# Not your parents' physical address space

*Simon Gerber, Gerd Zellweger, Reto Achermann,*
*Kornilios Kourtis, Timothy Roscoe, Dejan Milojicic*[†]
*Systems Group, Department of Computer Science, ETH Zurich* [†]*HP Labs, Palo Alto*

## Abstract

A physical memory address is no longer the stable concept it was. We demonstrate how modern computer systems from rack-scale to SoCs have multiple physical address spaces, which overlap and intersect in complex, dynamic ways, and may be too small to even address available memory in the near future.

We present a new model of representing and interpreting physical addresses in a machine for the purposes of memory management, and outline an implementation of the model in a memory system based on capabilities which can handle arbitrary translations between physical address spaces and still globally manage system memory.

Finally, we point out future challenges in managing physical memory, of which our model and design are merely a foundation.

## 1 Introduction

In the good old days, every memory location in a computer that could be addressed by the processor had a unique address. These addresses were also the size of machine words, such as 32 and 64 bits. Each processor had an address translation unit which turned word-sized virtual addresses into word-sized physical addresses. As far as the OS was concerned, to manage each core's MMU simply required knowing the important physical addresses (where RAM was, device registers, page table roots, etc.) and manage this address space accordingly.

We argue that this pastoral ideal is long gone in modern hardware (if, indeed, it ever existed), and current trends will render it even more irrelevant to OS design.

Virtual memory support is well-known as a complex element of both OS design and processor architecture [7, 8], but both rely on key simplifying assumptions about the underlying *physical* address space:

1. All RAM, and all memory-mapped I/O registers appear in a single physical address space.
2. Any processor core (via its MMU) can address any part of this physical address space at any given time.
3. All processors use the same physical address for a given memory cell or hardware register.

Unfortunately, none of these comforting assumptions are true for modern hardware, ranging from rack-scale systems to systems-on-a-chip (SoCs) in mobile devices. In the next section, we survey some of the many current and future violations of these assumptions, and point out the challenges this creates for effective OS management of physical memory as a resource.

We then present our response to this situation in two parts: firstly, an abstract model of physical addressing that captures the characteristics ignored in current systems, and secondly a way to implement the model in a real OS (Barrelfish) using an extended notion of capabilities. We start by discussing in more detail the growing challenge posed by modern and future physical addressing.

## 2 The Problem

Our argument is this: physical memory today violates the assumptions on which classical memory management is based. OS designers today face a choice: ignore the problem, continue with the old Unix-based "cores plus RAM" view, and have the OS manage the increasingly small areas of the machine where this old model fits, or find a better model of physical addressing which allows a better implementation to solve the problem.

The first observation is that **modern machines have multiple physical address spaces**. "Engineered systems" from vendors like Oracle, Teradata, SAP, and others consist of a collection of commodity PC servers and custom hardware tightly coupled in a single unit. Such systems frequently use remote direct memory access (RDMA) to allow applications to copy data from the RAM of one computer to that of another [24] without involving the OS. Scale-out NUMA [21] takes this idea further by integrating an RDMA-like interface into the cache coherency

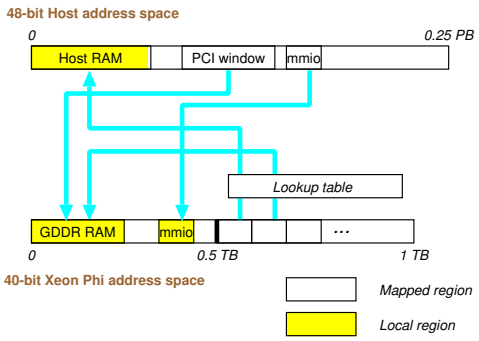


Figure 1: Xeon Phi Memory Layout

protocol, dramatically reducing the overhead of remote reads, writes, and atomic operations while allowing them to use virtual addresses.

Buffer allocation and correct placement of data is critical to performance here, but requires global allocation and protection of multiple physical address spaces.

Secondly, **multiple physical address spaces are not always disjoint, but often intersect**. For example, a PC hosting Xeon Phi accelerators [17] has a conventional 48-bit physical address space[1], which includes two PCI regions corresponding to each Xeon Phi card: one 16 GB region maps to RAM on the card, while the other, smaller region holds some of the card's IO registers (Figure 1).

However, the Xeon Phi cores see a different physical address space whose size is only 40 bits. The lower half of this contains the card's 16 GB RAM region and IO registers at fixed addresses. The upper half (512 GB) is divided into 32 separate 16 GB regions, each of which can be translated independently via a lookup table to any physical address on the host side (including the PCI apertures of this and other Xeon Phi cards).

This configuration violates all three assumptions in our introduction. Memory (including RAM on both the host and cards) is shared, but addresses are not. An OS managing RAM across the whole machine (e.g. [1]) can not use physical addresses to refer to regions of memory, and needs some other kind of descriptor. Worse, this hardware permits addressing loops to occur – something an OS would ideally be able to prevent.

This is just one example. Many PCI devices (GPUs, NICs, etc.) contain large amounts of on-card memory forming a separate physical address space and are capable of issuing bus cycles to host memory. Memory addresses must already be translated by software between address spaces, whether it is when copying data directly or programming DMA engines. It is clear from looking at OS code that IOMMUs [2] complicate, not simplify, this problem: they introduce additional address spaces (on the adaptor) and apertures between them and host memory.

Neither is the problem confined to high-end servers. System-on-chip parts used in phones contain a mix of heterogeneous cores with differing access to memory and devices, in effect living in different, though intersecting, physical address spaces. Such systems are comprised of a complex network of buses of different widths, with highly configurable translations and firewalls between them to facilitate sandboxing of low-level software on some of the cores. The redacted public programming manuals for such chips list multiple physical addresses for the same location, depending on which core is initiating the bus cycle (for example, [29]).

Even a low-end PC has multiple physical address spaces: every core (or, indeed, SMT thread) has its own memory-mapped local interrupt controller. We return to this simple example in the next section.

Finally, **we are running out of bits.** It is not unreasonable today to envisage a machine with a petabyte of main memory, and such scales are within scope of projects like The Machine [11, 15]. Note that a 64-bit PC today can only physically address 256 TB. In practice, many of these address bits tend to be used to route memory transactions to the appropriate part of the machine, further reducing the addressable memory.

Additional address bits could in principle be added, as was the case with Intel's PAE [16] and ARM's LPAE [4], but this incurs significant electrical cost in wires (and hence power), TCAM for cache lookup, etc. Furthermore, others have questioned whether conventional paging is workable at this scale at all [5, 30].

A more likely course for hardware designers is additional layers of physical address translation, such as that adopted for the (32-bit) Intel Single-Chip Cloud Computer [13, 14], which used a 256-entry per-core lookup table to translate each core's 32-bit physical addresses into 43-bit system-wide addresses.

We see three key implications of these trends:

- There will be more memory locations in a machine than a core can issue physical addresses for.
- There will be multiple translation steps of physical addresses between a core and memory it accesses.
- To access a memory cell, different cores in a machine must issue different physical addresses.

Virtual memory is no solution to this problem. Classical VM provides a per-process, opaque translation to a single physical AS, which is assumed to be the same for all cores. This is simply not the case for current and future machines. VM is just an additional level of translation, but solves none of the problems of address space size, intersection, or partial accessibility.

Instead, key OS design challenges, viewed as trivial for the last few decades (because they were), are raised anew:

1. How does an OS allocate, manage, and share regions

[1] Physical addresses on 64-bit x86 processors are limited to 48 bits.

of physical addresses (whether DRAM, persistent RAM, or memory-mapped I/O registers)?
2. How does OS code running on one core communicate a set of memory locations (such as a buffer) to code running on another core?
3. How can application code running on multiple cores use a single, shared virtual address space? To what extent is this guaranteed to be possible?

Our goal in the work reported in the rest of this paper is twofold: Firstly, we want to find workable OS design and implementation techniques to deal with hardware that has no single, global view of physical memory. As we have said, we are impelled to do this by trends in hardware design which are manifest in today's hardware, and which we see continuing into the future.

Secondly, however, in the spirit of Mogul et. al. [20] we do not want to be simply beholden to the hardware designers as in the past. We also realize that these hardware trends are happening for a reason, and by understanding their consequences for system software we hope to offer useful guidance and insight from a software perspective to hardware designers.

## 3 A better model for memory addressing

In this section, we present an abstract model for representing what a given physical address actually means in a machine. Our goal is to provide a logically firm foundation for designing a physical memory management system for complex modern machines.

Our representation is reminiscent of Device Trees [22], however we focus on the translation of physical addresses within a system, and within this narrower scope must accomodate non-hierarchical relationships (in particular, different cores and devices see different addresses for the same location).

We face a classic naming problem. Indeed, Saltzer's seminal paper [26] on naming and binding of objects takes memory addressing in Multics [9] as one of its two case-studies (the other, better-known example is the file system). As with all naming problems, we pay careful attention to defining the *context* in which each *name* (physical address in our case) is to be resolved.

Our approach is to start by identifying every distinct naming context in the system. We first define a *Physical Address Space*, shortened to *Address Space (AS)*. Each AS has a unique ID, a range of address values (typically a power of two in size), and a function from addresses to referents. A referent is either memory contents which can *only* be directly addressed in this AS, or else a new address in a different AS. An AS is partitioned into contiguous *regions*. Each region either contains only *local* referents (which can be RAM, memory-mapped hardware registers, or nothing at all), or is a function mapping addresses in its range into addresses in a different region of a *different* AS – in effect, the region is an "aperture" into another address space, with an associated translation function on addresses.

This translation function between regions in ASes might be an identity mapping (where a region in one AS shadows another), a simple static transformation (such as adding an offset or prepending bits), or dynamic translation using hardware like a lookup table or IOMMU.

We further enforce the rule that any "real" location (a RAM cell, or a hardware register) appears in exactly one AS. If such a location seems to be present locally in more than one AS, say ASes $x$ and $y$, then we create a new AS $z$ to contain it locally and replace the respective regions of $x$ and $y$ with mappings to $z$. This leads to more unique ASes than are considered by traditional approaches.

To summarize so far, a physical address is *always* relative to some AS, and resolves either to a unique, real location, or to other address in a different AS.

We next use the term *core* to refer to anything capable of issuing addresses: a processor core (or SMT thread), co-processor, or DMA-capable device. A core issues physical addresses to its own unique AS. This AS is local: even SMT threads on a PC "see" different ASes with different, memory-mapped local interrupt controllers.

When a core issues an address, it is progressively translated through successive address spaces until it hits a local referent in some AS, or else an error occurs. This process is not guaranteed to terminate – it is perfectly possible to set up routing loops inside a modern machine. In practice, this usually leads to a bus error.

A complete computer system is represented as an arrangement of cores and physical address spaces. Note that not all address spaces can be directly addressed by cores, and not all have actual contents – a novelty of our model is that many ASes serve purely to cleanly separate intermediate translation steps.

An example will help to make this clear. Figure 2 shows a decomposition of a simple, 2-core 64-bit PC with a 32-bit DMA capable network card. Even this small example (common 10 years ago) demonstrates the complexity of modern memory addressing. We model this system with three cores (two CPUs and a NIC), and four address spaces: one for each core and a host address space. RAM regions are local to the host address space, and identity-mapped to all other address spaces. Because the NIC is constrained to 32-bit addresses, it is incapable of accessing the RAM region with addresses above 4G in the system AS. The NIC's memory-mapped IO region (*mmio*) is local to the NIC's address space and exported to the host. Finally, each CPU mirrors the host address space and includes a local region for its local APIC.

Note Figure 1 is also an example of the model, omitting

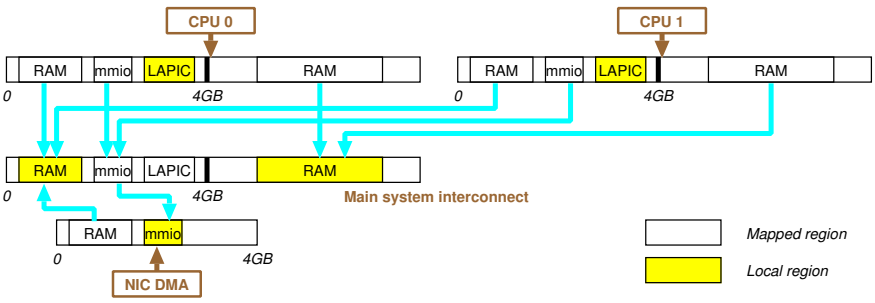


Figure 2: Address spaces in a simplified 2-core 64-bit PC with a 32-bit PCI network card with DMA.

8 host ASes (for a 4-core, 2-thread host) and 228 distinct Xeon Phi ASes (for a 57-core, 4-thread Xeon Phi).

**Discussion:** The principle that memory contents can only be "directly" accessed in a single, unique AS allows us to decompose a system into separate but connected ASes which translate addresses between themselves with well-defined behavior we can represent logically, configure online in an OS, and reason about.

We find our model leads to an increase in the number of conceptual ASes in a given real system, but not an explosion. For example, each core (or SMT thread) only introduces a single new AS. Even complex SoCs result in AS counts in the middle tens, rather than hundreds.

## 4 Implementation

We are not theoreticians. Our goal in devising a memory addressing model is to create a memory system for the Barrelfish OS [6] to capture the complexity of modern hardware and meet the challenges we mention above.

Representing a region in our model (a range of addresses and an associated AS identifier) is natural match for *descriptors* or, more generally, capabilities [12, 19, 28]. Capabilities allow us to refer to, and subdivide, any region of physical memory even when it cannot be mapped into a local virtual address space.

Barrelfish already represents physical memory regions using typed capabilities in a scheme extended from seL4 [18], but the representation also resembles Mach's memory objects [23] or Chorus segments [25]. One could even use POSIX file descriptors (and `mmap()`) for this purpose, while losing the scaling benefits of decentralized allocation afforded by full capabilities.

We extend Barrelfish's capabilities in a straighforward manner by adding a new field holding an identifier for the AS of the region referred to by the capability. The key implementation challenges concern how to access the memory referred to by a capability, revocation of rights to a region, and passing capabilities between cores.

**Capability resolution:** In Barrelfish and similar systems which represent physical memory with capabilities [12, 18, 28], a capability of the appropriate type confers the right to map the region into a virtual address space. In our new design we only allow this if the capability's AS is local to the core requesting the mapping. DMA programming is restricted in an analogous way.

If the region's AS is different from the core's, it must be *resolved*: a new derived capability must be created which represents a region of the local AS mapping to the remote region. There are three possibilities:

1. The translation is impossible. For example, a region referring to an RDMA buffer on a remote node in an Infiniband cluster cannot be translated to a local memory region. In this case, resolution fails. Note that holding a capability is still useful: this allows authorization for, and decentralized allocation of, remote buffer memory, for example.

2. A static translation exists. In this case the non-local region can *always* be trivially translated to a local region via a static function, and a new (local) capability can be created. In many cases this function is the identity (for example, when a local AS is mostly "shadowed" by a system-wide AS) or is a trivial operation (such as extending a 32-bit address in low memory to a local 64-bit address, or adding the address of a PCI Base register).

3. The region can be translated dynamically. This typically involves programming some translation hardware, such as a lookup table or an IOMMU.

The last case is the most complex to handle because hardware programming is involved, and a general solution is still an open design question for us.

One challenge is what action to take when required hardware resources (e.g lookup table or IOMMU entries) become exhausted. An option is to use translation faults and view hardware as a cache for a larger set of software-maintained translations, much as a TLB caches virtual

mappings, but the complexity and performance anomalies introduced by this approach make it unappealing.

A second challenge is what to do when the hardware resources required to resolve a capability are not accessible from the core itself. In the worst case, resolving a capability to a remote region of memory might require a complete map of the machine's memory system, and remote requests to other nodes in the system to poke translation hardware.

The former problem we can address using Barrelfish's centralized System Knowledge Base (SKB) [27] to maintain the full memory map (suitably cached), but remote translation requests introduce undesirable coupling between distributed parts of the OS which is known to have caused considerable difficulty in previous systems.

A simplifying assumption we are currently experimenting with is the restriction that a capability can only be resolved if the hardware resources required are local to the resolving core. This reduces the complexity considerably, at the cost of limited semantics: it is possible for situations to arise where resolution is unnecessarily forbidden. This can be mitigated by transferring *unresolved* versions of capabilities between nodes when possible, but more experience with a real system is required.

**Revocation:** Revoking a capability implies deletion of all capabilities derived from it – in effect, invalidating all mentions of the region referred to by the capability.

Barrelfish's capability system already provides revocation with semantics that are identical to seL4's revocation [10]. For example, any memory regions that are derived from a memory region $R$ need to be deleted when $R$ is revoked. Furthermore, since Barrelfish is a multikernel and the capability system is distributed (and replicated) across cores, the revocation mechanism already handles the distributed 2-phase commit required to remove all copies of a capability.

We extend this facility to track all capabilities resolved from the one being revoked. Tracking a chain of resolutions is straightforward for static translations. However, deleting capabilities that were the result of dynamic resolution requires reprogramming of the translation hardware to remove the established translation mappings. To ensure that this reprogramming happens, we make it a part of the capability deletion process. Note that only allowing local hardware to be programmed in the resolution process also simplifies revocation.

## 5 Conclusion and future work

Physical address space is not what it used to be. A clear and consistent scheme for naming and working with physical addresses in a modern machine is a surprisingly complex, but urgent, problem.

Our current work aims at a sound foundation for physical memory management and targets rack-scale appliances using RDMA, server-based SoCs like X-Gene [3] and accelerators like Xeon Phi. Consequently, we started by trying to capture the true complexity of physical memory in the design of a modern OS.

We also plan to evaluate our model using data processing applications running on The Machine [15], equipped with large amounts of non-volatile memory which is organized and addressed in different physical address spaces across many compute nodes. Another possible evaluation scenario is offloading and communication between the Xeon Phi co-processor and the host system.

However, this is only the first step. Large-scale platforms like The Machine [15] present an entirely new set of challenges that accompany persistent main memories and highly distributed architectures. One is reliability. Interconnects and memory will suffer partial and/or transient failures, and systems must adapt when erroneous data is retrieved remotely, or accesses simply fail.

The second is security. The principle of including the kernel on every core in the Trusted Computing Base may be insufficient at scale. Even today, mobile SoCs provide configurable firewalls on their interconnects to protect cores running critical code. The memory system on the experimental Intel SCC [13] could be programmed to completely sequester cores running in kernel mode.

Third, energy consumption is increasingly the major limiting factor in computing. Consequently, the OS must take power into consideration to effectively manage memories in large, high-density machines.

Finally, this is a *physical* and not (yet) a *virtual* memory system, though it may extend naturally to the extra, per-process translation layer provided by MMUs. However, a better question is whether virtual memory in its current form is still a useful concept. Per-process address translation and interposition on memory accesses to a given region are powerful facilities with many uses, but demand paging has questionable value in a world of very large persistent main memory.

The transparency of virtual memory is already an obstacle to performance: critical applications avoid paging and size data structures to fit TLB coverage, effectively second-guessing the MMU. In contrast, our approach in Barrelfish directly manages physical memory, and applications control their own virtual address spaces.

In any case, a prerequisite to addressing any of these problems is a clear basis for unambiguously referring to physical memory resources, and allowing an OS to reason about and control their location and accessibility from various parts of the system. It may also point the way towards hardware designs which mitigate the challenges we have outlined in this paper.

## References


[1] Achermann, R. *Message passing and bulk transport on heterogenous multiprocessors.* ETH Zurich, 2014. Master's Thesis, http://dx.doi.org/10.3929/ethz-a-010262232.

[2] Advanced Micro Devices, Inc. *AMD I/O Virtualization Technology (IOMMU) Specification.* Revision 2.0, Publication Number: 48882.

[3] AppliedMicro. *APM883208-X1-PRO-1 X-Gene X-C1 Evaluation Kit Product Brief.* Revision 0.1.

[4] ARM. *Cortex-A15 Technical Reference Manual.* Revision: r2p1, (ID122011).

[5] Bailey, K., Ceze, L., Gribble, S. D., and Levy, H. M. Operating System Implications of Fast, Cheap, Non-volatile Memory. In *Proceedings of the 13th Workshop on Hot Topics in Operating Systems* (2011).

[6] Baumann, A., Barham, P., Dagand, P.-E., Harris, T., Isaacs, R., Peter, S., Roscoe, T., Schüpbach, A., and Singhania, A. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles* (2009), pp. 29–44.

[7] Clements, A. T., Kaashoek, M. F., and Zeldovich, N. Scalable address spaces using RCU balanced trees. In *Proceedings of the 17th Conference on Architectural Support for Programming Languages and Operating Systems* (2012), pp. 199–210.

[8] Clements, A. T., Kaashoek, M. F., and Zeldovich, N. RadixVM: Scalable address spaces for multithreaded applications. In *Proceedings of the 8th ACM European Conference on Computer Systems* (2013), pp. 211–224.

[9] Corbató, F. J., and Vyssotsky, V. A. Introduction and overview of the Multics System. In *Proceedings of the Fall Joint Computer Conference, Part I* (1965), pp. 185–196.

[10] Elkaduwe, D., Derrin, P., and Elphinstone, K. A memory allocation model for an embedded microkernel. In *Proceedings of the 1st International Workshop on Microkernels for Embedded Systems (MIKES)* (2007), pp. 28–34.

[11] Faraboschi, P., Keeton, K., Marsland, T., and Milojicic, D. Beyond processor-centric operating systems. In *Proceedings of the 15th Workshop on Hot Topics in Operating Systems* (Karthause Ittingen, Switzerland, May 2015).

[12] Hardy, N. KeyKOS Architecture. *SIGOPS Operating Systems Review 19*, 4 (1985), 8–25.

[13] Howard, J. Rock Creek System Address Look up Table & Configuration Registers. External-architecture Specification (EAS) Revision 0.1, Intel Microprocessor Technology Laboratories, Jan 2010.

[14] Howard, J., Dighe, S., Hoskote, Y., Vangal, S., Finan, D., Ruhl, G., Jenkins, D., Wilson, H., Borkar, N., Schrom, G., Pailet, F., Jain, S., Jacob, T., Yada, S., Marella, S., Salihundam, P., Erraguntla, V., Konow, M., Riepen, M., Droege, G., Lindemann, J., Gries, M., Apel, T., Henriss, K., Lund-Larsen, T., Steibl, S., Borkar, S., De, V., Van Der Wijngaart, R., and Mattson, T. A 48-Core IA-32 message-passing processor with DVFS in 45nm CMOS. In *Solid-State Circuits Conference Digest of Technical Papers* (Feb 2010), pp. 108–109.

[15] HP Labs. The Machine. http://www.hpl.hp.com/research/systems-research/themachine/, January 2015.

[16] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual.* Order Number: 325384-052US.

[17] Intel Corporation. *Intel Xeon Phi Coprocessor System Software Developers Guide*, Mar 2014.

[18] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., and Winwood, S. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles* (2009).

[19] Levy, H. M. *Capability-Based Computer Systems.* Butterworth-Heinemann, Newton, MA, USA, 1984.

[20] Mogul, J. C., Baumann, A., Roscoe, T., and Soares, L. Mind the gap: reconnecting architecture and OS research. In *Proceedings of the 13th Workshop on Hot Topics in Operating Systems* (2011).

[21] Novakovic, S., Daglis, A., Bugnion, E., Falsafi, B., and Grot, B. Scale-out NUMA. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (2014), pp. 3–18.

[22] Power.org. *Standard for Embedded Power Architecture Platform Requirements*, Apr 2011. Revision 1.1.

[23] Rashid, R., Tevanian, A., Young, M., Golub, D., Baron, R., Black, D., Bolosky, W., and Chew, J.

Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. In *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems* (1987).

[24] Recio, R., Metzler, B., Culley, P., Hilland, J., and Garcia, D. RDMA Protocol Specification. RFC 5040, Oct 2007.

[25] Rozier, M., Abrossimov, V., Armand, F., Boule, I., Gien, M., Guillemont, M., Herrmann, F., Kaiser, C., Langlois, S., LÃl'onard, P., and Neuhauser, W. Overview of the CHORUS Distributed Operating Systems. *Computing Systems 1* (1991), 39–69.

[26] Saltzer, J. Naming and binding of objects. In *Operating Systems*, R. Bayer, R. Graham, and G. Seegmüller, Eds., vol. 60 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1978, pp. 99–208.

[27] Schuepbach, A., Peter, S., Baumann, A., Roscoe, T., Barham, P., Harris, T., and Isaacs, R. Embracing diversity in the Barrelfish manycore operating system. In *Proceedings of the Workshop on Managed Many-Core Systems (MMCS)* (Boston, MA, USA, June 2008).

[28] Shapiro, J. S., Smith, J. M., and Farber, D. J. EROS: A Fast Capability System. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles* (1999), pp. 170–185.

[29] Texas Instruments. *OMAP543X Multimedia Device Technical Reference Manual*, 2014. Revision AC.

[30] Witchel, E., Cates, J., and Asanović, K. Mondrian memory protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems* (2002), pp. 304–316.