Rethinking Communication in Multiple-kernel OSes for New Shared Memory Interconnects

Antonio Barbalace
Stevens Institute of Technology
antonio.barbalace@stevens.edu

Pierre Olivier
Virginia Tech
polivier@vt.edu

Binoy Ravindran
Virginia Tech
binoy@vt.edu

Abstract

Future computer platforms will likely be built with a multitude of on-chip and off-chip processing units being potentially of different ISAs, OS-capable, and sharing memory with a form of consistency. Multiple-kernel OSes, from multikernels to single-system image OSes, have been demonstrated to mange such platforms efficiently, but they assume no shared memory between kernels as a founding principle. This position paper proposes a new multiple-kernel OS design, which leverages consistent shared memory across homogeneous and heterogeneous processing units in a machine. Among other benefits, this design enables porting commodity SMP OSes to such future platforms, capitalizing on their shared memory programming model, and extend them to multiple-kernel OSes. Herein we present such design, based on two new software primitives tackling the problem of sharing and data format differences between eventually heterogeneous computing units: typed shared memory and type-morphable executable code. We also describe an initial implementation built around Popcorn Linux for x86 and ARM.

1 Introduction

Differences between memory and peripheral interconnects are becoming more blurry, due to lower latencies and higher bandwidths of emerging technologies, such as PCIe 5.0 [63]. Moreover, memory coherency extends outside the memory interconnect, now encompassing external accelerators and I/O devices through technologies such as OpenCAPI [49], CCIX [16], GenZ [26], or CXL [22]. This enables discrete processing units, including CPU, GPU, "smart" peripheral devices, etc., to operate on the same data without burdening the programmer with additional memory operations (i.e., data copy, conversions). Similarly, modern on-chip coherent interconnects, including CCI [3], IF [13], and the NoC in OpenPiton [5], link together on-chip CPU cores, GPU cores, accelerators, etc., facilitating their operations on data.

All such processing units are increasingly offering OS-like functionalities, while I/O devices are becoming "smart", i.e., programmable. For instance, near data/on-stream processing capabilities are now available within storage [24, 28] and network [15, 40, 42, 47] devices; and with GPUs becoming more OS-friendly [29], several studies [36, 56] tried to have them run non-strictly mathematical but control code. Hence, those I/O devices and accelerators may run Operating System (OS) code other than application code [8, 55]. Additionally, SoCs combining multiple heterogeneous-ISA general-purpose processors with cache coherent shared memory have been simulated [60], and prototyped on FPGAs [38]. The upcoming Intel Skylake [31] and Enzian project [53] open additional prototyping possibilities such as RISC-V and x86 or ARM. Each general-purpose processor on the SoC may run OS and application code.

On platforms that integrate multiple, perhaps heterogeneous, processing units, the classic approach is to run an entire software stack (SMP OS and applications) *for each* group of homogeneous processing units arranged on the same coherent interconnect sharing memory (processor island). OSes are independent, and applications need rewriting to run among them. However, pioneered by the multikernel [12, 48, 61], *multiple-kernel* OS designs [9, 11, 12, 39, 48, 50, 54, 59, 61, 64], which run a kernel instance per (group of) processing unit(s) while providing the same OS and OS services among all instances, demonstrated to be a more general OS architecture. In fact, unlike SMP OSes, multiple-kernel OSes are capable of seamlessly manage groups of homogeneous and heterogeneous processing units, from a single to multiple machines – thus, they are ideal for emerging platforms.

In today's multiple-kernel OSes, inter-kernel communication happens via message-passing because kernel instances assume no shared memory – they "share-nothing": *their OS services are written to implement a distributed protocol.* Hence, such OSes are unable to fully exploit the benefits of pervasive coherent shared memory on emerging platforms, where similarly to existent SMPs, communication by messages is expected to be more expensive than using shared memory itself [32, 35, 46]. On the other hand, SMP OSes that are built to exploit coherent shared memory cannot run among heterogeneous processing units [10]. With the perspective

PLOS '19, October 27, 2019, Huntsville, ON, Canada
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-7017-2/19/10…$15.00
https://doi.org/10.1145/3365137.3365399

hardware, is it possible to merge the multiple-kernel OS and classic SMP OS designs to support heterogeneous platforms while capitalizing on shared memory and benefit from the widespread deployability, large developer community, code maturity, large driver base, etc. of classic SMP OSes?

In this position paper we advocate that multiple-kernel OSes, currently based on message passing, should be rethought in order to capitalize on the consistent shared memory offered by emerging computing platforms – similarly to SMP OSes. Consequently, inter-kernels communication on shared memory will perform faster than paying the overhead of 1) marshalling and unmarshalling data to accommodate the inevitable issue of different data formats implemented by heterogeneous processing units; 2) sending/receiving data, including data copies, notify and wake up of the recipient; 3) running a distributed protocol for control and synchronization, etc. Notwithstanding, the problem of how to manage inter-kernel shared memory, perhaps with different data formats, has not been solved before.

Contribution. We propose a new OS architecture targeting emerging platforms with multiple (heterogeneous) groups of processing units interconnected via a form of coherent shared memory: *a multiple-kernel based on shared memory*. Focusing on the issues of sharing memory among kernel instances, and data format differences between heterogeneous processing units, the new multiple-kernel OS is based on two new primitives: *typed shared memory*, and *type-morphable executable code*. Together, these primitives may enable a stock SMP OS to be re-architected as a multiple-kernel OS with minimal effort – i.e., without the necessity of rewriting OS services for distribution. In the long run, we believe that the proposed technique will be streamlined in commodity SMP OSes, and thus maintained at a low cost. We foresee that the new multiple-kernel OS would ease the port of legacy applications using the shared memory programming model to platforms with multiple (heterogeneous) processing islands.

2 Background

The multikernel [12] is the principal example of a multiple-kernel OS – it operates a multicore/processor computer as a network of computing nodes, a "distributed OS in a box". Multiple-kernel OS implementations are either microkernels built from scratch [12, 48, 50, 54, 61, 64], or modifications of classic monolithic SMP kernels [9, 11, 39, 59]. Using exclusively message passing as the communication primitive between kernel instances – while shared memory can be used as an optimization [12], such OSes were conceived to boost scalability on large-SMP machines as well as to support platforms with multiple heterogeneous processing units/memory areas (e.g., Xeon and Xeon Phi, ARM and x86) [9, 11].

An OS that extends among ISA-heterogeneous processing units may face the problem of distribution/migration of applications among those. In fact, applications natively compiled for an ISA cannot run on another, nor span/migrate processes or threads among ISAs, due at least to the instruction and data format differences [9]. Managed code solves the first problem, but not the latter. Therefore, already in the 80s/90s, multiple projects studied the migration of applications in a network of heterogeneous machines [4, 33, 57]. During migration, data conversion to accommodate different formats adds large overheads. More recent works, such as HSA [52] or Popcorn Linux [9], improved the state-of-the-art by imposing a common data format, but this is not doable in all situations and may lead to a large memory overhead.

3 Platform Model

We target a hardware platform in which each processor has its own private memory, and it is sharing memory with a group of possibly heterogeneous processors. Memory is organized in memory islands, areas of memory with homogeneous access characteristics. Those are interconnected via emerging peripheral buses [16, 22, 26, 49] that provision shared memory with a sort of consistency [3, 13, 21, 30, 44]. Based on real prototype profiling [46] and previous work [2, 5] we assume that such platforms will be characterized by memory access latencies, locking scalability, and inter-processor interrupt times similar to existing multi NUMA-node SMPs. While memory access latencies and consistency guarantees vary based on the interconnection and the processor units, we will address these in future work.

Processing units may implement different ISAs, different word sizes (e.g., 32bit, 64bit), and support diverse data formats. Formats are defined by endianness and primitive data types sizes as well as alignment. Concerning endianness, we assume all processors support little-endian as the majority of modern processors does. Despite our multiple-kernel OS applies to platforms with homogeneous and heterogeneous-ISA processor islands, the rest of the paper focuses on the latter, for which we introduce a richer set of features.

4 Design Principles

We propose a new OS design at the midpoint of multiple-kernel and classic SMP OSes: a multiple-kernel OS based on shared memory, founded on two design principles: 1) *for inter-kernel communication, avoid or minimize the use of messaging, hence prefer hardware-implemented consistent shared memory (the opposite of the multikernel)*; 2) *when dealing with the same data having different formats and layouts among processors, instead of keeping codes as-is and convert the data, do the inverse: adapt the code to the data format.*

Communication. The first design principle is motivated by the observation that future computers will provide a form of consistent shared memory between diverse processing units. Despite message passing code being generally considered to scale better than shared memory code [12] on consistent shared memory, message passing has its own, non-negligible sources of overheads: when using messages a conspicuous amount of time is spent in marshaling and unmarshalling, serializing and deserializing, or just converting data – which may outweigh the scalability benefits [35]. Moreover, a lot of work has been devoted to solve the shared memory scalability problems [1], some of which has been integrated in classic SMP OSes, including Linux [23]. Finally, when using shared memory processors are decoupled: they can independently operate on shared data with minimal overhead, if not for synchronizations. Instead with messages, any operation on data creates overhead for both sender and receiver, even without synchronizations [35]. In fact, a message have to be placed in a buffer, a notification sent (e.g., an IPI that may cost tens of us [41]), the receiver interrupts processing needs to handle the message, and it may have to send a message back in the same way. Thus, for performance, shared memory is preferable than message passing.

Using shared memory at the base of a multi-kernel calls for a possible easy adoption of commodity SMP OSes to heterogeneous platforms, which currently have to be re-designed as multiple-kernel OSes: SMP OSes have to be rewritten to support messaging, the standard communication method in today's multiple-kernel OSes. Such rewriting is expensive and requires the addition of distributed protocols as well as the code to handle each data structure [11]. On the contrary, relying on consistent shared memory should allow porting commodity SMP OSes with minimal effort.

Data Format. The second design principle is motivated by the observation that most of the software that we are producing has a fixed data format and a cast in stone executable code segment. In virtue of this, previous approaches to migrate code among heterogeneous processors focused on keeping the code as-is and adapting the data to the new format [4, 33, 57], incurring significant overheads at every migration. A fixed data format and executable code segment ease debugging, and potentially simplify the scan of latent virus signatures. However, previous work on compilers claimed the possible advantages of having continuously optimized code, thus changing the executable code at runtime [37]. Updating code rather than data holds the promise of significant runtime overheads reduction, especially in scenarios where the amount of data to process is significant.

5 Architecture

The proposed OS architecture merges concepts from multiple-kernel and classic SMP OSes by running a different SMP OS kernel instance per processor island, while kernel instances communicates via shared memory. Among other goals, this architecture aims at improving the performance of multiple-kernel OSes adapted from commodity SMP OSes, such as Popcorn Linux [9, 11], reducing the overhead due to message passing by exploiting emerging hardware shared memory.

Because the OS problems of booting and creating a single system environment have been solved already [6, 9, 11, 12, 43], herein we focus on two new software primitives for "transparent" inter-kernel shared memory, namely *typed shared memory* and *type-morphable executable code*, which implement the above design principles, and together the contract abstraction [7]. A contract is metadata that describes the format of a memory area, all processors accessing that memory area adhere to the format by virtue of the contract – the supervisor code on each processor island is responsible to make sure all software running on it observes the contract.

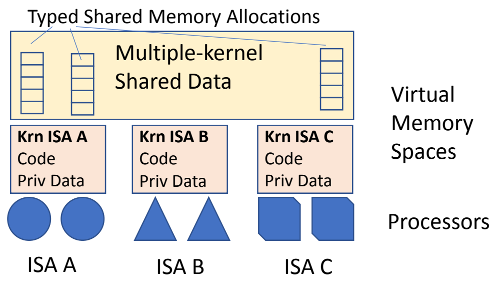


Figure 1. Virtual address space in our design.

Heterogeneous processing platforms are characterized by processors of various ISA and ABI, thus with incompatible executable code, address space layout, data conventions, etc. Each of such processors requires a different kernel executable code, and processor-specific handling. Because of this, for each kernel's address space we enforce a private part and a public part. The latter, contains data shared with other processors. Figure 1 illustrates such virtual address space.

Overview. A multiple-kernel OS boots a kernel image per (group of) processing unit(s). Kernel images are produced from a mostly common codebase, with architecture dependent and independent code. For example, for x86-64 94% of the Linux's (4.9.0) source code is architecture independent (99% for Arm64), not including drivers.

In the proposed architecture, OS services that share data structures among kernel instances still allocate memory through the traditional OS's memory allocators, such as the page or slab allocators. These allocators are extended to take an additional parameter representing a description of the format of the memory object to be allocated. The object description is produced at compile time, and it can be in any format. When a first kernel, called *creator*, allocates a specific shared memory area, the same area does not have to be reallocated neither reinitialized by another kernel wishing to attach to that shared area. Attaching kernels are called *attaché*. The kernel source code is edited to bypass the regular initialization process so that attaché kernels may instead use suspend/resume functionalities already available in some OS kernel subsystems. Moreover, sharing of global variables (`.bss`, `.data`) is supported by an extended kernel loader. Each shared memory area is identified by a global name, a list of which, including an area description, is globally shared among kernels on shared memory. When an attaché kernel attaches to a shared memory area, it checks the object format first. If the format is different from the one the code has been

		ISA 2		
		arm64	**x86-32**	**arm32**
ISA 1	**x86-64**	0.21 MB / 5%	1.71 MB / 55%	1.12 MB / 27%
	arm64		1.2 MB / 34%	1.58 MB / 41%
	x86-32			0.12 MB / 4%

Table 1. Static data memory overhead (in MB and as a percentage of total static data) in bi-ISA heterogeneous systems for a unified data format.

compiled for, the code accessing the data (*accessor code*) have to be updated at runtime, thus keeping the data as-is. Hence, the data format for a memory area is chosen at allocation time. That choice can be made towards different objectives. For example, in order to optimize for memory usage, in a heterogeneous setup including 32 and 64 bits ISAs, one would choose the 32 bit format. Instead, if the data is expected to be mostly manipulated by a particular ISA, to optimize for performance one would choose this ISA's native format.

We believe the described architecture can be integrated in most of the existent SMP OS kernel code bases transparently. However, compiler and runtime code have to be extended to support the proposed functionalities.

Why Not a Pre-defined Unified Format? A unified data format may be enforced [9, 60], but at the cost of memory overhead to satisfy ISA combinations: maximum sizes need to be selected for primitive types, and for alignment constraints the least common multiple must be chosen [9, 60]. This may also impact performance: because the maximum sizes are chosen, ISAs with smaller word size have to use several instructions to access data types.

To demonstrate the negative impact of a unified data format, we studied the static data of the Linux kernel (4.19.0) in the x86-64, arm64, x86-32 and arm32 ISAs. We considered the core kernel (no modules) from the Debian repositories. We found very large numbers of data symbols that are present in all ISAs but have different sizes: from 33% of the total symbols when comparing x86-64 to arm64 up to 80% when comparing arm32 to x86-64. We estimated the memory overhead brought by selecting the largest size to create an hypothetical unified format for a bi-ISA system. Results are presented in Table 1. As one can observe, a unified data format can incur large (up to 55%) memory footprint increases when considering systems composed of 64 and 32 bits processors. This overhead is also a lower bound as we only consider size and not alignment in this study. Thus, sticking with one native format is the best solution – with the observation that in our system the format choice is realized according to memory consumption or performance considerations.

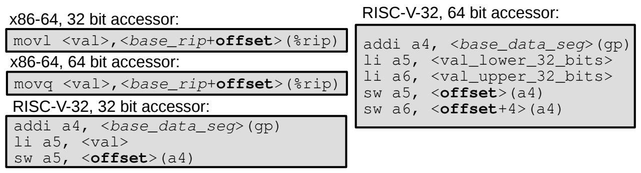
Programming Model. With consistent shared memory between processor islands it would be ideal to use the shared memory programming model for a multiple-kernel OS. However, because of processing units' heterogeneity it is not possible to share the entire memory among kernels, e.g., processor specific boot up variables and routines shouldn't be shared. Thus, the programming model is not straight shared memory, but it incorporates the same memory splitting of PGAS [17, 18, 25]. The proposed typed shared memory is the global part of the PGAS address space, while the partitioned part of the PGAS is the rest (cf. Figure 1). Differently from most PGAS runtimes, there are no explicit remote memory accesses, memory management is still transparent to the programmer and handled by the compiler. Similarly, there are different allocators for local and global memory.

Typed Shared Memory. Typed shared memory is an attempt to create a (shared) memory allocator that associates a data format with an area of memory and a global name. The format describes the exact data types (with sizes), fields offsets in structures, item offsets in arrays, array's memory layout, etc. All accesses to such memory must enforce the specified data type format. The data type format of a typed shared memory is defined at creation time, together with a global name (hash) used to tag the area so that an attaché kernel can look it up and attach to it. When attaching to a typed shared memory area a kernel receives the data type format that it has to use accessing that memory. Below we describe how this information is used by the kernel. The description of the data format may come itself in any format. Examples of formats emitted by popular compilers include DWARF [20] or XML [14] – we are using the former.

In order to fully implement the contract abstraction [7], in addition to the data format specification, a typed shared memory allocator provides also topological information of a specific area of memory. This includes consistency guarantees and the memory latencies in respect to a processor – information that can be used for runtime optimizations.

A practical and safe implementation of typed shared memory requires compiler support. We propose that variables to be used in typed shared memory areas should be marked by the developer with the keyword *__typed*, so that the compiler emits special code and generates type information. Note that this is similar to the way thread local variables are declared in C (`__thread`). For ease of use, automatic conversion routines may be inserted by the compiler where the developer does operate between two instances of the same object that are respectively *typed* and not *typed*. For security, *typed* variables may be checked on pointer access, to make sure the access is within the same typed shared memory area, or another one with appropriate permissions – but not outside.

Type-morphable Executable Code. With the introduction of typed shared memory, an address space presents memory areas with runtime-defined data formats. Note that memory areas with predefined data formats in physical address space are very well known, for example peripheral device's memory; however, their data format is just fixed. To support runtime-defined data formats, the executable code



x86-64, 32 bit accessor:

```
movl <val>,<base_rip+offset>(%rip)
```

x86-64, 64 bit accessor:

```
movq <val>,<base_rip+offset>(%rip)
```

RISC-V-32, 32 bit accessor:

```
addi a4, <base_data_seg>(gp)
li a5, <val>
sw a5, <offset>(a4)
```

RISC-V-32, 64 bit accessor:

```
addi a4, <base_data_seg>(gp)
li a5, <val_lower_32_bits>
li a6, <val_upper_32_bits>
sw a5, <offset>(a4)
sw a6, <offset+4>(a4)
```

Figure 2. Accessor code examples for setting the value of a struct member of different sizes in different ISAs.

must be changed/switched/adapted at runtime to adhere to the published format of a specific typed shared memory area: padding/alignment, data offset, data size, array's memory layout, etc. Therefore, we assume that two kernel instances accessing the same typed shared memory, for example an array of structs, have the same type definition for it, or one's definition is a subset of the other. We also assume that there is no type mismatch or that type mismatches can be converted without losing information. The typed shared memory subsystem makes such checks while attaching a typed shared memory area into a new kernel address space.

Figure 2 shows the generated code for a simple example: setting the value of a `struct` member. We consider x86-64 and RISC-V 32 bits, and give accessor code examples for two possible sizes for the member: 32 or 64 bits. The generated code varies with the data format. First, according to the member size, `movq` vs `movl` will be used for x86-64, or one vs two phases stores for RISC-V. Second, the **`offset`** in bytes of the member within the containing object (in bold in Figure 2) will vary according to previous members sizes/alignment constraints, and the alignment of the member. Finally, the location of the data structure in the address space (in italic) will be PC-relative for x86-64, `base_rip`, or relative to the start of the data segment for RISC-V, `base_data_seg`.

To change, switch, or adapt the code at runtime (generically called morph) there are multiple options. The most naive is to pre-compile the code that will access typed shared memory in multiple formats and at runtime choose which version to use. As the number of existing popular ISA/ABI is not enormous, this appears feasible, but it may not scale to complex data types that could require hundreds of accessor code versions. Another option is to hot-patch [51, 62] the code or rewrite it at runtime. Further options include compiling the accessor code in an intermediate language and JIT compile it to adhere to the advertised format – there is no need to have a fully fledged JIT compiler for this, but just one supporting a limited number of instructions. Finally, code can adapt to data structures by simply using offsets and (built-in) data conversion routines. The type-morphing method is an implementation detail (see Section 6).

Inter-kernel Data Sharing. The long-term goal of the proposed design is to share as many data structures as possible between the multiple kernel instances of the OS. In fact, for data structures that can be accessed from each compute unit, this allows to save memory and also provides transparent synchronization as opposed to the use of a complex distribution/replication protocol. We describe a few examples of those below. Each may apply to any of the classic OS designs, including monolithic, microkernel, and exokernel.

Process Management. Multiple data-structures related to process management can be shared. For example, the process descriptors (in Linux the `task_struct`) are mostly architecture-independent and should be shared. It would ease maintaining the state of threads migrated, spawned, or distributed among heterogeneous computing units.

Memory Management. Sharing some data structures related to memory management such as virtual memory area descriptors would help the different kernel instances to maintain a consistent view of the PGAS virtual address space.

Statistics and Profiling. Sharing data structures related to hardware performance counters would help heterogeneous schedulers to make more sound decisions. For example, in a work stealing scheduler one processing unit may want to examine the performance of a thread on another unit to decide if stealing should happen or not.

IO Subsystem. Finally, sharing data structures of I/O subsystems would significantly help in efficiently offering an homogeneous view of I/O resources to programs. Tasks should see a consistent view of the filesystem, which involves sharing many control data structures of the virtual and actual filesystems, as well as sharing caches such as the page/inode/dentry cache for Linux. Sharing socket buffers also allows software on different processing units to deal with incoming packets independently of the device that produced them.

6 Implementation

We are working on a proof of concept based on the Popcorn Linux [9, 11] kernel and compiler framework targeting a heterogeneous platform with an x86-64 (AMD Epyc 7451) and an ARM 64bit server (Cavium ThunderX). The servers are interconnected via Dolphin PXH810 PCIe adapters [58] providing sub-microsecond shared memory as well as DMA transfers. This setup is at an early stage of development, and aims to emulate emerging platforms with shared memory across the peripheral bus.

Extended slab allocator. We modified the Linux's slab allocator (slub) to accept on create the data format associated with a memory area. Because Linux's slab creation runtime (`kmem_cache_create()`) already requests an identification string, we used it to identify a typed shared memory (and its format description). An array with fixed data format is used to advertise typed shared memory objects on globally accessible memory to other kernels. A kernel boot parameter provides the physical memory address of this array to the typed shared memory slab allocator. Thus, only the first kernel to boot creates such a table. When the slab's create function is called, the code checks in the table if an area of memory with the same identifier has been already created and if the data type format differs, it returns an error. Hence, the code uses the new `kmem_cache_query()` to request the data type format, "morphs" the data access code (see below), and calls the slab's create function again. Note that there are parts of the kernel code that are not using the slab allocator, thus calling the page allocator directly or through `kmalloc()`, for the sake of a prototype we converted a minimum amount of `kmalloc()` in slab calls.

We are currently modifying OS's initialization and teardown routines in order to avoid for attaché kernels reallocation or earlier deallocation of data structures in a typed shared memory. This includes also code in the early boot process (Linux's code in `arch/*/boot/`) that enable the sharing of static memory (`.bss` and `.data` variables) (cf. Table 1).

Compiler support. For the time being we use pre-processed sources in order to construct data structures descriptions and compile them into the kernel image itself (similar to `asm-offsets.c`). However, we are planning to automate the generation of such information with compiler support.

We added the `__typed` compiler's storage class keyword in GCC, and modified the compiler to generate machine code that supports runtime morphable data types. For each `__typed` variable, the compiler allocates a global data structure, the *type table*, defining for each struct's data field its offset, size, and eventually a conversion function; for arrays, each element displacement, helper code to support different array's memory layouts, etc. When a `__typed` variable is accessed, the type table is used to generate the address of its elements. Type tables are stored in memory and can be accessed and changed by any kernel code. To protect them from unintentional modifications they are in a specific compilation section that is protected in writing. Currently, only functions from the slab allocator are allowed to unprotect, modify, and protect such areas. We plan to evaluate the added overhead of using type tables, as well as automate the process of declaring variables as `__typed`.

When invoking C's `sizeof()` on `__typed` variables, it will return the size value saved in the type table. Thus, making `sizeof()` dynamic. Finally, we plan to force the compiler to bound check memory accesses to the typed shared memory.

Inter-kernel communication. To simplify constructing our prototype, only a subset of OS services are being converted to use the proposed primitives. We use Popcorn Linux's provided ones for the rest – which is the reason we used Popcorn Linux instead of vanilla Linux. Therefore, not all slab allocations are rewritten as typed shared memory, Finally, as a design choice that fits with the PGAS programming model the same typed shared memory is present at the same virtual address in all kernel instances.

OS Subsystems. On the target hardware, we plan to share data structures from the storage and network OS subsystems – we will consider other subsystems in future work. Specifically, we aim to share filesystems control structures (below VFS) within the storage subsystem, including the page cache data and the page cache control structures. In the network subsystem, we aim to shared the Linux's socket buffers.

7 Related Work

Typed Shared Memory. The most similar concept to typed shared memory are data units [19]. Like typed shared memory, data units aim to replace message passing via a higher-level shared memory abstraction. However, data units do not preserve the shared memory programming model.

Debuggers, high-level languages, and language runtimes (e.g., Java) implement a (runtime) type system. Assembly and C languages, used to develop traditional OS kernels do not, thus our proposal builds up on such previous works.

Morphable code and flexible address spaces. Morphable code, or self-modifying code, not intended as a virus nor a bug, has been introduced before for various applications [27, 34]. Moreover, LLVM [37] proposed continuous runtime optimization for performance. Additionally, hot and runtime patching have been used for live fixing bugs [51, 62] and performance improvements based on different hardware (Linux's alternatives). Differently, in this paper code modifications are introduced to adapt the data layout at runtime.

Operating Systems. We introduce a new design to build a multiple-kernels OS where kernel instances access each other internal data structures over shared memory – like a SMP OS. MutekH [45] is the earliest work introducing "shared everything" among OS kernel instances. However, it is based on an exokernel and libOS – the exokernel, the lowest layer of the software, does not exploit shared memory, while in our proposal, the lowest level of the software use shared memory, which is fairly more complicated and performant. K2 [39] claims to adopt a similar concept of "shared-something" in Linux. However, its implementation fully differs from ours, does not introduce any new OS abstraction, and it doesn't apply to heterogeneous-ISA platforms.

8 Conclusion

Considering the trend through pervasive consistent shared memory among eventually heterogeneous and OS-capable processing units in emerging computing platforms, we observed that existent multiple-kernel OSes do not exploit it. By fully leveraging consistent shared memory multiple-kernel OSes can boost their performance, while SMP OSes can be extended to run on such hardware by leveraging ideas from multiple-kernel OSes – with relatively low effort. We described a new OS architecture including two central concepts to leverage inter-kernel shared memory as well as heterogeneity, *typed shared memory* and *type-morphable code*, as well as our initial implementation based on Popcorn Linux.

Acknowledgments

This work is supported in part by ONR under grants N00014-16-1-2104, N00014-16-1-2711, and N00014-18-1-2022.

References

[1] Shameem Akhter and Jason Roberts. 2006. *Multi-core programming*. Vol. 33. Intel press Hillsboro.

[2] Johnathan Alsop, Matthew D Sinclair, and Sarita V Adve. 2018. Spandex: a flexible interface for efficient heterogeneous coherence. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*. IEEE Press, 261–274.

[3] ARM. 2019. ARM Website - CoreLink Cache Coherent Interconnect Family. https://developer.arm.com/ip-products/system-ip/corelink-interconnect/corelink-cache-coherent-interconnect-family.

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

[5] Jonathan Balkind, Michael McKeown, Yaosheng Fu, Tri Nguyen, Yanqi Zhou, Alexey Lavrov, Mohammad Shahrad, Adi Fuchs, Samuel Payne, Xiaohua Liang, et al. 2016. Openpiton: An open source manycore research framework. In *ACM SIGARCH Computer Architecture News*, Vol. 44. ACM, 217–232.

[6] Amnon Barak and Oren La'adan. 1998. The MOSIX multicomputer operating system for high performance cluster computing. *Future Generation Comp. Syst.* 13, 4-5 (1998), 361–372.

[7] Antonio Barbalace and Anthony Iliopoulos. 2017. Address Space and Executable Formats, Such Old Topics!. In *Proceedings of the 7th Workshop on Multicore and Rack-Scale Systems (MaRS's 17)*.

[8] Antonio Barbalace, Anthony Iliopoulos, Holm Rauchfuss, and Goetz Brasche. 2017. It's Time to Think About an Operating System for Near Data Processing Architectures. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*. ACM, 56–61.

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

[10] Antonio Barbalace, Alastair Murray, Rob Lyerly, and Binoy Ravindran. 2014. Towards operating system support for heterogeneous-isa platforms. In *In Proceedings of The 4th Workshop on Systems for Future Multicore Architectures (SFMA)*.

[11] Antonio Barbalace, Marina Sadini, Saif Ansary, Christopher Jelesnianski, Akshay Ravichandran, Cagil Kendir, Alastair Murray, and Binoy Ravindran. 2015. Popcorn: Bridging the Programmability Gap in heterogeneous-ISA Platforms. In *Proceedings of the Tenth European Conference on Computer Systems (EuroSys '15)*. 29:1–29:16.

[12] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. 2009. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP '09)*. 29–44.

[13] Noah Beck, Sean White, Milam Paraschou, and Samuel Naffziger. 2018. 'Zeppelin': An SoC for multichip architectures. In *2018 IEEE International Solid-State Circuits Conference-(ISSCC)*. IEEE, 40–42.

[14] Tim Bray, Jean Paoli, C Michael Sperberg-McQueen, Eve Maler, and François Yergeau. 1997. Extensible markup language (XML). *World Wide Web Journal* 2, 4 (1997), 27–66.

[15] Broadcom. 2019. Stingray SmartNIC Adapters and IC. https://www.broadcom.com/products/ethernet-connectivity/smartnic.

[16] CCIX Consortium. 2017. Cache Coherent Interconnect for Accelerators (CCIX). http://www.ccixconsortium.com/.

[17] Bradford L Chamberlain, David Callahan, and Hans P Zima. 2007. Parallel programmability and the chapel language. *The International Journal of High Performance Computing Applications* 21, 3 (2007), 291–312.

[18] Barbara Chapman, Tony Curtis, Swaroop Pophale, Stephen Poole, Jeff Kuehn, Chuck Koelbel, and Lauren Smith. 2010. Introducing OpenSHMEM: SHMEM for the PGAS community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*. ACM, 2.

[19] D. L. Cohn, A. Banerji, P. M. Greenwalt, M. R. Casey, and D. C. Kulkarni. 1992. Workstation cooperation through a typed distributed shared memory abstraction. In *[1992] Proceedings Third Workshop on Workstation Operating Systems*. 70–74. https://doi.org/10.1109/WWOS.1992.275686

[20] DWARF Debugging Information Format Committee et al. 2010. DWARF debugging information format, version 4. *Free Standards Group* (2010).

[21] HyperTransport Consortium. 2004. HyperTransport I/O Technology Overview.

[22] CXL Members. 2019. Computer Express Link Specifications. https://www.computeexpresslink.org.

[23] Davidlohr Bueso, Scott Norton. 2014. An Overview of Kernel Lock Improvements. http://events17.linuxfoundation.org/sites/events/files/slides/linuxcon-2014-locking-final.pdf.

[24] Eideticom. 2017. Eideticom NoLoad FPGA Accelerator. https://www.eideticom.com/uploads/images/NoLoad_Product_Spec.pdf. Online, accessed 01/05/2019.

[25] Tarek El-Ghazawi and Lauren Smith. 2006. UPC: unified parallel C. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. ACM, 27.

[26] Gen-Z Consortium. 2017. Gen-Z – A New Approach to Data Access. http://genzconsortium.org/.

[27] Jonathon T. Giffin, Mihai Christodorescu, and Louis Kruger. 2005. Strengthening Software Self-Checksumming via Self-Modifying Code. In *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC '05)*. IEEE Computer Society, Washington, DC, USA, 23–32. https://doi.org/10.1109/CSAC.2005.53

[28] Boncheol Gu, Andre S Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, et al. 2016. Biscuit: A framework for near-data processing of big data workloads. In *ACM SIGARCH Computer Architecture News*, Vol. 44. IEEE Press, 153–165.

[29] John Hubbard and Jerome Glisse. 2017. GPUs: HMM: Heterogeneous Memory Management. Red Hat Summit. https://www.redhat.com/files/summit/session-assets/2017/S104078-hubbard.pdf, Online, accessed 01/05/2019.

[30] Intel. 2009. An Introduction to the Intel QuickPath Interconnect.

[31] Intel. 2018. Intel Xeon Processor Scalable Family. https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/xeon-scalable-datasheet-vol-1.pdf, Online, accessed 01/10/2019.

[32] Gabriele Jost, Hao-Qiang Jin, Dieter anMey, and Ferhat F Hatay. 2003. Comparing the OpenMP, MPI, and hybrid programming paradigm on an SMP cluster. (2003).

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

[34] Y. Kanzaki, A. Monden, M. Nakamura, and K. Matsumoto. 2003. Exploiting self-modification mechanism for program protection. In *Proceedings 27th Annual International Computer Software and Applications Conference. COMPAC 2003*. 170–179. https://doi.org/10.1109/CMPSAC.2003.1245338

[35] D. Katz, A. Barbalace, S. Ansary, A. Ravichandran, and B. Ravindran. 2015. Thread Migration in a Replicated-Kernel OS. In *2015 IEEE 35th International Conference on Distributed Computing Systems*. 278–287.

[36] Sangman Kim, Seonggu Huh, Yige Hu, Xinya Zhang, Emmett Witchel, Amir Wated, and Mark Silberstein. 2014. GPUnet: Networking Abstractions for GPU Programs. In *Proceedings of the 11th USENIX Conference*

on Operating Systems Design and Implementation (OSDI'14). 201–216.

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

[38] Katie Lim, Jonathan Balkind, and David Wentzlaff. 2018. Juxtapiton: Enabling heterogeneous-isa research with RISC-V and SPARC FPGA soft-cores. *arXiv preprint arXiv:1811.08091* (2018).

[39] Felix Xiaozhu Lin, Zhen Wang, and Lin Zhong. 2014. K2: A Mobile Operating System for Heterogeneous Coherence Domains. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. 285–300.

[40] Mellanox Technologies. 2017. BlueField Multicore System on Chip. http://www.mellanox.com/related-docs/npu-multicore-processors/PB_Bluefield_SoC.pdf. Online, accessed 01/05/2019.

[41] Zeyu Mi, Dingji Li, Zihan Yang, Xinran Wang, and Haibo Chen. 2019. SkyBridge: Fast and Secure Inter-Process Communication for Microkernels. In *Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys '19)*. ACM, New York, NY, USA, Article 9, 15 pages. https://doi.org/10.1145/3302424.3303946

[42] Timothy P. Morgan. 2013. Tilera Rescues CPU Cycles with Network Coprocessors. https://bit.ly/2DfM53R, Online, accessed 01/05/2019.

[43] Christine Morin, Renaud Lottiaux, Geoffroy Vallée, Pascal Gallard, David Margery, J-Y Berthou, and Isaac D Scherson. 2004. Kerrighed and data parallelism: Cluster computing on single system image operating systems. In *Cluster Computing, 2004 IEEE International Conference on*. IEEE, 277–286.

[44] David Mulnix. 2017. Intel Xeon Processor Scalable Family Technical Overview. https://software.intel.com/en-us/articles/intel-xeon-processor-scalable-family-technical-overview.

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

[46] Myron Slota. 2018. OpenCAPI Technology. https://openpowerfoundation.org/wp-content/uploads/2018/04/Myron-Slota.pdf.

[47] Netronome. 2019. About Agilio SmartNICs. https://www.netronome.com/products/smartnic/overview/, Online, accessed 01/05/2019.

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

[49] OpenCAPI Consortium. 2017. Welcom to OpenCAPI Consortium. http://opencapi.org/.

[50] Simon Peter, Jialin Li, Irene Zhang, Dan RK Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. 2016. Arrakis: The operating system is the control plane. *ACM Transactions on Computer Systems (TOCS)* 33, 4 (2016), 11.

[51] Josh Poimboeuf. 2014. kpatch: dynamic kernel patching. Linux Weekly News. https://lwn.net/Articles/597123/, Online, accessed 01/10/2019.

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

[53] Timothy Roscoe. 2019. Building Enzian: a research computer. The 9th Workshop on Systems for Multi-core and Heterogeneous Architectures.

[54] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. 2018. LegoOS: A Disseminated, Distributed {OS} for Hardware Resource Disaggregation. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 69–87.

[55] Mark Silberstein. 2017. OmniX: an accelerator-centric OS for omni-programmable systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*. ACM, 69–75.

[56] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. 2013. GPUfs: Integrating a File System with GPUs. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. 485–498.

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

[58] Dolphin Interconnect Solutions. 2015. PXH810 PCI Express Gen3 Host Adapter. https://www.dolphinics.com/download/PX/OPEN_DOC/PXH810_Product_Brief.pdf, Online, accessed 01/10/2019.

[59] Xiang Song, Haibo Chen, Rong Chen, Yuanxuan Wang, and Binyu Zang. 2011. A Case for Scaling Applications to Many-core with OS Clustering. In *Proceedings of the Sixth Conference on Computer Systems (EuroSys '11)*. ACM, New York, NY, USA, 61–76. https://doi.org/10.1145/1966445.1966452

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

[61] David Wentzlaff and Anant Agarwal. 2009. Factored operating systems (fos): the case for a scalable operating system for multicores. *ACM SIGOPS Operating Systems Review* 43, 2 (2009), 76–85.

[62] Konrad R. Wilk and Ross Lagerwall. 2016. Patching with Xen LivePatch. Xen Project Developpers Summit. Online, accessed 01/10/2019.

[63] Al Yanes. 2018. Doubling Bandwidth in Under Two Years: PCI Express® Base Specification Revision 5.0, Version 0.9 is Now Available to Members. https://bit.ly/2ClJ9AT, Online, accessed 01/05/2019.

[64] Gerd Zellweger, Simon Gerber, Kornilios Kourtis, and Timothy Roscoe. 2014. Decoupling Cores, Kernels, and Operating Systems.. In *OSDI*, Vol. 14. 17–31.