8 shows NRD’s design. Multiple daemon instances run on the server, and each embedded application/VM connects to a unique daemon instance ①. This design provides isolation between the remote memory of each application. Each daemon instance manages the remote memory for its client application and maintains local VA-to-remote VA address translations ②. Multiple instances can also manage applications running on multiple different boards.

### E. 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 [60], [61] 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 or support unikernels. Moreover, some have poor or no support for managing nodes of various ISAs and managing embedded systems (due to the number 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 in Figure 3) by the cluster scheduler, with an amount of resources similar to that of the server. The HEXO scheduler runs on the server and makes job migration decisions between the server and the boards. Thus, the cluster scheduler needs only to be updated with the resources used on the server as jobs move between the server and the boards.

How the single board computers we consider can be integrated in the datacenter is another point of concern. A possible way to achieve transparently such integration within existing datacenter infrastructures would be to have them in the form of PCIe boards plugged into traditional servers: PCIe-socketable single board computers have already been proposed in the past, e.g. Intel Xeon Phi [62] or Ghost Canyon NUC [63].

**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. 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 [64]. 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 I) the instructions per second, last level cache references per second, and last level cache misses per second for each of the NPB [27] 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 did not observe such behavior for any of the macro-benchmarks presented in our evaluation.

An additional scheduling criterion is the number 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 [1], 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 jobs that are performance insensitive (for example development/debug jobs) may be executed on the embedded system.

## V. IMPLEMENTATION

HEXO's implementation is divided into 3 components: (1) HermitCore's kernel port to Arm64 (4,583 LoC added/modified); (2) support for heterogeneous semantic migration, subdivided into (2.1) kernel and hypervisor support (7,500 LoC, including 1132 LoC for Netswap) and (2.2) toolchain support (1,806 LoC); (3) the scheduler (400 LoC).

### A. Porting HermitCore to ARM

HermitCore [65] is a unikernel designed for HPC and cloud workloads on x86-64 processors. HermitCore initially focused on HPC in common multicore clusters and combined multi-kernel designs like FusedOS [66], mOS [67], and McKernel [68] with a unikernel design. It was later extended for standalone execution without Linux running alongside [65].

Hypervisors such as QEMU [11] exhibit a large overhead in a unikernel context, in particular during initialization which is critical in HEXO when resuming after migration. HermitCore includes a lightweight specialized hypervisor called *Uhyve*, extended from Solo5 [69] 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 is thus reduced from  $\approx 2500$  ms in QEMU to  $\approx 25$  ms in Uhyve. In general, Uhyve has a higher abstraction layer than traditional hypervisors, as it does not virtualize the hardware layer, but rather provides an interface to the host's system software. Due to Uhyve's swift boot time, we use and heavily modify it.

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.

### B. Semantic Heterogeneous Migration

**State Transformation.** We adapted Popcorn [23]’s LLVM compiler and GNU gold linker modifications, 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.

We also adapted Popcorn [23]’s runtime system, which is responsible for stack and register translation, to the unikernel 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 multistep 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

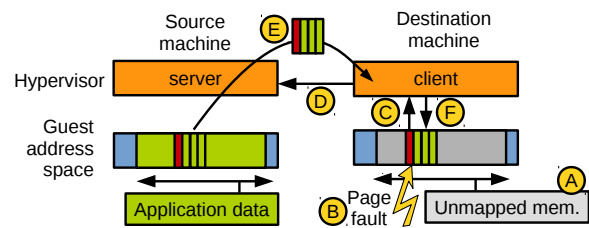


Fig. 9. Post-copy batch page fault handling.

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. Afterward, 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 ISA-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 socket migration.

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’s address space is then restored by the hypervisor from the checkpoint file on 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 the 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.

In post-copy, to transfer the state, the hypervisor snapshots a minimal checkpoint of the guest. The snapshot contains the transformed CPU register set and stack content, and some metadata. The hypervisor then spawns a TCP/IP server. The checkpoint is transferred to the target machine where it is restored after guest kernel initialization.

Figure 9 illustrates the next steps. When booting, the guest kernel ensures that the areas of the address space corresponding to remote memory (e.g., application data) are unmapped (Figure 9's A). The application resumes and accesses the remote memory, triggering a page fault B. The page fault handler issues a hypercall C. The hypervisor is connected to the server, i.e., the other hypervisor on the source machine, and requests the virtual address D. The server then transfers the page to the client E.

We observed that on-demand paging involves a significant per-page overhead, including a large latency due to the slow networks found in embedded systems. Thus, we use batching: the server transfers sets of pages whose 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 yields a speed improvement of 50% compared to transferring pages one by one. Once the pages are received, the client writes them in guest memory F 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 migration, resources remain occupied on the source machine until the entire memory to be transferred is transmitted. This mainly involves RAM, as CPU usage while serving remote memory is minimal. The amount of time to reach the end of a post-copy migration can be nondeterministic 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 of 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.

### C. Netswap Implementation

Figure 10 shows Netswap's control flow. Once an application launches, it establishes a TCP/IP socket connection with the NRD. Netswap introduces an hypercall which is

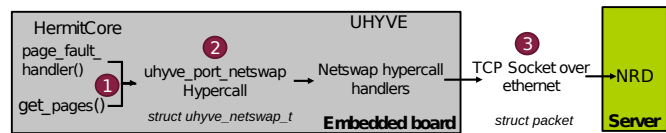


Fig. 10. The Netswap Remote Daemon.

used by the unikernel to communicate swap-in/out requests to Uhyve ②. Swap-out requests originate from HermitCore's physical memory allocator when the free memory drops below a determined threshold, and swap-in requests originates from the unikernel's page fault handler triggered when a swapped-out page is accessed. Uhyve communicates with the NRD via a TCP/IP socket connection over Ethernet ③.

**Swap-out Component.** Figure 11 shows the swap-out flow. When the embedded board runs out of memory, HermitCore's VMM initiates the swap-out sequence. When the physical memory allocator is called to allocate one or more physical pages, a check is made to assess if the amount of free physical memory is less than *free\_threshold*. If it is, the swap-out sequence is triggered. Depending on the configured eviction policy, an eviction candidate is selected from the heap's range ① (We discuss the policy's implementation later in this section). We select the heap as target for Netswap because in the high-memory demand scenarios we target, the vast majority of the memory used by the unikernel will be located in that area.

Netswap attempts to batch 512 adjacent pages ②. Our experiments determined that 512 is the ideal swap batch size for our hardware configuration. Batching will be terminated if a page's *present* or *valid* bit is not set in the Page Table Entry (PTE). The present bit indicates that the page is allocated physically, and the valid bit indicates whether the page is locally available or swapped out. The eviction address and the number of pages in the batch are passed to Uhyve via a custom hypercall.

The swap-out handler then translates the local virtual address to the local physical address for each page in the batch ③ and fills the transfer buffer ④. Uhyve sends the swap-out request to NRD ⑤. NRD allocates the memory and creates mappings between local and remote virtual addresses in the Remote Translation Table (RTT) ⑥. The transfer buffer is then transferred from Uhyve to NRD ⑦. After the transfer, the valid bit for all the transferred pages is set to 0, indicating that the page is in remote memory ⑧, and the physical pages are freed for further allocation ⑨.

**Swap-in Component.** Figure 12 shows Netswap's swap-in technique's implementation. When an application tries to access a page that is in remote memory, a page fault is triggered. In the page fault handler, we verify if the fault was caused by the valid bit being 0. If so, the Netswap swap-in sequence is triggered ①. Batching is done until either a page with valid bit 1 is encountered or the swap batch size (512) is reached ②. New physical pages are allocated to accommodate the pages that must be swapped ③.

Once the new physical pages are allocated, HermitCore passes the address and batch size to Uhyve via the custom

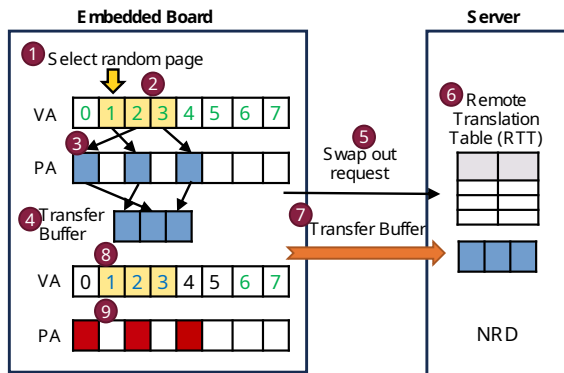


Fig. 11. Swap-out execution flow.

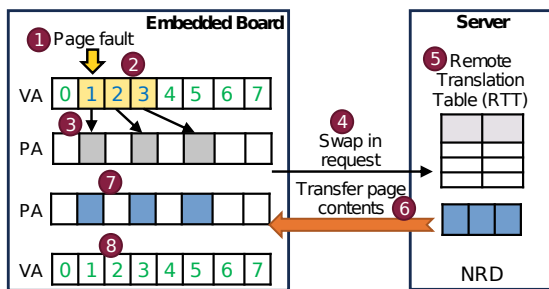


Fig. 12. Swap-in execution flow.

hypercall. Uhyve then sends a swap-in request to the NRD ④. NRD refers to the RTT to translate all the local virtual addresses to their corresponding remote virtual addresses and prepares the transfer buffer ⑤. The contents of the pages are then transferred to client Uhyve ⑥. Upon receiving the page contents, Uhyve saves them into the newly allocated physical pages ⑦. The page table entries for the virtual address are then updated by setting their valid bit to 1, indicating that the pages are now available in local memory ⑧.

**Eviction Policy.** The random eviction policy first determines the number of pages in the heap by finding the difference between the start of the heap and the end of the heap. Since the end of the heap can change during program execution, the range must be recalculated each time. Then a random index within the range is selected and converted to the corresponding address to be evicted in the heap.

As discussed in Section IV-D, Netswap uses LRU’s clock-based approximation [56]. The *lru\_pivot* is used to sequentially examine all the addresses in a clock-like fashion while checking if a page is accessed or not. When a swap-out request is triggered, the *accessed* bit in the PTE corresponding to the address pointed to by *lru\_pivot* is verified. The accessed bit is reset if the page was previously accessed, and the *lru\_pivot* is incremented to the next address if the minimum swap-out batch size is not achieved.

In ARMv8.0, the CPU does not set the accessed bit as the page is accessed [59]. Instead, a page fault is triggered when a page is accessed while the accessed bit is set to 0 and the *page\_fault\_handler()* sets the accessed bit to 1. This page fault is an additional overhead for the LRU policy.

However, Netswap handles this case at the beginning of the *page\_fault\_handler()* to reduce the overhead.

**Free Memory Manager.** Netswap uses a bit-map-based Free Memory Manager (FMM), which was implemented to overcome some of the challenges for implementing the virtual address-based eviction candidate technique discussed in Section IV-D. A bitmap is defined to represent each physical page with a bit. We implemented helper functions to read, modify, and traverse the bitmap.

When a page needs to be evicted, the pivot initially starts from the end of the heap and walks the bit-map backward searching for a candidate. Our experiments (see Section VI) suggested that walking the bit-map from the start to the end of the heap is more likely to evict hot pages due to the memory access pattern of applications. When the pivot reaches the start of the heap, it wraps over to the end of the heap.

#### D. 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 long-running 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.

With post-copy migration and Netswap, the scheduler has to keep track of the state of each unikernel: where it runs (server/board) and if it is pulling remote memory. To that aim, a file is maintained on the host containing the current state for every unikernel. It is updated by the hypervisor and read by the scheduler to determine if a unikernel is migratable or not.

## VI. 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 details are in Table I. They are linked with Ethernet, capped at 100Mb/s by 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 long-running compute-intensive datacenter workloads: the shared-memory MapReduce implementation Phoenix [70], PARSEC [71], NPB [27], and the micro-benchmarks Linpack, Dhrystone and Whetstone.

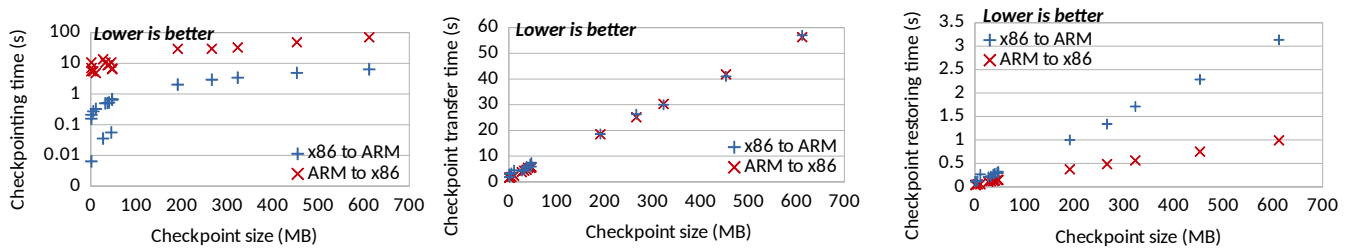


Fig. 13. Full checkpoint checkpointing (left), transferring (middle) and restoring (right) times.

TABLE II  
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	Linpack	1.5 MB
NPB MG (A)	453 MB	Dhrystone	1.2 MB
NPB SP (A)	47 MB	Whetstone	1.2 MB
NPB EP (A)	11 MB		

### A. Migration Overhead

**Full Checkpoint/Restart.** We use Table II’s benchmarks 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 II shows the checkpoint size for each benchmark. It is dependent on the application’s memory size and varies among programs, from 1.2 MB (Dhry/Whetstone) to 612 MB (Blackscholes). Application heap, `.data`, and `.bss` constitute the major part of these checkpoints.

Results are presented in Figure 13 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 a function of the checkpoint size. This is not surprising; the larger 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

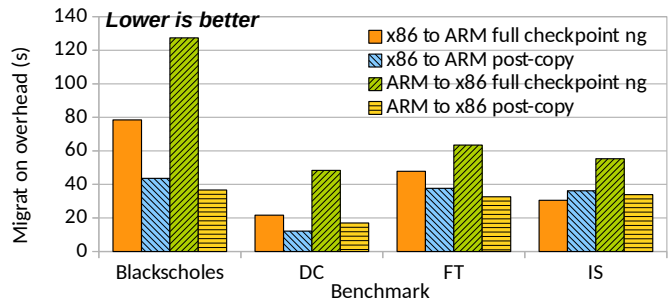


Fig. 14. Migration overhead comparison.

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 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 where a job finishes on the target machine without requesting the entire data set in memory. Migration overheads can then be reduced by transferring, on-demand, only the necessary application’s state, as well as batching transfers to benefit from certain access patterns, post migration.

To measure the efficiency of post-copy over full checkpoint transfer, we select jobs with large datasets (see Table II) 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.

Figure 14 shows the results. As one can observe, the overhead reduction brought by post-copy is significant for benchmarks that do not require the entire memory to finish

TABLE III  
THE 4 SETUPS USED IN CONSOLIDATED EXPERIMENTS.

Setup	1 Xeon server	1 server + 1 Potato board	1 server + 3 boards	2 servers
Abbreviation	<i>X</i>	<i>XP</i>	<i>X3P</i>	<i>2X</i>

– 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 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.

### B. Consolidated Scenarios

Here we demonstrate the benefits of HEXO in consolidated scenarios, showing that a server augmented with one or a few embedded boards (1) offers better throughput than a single server and (2) is more energy-efficient than 2 servers. We consider 4 setups, presented in Table III: 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 I). 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 [72].

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.

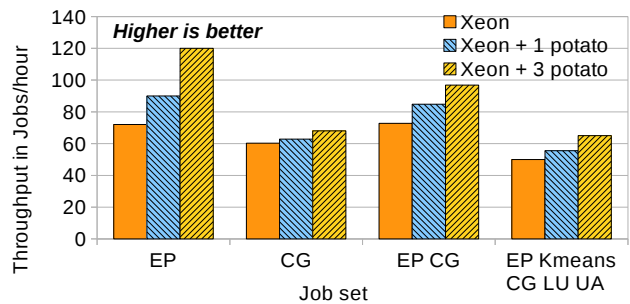


Fig. 15. Throughput for HEXO versus 1 Xeon.

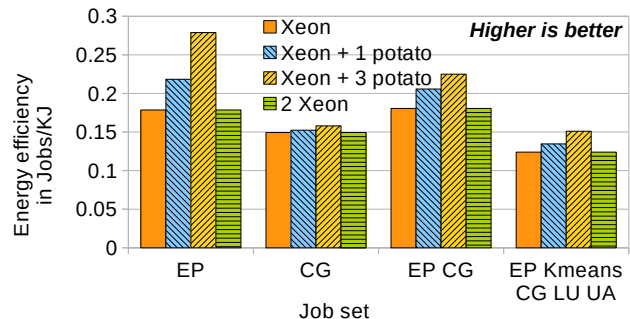


Fig. 16. Energy efficiency for HEXO vs. 1 and 2 Xeons.

**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 15. 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 improvements that are far superior to the price increase of *XP* (1.5% more expensive) and *X3P* (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. Figure 16 shows the results. HEXO's gains in energy efficiency are somewhat lower than the throughput gains 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

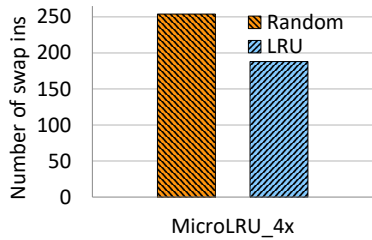


Fig. 17. Number of swaps: random vs. LRU.

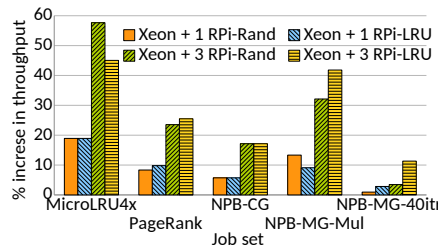


Fig. 18. Netswap's throughput vs. 1 Xeon for Random and LRU policies.

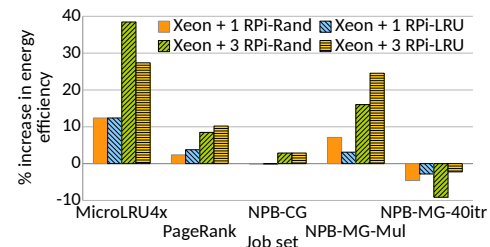


Fig. 19. Increase in energy efficiency for Netswap: Random vs. LRU.

method and found no noticeable difference because each of these benchmarks requires the entire data set after migration.

### C. Netswap's Effectiveness

In evaluating Netswap, our goals include understanding 1) its throughput and energy efficiency gains when considering applications whose memory demand exceeds the embedded board's memory; 2) the effect of the random and LRU eviction policies on throughput and energy efficiency; 3) the effect of application slowdown on Netswap's performance; and 4) the effect of memory demand on Netswap's performance.

**Hardware and Workloads.** We used different hardware for these experiments: Table I's Xeon machine as the server and Arm64-based Raspberry Pi 4 [73] boards (hereafter RPi4) as the embedded boards. In contrast to Potato board's 100Mb Ethernet, the RPi4 features a 1Gb Ethernet connection, which improves network performance. The RPi4 has a total memory capacity of 2 GB, of which 1.5 GB is free after booting up the Ubuntu image. Therefore, any unikernel with memory greater than 1.5 GB would not run without Netswap.

To perform controlled experiments for answering our evaluation questions, we developed a microbenchmark with a configurable Working Set Size (WSS) ranging from 2 GB to 11 GB, and a configurable slowdown on the embedded board with respect to the server (1x, 2x, 3x, and 4x). We also used three programs from the NPB benchmark suite [27]: NPB-CG class B with 235 MB WSS, NPB-MG-MUL, and NPB-MG-40itr with 480 MB WSS. In addition, we wrote a handcrafted pagerank benchmark with 750 MB WSS. The board's available memory is artificially limited for these benchmarks, so they cannot run without Netswap. The details of these micro- and macro benchmarks are:

- **MicroRand1/2/3/4x:** Random policy affine-microbenchmarks with  $\approx 1x/2x/3x/4x$  slowdown between Xeon and RPi4.
- **MicroLRU1/2/3/4x:** LRU policy affine-microbenchmark with  $\approx 1x/2x/3x/4x$  slowdown between Xeon and RPi4.
- **PageRank:** Handcrafted pagerank benchmark.
- **NPB-CG:** NPB-CG class B benchmark.
- **NPB-MG-Mul:** Modified NPB-MG class B benchmark with multiplication statements in a for loop to introduce a delay that matches the slowdown of MG (4x).
- **NPB-MG-40itr:** Modified NPB-MG class B with the number of iterations set to 40 from 20 to increase the execution time with more memory accesses.

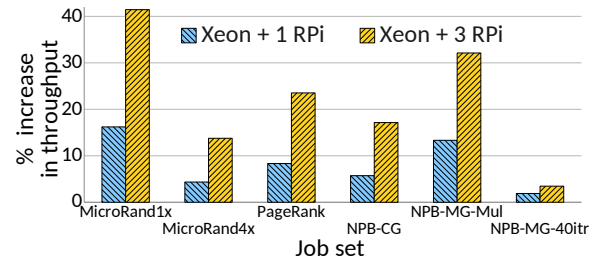


Fig. 20. Netswap's throughput increase vs. 1 Xeon.

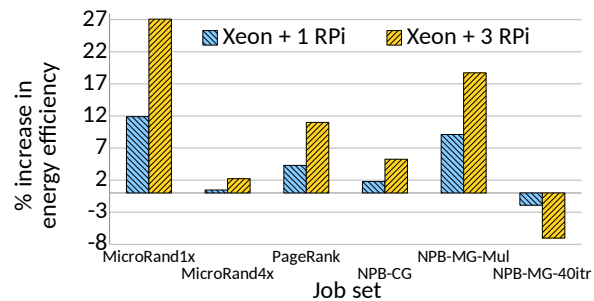


Fig. 21. Increase in energy efficiency for Netswap vs. 1 Xeon.

Similar to Section VI-B's experiments, we launched a queue of jobs on the server, and HEXO's scheduler dynamically migrated jobs to the boards. If a board runs out of free memory during a job execution, Netswap utilizes remote memory.

**Throughput and Energy.** Figure 20 shows the throughput. The results show that for the 1 Xeon and 1 RPi4 cluster (X1P), MicroRand1x has a maximum performance gain of 16%, followed by NPB-MG-MUL with 13% gain. NPB-MG-40itr has the lowest performance gain of 2% as it involves a greater number of iterations of memory accesses, which adds significant network overhead. Similarly, for the 1 Xeon and 3 RPi4 cluster (X3P), MicroRand1x experienced 41% throughput gain, and the least gain is 4% for NPB-MG-40itr.

The energy experiments produced similar results, as shown in Figure 21. MicroRand1x with X3P achieved the most significant energy gain of 28%, followed by NPB-MG-MUL X3P with a 17% gain. However, NPB-MG-40itr's energy efficiency decreased in spite of a throughput gain. This is due to the fact that NPB-MG-40itr's greater memory access reduces the percentage of jobs executing on the board. Most of the throughput is achieved from the Xeon server.

Figure 22(a) shows that 33% of jobs are executed on the

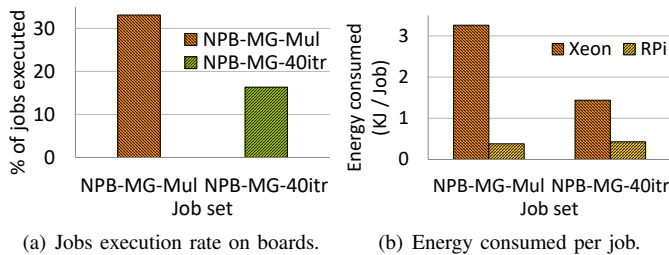


Fig. 22. Execution rate and energy per job for NPB-MG.

embedded boards for NPB-MG-MUL, whereas only 16% of jobs are executed on the boards for NPB-MG-40itr. Also, Figure 22(b) shows that energy consumed per job in RPi4 for NPB-MG-40itr is higher compared to NPB-MG-MUL. In contrast, the energy consumed per job on the Xeon server for NPB-MG-40itr is lower than that of NPB-MG-MUL.

**Random vs. LRU Policies.** We evaluated Netswap’s performance with the random and the LRU policies. We developed an LRU policy-affine microbenchmark, MicroLRU4x, with a memory access pattern that can leverage the LRU policy. We first evaluated the number of swap-in requests by running this microbenchmark under both policies. The application was directly launched on an embedded board instead of being migrated from the Xeon server. Figure 17 shows the results. The figure shows that the LRU-affine application has fewer swap-in requests with LRU than random. However, we observed that random performed better when the jobs were launched on Xeon and migrated to the board.

Figure 18 shows the random versus LRU eviction policy results. The benchmarks pagerank and NPB-MG-40itr performed better with the LRU eviction policy.

Similar to the throughput, energy efficiency also increased in a similar pattern. The results are shown in Figure 19. The random policy yielded better energy efficiency for MicroLRU4x due to the same reason as discussed before. However, for NPB-MG-40itr, energy efficiency decreased with the LRU policy, but only by 2%, compared to a decrease of 9% with the random policy in the X3P setup.

**Effect of Slowdown on Performance.** We investigate the impact on throughput and energy of an application’s slowdown on the embedded board with respect to execution on the server. To that aim, we conducted throughput and energy measurements on the microbenchmarks with increasing slowdown (1x, 2x, 3x, and 4x). Throughput gains (Figure 23 left) decrease consistently as the slowdown increases. The number of jobs executed on the embedded boards decreases as the slowdown increases, which in turn, reduces the overall throughput. Similar to throughput, energy efficiency gains (Figure 23 right) decrease consistently as the slowdown increases.

**Effect of Memory Demand on Performance.** As with other swapping algorithms, Netswap’s swap activity increases with higher memory demand, degrading application performance. To investigate that phenomenon, we used a configurable microbenchmark, MicroRand4x, with a fixed slowdown of 4x and measured throughput and energy efficiency gains. Throughput gains (Figure 24) decrease consistently as the memory demand

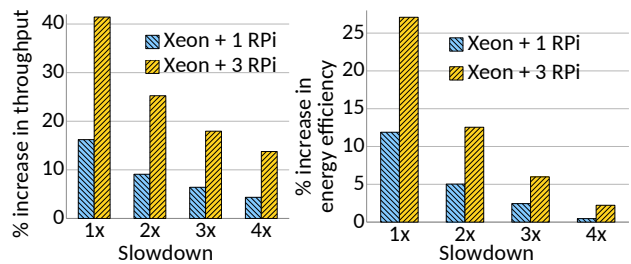


Fig. 23. Netswap’s throughput (left) and energy gains (right) vs. slowdown.

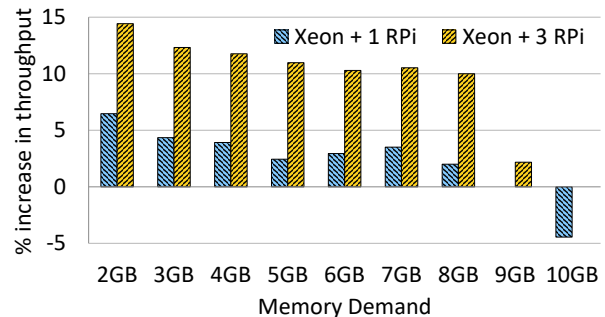


Fig. 24. Netswap’s throughput increase vs. memory demand.

increases. The point of diminishing returns for X1P is  $\approx 9$  GB. X3P fails to run from 10 GB as the server’s free memory is not sufficient to accommodate all the swap requests.

The energy experiments (Figure 25) indicate that energy efficiency decreases from  $\approx 5$  GB. This point of diminishing returns is before throughput’s point of diminishing returns, which is consistent with the trends in Figures 20 and 21.

## VII. CONCLUSION

We advocate augmenting datacenter servers with embedded boards of negligible price and power consumption. HEXO offloads at runtime long-running 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. The evaluation shows a significant increase in throughput and energy efficiency in consolidated scenarios. Notable avenues for future work

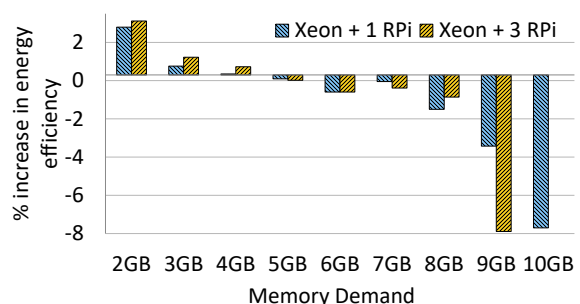


Fig. 25. Netswap’s energy efficiency gain vs. memory demand.

include extending the toolchain to support further languages and libraries, extending Netswap to work across multiple distributed servers and boards, developing advanced scheduling policies and investigating how static analysis can help in that context, considering HEXO's application to ARM servers, and implementing support for socket migration. HEXO's code is available online: <http://popcornlinux.org/index.php/hexo>.

#### ACKNOWLEDGMENTS

This work is supported in part by the US Office of Naval Research (ONR) under grants N00014-16-1-2104, N00014-16-1-2711, N00014-19-1-2493, and N00014-22-1-2672; the US National Science Foundation (NSF) under grant CNS 2127491; the German Federal Ministry of Education and Research (BMBF) under Grant 16ME0688 (Project ScalNEXT); the Institute of Information & communications Technology Planning & Evaluation (IITP) under the Artificial Intelligence Convergence Innovation Human Resources Development (IITP-2024-RS-2023-00255968) grant funded by the Korea government (MSIT); and the UK EPSRC grants EP/V012134/1, EP/V000225/1, and EP/X015610/1.

#### REFERENCES

- [1] R. Bianchini, "Improving datacenter efficiency," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.
- [2] N. Rajovic, P. M. Carpenter, I. Gelado, N. Puzovic, A. Ramirez, and M. Valero, "Supercomputing with commodity CPUs: Are mobile SoCs ready for HPC?" in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013.
- [3] K. Lim, P. Ranganathan, J. Chang, C. Patel, T. Mudge, and S. Reinhardt, "Understanding and designing new server architectures for emerging warehouse-computing environments," in *ACM SIGARCH Computer Architecture News*, vol. 36, no. 3, 2008, pp. 315–326.
- [4] Scaleway, "Scaleway cloud pricing," 2018. [Online]. Available: <https://www.scaleway.com/pricing/>
- [5] A. Krioukov, P. Mohan, S. Alspaugh, L. Keys, D. Culler, and R. Katz, "Napsac: design and implementation of a power-proportional web cluster," *ACM SIGCOMM computer communication review*, 2011.
- [6] Y. Durand, P. M. Carpenter, S. Adami, A. Bilas, D. Dutoit, A. Farcy, G. Gaydadjiev, J. Goodacre, M. Katevenis, M. Marazakis *et al.*, "Euroserver: energy efficient node for European micro-servers," in *17th Euromicro Conference on Digital System Design (DSD)*, 2014.
- [7] K. Keeton, "The Machine: an architecture for memory-centric computing," in *Workshop on Runtime and Operating Systems for Supercomputers (ROSS)*, 2015.
- [8] Hewlett Packard, "HPE Moonshot system," 2018. [Online]. Available: <https://www.hpe.com/us/en/servers/moonshot.html>
- [9] W. Lang, J. M. Patel, and S. Shankar, "Wimpy node clusters: What about non-wimpy workloads?" in *Proceedings of the Sixth International Workshop on Data Management on New Hardware (DaMoN)*. New York, NY, USA: ACM, 2010, pp. 47–55.
- [10] M. S. Gordon, D. A. Jamshidi, S. A. Mahlke, Z. M. Mao, and X. Chen, "COMET: Code offload by migrating execution transparently," in *OSDI*, vol. 12, 2012, pp. 93–106.
- [11] F. Bellard, "QEMU, a fast and portable dynamic translator," in *USENIX Annual Technical Conference, FREENIX Track*, vol. 41, 2005, p. 46.
- [12] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "MAUI: Making smartphones last longer with code offload," in *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2010.
- [13] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "Clonecloud: elastic execution between mobile device and cloud," in *Proceedings of the sixth conference on Computer systems*. ACM, 2011, pp. 301–314.
- [14] E. Bugnion, J. Nieh, and D. Tsafir, "Hardware and software support for virtualization," *Synthesis Lectures on Computer Architecture*, 2017.
- [15] G. Lee, H. Park, S. Heo, K.-A. Chang, H. Lee, and H. Kim, "Architecture-aware automatic computation offload for native applications," in *Proceedings of the 48th international symposium on microarchitecture*. ACM, 2015, pp. 521–532.
- [16] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft, "Unikernels: library operating systems for the cloud," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2013, pp. 461–472.
- [17] S. Lankes, S. Pickartz, and J. Breitbart, "Hermitcore: a unikernel for extreme scale computing," in *Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers*, 2016.
- [18] A. Kantee and J. Cormack, "Rump kernels: No os? no problem!" *USENIX; login: magazine*, 2014.
- [19] F. Manco, C. Lupu, F. Schmidt, J. Mendes, S. Kuenzer, S. Sati, K. Yasukata, C. Raiciu, and F. Hui, "My VM is lighter (and safer) than your container," in *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, 2017.
- [20] P. Olivier, D. Chiba, S. Lankes, C. Min, and B. Ravindran, "A binary-compatible unikernel," in *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, 2019, pp. 59–73.
- [21] P. Olivier, H. Lefevre, D. Chiba, S. Lankes, C. Min, and B. Ravindran, "A syscall-level binary-compatible unikernel," *IEEE Transactions on Computers*, vol. 71, no. 9, pp. 2116–2127, 2021.
- [22] A. Madhavapeddy, T. Leonard, M. Skjogstad, T. Gazagnaire, D. Sheets, D. J. Scott, R. Mortier, A. Chaudhry, B. Singh, J. Ludlam *et al.*, "Jitsu: Just-in-time summoning of unikernels," in *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2015, pp. 559–573.
- [23] A. Barbalace, R. Lyerly, C. Jelesnianski, A. Carno, H.-R. Chuang, V. Legout, and B. Ravindran, "Breaking the boundaries in heterogeneous-ISA datacenters," in *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.
- [24] A. Venkat and D. M. Tullsen, "Harnessing ISA diversity: Design of a heterogeneous-ISA chip multiprocessor," *Proceedings of the 41st International Symposium on Computer Architecture (ISCA)*, 2014.
- [25] M. DeVuyst, A. Venkat, and D. M. Tullsen, "Execution migration in a heterogeneous-isa chip multiprocessor," in *Proceedings of the 17th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2012.
- [26] A. K. Mishra, J. L. Hellerstein, W. Cirne, and C. R. Das, "Towards characterizing cloud backend workloads: insights from google compute clusters," *ACM SIGMETRICS Performance Evaluation Review*, 2010.
- [27] 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 *et al.*, "The NAS parallel benchmarks," *The International Journal of Supercomputing Applications*, vol. 5, no. 3, pp. 63–73, 1991.
- [28] P. Olivier, S.-H. Kim, and B. Ravindran, "OS support for thread migration and distribution in the fully heterogeneous datacenter," in *16th Workshop on Hot Topics in Operating Systems (HotOS)*, 2017.
- [29] A. Barbalace, M. L. Karaoui, W. Wang, T. Xing, P. Olivier, and B. Ravindran, "Edge computing: The case for heterogeneous-isa container migration," in *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2020.
- [30] T. Xing, A. Barbalace, P. Olivier, M. L. Karaoui, W. Wang, and B. Ravindran, "H-container: Enabling heterogeneous-ISA container migration in edge computing," *ACM Transactions on Computer Systems*, 2022.
- [31] E. B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt, "Helios: heterogeneous multiprocessing with satellite kernels," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 2009, pp. 221–234.
- [32] L. Zhang, J. Litton, F. Cangialosi, T. Benson, D. Levin, and A. Mislove, "Picocenter: Supporting long-lived, mostly-idle applications in cloud environments," in *Proceedings of the Eleventh European Conference on Computer Systems*, 2016.
- [33] T. Knauth and C. Fetzer, "Dreamserver: Truly on-demand cloud services," in *International Conference on Systems and Storage*, 2014.
- [34] T. Knauth, P. Kiruvale, M. Hiltunen, and C. Fetzer, "Sloth: SDN-enabled activity-based virtual machine deployment," in *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, 2014.
- [35] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin, "Efficient memory disaggregation with Infiniswap," in *NSDI*, 2017, pp. 649–667.
- [36] H. Al Maruf and M. Chowdhury, "Effectively prefetching remote memory with Leap," in *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*, 2020, pp. 843–857.

- [37] E. Amaro, C. Branner-Augmon, Z. Luo, A. Ousterhout, M. K. Aguilera, A. Panda, S. Ratnasamy, and S. Shenker, "Can far memory improve job throughput?" in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020, pp. 1–16.
- [38] A. Lagar-Cavilla, J. Ahn, S. Souhlal, N. Agarwal, R. Burny, S. Butt, J. Chang, A. Chaugule, N. Deng, J. Shahid *et al.*, "Software-defined far memory in warehouse-scale computers," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 317–330.
- [39] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang, "LegoOs: A disseminated, distributed OS for hardware resource disaggregation," in *13th USENIX Symposium on Operating Systems Design and Implementation*, 2018.
- [40] I. Calciu, M. T. Imran, I. Puddu, S. Kashyap, H. A. Maruf, O. Mutlu, and A. Kollli, "Rethinking software runtimes for disaggregated memory," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021.
- [41] Intel Corp., "Intel scale design architecture," 2018. [Online]. Available: <https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/rack-scale-design-architecture-white-paper.pdf>
- [42] C. Wang, K. He, R. Fan, X. Wang, W. Wang, and Q. Hao, "CXL over Ethernet: A novel FPGA-based memory disaggregation design in data centers," in *2023 IEEE 31st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2023, pp. 75–82.
- [43] Z. Ruan, M. Schwarzkopf, M. K. Aguilera, and A. Belay, "AIFM: High-performance, application-integrated far memory," in *USENIX Conference on Operating Systems Design and Implementation*, 2020.
- [44] C. Wang, H. Ma, S. Liu, Y. Li, Z. Ruan, K. Nguyen, M. D. Bond, R. Netravali, M. Kim, and G. H. Xu, "Semeru: A memory-disaggregated managed runtime," in *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, 2020, pp. 261–280.
- [45] J. de Gelas, "Investigating Cavium's ThunderX: The first ARM server SoC with ambition," 2016. [Online]. Available: <https://www.anandtech.com/show/10353/investigating-cavium-thunderx-48-arm-cores>
- [46] E. Brown, "Catalog of 116 open-spec hacker boards," 2018. [Online]. Available: <https://linuxgizmos.com/catalog-of-116-open-spec-hacker-boards/>
- [47] P. Garraghan, P. Townend, and J. Xu, "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, 2013, pp. 124–131.
- [48] Intel Corp., "Intel Clear Containers," 2018. [Online]. Available: <https://clearlinux.org/documentation/clear-containers>
- [49] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation*. USENIX Association, 2005, pp. 273–286.
- [50] M. R. Hines, U. Deshpande, and K. Gopalan, "Post-copy live migration of virtual machines," *ACM SIGOPS operating systems review*, 2009.
- [51] D. G. Von Bank, C. M. Shub, and R. W. Sebesta, "A unified model of pointwise equivalence of procedural computations," *ACM Transactions on Programming Languages and Systems*, 1994.
- [52] C. Lattner and V. Adve, "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, 2004, p. 75.
- [53] V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis, "ret2dir: Rethinking kernel isolation," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 957–972.
- [54] M. Chrobak and J. Noga, "LRU is better than FIFO," *Algorithmica*, 1999.
- [55] J. Alghazo, A. Akaaboune, and N. Botros, "SF-LRU cache replacement algorithm," in *Records of the 2004 International Workshop on Memory Technology, Design and Testing, 2004*. IEEE, 2004, pp. 19–24.
- [56] F. J. Corbato, "A paging experiment with the Multics system," Massachusetts Institute of Technology, Tech. Rep., 1968.
- [57] S. Jiang, F. Chen, and X. Zhang, "CLOCK-Pro: An effective improvement of the CLOCK replacement," in *USENIX Annual Technical Conference*, 2005.
- [58] OSDDev Wiki, "Paging in x86-64," 2024. [Online]. Available: <https://wiki.osdev.org/Paging>
- [59] C. Marinas, "arm64: Add support for hardware updates of the access and dirty pte bits," 2015. [Online]. Available: <https://patchwork.kernel.org/project/linux-arm-kernel/patch/1436545468-1549-1-git-send-email-catalin.marinas@arm.com/>
- [60] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center," in *NSDI*, 2011.
- [61] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth *et al.*, "Apache Hadoop YARN: Yet another resource negotiator," in *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 2013, p. 5.
- [62] G. Chrysos, "Intel Xeon Phi coprocessor - the architecture," *Intel Whitepaper*, 2014.
- [63] T. S. Ganest, "Intel Ghost Canyon NUC9i9QNX review," 2020. [Online]. Available: <https://www.anandtech.com/show/15720/intel-ghost-canyon-nuc9i9qnx-review>
- [64] V. Petrucci, O. Loques, and D. Mossé, "Lucky scheduling for energy-efficient heterogeneous multi-core systems," in *HotPower*, 2012.
- [65] S. Lankes, S. Pickartz, and J. Breitbart, "A Low Noise Unikernel for Extrem-Scale Systems," in *30th International Conference on Architecture of Computing Systems*, 2017.
- [66] Y. Park, E. V. Hensbergen, M. Hillenbrand, T. Inglett, B. S. Rosenburg, K. D. Ryu, and R. W. Wisniewski, "FusedOS: Fusing LWK Performance with FWK Functionality in a Heterogeneous Environment," *Symposium on Computer Architecture and High Performance Computing*, 2012.
- [67] R. Wisniewski, T. Inglett, P. Keppel, R. Murty, and R. Riesen, "mOS: An architecture for extreme-scale operating systems," in *Proceedings of the 4th International Workshop on Runtime and Operating Systems for Supercomputers (ROSS '14)*. New York, New York, USA: ACM Request Permissions, Jun. 2014, pp. 1–8.
- [68] T. Shimosawa, B. Gerofi, M. Takagi, G. Nakamura, T. Shirasawa, Y. Saeki, M. Shimizu, A. Hori, and Y. Ishikawa, "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*, 2014.
- [69] D. Williams and R. Koller, "Unikernel monitors: Extending minimalism outside of the box," in *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*. Denver, CO, USA: USENIX Association, 2016. [Online]. Available: <https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/williams>
- [70] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrak, "Evaluating mapreduce for multi-core and multiprocessor systems," in *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*. Ieee, 2007, pp. 13–24.
- [71] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. ACM, 2008, pp. 72–81.
- [72] Xen Wiki, "Xen project best practices," 2015. [Online]. Available: [https://wiki.xenproject.org/wiki/Xen\\_Project\\_Best\\_Practices](https://wiki.xenproject.org/wiki/Xen_Project_Best_Practices)
- [73] Raspberry Pi Foundation, "Raspberry Pi 4," 2019. [Online]. Available: <https://www.raspberrypi.com/products/raspberry-pi-4-model-b/>

**Pierre Olivier** is a senior lecturer in the Department of Computer Science at the University of Manchester, UK. His research interests include computer systems and systems security.

**A K M Fazla Mehrab** is a former graduate student at Virginia Tech and is currently a Linux Kernel Engineer at ByteDance Ltd. His research broadly focuses on computer systems.

**Sandeep Errabally** is a former graduate student at Virginia Tech and is currently working at Apple Inc. as System Software Engineer. His research broadly focuses on computer systems.

**Stefan Lankes** is academic director at the Institute of Automation Complex Power Systems, RWTH Aachen University, Germany. His research interests include operating systems, memory systems, and system software for cloud environments.

**Mohamed Lamine Karaoui** is a former Postdoctoral Fellow at Virginia Tech and is currently a senior research engineer at Huawei France.

**Robert Lyerly** is a former graduate student at Virginia Tech and is currently working at Meta Platforms, Inc. as a Research Scientist. His research broadly focuses on operating and distributed systems.

**Sang-Hoon Kim** is an Associate Professor at Ajou University, South Korea. His research interests include operating systems, memory systems, and storage systems.

**Antonio Barbalace** is a senior lecturer in the School of Informatics, at the University of Edinburgh, Scotland, where he works in the Institute for Computing Systems Architecture (ICSA). His research is broadly around systems software.

**Binoy Ravindran** is a Professor of Electrical and Computer Engineering and Bradley Senior Faculty Fellow Endowed Professor at Virginia Tech, where he leads the Systems Software Research Group. His research is broadly in computer systems.