DISS. ETH NO. 26619

On Memory Addressing

A thesis submitted to attain the degree of

DOCTOR OF SCIENCES OF ETH ZURICH

(Dr. sc. ETH Zurich)

presented by

RETO ACHERMANN

Master of Science ETH in Computer Science, ETH Zurich

born on 04. June 1989

citizen of Ennetbürgen, Switzerland

accepted on the recommendation of

Prof. Dr. Timothy Roscoe (ETH Zurich), examiner
Prof. Dr. David Basin (ETH Zurich), co-examiner
Prof. Dr. Gernot Heiser (UNSW Sydney), co-examiner
Dr. David Cock (ETH Zurich), co-examiner

2020

DOI: 10.3929/ethz-b-000400029

Abstract

Operating systems manage and configure a machine's physical resources such as memory and translation hardware. This task is mission critical: the operating system must always correctly configure memory address translations and unambiguously name the physical resources of a system. However, operating systems today use abstractions and assumptions which unfaithfully represent the actual topology of the hardware they are managing. This mismatch leads to bugs and security vulnerabilities in system software. This is a problem.

This dissertation presents a new abstraction model to faithfully represent the memory subsystem of a hardware platform as seen by software. The core abstraction of the new model is the address space, which defines the context for address decoding. An address space either translates addresses or terminates address resolution within its context. The *Decoding Net* formally specifies the semantics of address decoding behavior of address spaces in the Isabelle/HOL theorem prover. This provides a sound basis for reasoning about the current hardware configuration of a platform.

Address spaces are inherently dynamic in two ways: *i)* new devices are discovered, powered on or off, or hot-plugged introducing new address spaces in the system, and *ii)* a memory allocation request requires an update of the translation configuration of an address space. Changing the configuration of an address space is a privileged operation and requires a certain authority. This is expressed as an extension to the *Decoding Net* with a layer adding a notion of configurability and fine-grained authority following the principle of least-privilege.

Guided by an executable specification of the dynamic *Decoding Net* model, the implementation in *Barrelfish/MAS* is driven following the principle of least-privilege. The resulting implementation demonstrates that it is possible to implement the detailed address space model and least-privilege memory management in an operating system efficiently and with little overhead and matching performance to the Linux operating system.

Zusammenfassung

Betriebssysteme verwalten und konfigurieren die physikalischen Ressourcen wie Hauptspeicher und Adressierungsübersetzungshardware einer Rechenmaschine. Dieser Arbeitsschritt ist missionskritisch: das Betriebssystem muss die Hardware stets korrekt konfigurieren wie auch die physikalischen Ressourcen des Systems eindeutig benennen können. Die Betriebssysteme von heute benutzen jedoch Abstraktionen und Annahmen welche gerade die eigentliche Topologie der verwalteten Hardware inakkurat repräsentieren. Diese Diskrepanz führt zu verschiedensten Defekten und Sicherheitslücken in System Software. Dies ist ein Problem.

Diese Dissertation präsentiert einen neues Abstraktionsmodell welches die Hardwarekonfiguration einer Rechenmaschine, wie sie von der Software gesehen wird, akkurat repräsentiert. Die Zentrale Abstraktion dieses neuen Modells ist der Adressraum, welcher einen Kontext für Adressdekodierung definiert. Ein Adressraum übersetzt oder schliesst die Adressauflösung für eine Adresse innerhalb seines Kontextes ab. Die Semantik der Adressauflösung und der Adressräume ist dann formalisiert im "*Decoding Net*", eine Spezifikation der Adressraumabstraktion in Isabelle/HOL. Dies bildet eine wohldefinierte Grundlage, um über die gegenwärtige Hardwarekonfiguration einer Plattform zu argumentieren.

Die Adressräume sind von Natur aus dynamisch in zwei Arten: *i)* das Auffinden, Anschliessen oder Entfernen von neuen Hardwarekomponenten verändert die Anzahl der Adressräume im System, und *ii)* die Konfiguration dieser Adressräume kann verändert werden. Diese privilegierte Aktion benötigt die entsprechenden Befugnisse. Das "*Decoding Net*" Modell wird erweitert mit einer Auffassung von Konfigurierbarkeit und detailgenauen Befugnissen im Sinne von Prinzip des minimalen Rechts.

Gelenkt von einer ausführbaren Spezifikation des dynamischen "*Decoding Net*" Modells wird die Implementierung in *Barrelfish/MAS* durchgeführt unter dem Prinzip des minimalen Rechts. Die resultierende Implementierung zeigt, dass es möglich ist das detaillierte Adressraummodell und eine Speicherverwaltung mit minimalen Rechten effizient in einem Betriebssystem zu realisieren.

Acknowledgments

I am exceptionally thankful to my many amazing and wonderful friends and colleagues, who contributed, in some way or another, to the research presented in this thesis.

To my adviser, Prof. Timothy "Mothy" Roscoe. Thank you for giving me the opportunity to do my masters and doctoral studies in the Systems Group. I am truly grateful for your openness, support, and the feedback and advice I have received from you during my time at ETH Zurich. It was a great pleasure to work with you.

To Dr. David Cock. I am exceptionally grateful for the opportunity to work with you. Thank you for your inputs to the formal modeling, constructive feedback, and Isabelle/HOL expertise, which enabled and propelled many aspects of the work presented in this thesis.

To Prof. Gernot Heiser. Thank you for agreeing to be on my committee, and your outstanding dedication in providing valuable feedback and insightful comments, which greatly improved my dissertation.

To Prof. David Basin. I appreciate that you have agreed to be on my committee, and your valuable and assuring feedback on the dissertation.

To my outstanding and truly exceptional collaborators at ETH Zurich, Hewlett-Packard Labs, VMware Research and elsewhere. I would like to express my greatest gratitude for giving me the opportunity to work and collaborate with you. Your dedication, comments and feedback during many meetings and interactions, and contributions to papers, reports and projects either directly or indirectly contributed to my dissertation: David Cock, Lukas Humbel, Roni Häcki, Simon Gerber, Jayneel Gandhi, Dejan Milojicic, Kornilios Kourtis, Stefan Kästle, Michael Giardino, Nora Hossle, Daniel Schwyn, Gerd Zellweger, Moritz Hoffmann, Ashish Panwar and Abhishek Bhattacharjee.

To my friends and colleagues at ETH Zurich. Roni, Pravin, Renato, Simon, Stefan, Frances, Michael, Claude, Moritz, Gerd, Daniel, David, Nora,

Anastasiia, Andrea, Lukas, Melissa, Michael, Monica, Raphael, Daniel and Barbara. Thank you for making my time in the Systems Group such a great experience. A special thank you to the amazing admins of the Systems Group: Simonetta, Nadia, and Jena.

Finally, I would like to thank my parents, Irene and Beppi, for their unconditional support.

Zurich, February 2020.

Contents

1

Introduction

This dissertation applies formal methods to the design and implementation of memory management and authorization components in operating systems. It uses a formal model to capture the complexity of memory addressing on modern hardware and adopts the principle of least-privilege to configure address translation hardware.

The principal goals of the formal model and its application in the context of system software are the following:

1. Provide an accurate representation of the memory subsystem of any hardware platform as seen by system software including non-uniform and heterogeneous translations and memories.

2. Establish a sound foundation to unambiguously name memory resources and to formally reason about address resolution.

3. Enable the implementation of system software components to correctly manage memory resources and configure translation hardware.
4. Define and identify the semantics of address space configuration and the required authority to do so.
5. Apply the principle of least-privilege to the task of address space configuration.

The remainder of this chapter sets the stage for the work presented in this thesis by providing the motivational aspects followed by the problem statement and the contributions of this dissertation to systems research. Finally, an overview of the structure of the thesis is presented.

1.1 Motivation

While application processes generally run in user space and in their own virtual address space, it is the task of system software (e.g. the operating system kernel) to provide this illusion of a uniform, linear address space to the application. Likewise, the kernel also runs in its own, linear address space separated from user space processes. To uphold this illusion, system software needs to correctly program the relevant translation units (e.g. the processor's memory management unit (MMU)). This in turn, requires system software to know the corresponding address a memory resource (e.g. DRAM or memory mapped device registers) appears on the processor's system bus.

The actual location of memory resources and devices, especially under which address they appear, depends on the current hardware configuration of the platform at hand where the configuration refers to the hardware components, their interconnection and state. Two platforms can have a very different configuration, especially for system-on-a-chip (SoC) architectures. Worse, the observed address ranges differ between two distinct cores, most prominently between the CPU and a direct memory access

(DMA) capable device. This inherent heterogeneity, not only between two different platforms but also within a single platform, makes the design and implementation of system software tedious and error prone: writing to the wrong device register or accessing the wrong memory location can lead to data corruption, unexpected behavior, or security vulnerabilities.

While one may argue that the platform configuration information is passed to the operating system through UEFI [UEF17], ACPI tables [UEF15], those services are in fact other instances of system software and therefore need to know the locations of the resources present on the platform. DeviceTrees [dev17] on the other hand are a file format than a topology description and are incapable of expressing complex address typologies.

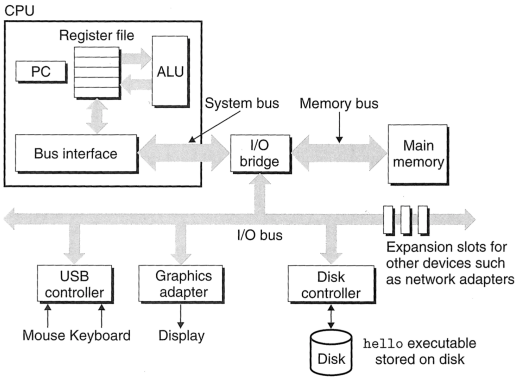
Moreover, UEFI services may not be present at all. A machine can be booted in legacy mode, or there is simply no UEFI available for that platform. Tiano Core [Tia19], for instance, is compiled for a specific platform including all information about memory and devices compiled into the UEFI image.

In summary, the hardware representation is, in some sense, too abstract which hides important details such as the actual topology, interconnect configurations, and locations of hardware firewalls. Moreover, this can be misleading. For instance, a resource access can have different characteristics from where it is accessed from (processor, device, accelerators, etc.), and this resource may actually have different addresses it appears.

The misleading abstractions need to be replaced with a more accurate representation, which reveals enough details of the underlying hardware with its configuration and characteristics to give system software a chance to actually handle and configure hardware correctly.

Therefore, system software needs to be written with an accurate description of the platform at hand, including the memory topology, addresses and sizes of memory resource and devices as seen from a particular core. Yet, the platform architecture diagrams presented in recent textbooks about operating systems (an example is shown in Figure 1.1) completely ignore the possibility of heterogeneity and different observed address ranges

Figure 1.4
Hardware organization of a typical system. CPU: Central Processing Unit, ALU: Arithmetic/Logic Unit, PC: Program counter, USB: Universal Serial Bus.



systems, but all systems have a similar look and feel. Don't worry about the complexity of this figure just now. We will get to its various details in stages throughout the course of the book.

Figure 1.1: System Architecture as presented in *Computer Systems, A Programmer's Perspective* by Randal E. Bryant and David R. O'Hallaron [BO15].

between two distinct cores. Hence, the diagrams shown are oversimplified and inaccurate, one may even say plain wrong.

Another aspect why an accurate representation of the hardware platform is system software verification. Correctness is a central property for system software and one way to guarantee correct operational behavior is through verification (e.g. seL4 [Kle+09] or CertiKOS [Gu+16]). While these projects prove or certify correct operational semantics with respect to an execution model, the proofs are based on an abstract machine representation, which is again greatly simplified and does not accurately represent the complexity and heterogeneity of a real system.

Instead of applying heuristics and using vague assumptions to configure translation units and abstract hardware as seen by software, this dissertation presents a sound and well-founded formal model to express the hardware configuration of computing platforms as a network of address spaces. This provides system software with enough details about the hardware to enable system software to correctly manage and configure the resources of a hardware platform.

1.2 Problem Statement

This dissertation investigates the following research questions and problems from the angle of systems software.

Multiple-Physical Address Spaces This thesis makes the case that the presence of a single, globally uniform physical address space has always been an illusion, and promotes the address space as a first-class abstraction to express the configuration of a platform. Processor cores and DMA-capable devices have different *views* of the platform: within a single address space, memory resources can appear at different addresses, they can be aliased or the same address refers to a different resource in two distinct address spaces. There is no longer *the (unique) physical address* of a memory resource. It is rather a question of what the set of addresses is

this memory resource appears at, and in which address spaces, or whether the same address in two address spaces really refer to the same resource. Getting this wrong leads to correctness problems, e.g. because of the misinterpretation of the address, software accesses the wrong underlying resource, which results in data corruption or information leakage.

Platforms as a Network of Address Spaces This dissertation argues that the configuration of a platform can be expressed as a network of address spaces which can be overlapping, isolated or intersecting in arbitrary ways. The connections between address spaces can be fixed or configurable. These aspects are hidden in the current hardware abstraction model. The dissertation explores a variety of platform architectures of different types and sizes. It further shows the applicability of a network of address spaces to express the configuration of the various platforms. This reveals important details of the underlying hardware and its characteristics to system software.

***Decoding Net* Model of Address Spaces** The thesis shows that the network of address spaces can be formally specified as a decoding network. The *Decoding Net* is a directed graph of nodes corresponding to address spaces of a platform, while edges are mappings of addresses between two address spaces. This dissertation demonstrates that this formal specification is capable of accurately expressing inherently complex and heterogeneous platforms and execution modes of processors. Moreover, it is possible to define transformations on the *Decoding Net* without changing the *view* from within a particular address space. Lastly, the abstract *Decoding Net* can be refined to express existing hardware devices such as translation lookaside buffers (TLBs), memory management units, memory controllers and lookup tables.

Authorization and Configurable *Decoding Nets* The *Decoding Net* model captures a snapshot of the system configuration. This thesis further shows how the *Decoding Net* model can be extended with a notion of configuration and authorization using a least privilege approach. The central question is “who can configure this translation hardware and what

rights does one need to have to do so?". The dissertation addresses this question by applying a fine-grained decomposition of the address translation configuration process into subjects, objects and the necessary authority on top of the *Decoding Net* model.

A Fast Implementation of the Authorization Model The thesis demonstrates that it is possible to efficiently implement the fine-grained authorization model following the principle of least-privilege plus the detailed address space model in an operating system. To do so, the thesis presents an operating system architecture that separates low-level protection primitives from high-level policy mechanisms to manage memory resources and configure address translation units.

1.3 Structure of the Dissertation

Chapter 2 provides background information on the definitions used for addresses and address spaces in hardware manuals, and a survey of real and proposed hardware translation schemes. It then presents the current abstractions used by operating systems and highlights the problems that arise with respect to the presented address translation schemes and platforms.

Chapter 3 surveys work related to memory management and abstractions used in operating systems, runtimes for scheduling and memory allocation policies, and models of processor and the semantics of memory accesses.

Chapter 4 presents the address space model to capture the complexity of the memory topology of any platform. Moreover, it defines the Decoding Net, a formalization of the address space model in Isabelle/HOL which rigorously specifies the semantics of address decoding, and defines correct transformation algorithms on top of the model. This is joint work with David Cock (who significantly helped with formalizing the address space model), Lukas Humbel, Gerd Zellweger, Kornilios Kourtis, Timothy

Roscoe, and Dejan Milojicic and it appeared in related publications [Ach14; Ger+15; Ach+17b; Ach+18].

Chapter 5 extends the decoding net model of Chapter 4 to add support for dynamic address spaces in both their translation configuration and the total number thereof. The chapter further presents a fine-grained authorization model for configuration changes. This is based on joint work with Nora Hossle (who implemented the executable specification), Lukas Humbel, Daniel Schwyn, David Cock and Timothy Roscoe in *Least-Privilege Memory Protection Model for Modern hardware* [Ach+19a] and [Hos19].

Chapter 6 describes the needed mechanisms for an efficient implementation of the address space model of Chapter 5 in operating system software. The chapter presents *Barrelfish/MAS*, an extension to the Barrelfish research operating system that implements the address space model following the principle of least-privilege for translation configuration. This is based on joint work with Nora Hossle, Lukas Humbel (who contributed significantly to the runtime representation), Daniel Schwyn, David Cock and Timothy Roscoe in *Least-Privilege Memory Protection Model for Modern hardware* [Ach+19a].

Chapter 7 evaluates the implementation of *Barrelfish/MAS* in the context of memory management, least-privilege address translation configuration, scaling and bookkeeping overheads. This is based on joint work with Nora Hossle, Lukas Humbel, Daniel Schwyn, David Cock and Timothy Roscoe in *Least-Privilege Memory Protection Model for Modern hardware* [Ach+19a].

Chapter 8 draws the conclusions of this dissertation and presents future directions of the address space memory model and its applications in system software.

1.4 Related publications

This dissertation presents work that is part of the Barrelfish research operating system (http://www.barrelfish.org). Many people have contributed to the existing infrastructure and the Barrelfish ecosystem, which enabled the research of this dissertation.

The following list provides references to work presented in this dissertation, which is published in some form.

[Ach14] Reto Achermann. "Message Passing and Bulk Transport on Heterogenous Multiprocessors". Master's Thesis. Department of Computer Science, ETH Zurich, 2014.

[Ach+17b] Reto Achermann, Lukas Humbel, David Cock, and Timothy Roscoe. "Formalizing Memory Accesses and Interrupts". In: *Proceedings of the 2nd Workshop on Models for Formal Analysis of Real Systems*. MARS 2017. 2017.

[Ach+18] Reto Achermann, Lukas Humbel, David Cock, and Timothy Roscoe. "Physical Addressing on Real Hardware in Isabelle/HOL". in: *Proceedings of the 9th International Conference on Interactive Theorem Proving*. ITP'18. 2018.

[Ach+19a] Reto Achermann, Nora Hossle, Lukas Humbel, Daniel Schwyn, David Cock, and Timothy Roscoe. *A Least-Privilege Memory Protection Model for Modern Hardware*. 2019. arXiv: 1908.08707 [cs.OS].

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

[Ger+15] Simon Gerber, Gerd Zellweger, Reto Achermann, Kornilios Kourtis, Timothy Roscoe, and Dejan Milojicic. "Not Your Parents'

Physical Address Space". In: *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems*. HOTOS'15. 2015.

[Ger18] Simon Gerber. "Authorization, Protection, and Allocation of Memory in a Large System". PhD thesis. ETH Zurich, 2018.

[Hos19] Nora Hossle. "Multiple Address Spaces in a Distributed Capability System". Master's Thesis. Department of Computer Science, ETH Zurich, 2019.

[Sch17] Daniel Schwyn. "Hardware Configuration With Dynamically-Queried Formal Models". Master's Thesis. Department of Computer Science, ETH Zurich, 2017.

2

Problem Statement

This chapter makes the case for a new model for representing and interpreting physical addresses in a machine for the purposes of memory management and memory subsystem configuration. Secondly, it presents a survey of some of the many violations of these assumptions in past, current, and proposed future hardware. Moreover, it points to the challenges this creates for effective management of physical memory as a resource in systems software.

2.1 Motivation

To this day, hardware designers have come up with many memory address translation schemes among which page-based virtual memory [Den70]

and virtualization [PG74; Int19a] being one of the most ubiquitous. Each address translation scheme has different features, translation granularity, and means of configuration. The common denominator of those translation mechanisms is that system software is responsible for *correctly* configure them. Failing to do so leads to severe problems such as algorithms producing the wrong results, data corruptions, information leakage, crashes and security vulnerabilities.

The objective of this chapter is to demonstrate the existing mismatch between the hardware abstractions and assumption about the memory system used in operating systems today on the one hand, and the architecture and configuration of real hardware as it is seen by software running on the platform on the other hand. This mismatch is a problem. It has lead to various security vulnerabilities and bugs in system software (Section 2.4.2) e.g. 33% of code-changes to the Linux memory manager are bug fixes [HQS16].

The chapter is structured as follows:

1. Section 2.2 presents a survey of real and proposed memory address translation hardware components including their translation and configuration mechanisms.
2. Section 2.3 describes the "single global physical abstraction".
3. Section 2.4 shows the problems that arise with the abstractions currently used by operating system and actual memory subsystem of the hardware platforms presented in the survey.
4. Section 2.5 analyzes the implications of this mismatch for operating system design and implementation.

2.2 Survey of Memory Address Translation

With virtual memory [Den70] and virtualization in general [PG74], processing units, processor cores or DMA-capable devices, only ever deal

with an opaque handle (the virtual address) to the physical resources it accesses. This is one of the corner stones of computing, providing isolation and protection between two tasks running on the same machine.

Processes and even operating systems in virtual machines are given the impression of being the only task or operating system running on the machine providing the illusion of having exclusive access to all resources, sometimes even more resources than exist in reality (e.g. demand paging [Fot61]). This illusion, however, does not stop at the physical machine level: in memory-centric computing [Far+15; Bre+19] there is more memory available than the machine can ever issue addresses for. Therefore, only a configurable subset can be accessed at the same time.

In summary, there are multiple types of addresses (the most prominent of which are virtual and physical addresses), and hardware translates them using different translation schemes configured by system software.

The remainder of this section analyses the terminology of memory addresses used in technical reference manuals (Section 2.2.1) and then presents a survey of memory translation mechanisms, either present in existing, real hardware (Section 2.2.2), or proposed as part of conference submissions or white papers (Section 2.2.3). This provides an overview of the complexity of memory translation schemes.

2.2.1 Address Spaces and Address Definitions

In technical reference or software developer's manuals, hardware vendors describe the features of their products and how to use them. To avoid confusion, the manuals include a section or table about the used terminology. The purpose of this section is to compare the definitions and terminologies of different kinds of address and address spaces found in those documents. The address type and its context are important to precisely refer to a resource in the system e.g. a device virtual address might be different to a guest virtual address.

Intel	**AMD**	**ARM**	**IBM Power**
physical	physical	physical / intermediate	real
linear	virtual = linear	-	virtual
virtual	virtual = linear	virtual	-
logical	logical	-	effective
effective	effective	-	
host physical	host	physical	host real
-	system physical	-	
guest physical	guest physical	intermediate physical	guest real
guest virtual	guest virtual	virtual	fully qualified
DMA	-	-	
I/O virtual	-	virtual	-
-	device virtual	virtual	effective
-	local memory	-	-

Table 2.1: Summary of Different Address Terminologies Found in Hardware Manuals.

Table 2.1 summarizes the address types. A row corresponds to set of terms used by different vendors referring to *similar* concepts.

Intel Terminology The Intel 64 and IA-32 Architectures Software Developer's Manual [Int19a] and the Intel Virtualization Technology for Directed I/O Architecture Specification [Int19b] describe multiple memory models and address types. Memory attached to a processor bus is referred to as *physical memory* and each byte of it assigned a *unique physical address*. The *physical address space* is then a range of addresses from zero to $2^b - 1$ where b is the maximum supported address width. Itanium [Int10a] further introduced multiple virtual address space regions where the combination of them formed a large 85-bit *global* address space.

Physical memory is generally not accessed directly. The processor has three different memory models: flat, segmented, and real-address mode. The memory model defines how a *logical address* (segment selector + effective address) used by the processor is converted into a *linear address* (See Section 2.2.2.1 for an illustration).

The linear address is then translated to a physical address either through a one-to-one mapping or through a page-based translation mechanism (paging). Paging effectively virtualizes physical memory. Depending on its configuration, the linear address corresponds to the physical address (one-to-one mapping), or to the virtual address (with paging). An address is in its *canonical* form if the topmost $(63 - b)$ bits are either all zero or all ones, with b being the number of bits implemented by hardware.

Virtualization partitions the machine and adds another layer of address types. The physical address above is now referred to *host physical address* and is defined as:

> *"Physical address used by hardware to access memory and memory-mapped resources."* – Intel 64 and IA-32 Architectures Software Developer's Manual [Int19a]

System Software running inside a partition or virtual machine sees *guest physical addresses* and applications use *guest virtual addresses*. The

manual speaks about the *view* of physical memory in this context. Addresses used by software on the host processor are *virtual addresses*. Devices operate on *DMA addresses* which may refer to either a host/guest physical or virtual address, or an I/O virtual address.

AMD Terminology AMD uses almost identical terminology to Intel. The AMD64 Architecture Programmer's Manual [AMD19] makes an explicit association between *virtual* and *linear* addresses and the virtual address is translated into *physical* addresses through paging. Similarly, the logical address is then formed by a segment selector and an *effective* address.

The AMD I/O Virtualization Technology (IOMMU) Specification [AMD16] further defines a *device virtual address*, which is either a *guest physical* or a *host / system physical address*. The manual states that the host physical address is "*in most systems identical with the System Physical Address*". The *local memory address* is the device local physical address used to access device private resources, which may or may not be mapped into the system physical address space.

Arm Terminology The Architecture Reference Manual for ARMv8-A [ARM19a] describes the virtual memory system architecture for the 32-bit and 64-bit operating modes. It defines the *virtual address* as an address which is used in an instruction. Similar to the x86 canonical address, virtual addresses have the top bits either all ones or all zeroes.

The virtual address gets translated to an *intermediate physical address* which is the output address of the stage-one translation and the input address of the stage-two translation. The intermediate address then gets translated to a physical address, which corresponds to "*a location in a physical memory map*." If there is just a single stage translation, then the intermediate address is identical to the physical address.

The Arm architecture further defines two address spaces that exist in parallel: secure and non-secure. This distinction provides a mechanism to isolate certain resources and devices from unprivileged accesses or

interference. For instance, only software running in the secure world may access resources within the secure address space. Processors or devices make memory accesses either *secure* or *non-secure* [ARM09]. This implies:

> *"Secure 0x8000 and Non-secure 0x8000 are, technically speaking, different physical addresses"* – ARMv8-A Address Translation [ARM17]

There are no special terms for System MMU [ARM16] (the Arm IOMMU equivalent). The same translation regimes as for processor cores are used.

IBM Power Terminology On the IBM POWER9 platform [IBM18; IBM17], threads use 64-bit *effective addresses* which are comprised of an effective segment identifier and offset to access different storage objects. The address is a *fully qualified address* if it also includes the *effective logical partition identifier* which uniquely identifies the processing thread.

The (process-local) effective address is translated to a 68-bit operating system global *virtual address*. The process addressing context, a segment descriptor defines this translation. The effective segment ID is converted into a virtual segment ID. Together with the offset, this virtual segment ID forms the virtual address. Lastly, the page-based translation mechanism converts the virtual address to a 56-bit *real address*. With virtualization enabled, a partition-scoped page table translates *guest real addresses* to *host real addresses*.

Conclusion Hardware vendors use similar terminology to refer to different address types (Table 2.1). Yet, there are subtle differences especially when referring to addresses a device is using. From the analyzed architectures, IBM Power diverges the most. The difference between virtual and linear addresses are vague, sometimes they are equivalent. Arm explicitly acknowledges the existence of multiple address spaces and AMD the presence of device private resources which may not be accessible from the main processor cores.

2.2.2 Translation Schemes in Real Hardware

Processors, interconnects and memory controller use different addressing modes requiring addresses of different types to be converted between one another. This section presents a survey of different translation schemes which various vendors have implemented in production hardware.

2.2.2.1 Segmentation

In segmentation-based memory systems [RK68] such as on x86 [AMD19; Int19a] or Power [IBM18], software accesses memory through *segments* that have a defined fixed or configurable size, protection attributes and map contiguously onto a linear address space. The remainder of this section focuses on the x86 architecture as an example of segmentation.

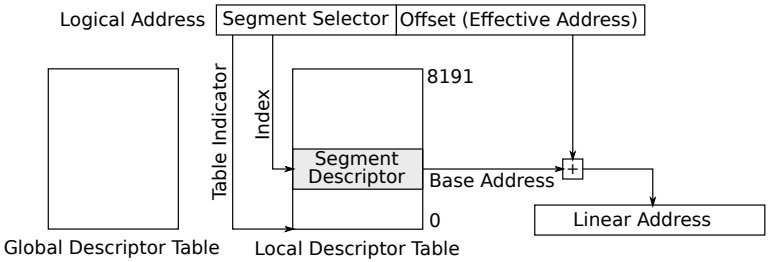


Figure 2.1: Logical to Linear Address Translation Using Segmentation on the x86 Architecture [Int19a].

Figure 2.1 illustrates the conversion of a logical address to a linear address using segments on the x86 architecture. The logical address consists of a segment selector and an offset into the segment (or effective address), combined using the colon as visualization.

$$\textit{Logical Address} = \textit{Segment Selector} : \textit{Effective Address}$$

The segment selector indicates whether to use the global or local descriptor table and specifies the segment descriptor by an index into the table. The descriptor tables are stored in memory. The segment descriptor stores the base and limit of the segment. The linear address is then calculated by adding the offset (effective address) to the segment's base address.

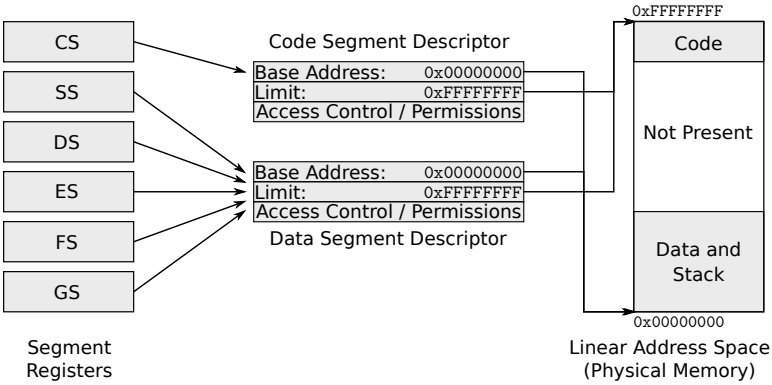
$$\textit{Linear Address} = \textit{Segment Base Address} + \textit{Effective Address}$$

There are six segment registers available on the x86 architecture to speed up the translation process by storing an entry of the descriptor table. Their name indicate their historic usage: CS (Code Segment), SS (Stack Segment) and four data segments DS, ES, FS, GS. Their use depends on the processor mode and the segmentation model. Segments can have an intended use e.g. code, data and stack. Data can be accessed explicitly by stating the segment e.g. `DS:0x1000` or implicitly e.g. through an instruction fetch or stack operations. The x86 architecture offers three segmentation models.

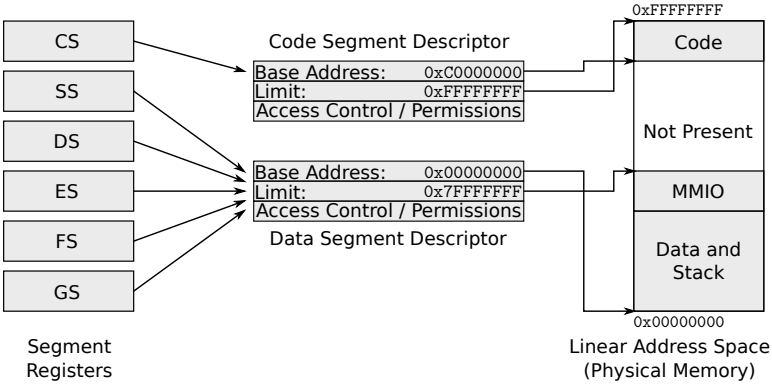
Basic and Protected Flat Models The basic flat model is arguably "*the simplest memory model*" [Int19a] which effectively enables access to a contiguous, linear address space by hiding segmentation as much as possible. There are two segment descriptors required, one for code and one for data segments as illustrated in Figure 2.2. Both descriptors are set up to map the entire linear address space including regions which are not backed by physical resources like RAM or memory-mapped device registers. The processor can therefore access and execute from every memory location including modification of its own code.

The protected flat model sets the base and limits of the segment descriptors to match the actual physical resources present in the machine. Accesses to non-existing memory resources are now caught by the segment limits and trigger a general protection fault.

Multi-Segment Model The basic models above do not enforce separation of code, stack and other data structures. The multi-segment model enables the use of all six segments. Each process has its own descriptor



(a) Basic Flat Model.



(b) Protected Flat Model.

Figure 2.2: Segmentation with the Basic and Protected Flat Model on the x86 Architecture [Int19a].

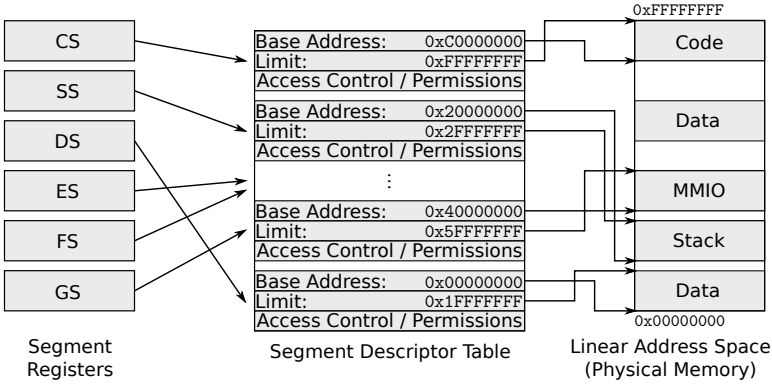


Figure 2.3: Segmentation with the Multi Segment Model on the x86 Architecture [Int19a].

table and uses different segments for code, stack and data which have different access protections e.g. a process cannot modify its own code. Figure 2.3 illustrates the use of multiple segments and how they are mapped onto the linear address space.

Segmentation with Paging Segmentation can be used with and without a page-based translation mechanism. With paging, this results in a two-stage translation mechanism where a logical address is first converted into a linear address using segmentation. The linear address space is then split up into pages which map onto the physical address space. Each page translates a contiguous block of addresses. This translation is defined by an in-memory data structure (more about page-based translation schemes in Section 2.2.2.2). Figure 2.4 illustrates the combined translation mechanism with segmentation and paging.

64-bit Operation Mode During initialization, system software enables the 64-bit operation mode (AMD64 [AMD19; Int19a]) which mostly, but not completely, disables segmentation: limit checks in all segments are

disabled and the processor ignores segment bases for CS, DS, ES and SS and hard-wires them to zero. This results in a flat, 64-bit linear address space where the linear address is equal to the effective address.

The two exceptions are the FS and GS segment registers, which can still hold a value other than zero as base address. Software can use those two additional registers to hold pointers to local data structures, for instance.

2.2.2.2 Page-based Translation Mechanism

Page-based translation mechanisms, or paging for short, are widely used e.g. on x86 platforms [Int19a; AMD19], IBM Power [IBM18], ARMv7 and ARMv8 [ARM19a; ARM17] including virtualization, IOMMUs [Int19b; AMD16], and SMMUs [ARM16]. The next two paragraphs illustrate paging in the native and virtualized environments.

Paging Using paging, the virtual (or linear) address space is divided into naturally aligned pages of equal size, e.g. 4 KiB on x86. Depending on the architecture and processor, larger page sizes may be supported (e.g. 2 MiB

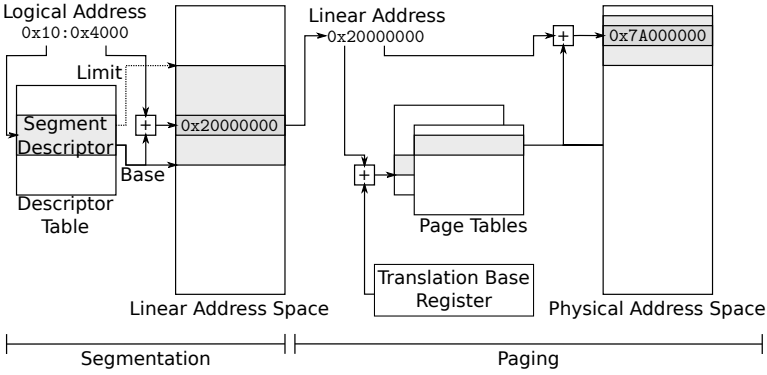


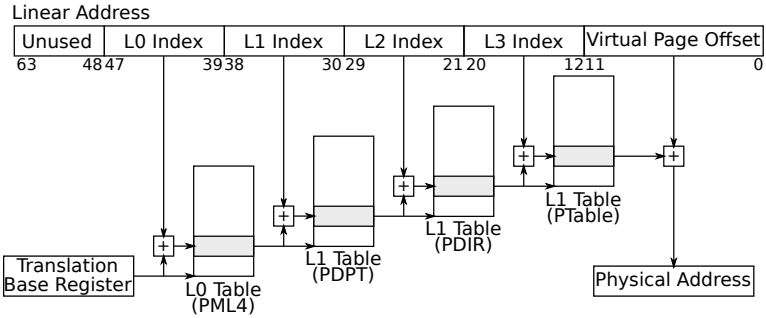
Figure 2.4: Segmentation with Paging on the x86 Architecture [Int19a].

large and 1 GiB huge pages on x86), Some other architectures like Intel Itanium [Int10a], Arm [ARM17] or MIPS R4600 [ITD95] support more page-sizes, including the size of the page tables themselves.

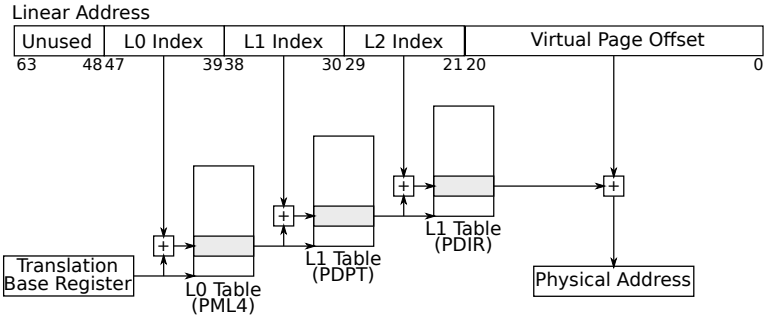
The translation is defined by an in-memory data structure called the page table which defines a mapping from virtual pages to physical frames. The page tables themselves operate on physical addresses. Each entry in the page table defines the target address, permissions, access characteristics and caching properties of a page of memory. Page-table entries may not always be strictly independent e.g. super sections on ARMv7 [ARM19a]. The page table is a multi-level radix tree where each level translates a portion of the linear address space.

Figure 2.5 shows the translation of a linear address into a physical address using a multi-level page table for 4 KiB and 2 MiB pages. Translation starts at the translation base register (cr3 on x86), which serves as the base address of the top-level page table. Parts of the linear address define the indices into the different tables. For each level of the page table, the entry at the extracted index defines the base address of the next table. This process is repeated until the last level is reached. The frame base address of the entry is then added to the page offset to obtain the physical address. Using multi-level page tables, large and huge pages can be implemented by stopping the page-table walk early as shown in Figure 2.5b.

Virtualization using Nested-Paging A virtual machine [PG74; Gol73], emulates a computer system with a defined configuration (CPUs, memory, devices) on top of a physical machine. Virtual machines allow running multiple operating systems on the same server. Using special hardware extensions (e.g. in the processor [Uhl+05]), memory accesses in virtual machines are translated twice. From guest virtual to guest physical to host physical addresses, or virtual-intermediate-physical using ARM terminology for two-stage translations (Recall Section 2.2.1). Each translation stage itself is defined by a page-table structure. The first stage is managed by the guest operating system, and the second stage by the hypervisor or virtual machine monitor using “extended page tables (EPT)”.



(a) Linear to Physical Address Translation with 4 KiB Page Size.



(b) Linear to Physical Address Translation with 2 MiB Large Page Size.

Figure 2.5: Illustration of a Linear to Physical Address Translation Using a Multi-Level Page Table [Int19a].

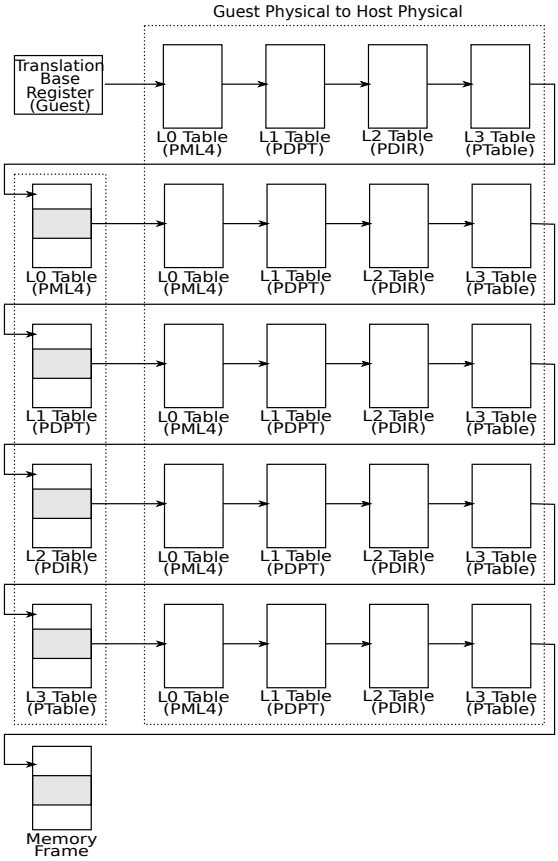


Figure 2.6: Illustration of a Two-Stage Guest-Virtual to Host-Physical Translation with Nested Paging on the x86 Architecture [Int19a].

The guest page tables contain guest physical addresses. Therefore, for each translation step the EPT needs to be walked to obtain the physical address of the next page table.

2.2.2.3 Translation Lookaside Buffers (TLB)

Translating addresses using a full page-table walk results in up to four memory accesses in the native case and up to 24 memory accesses using nested paging. Recall,page tables operate on physical addresses themselves [Int19a; ARM17; AMD19], and hence those memory accesses are physical addresses and do not need to be translated. Consequently, translating an address using a multi-level page table is expensive [Bas+13]. TLBs cache successfully translated addresses to reduce the number of page-table walks required. Consequently, a TLB holds a subset of the translations defined by the page table. Section 4.6 will present an operational model of a software-loaded TLB.

In the event software issues an address which is not present in the TLB, a page-fault exception is triggered. Either the operating system (in the case of a software-loaded TLB), or a hardware page-table walker reads the page table and updates the cache with the new translation. This results in additional memory accesses to translate a linear to a physical address.

TLBs are a cache, which the operating system needs to keep consistent with the backing page table. To make updates to the page table visible to software, existing entries must either be explicitly updated, or invalidated and the hardware page-table walker fetches the new translation from the updated page table.

2.2.2.4 Register-Based Lookup Tables

Page-based translation schemes do not have to use multi-level, in-memory page tables: if the number of pages is small enough, an array of registers may be sufficient to configure the translation. An example of such a lookup table is the system memory page table (SMPT) of the Intel Xeon Phi coprocessor [Int14a] or the Intel Single-Chip Cloud Computer (SCC) [Int10b].

The system memory page table (SMPT) translates addresses from a 512 GiB region in the 40-bit Xeon Phi address space to the 48-bit system address space (possibly protected by an IOMMU).

Figure 2.7 illustrates this translation. The lookup table consists of 32 registers each controlling how a 16 GiB "page" is being translated. There exists a fixed relationship between the register and the region of co-processor physical addresses controlled by the register. Any 16 GiB aligned target address is possible. In contrast to the page tables, an entry of the SMPT cannot be invalidated, it always translates somewhere.

Similarly, the SCC has a 256-entry lookup table each mapping a 16 MiB physical memory frame in the processor's 32-bit address space to the extended memory map of the system. More on SCC memory addressing in Section 2.2.3.1.

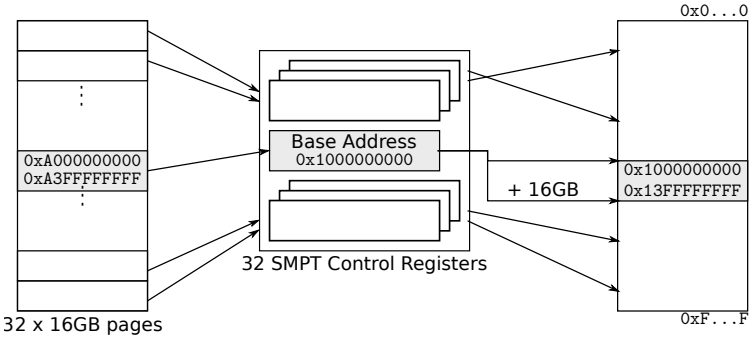


Figure 2.7: The System Memory Page Table on the Intel Xeon Phi Co-Processor.

2.2.2.5 Fully Associative Lookup Table

One way to interpret the SMPT is like a "*direct mapped*" translation cache: each entry of the SMPT holds the translation for one specific, fixed page of memory. On the other side of the spectrum, a *fully associative* translation cache decouples the entry from the memory page it translates, while N-way set associative caches are in between.

The MIPS R4600 TLB [ITD95] is an example of a fully-associative lookup table. It is a software-loaded, fully associative TLB with 48 entry-pairs. The translation scheme is illustrated in Figure 2.8 and further used as an example in Section 4.6. Note, that some TLBs implement a multi-level caching scheme and are N-way set associative.

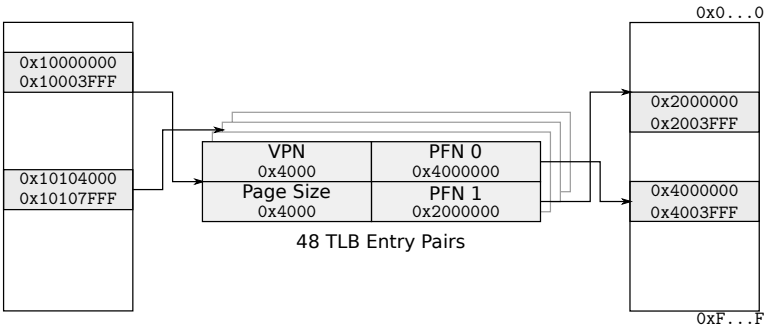


Figure 2.8: Fully Associative, Software Loaded Translation Lookaside Buffer.

Each entry of the MIPS R4600 TLB maps two consecutive pages of virtual memory to two independent physical memory frames. Each entry can map *any* virtual page to *any* physical frame. In theory, it is possible to have two entries translate the same virtual page which leads to undefined behavior and possibly damage the chip [ITD94].

2.2.2.6 Multi-Stage Translation Schemes

Memory addresses might be translated multiple times while a memory request traverses the network of interconnects in the system. Examples of this include the memory controller on multi-socket x86 machines, accesses to the host DRAM from the Intel Xeon Phi co-processor (Section 5.5), and the ARM Cortex-M3 Subsystem on the Texas Instruments OMAP4460 [Tex14].

Memory controllers and multi-socket machines Large server machines consist of multiple processors each having a collection of cores. Each processor has memory controllers with DRAM memory attached (Figure 2.9). The memory access characteristics of such a machine are non-uniform, or NUMA [Lam13] for short, where accesses to a processor's local memory is faster than accessing memory attached to another processor [Gau+15; Kae+15].

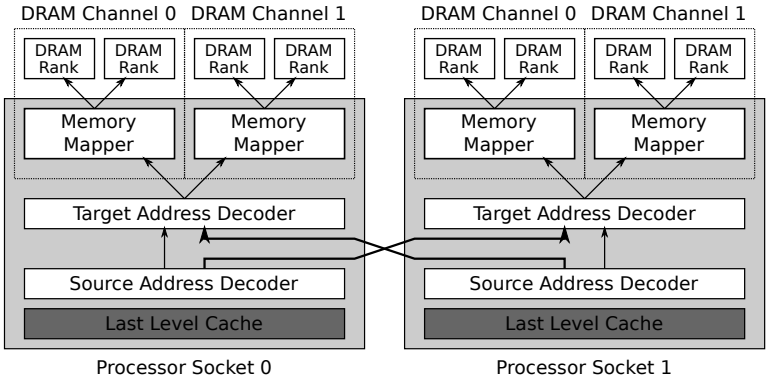


Figure 2.9: Memory Controller Configuration of a Two-Socket Intel Haswell.

Memory controllers of modern processors are configurable. However, there exist hardly any public documentation on the multi-stage translation and for-

warding process of memory controllers. Nevertheless, reverse engineering efforts can reveal the translation scheme and register descriptions [Hil17; Hil+17].

The BIOS (firmware initializing hardware during boot) might display an option to select the visibility of the NUMA topology to the operating system. This corresponds to partitioning of memory into the NUMA regions, or striping among all NUMA nodes at cache-line granularity. The system software can configure the memory controllers using memory mapped registers where each socket can be configured individually.

Figure 2.9 depicts the address translation scheme of Intel Xeon processors of the Haswell and Broadwell generations based on the description in [Hil17; Hil+17]. Note, that the address decoding happens after the last-level cache and therefore in *physical addresses*. The translation is triggered when a memory access fetches data from DRAM. The steps are as follows:

1. The source address decoder forwards the request to the memory controller of the target NUMA node depending on the physical address of the memory access and the configuration of the source address decoder.

2. The target address decoder forwards the requests to the memory mapper of the correct DRAM channel. While doing so, the target address decoder converts the physical address from the system address to the mapper-local address space. Again, this depends on the configuration and the physical address.

3. The memory mapper forwards the request to the corresponding DRAM rank, as defined by its configuration. Again, this converts the address into the DRAM rank local representation.

Secondary Translations on the Ti OMAP 4460 The ARM Cortex-M3 subsystem of the Texas Instrument OMAP 4460 [Tex14] presents an interesting case: the two ARM Cortex-M3 cores share a two-stage MMU setup as shown in Figure 2.10. The MMUs of the Cortex-M3 cores are

configured from the Cortex-A9 processor on the OMAP 4460 through a dedicated configuration port.

The shared L1 MMU translates memory requests and forwards them to the L2 master interface (L2 MIF). This acts as an address splitter, forwarding requests either to the local RAM, ROM, or to the second-stage translation unit (L2 MMU, on Figure 2.10) which translates the address and routes the request to the L3 interconnect. This makes other system resources such as RAM accessible to the Cortex-M3 core.

The interesting case, however, is when the L2 MMU emits an address to the L3 interconnect, which falls into the address range of the L2 MPORT. This creates a cycle: L2 MIF-L2 MMU-L3 interconnect-L2 MIF. Note, a similar cycle exists on the Intel Xeon Phi co-processor. In both instances, they are benign.

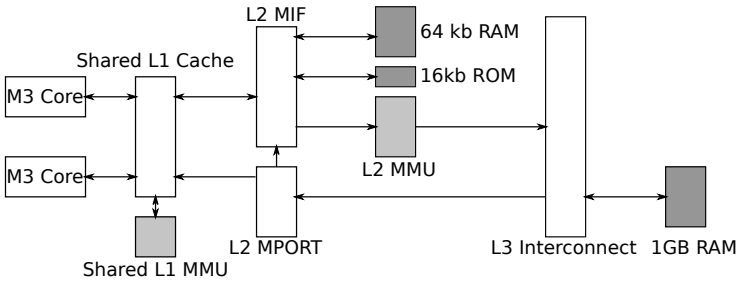


Figure 2.10: Texas Instruments OMAP4460 Cortex-M3 Subsystem.

2.2.2.7 System on a Chip Firewalls

SoC platforms are networks of components which are connected through interconnect networks and buses. Firewalls mediate access from and to hardware components connected to a bus or interconnect.

Based on in-band attributes including the request type (read/write), and connection identifiers for example, the firewalls of the Texas Instruments OMAP 4460 [Tex14] can block or allow access to an entire subsystem.

The firewall is region based, where each region has a start and end address, an assigned priority level, and can have different access permissions. Regions with a higher priority take precedence over lower priorities. Figure 2.11 illustrates a firewall configuration using regions with different priorities shown as levels in the figure. The regions are then projected on one another onto the result plane.

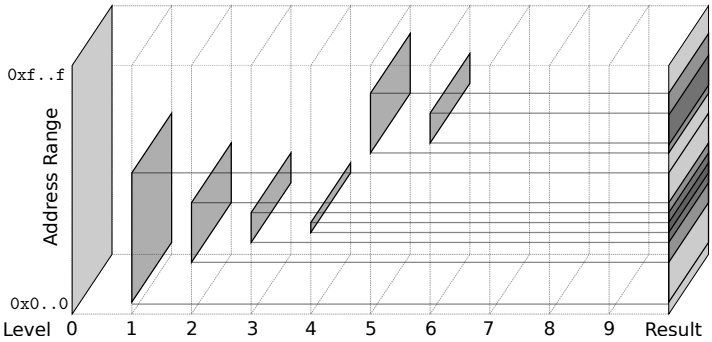


Figure 2.11: Address Filtering in the Firewalls on the Texas Instruments OMAP 4460 [Tex14].

2.2.2.8 IOMMU / System MMU

IOMMUs or System MMUs translate memory requests coming from I/O devices. This effectively restricts the resources a device can access e.g. which RAM regions it can write to. IOMMUs are often configured using the same paging-structure to configure the translations as processor cores use for paging [Int19b; AMD16; ARM16]. In contrast to a processor core, there might be *multiple* devices covered by a single IOMMU which

maintains a context per device. This context then contains a pointer to the root-level page table to be used to translate requests from this device. Similar to "normal" paging, IOMMU also support virtualization using nested paging. Figure 2.12 shows an illustration of the translation process.

The IOMMU then selects the context based on the *source* from which the memory request comes from. This source might be, for example, the PCI bus-device-function triple contained in the PCI transaction. Consequently, even though two devices access the same address, they can get translated differently by the IOMMU.

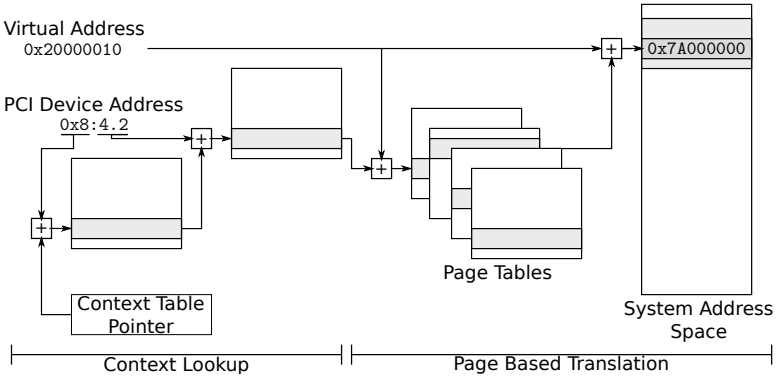


Figure 2.12: Example of a Translation Scheme of an IOMMU Based on Intel VT-d [Int19b].

2.2.2.9 Enclaves

Applications running on an operating system or virtual machines running in the cloud are isolated from each other through hardware mechanisms such as virtual memory. However, they have to trust the operating system, hypervisor or cloud vendor not to interfere with the integrity and confidentiality of their data.

Hardware mechanisms such as the Intel Software Guard Extensions [Int19a] (SGX) are designed to prevent the operating system or hypervisor from accessing application data (e.g. [Fer+17; BPH15]) through memory isolation and encryption. This enclave *"is a protected area in the application's address space, which provides confidentiality and integrity even in the presence of privileged malware."* [Sel16]. However, information may still be leaking out of the enclave through side channels [Van+18].

The Intel Software Guard Extensions (SGX) [Int19a] explicitly assigns memory to the Enclave Page Cache (EPC) to make it accessible to the enclave. Memory accesses to a page of the EPC is subject to additional hardware access control checks e.g. from within an enclave, code cannot be loaded from outside the enclave, and access to pages in the EPC from outside the enclave results in undefined behavior. Normal protection such as paging and segmentation are still effective as the enclave effectively runs with the linear address space of a user-level process.

2.2.2.10 ARM TrustZone

Recall, ARM-based platforms that implement TrustZone [ARM09] split the memory subsystem into two: a *secure* world and a *non-secure* world. Effectively the "*ARM architecture defines two physical address spaces*" [ARM17]. Memory, devices and processors can be assigned to either the secure or non-secure world. Some of which are capable of being in both, or switch between worlds by changing the operation mode (e.g. secure monitor calls on the Cortex processors)

Figure 2.13 illustrates a simplified setup with one ARM Cortex processor, some DRAM and a device – the generic interrupt controller (GIC). A portion of DRAM is *configured* to be secure memory and hence only accessible from secure devices or processors. Similarly, the GIC is TrustZone-aware and some of its registers are "*security banked*" meaning despite being located at the same address, a different register is accessed depending on whether the access originated from the secure or on-secure world. A formalization of TrustZone revealed several imprecise or ambiguous specifications [Arc19].

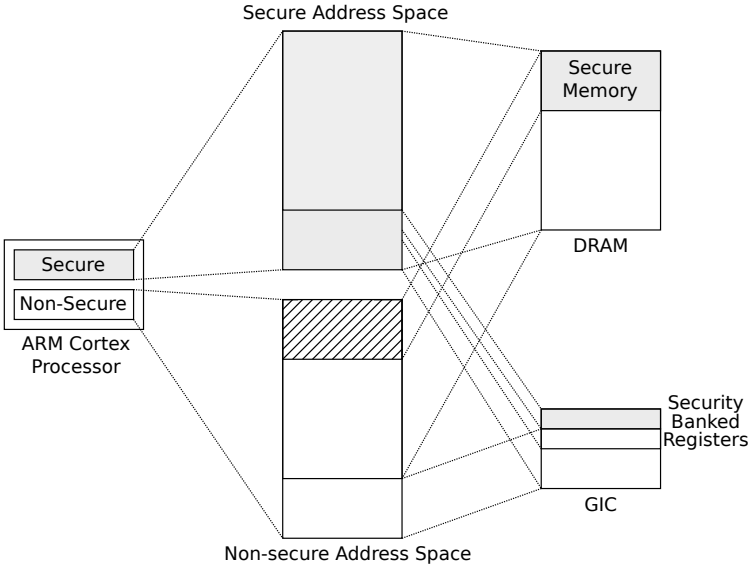


Figure 2.13: Two Physical Address Spaces in ARM TrustZone.

The processor can operate in secure and non-secure mode which changes the view of the world: only in the secure mode, the processor is able to access the secure memory region.

2.2.3 Proposed Translation Schemes

The implementation of address translation schemes usually involves trade-off decisions such as the granularity of protection and translation vs. required resources to configure how hardware translates addresses. This section surveys improvements and alternatives to the translation schemes described in the previous section and prototype hardware.

2.2.3.1 Intel's Single-Chip Cloud Computer

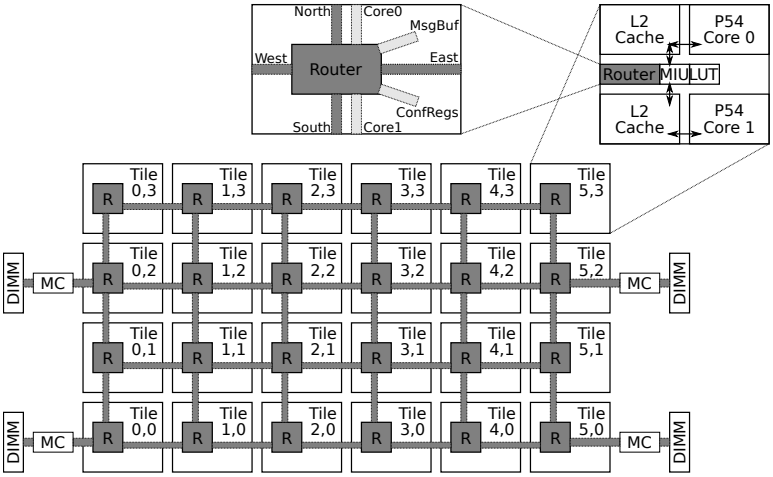
The Intel Single Chip Cloud Computer (SCC) [Int10b] consists of 24 tiles with two Pentium cores each. The tiles form a two-dimensional mesh structure where each tile can be identified using its (x, y)-coordinates. Figure 2.14 on page 37 illustrates the top-level layout of the chip. The architecture has four memory controllers (MC) that allow up to 64 GiB of DRAM. In addition, each core has private, on-die memory for message passing buffer (MBP) and system configuration space.

The cores operate as normal Intel Pentium processors including segmentation and paging as described earlier. Logical and linear addresses are translated to (core-local) physical addresses. Upon a miss in the core's L2 cache, the request is forwarded to the mesh interface unit (MIU) on the tile which then consults the system address lookup table (LUT) to obtain the destination tile of the request.

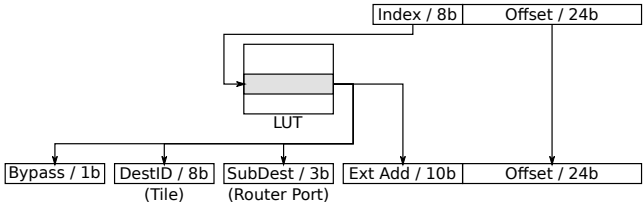
The memory request is then forwarded to the destination tile using the routing network. The router in the destination tile forwards the request to the sub-destination ID passed as part of the address. The bypass bit indicates whether to bypass the mesh interface unit to access the tile-local memory buffer directly. Each core having its own lookup table and configuration registers being memory mapped, it can be configured such that the core is completely sequestered and cannot change its own lookup table, which defines what resources the core can access.

2.2.3.2 Reducing TLB Miss Overheads

Big-memory workloads experience a high TLB misses rate resulting in overheads due to walking the page-table structures. This is a problem as such workloads end up spending 50% or more of their clock cycles handling TLB misses [Bha17], while at the same time those workloads seldom use the rich features provided by page-based virtual memory. Direct Segments [Bas+13] map a part of the linear address space using segmentation-based technique, while mapping the remaining part of the



(a) SCC Top-level Tile Network.



(b) Local to System Address Translation on the SCC.

Figure 2.14: Memory Addressing in the Intel Single-Chip Cloud Computer (SCC).

linear address space using page-based virtual memory translation (Figure 2.15).

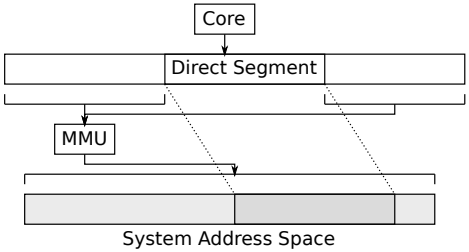


Figure 2.15: Illustration of a Direct Segment Mapping.

This effectively preempts the page-table walk, similar to large-pages which may not necessarily be good in NUMA systems because of memory access imbalances causing memory controller congestion [Gau+14]. The overhead of radix-tree based virtual memory translations can be reduced by leveraging application specific translation schemes and exposing physical memory resources to applications [Ala+17]. The direct segment approach is also applicable in the virtualized case where it can reduce the full nested page-table walk to two direct segment translations [Gan+14]. Range Translations [Gan+16] and redundant memory mappings [Kar+15] added support for multiple ranges or segments to be mapped. Those approaches speed up translations at the cost of coarse-grained protection and requirement of large, contiguous blocks of 'physical' memory. Agile paging [GHS16] combines nested page-table walks and shadow-paging to reduce overheads of 2D page-table walks and virtual machine monitor involvement in page-table updates. Those proposals effectively carve out a contiguous region of memory which is translated differently. Devirtualized memory [HHS18] tries to identity-map memory to avoid virtual to physical translations by leveraging permission validation techniques [KCE92; WCA02].

2.2.3.3 Hardware Capabilities

Page-based virtual memory introduces a tradeoff between translation and protection: while the use of large and huge pages speed up TLB misses by cutting page-table walks short, the protection granularity is coarser grained as a result. Instruction set capabilities such as CHERI [Woo+14] or "Matching Key Capabilities" [Ach+17a; Azr+19; Bre+19] separate translation from protection as illustrated in Figure 2.16. To some extent, this adds a segmentation-like layer on top of the virtual address space, where the segment descriptor is encoded into the hardware capability. The set of physical resources that are accessible is defined by the combination of the capability protection and the virtual memory translation.

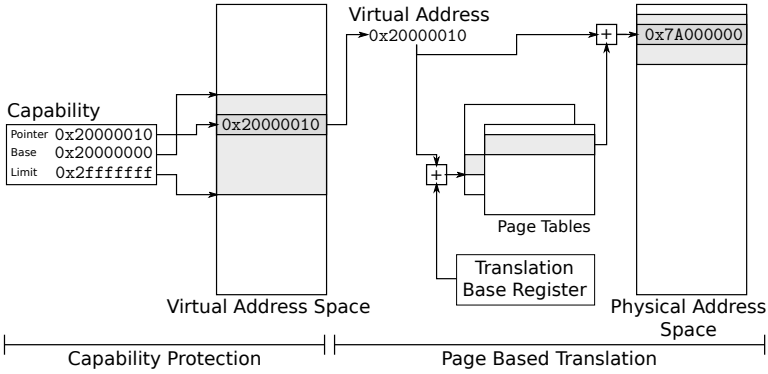


Figure 2.16: Hardware Capabilities used as "Fat Pointers".

Memory accesses are sanitized against the size base and limit of the hardware capability (in the case of CHERI encoded in a "fat pointer") and any attempts of out-of-bounds accesses are blocked. CODOMs [Vil+14] tags every page in memory while using the instruction pointer as a capability that defines which page tags a program can access. Single address space operating systems use global address translations and enforce isolation by

maintaining protection domains [Cha+94]. Protection lookaside buffers caches the (domain,page) protection information [KCE92]. Processes running on the Cambridge CAP Computer [NW77; Lev84] access memory through capabilities stored in the process resource list. In contrast, the Hypernel [Kwo+18] uses a hardware module between the processor and DRAM supporting protection at word granularity.

2.2.3.4 Heterogeneous Memory Architectures

The Part-of-Memory (POM) [Sim+14] manages two types of memory (slow and fast). POM distinguishes between a *page-table physical address (PTPA)* and a *DRAM physical address (DPA)*. On a memory access, the PTPA is further translated to the DPA using a segmentation based translation scheme to either a region within the fast or slow memory (Figure 2.17). The configuration of the secondary translation step is defined by the segment remapping cache (SRC) and the segment remapping tables (STR) residing in fast memory. This mapping can be changed dynamically depending on the access patterns of the application. Unified memory architectures for memory-mapped SSDs [Hua+15; Abu+19] us a similar approach to access data on SSDs.

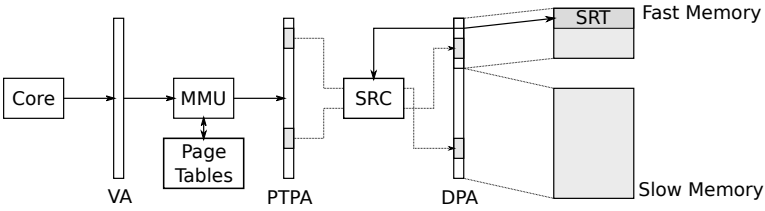


Figure 2.17: Illustration of the Part-of-Memory POM Architecture.

Impulse [Car+99; Zha+01; ZPC06] adds address translation hardware to the memory controller that allows transforming a data structure scattered across pages or cache-lines in main memory to a single, contiguous region

on the systems bus. Example applications include sparse matrices, struct-of-arrays and array-of-structs, column-store vs row-store. In contrast to POM [Sim+14], Impulse allows for a more flexible, secondary translation step from the system bus to the memory controller.

2.2.3.5 Multiple Views of Memory

Instead of the multi-stage address translations mentioned above, the same byte in DRAM can appear at different addresses in the processors local physical address space, where each mapping has a different memory layout e.g. array-of-structs and struct-of-arrays views as in SAMS [GKG08; GKG10]. Memory controllers of modern processors already allow a limited set of transformations [Hil+17; Hil17]. Page overlays [Ses+15] allow memory management at cache-line-granularity where each virtual page is mapped to a regular physical page and an overlay page using a direct mapping where the overlay address is a concatenation of the process ID and the virtual address. (Figure 2.18). The overlay mapping table (OMT) stores which pages have an overlay. The memory controller receiving the request in the overlay memory store region, returns the cache-line from normal main memory depending on the configuration of the overlay.

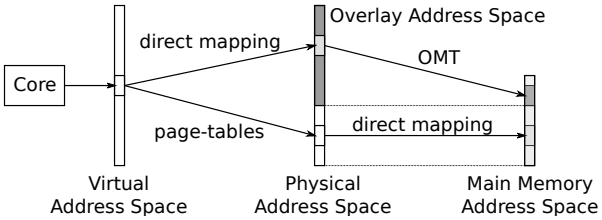


Figure 2.18: Address Translation with Page Overlays.

2.2.3.6 Large Rack-Scale Pooled Memory Resources

Technological advancements in optical interconnects [ORS13] and nonvolatile memory allow low-latency, high-bandwidth access to rack-scale memory pools or "fabric-attached memory" [Far+15; Bre+19; Kee15]. Fabrics such as Gen-Z [Gen18] provide a "memory-semantic" interconnect to access volatile and nonvolatile storage in a machine through a byte-addressable load/store interface. Mediating access and enforcing authority in such an architecture poses new challenges to the system software developers [Azr+19; Bre+19].

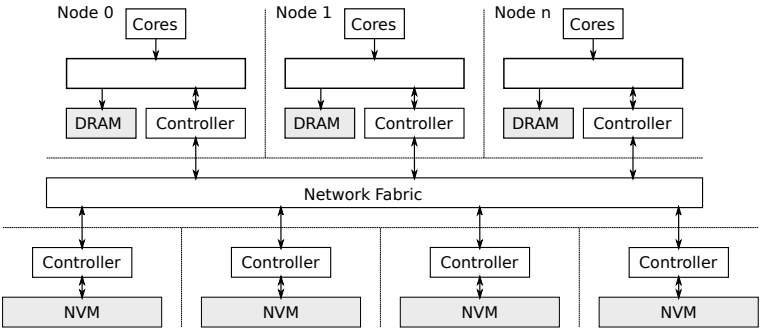


Figure 2.19: Rack-Scale Pooled Memory Resources (as in [Kee15]).

An example of pooled memory resources is shown in Figure 2.19. There is a collection of compute nodes, each having a set of cores and local memory resources. Parts of the node's local address space is mapped onto the network fabric where storage nodes are attached. This is yet another instance of a multi-stage translation scheme. In such a system, the size of the memory pool is larger than can be addressed by the core. Memory resources in the memory pool have a larger address.

2.2.3.7 Near Data Processing / In Memory Processing

Executing operations on data often means fetching it from off-chip memory into the processor caches, apply the operation and write it back to off-chip memory. This induces overheads in terms of latency, bandwidth and memory consumption. Processing in memory (PIM) or near-data processing (NDP) architectures try to reduce this data transfers by deploying small cores close to memory that are capable of executing simple operations. Proposed architectures (e.g. [Ver+16; Ver+17; HS01]) employ various translation and protection schemes providing a global address space or local memory regions, for instance. System software needs to manage those translations and components [Bar+17].

2.2.4 Summary

The survey presented a list of different address translation schemes. In summary, translations are defined using in-memory data structures (e.g. page tables) or registers (e.g. SMPT of the Xeon Phi co-processor). The translation granularity is either fixed or configurable. The actual translation depends on the target or sources address. Memory requests traverse a multi-stage translation scheme with possible loops. Hardware extensions such as enclaves and capabilities put another layer on top of virtual memory mediating access to the physical resources. From a single address space, addresses can be translated using multiple, different translation schemes (e.g. direct segments and paging).

2.3 Current Physical Address Space Model

When opening a computer systems textbook, one gets presented with a nice and simple view of a computer, where a processor has an address translation unit which turns virtual addresses into physical addresses (refer to Figure 1.1 on page 4).

In the case of multi-processor systems, each core has its own translation unit which can be configured independently. As far as the operating system is concerned, running a process in parallel on multiple cores simply means programming the translation units with the *same* page tables, defining the virtual-to-physical address translation. To set up the page tables for a process, the operating system simply needs to know the important physical addresses (e.g. RAM, device registers, page table roots, etc.) and manage this address space accordingly. Every byte-addressable resource such as RAM or device register, that could be addressed by a processor has a unique (physical) address.

Virtual-memory support is a complex element of both operating system design and processor architecture [CKZ12; CKZ13], providing translation and protection mechanisms, and a uniform and flat view of memory for each process running on the system. To make this work, the following key simplifying assumptions about the underlying *physical* address space play a central role:

1. All physical resources such as RAM and memory-mapped I/O registers, appear in a single, uniform 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 physical resource – memory cell or hardware register.

This view of the world with the assumptions stated above does, provides a nice and simple model of a platform and to write software for it. However, it does not hold for modern hardware ranging from rack-scale systems to systems-on-a-chip (SoCs) in mobile devices – if it ever has held in the past. The next section examines where these assumptions break. This provides evidence that the concept of a globally unique physical address (as seen from the processor) will become even more ambiguous and increasingly unsuitable for a using it as a unique identifier for resource management operating systems.

2.4 Problems with the Current Model

The previous two sections presented a survey of different memory addressing schemes and the commonly used abstraction of the memory subsystem using the single address-space-assumption. This section now highlights cases where there is a discrepancy between real hardware and the single-address-space assumption (Section 2.4.1) and resulting problems in systems software (Section 2.4.2).

2.4.1 Observations

Based on the survey (Section 2.2), this section highlights several observations which provide evidence of the mismatch between real hardware and the operating system abstractions.

2.4.1.1 First Observation: Disjoint Address Spaces

Rack-scale systems are a tightly interconnected collection of individual machines forming a single unit, in this case a rack. Hardware vendors like Oracle, Teradata, SAP, and others combine commodity servers and custom-built hardware to a single system, where servers are connected through a high-speed, low-latency network. A diagram of an example system can be seen in Figure 2.20. Each machine has processors, memory resources and a hardware module (labeled "RDMA Card" in Figure 2.20) connecting the machine to the global interconnect network.

One of the features of such systems is the capability to access memory resources of another machine directly. Remote direct memory access (RDMA) allows applications to copy data to and from the RAM of another [Rec+07] without involving the operating system. The RDMA card issues a memory read or write request to the global interconnect network. This request contains the target machine identifier and the address within the target machine's local address space. This address may be further translated before issued to the machine local interconnect. Memory resources are therefore identified with a machine identifier and an address.

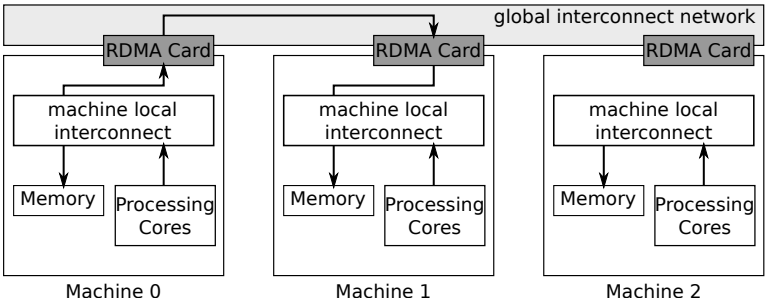


Figure 2.20: An Example of a Tightly Interconnected Rack-Scale Machine.

Scale-out NUMA [Nov+14] pushes this idea further by integrating an RDMA-like interface into the processor's cache coherence protocol to reduce or eliminate overheads from network stacks, PCI express and DMA copying resulting in a dramatically lower overhead for remote reads, writes, and atomic operations. Applications simply use loads and stores to access remote memory. Scale-out NUMA exposes some of a machine's local resources to other machines of the cluster. Exposed resources are globally identified with a node identifier, a context and an offset. This triple is then converted into a local address in the destination machine.

The machine boundaries of such tightly interconnected systems is increasingly blurred. As in NUMA machines [Lam13], buffer allocation and correct placement of data is critical to performance [Kae+15], but in contrast requires global coordination and protection of multiple physical address spaces. Moreover, the address at which a buffer appears might not be the same on all the machines.

2.4.1.2 Second Observation: Parallel Address Spaces

The ARM architecture [ARM17; ARM09] defines two physical address spaces explicitly: "secure" and "non-secure" where the normal world cannot access the secure address space, whereas the secure world can

access both address spaces. Some resources are aliased in both address spaces, whereas others only ever appear in one (e.g. banked registers)

Accessing the same physical address from the secure and non-secure world, therefore, resolves the addresses in two different address spaces (Figure 2.13). Memory and devices may behave differently depending on whether the access originated from the secure world or not.

2.4.1.3 Third Observation: Different Views

PCI Express [PCI17] is a programmable, high-speed bus connecting peripheral devices such as graphic cards (GPGPUs), network cards (NICs), FPGA accelerators etc., with memory bus of the host system. Many of those PCI devices contain large amounts of on-card memory forming a separate physical address space. An example is given in Figure 2.21.

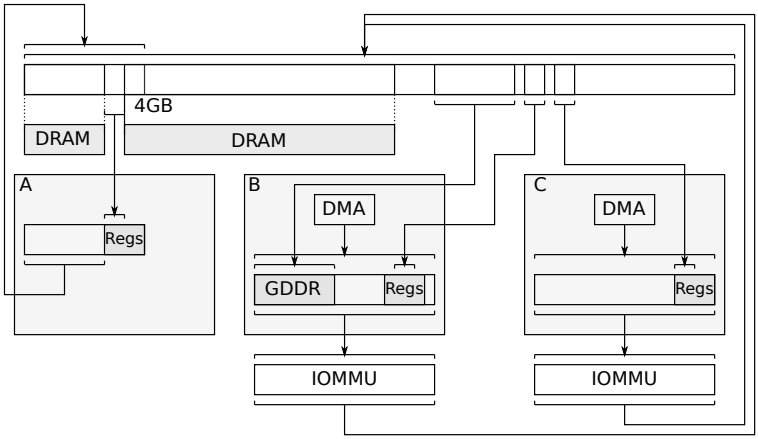


Figure 2.21: Address Spaces of a System with PCI Express Attached Co-Processors.

To perform useful work, data needs to be transferred over the PCI bus between host memory and the device. This can be done using loads/stores by the processor, a DMA transfer on the device, or even using a separate DMA engine to perform a transfer as shown in [Ach14]. Setting up data transfers is non-trivial:

1. Not all memory resources may be reachable from the processor or the DMA engine doing the transfer (e.g. device A in Figure 2.21 only supports 32-bit addressing).

2. Even if all resources are reachable, only one direction of transfer may be supported e.g. Intel CrystalBeach 3 DMA controllers [Int13; Int17] (device C in Figure 2.21) can only copy from DRAM to GDDR, but not the other direction.

3. Depending on the current configuration, different source and destination addresses must be used e.g. devices B and C in Figure 2.21.

In any case, software has to translate addresses correctly independently which transfer mode is being used. IOMMUs [Int19b; AMD16] or System MMUs [ARM16], do not facilitate the problem at hand, they rather add complexity leading to vulnerabilities [Mar+19; MMT16], as they effectively introduce additional address spaces.

2.4.1.4 Fourth Observation: Intersecting Address Spaces

Some PCI Express devices such as GPGPUs, accelerators or FPGAs contain a significant amount of memory resources (e.g. GDDR in the case of GPGPUs) and they are capable of running general purpose workloads. The system is heterogeneous, consisting of at least two types of cores and memory. An example of such a system is shown on Figure 2.22, where two Xeon Phi co-processors [Int14b] are plugged into PCI Express slots, resulting in an intersecting address space configuration where the host and Xeon Phi address space have *windows* into each other's address space.

On the one hand, the Xeon Phi co-processor appears in the host's address space as a PCI Express device with two distinct memory regions: one 16

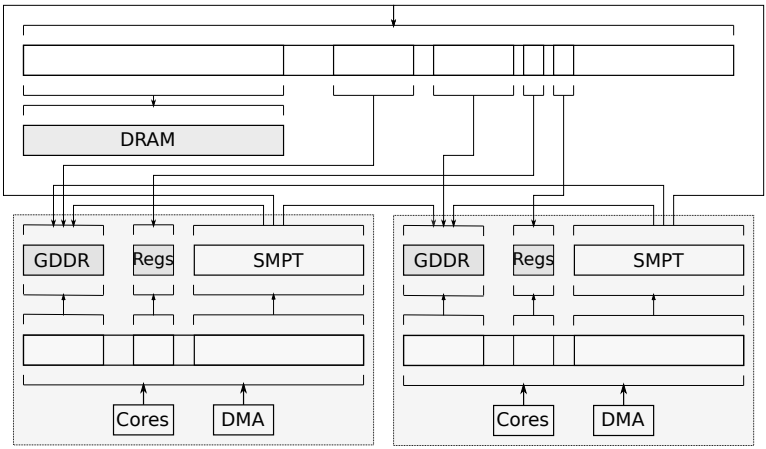


Figure 2.22: A System with Two Intel Xeon Phi Co-Processors.

GiB region maps to the GDDR RAM on the co-processor card, while the other, smaller region holds the Xeon Phi's memory mapped I/O registers. Where those regions appear is configurable and determined by the PCI Express bridge programming algorithm.

On the other hand, the Xeon Phi cores have a different view of the world: GDDR RAM and MMIO registers appear at fixed locations in the local address space. The upper half of the address space (512 GiB) is where the window to the host address space resides. This region is divided into 32 separate 16 GiB regions, each of which can be translated independently via a lookup table to any physical address on the host side – this includes the GDDR RAM of both Xeon Phi cards that may be present in the system.

This example invalidates all three assumptions stated earlier: there are at least two distinct physical address spaces, the host address space is larger than the Xeon Phi address space and therefore not everything is reachable at any point in time, and memory resources have at least two different physical addresses. Even worse, this real-world hardware example permits

addressing loops to occur, as highlighted in Figure 2.22 – something an operating system would ideally be able to prevent. True loops might cause bus errors or data aborts, whereas benign loops alias memory at different addresses which is a correctness problem for address-based access control and resource management.

Programming frameworks like CUDA Unified Memory [NVI13], OpenCL SVM [Khr18], AMD HSA[HSA14] or Linux Heterogeneous Memory Management (HMM) [Lin19c] try to unify host and graphics accelerator memory, which works for their *specific* setup, but does not work in the general case. Those frameworks provide a rather opaque solution: Linux HMM for instance migrates memory between host and device memory transparently without explicit application request.

2.4.1.5 Fifth Observation: Dynamic Address Spaces

Parts of the system may change during runtime. Power constraints prohibit running all cores and devices in the system at the same time (dark silicon [Esm+11]). Depending on the workload, specialized accelerators are turned on, while other ones are switched off. Similarly, PCI Express supports hot-plugging of devices [PCI13].

This effectively changes the number of address spaces in the system, including the physical resources within them. Not only the number of address spaces changes, but also how the remaining address spaces map regions is adapted to account for the vanished resources.

2.4.1.6 Sixth Observation: Code, Data and Stack Segments

Segmentation is one of the protection modes of x86 processors. The Intel 64 and IA-32 Architectures Software Developer's Manual [Int19a] specifies three segmentation models using up to six different segments which can be classified into code, stack and data segments. Depending on the type of memory access and the segmentation mode, the processor will access the memory location through another segment i.e. an instruction

fetch, stack or data access to the same address results in different locations being accessed as illustrated in Figure 2.23.

Segment registers are still used today to hold pointers to core-local data structures, for instance. Moreover, proposals published at architecture conferences such as direct segments [Bas+13] advocate the use of special memory regions which are translated through segmentation instead of paging to eliminate page-walk time. In addition, hardware capabilities such as CHERI [Woo+14] provide segment-like base-limit access restrictions on top of virtual memory.

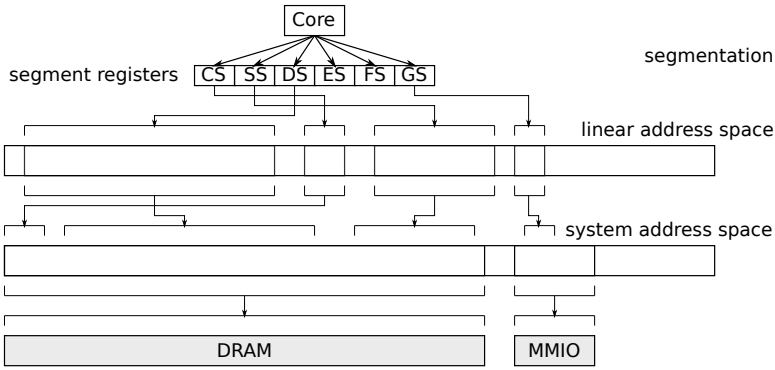


Figure 2.23: Memory Access Through Segmentation.

2.4.1.7 Seventh Observation: Configurable NUMA

Large servers typically consist of multiple processors connected through a high-speed coherency interconnect such as Intel's QPI [Int09] or AMD's HyperTransport [Hyp08]. Each processor contains memory controllers which have some RAM attached. Memory accesses from cores are non-uniform: accessing local memory has different characteristics than accesses to memory attached to a remote processor, also called NUMA [Lam13]. An

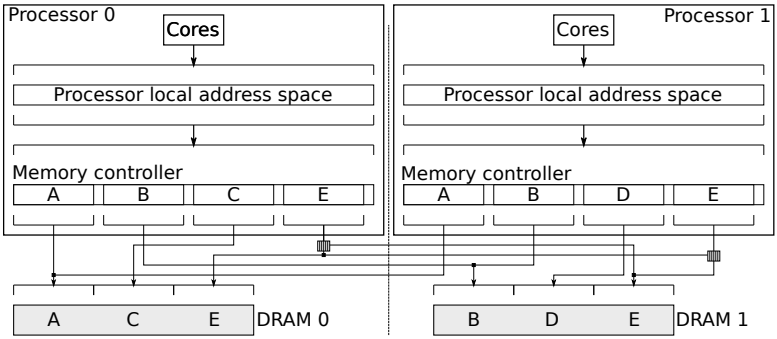


Figure 2.24: NUMA Memory Controller Configuration with Private Memory (C and E).

example of a two-socket system is shown in Figure 2.24: Each processor has multiple cores and a memory controller which maps four memory regions: A and B are shared, C and D are private, and E is interleaved between the two DRAM modules.

The memory controller in each processor effectively works as an arbiter forwarding memory accesses to the local memory or to the remote node. This can be configured at cache-line granularity. It is possible to alias memory regions with different interleaving. Depending on the workload, memory can be allocated with the optimal interleaving resulting in performance improvements [Hil+17]. Moreover, this configuration can be different for each processor socket, even parts of the memory resources can be made inaccessible to implement private memory.

2.4.1.8 Eighth Observation: Heterogeneous SoC

The problem is not just confined to high-end servers with custom-built hardware and accelerator cards. System-on-a-Chip (SoC) based platforms such as the Texas Instruments OMAP series [Tex14], nVidia Tegra [NVI17] or

QualComm's Snapdragon [Qua18] are inherently heterogeneous consisting of a mix of different cores, memory, devices and interconnects, each of which effectively forming a different address space. Figure 2.25 shows the block diagram of the Texas Instruments OMAP 44xx SoC series.

This chip has two ARM Cortex A9 cores as main application processors and consists of at least 13 distinct interconnect each of which is guarded by firewall modules defining which subsystems can access which parts of the chip and at which addresses. In addition, there are at least seven processing cores (most prominently two ARM Cortex A9 and two ARM Cortex M3, plus DSP and graphics processors), last but not least a collection of on-chip DMA-capable devices. Each of those processing cores and devices have distinct views of the system, often having a private access port to other subsystems. Finally, the address translation of this chip can be configured such that a memory access is routed through the very same interconnect twice. This effectively results in a loop.

In summary, the interconnect of such SoC chips forms a complex network of buses of different widths. Highly configurable translations and firewalls between the buses and the cores, devices or memory attached to them enable sand-boxing of low-level software on some cores. Security extensions such as ARM TrustZone [ARM09] divide the components into secure and non-secure worlds, where some resources are only accessible when the core is in a particular mode. Moreover, devices can behave differently whether the access originated from a secure or non-secure context.

In addition, the memory maps printed in the publicly available manuals for such SoC chips list multiple physical addresses for the same device register or memory cell. The address is different depending on which core is initiating the bus cycle (e.g. [Tex14]).

2.4.1.9 Ninth Observation: Core-Local Resources

Even an off-the-shelf, low-end personal computer consists of multiple physical address spaces. PCI Express devices aside, every core has its own memory mapped-local interrupt controller (local APIC [Int19a]).

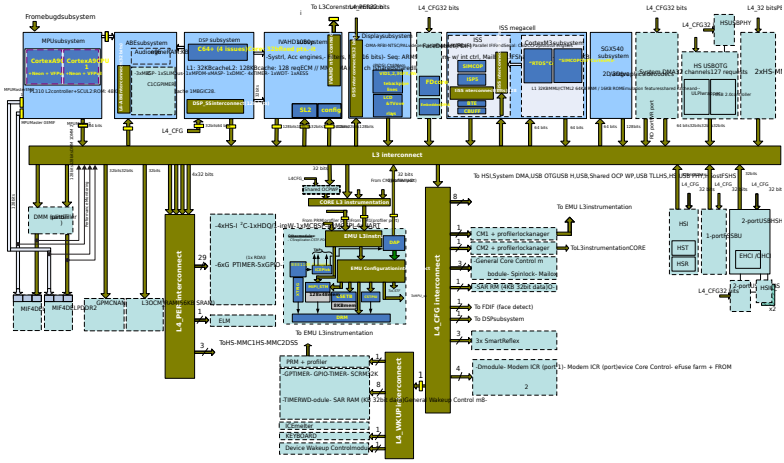


Figure 2.25: Schematics of the Texas Instruments OMAP4460 SoC [Tex14].

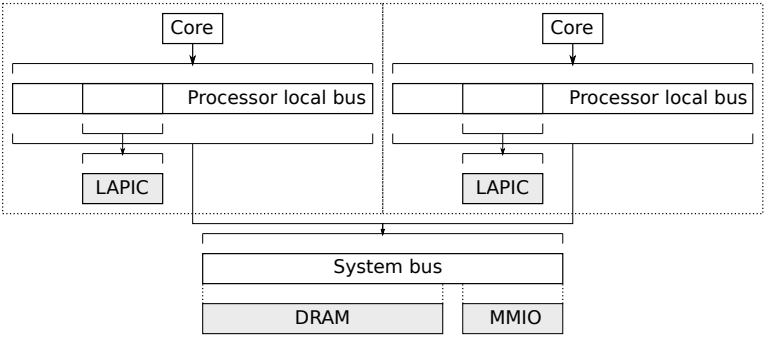


Figure 2.26: Core-Private Resources of the Local APIC on x86 processors.

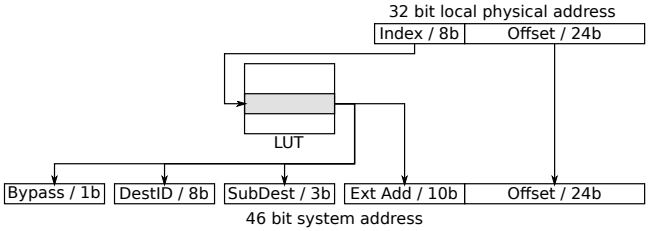


Figure 2.27: Processor's Address Space Size is Smaller than the System Address Space at the Example of the SCC.

Figure 2.26 shows an illustration of this scenario. This also applies to hyper-threading or simultaneous multi-threading where there are multiple, independent execution contexts per physical core. The local APIC has a well-known, but configurable address. Consequently, in a multi-core system the address, by default, is the same for all and therefore even though the cores issue the same address different resources are reached.

2.4.1.10 Tenth Observation: Limited Number of Bits

Finally, the size of available memory simply exceeds the number of addressable bits of an address space. Machines with terabytes of main memory are already available today and projects like "The Machine" [Far+15; Kee15] envision main memory in the order of petabytes. Note, that current platforms support up to 48 bits or 256 TB of addressable memory, which is also a limitation for virtual address spaces [El +16]. In addition, the number of available bits may be even lower, as some of them are used to route memory transactions to the appropriate part of the machine, or mapping of kernel windows etc, and hence further reduce the amount of addressable memory.

Hardware vendors proposed designs to *extend* the physical address by adding additional address bits (e.g. Intel's Physical Address Extension (PAE) [Int19a], Arm's LPAE [ARM11], or Intel's Single Chip Cloud

Computer [Mat+10]). An example is shown on Figure 2.27. Increasing the number of available bits is costly, as it requires additional wires and space in the caches etc.

Furthermore, efficient scaling of conventional page-based translation mechanisms to very large memories is questionable [Bai+11; WCA02]. It seems more likely for hardware designers to add additional layers of physical address translation: the Intel Single-Chip Cloud Computer [Mat+10; How10] extended a 32-bit address to a 46-bit address using a 256-entry lookup table, the Intel Xeon Phi [Int14b] extends a 40-bit address to a 48-bit address. Similar approaches are followed by rack-wide persistent memory pools as in [Ach+17a; Azr+19; Bre+19].

2.4.2 Resulting Problems in Operating Systems

The complexity of the memory subsystem and its configuration of hardware platforms is also reflected in the operating system, which in turn is often complex itself, and relevant source code is poorly documented [Gor04]. The struggle with wrong assumptions about memory addressing on various platforms by introducing special cases and actually getting it right in an operating system is revealed by numerous bugs and vulnerabilities [HQS16]. Examples include:

- CVE-1999-1166 -Potential map of kernel memory to user-space.
- CVE-2011-1898 - Using DMA transfers to write to interrupt registers.
- CVE-2013-4329 - Using DMA before the IOMMU is setup properly.
- CVE-2014-0972 - IOMMU registers not write-protected.
- CVE-2014-3601 - miscalculation of the number of affected pages.
- CVE-2014-8369 - Due to bug in fix of CVE-2014-3601.
- CVE-2014-9888 - Access rights for data pages.

- CVE-2014-9932 - Improper address range calculation in TrustZone.
- CVE-2016-3960 - Handling of shadow page tables.
- CVE-2016-5195 - Race conditions using memory hardware features (DirtyCow).
- CVE-2016-5349 - Not providing enough memory address information to secure execution environment.
- CVE-2017-5925 - Potential ASLR break due to page-table walks.
- CVE-2017-6295 - Reading the wrong buffer location in TrustZone.
- CVE-2017-8061 - Interactions of virtually mapped stack with DMA scatter lists.
- CVE-2017-12188 - Not properly translate guest virtual to guest physical addresses.
- CVE-2017-16994 - Ignoring holes in huge pages.
- CVE-2018-11994 - SMMU misconfiguration allows access to HLOS memory.
- CVE-2018-1038 - Allowing a process to modify its own page-table entries.
- CVE-2019-0152 - Insufficient memory protection in System Management Mode.
- CVE-2019-2250 - Writing to arbitrary memory location.
- CVE-2019-2182 - Wrong page permissions leave read-only pages writable.
- CVE-2019-10540 - Insecure setup of SMMU.
- CVE-2019-15099 - Wrong DMA address in descriptor.
- CVE-2019-19579 - Possible DMA to host memory from Xen guest.

Out-of-bounds memory accesses, address miscalculations or misinterpretations, under- or overflows of addresses and offsets, can lead to unexpected behavior, including crashes and memory corruption [KA18]. DMA-based attacks [SB13] and misconfiguration of IOMMUs or SMMUs [Mar+19] further lead to security implications.

For example, violations of system integrity policies allowed the WiFi co-processor to request a change in the IOMMU configuration to gain access to memory at will [Gon19]. Device drivers being one of the major causes of operating system crashes [SBL03; Mat+14]. Overlapping memory addresses with MSI-X interrupt ranges (Linux commit 17f5b569e09cf) can lead to unintended interrupts, and no proper memory writes.

In summary, the complexity of the memory subsystem of real hardware and the lack of a faithful and sound representation thereof in system software has been the source of serious security vulnerabilities and bugs. This is a problem for which this thesis proposes a possible solution.

2.5 Implications for Operating System Architectures

Based on the survey and the observations above, there are two options:

- *Ignore the problem.* One can simply ignore the problem and continue with the old "cores plus RAM" view of the world. As a consequence, have the operating system manage the increasingly small areas of the machine where this old model still fits.
- *Embrace the problem.* One can acknowledge the problem and find a better model to express the memory subsystem topology with its configuration and physical addressing.

This section describes implications and challenges for the design and implementation of system software.

2.5.1 Operating System Design Implications

The observations above suggest five key implications for operating system design and implementation with respect to memory management

Naming of resources. There are multiple-levels of translation and how they map addresses may change. Therefore, physical addresses as seen by a processor core are not a stable concept. Ultimately, there is *some* local physical address that refers to the resource, but which one is it? This is important for memory management. Allocating a physical frame on one core and freeing it on another core must operate on the same underlying physical frame. System software must be able to unambiguously name a specific resource. Failing to do so, can lead to double allocations or releasing of physical resources and this can lead to data corruption due to unintended sharing of memory.

Different views. Memory resources can appear at different addresses on different cores, or the same address may resolve to another memory resources. To access a particular resource, a core may need to issue a different local physical address. This is important when sharing data structures between multiple cores and devices: what is the local physical address in the address space of the other device or core? Getting this ad-hoc translation wrong results in memory accesses using invalid pointers to the wrong data structures, causing data corruption or information leakages. System software needs to explicitly convert the name of a resource into a valid local physical address e.g. when updating the page tables of a process.

Configurable multi-stage translation. Memory requests traverse multiple interconnects and translation units. This process transforms the request with its address several times between the issuing core and the destination resource. This chain of translation steps is configurable which implies that the core-local physical address is not a stable concept. It can change. System software needs to ensure that this change is reflected in the relevant translation layers e.g. invalidate a process' corresponding virtual address space region to avoid unintended data accesses. Moreover, the view of a

process running its virtual address space must remain consistent across cores and reconfiguration steps (i.e. pointers must always refer to the same resource). System software needs to track every part of an address space that is mapped in other address spaces.

Limited reachability. The possible number of memory locations in a machine exceeds the range of physical addresses a core can issue. This implies that some memory locations are not directly or just temporary reachable from a subset of the cores. System software needs to be aware of this when allocating memory. Allocation policies must include constraints on where the memory is used from. Failing to do so renders the memory unusable on some cores, worse pointers could be truncated or refer to different memory resources resulting in accesses to the wrong data structures or registers.

Access to kernel data structures. Processor cores are not the only source of bus cycles: DMA capable devices and other hardware components can access memory resources. System software needs to be aware of this when allocating memory for kernel data structures. As a security property, it must be impossible for any user-level process running on a core, co-processor or device to access kernel data structures. This suggests a distinction between user-accessible and privileged memory regions.

2.5.2 Virtualization as a Solution?

Virtualization technologies are able to emulate a particular environment and topology to their guests. Memory virtualization can provide a unified address space: virtual machines can be configured to emulate a single, uniform guest physical address space per-virtual machine. This is similar to the linear address space of processes.

However, this still does not solve the problem: the lowest layer in the stack (the operating system or hypervisor) still needs to deal with different views, reachability and multi-stage translation described above. Virtualization

adds another layer of translation on top. Moreover, running concurrently on multiple cores requires careful synchronization of translation structures.

2.5.3 Operating System Design Challenges

In the presence of multiple, intersecting and configurable address spaces, managing translations and physical resources remains a key operating system design challenge:

1. How does an operating system correctly and efficiently allocate, manage, and share regions of physical memory regardless whether it is DRAM, byte-addressable non-volatile memory, or memory-mapped I/O registers?

2. Which translation units does system software need to configure to make the allocated region accessible and what is the correct configuration to apply?

3. How does an unambiguous naming scheme for physical resources look like?

4. How does systems software communicate and share a set of memory locations (e.g. to set up a shared buffer) between two cores, co-processors or devices of the machine?

5. When can processes run on multiple cores concurrently using a single, shared virtual address space? Under what circumstances is it guaranteed to be feasible and when is it not? How does system software need to configure the relevant translation hardware?

6. How can system software developers get assurance that the address being issued by a process or device actually ends up at the right memory resource?

Large-scale platforms such as Hewlett-Packard's "The Machine" [Far+15; Kee15] raise additional challenges in memory management regarding

different memory types, reliability, and security and authorization aspects in highly distributed architectures.

- *Heterogeneous Memory.* Already today, machines consist of a highly heterogeneous memory architecture including DRAM, Graphics Memory, byte addressable read-only memory (ROM) and device registers. The introduction of persistent memory [Cut19] forming large pools of non-volatile memory holding operating system and application data structures requires careful allocation and tracking of memory regions [Bre+19].

- *Reliability.* With the distributed nature of tightly interconnected machines, memory or interconnects may fail transiently or partially. Systems software and hardware must be aware of potential failures when accessing memory nodes, either local or remote [Kha+17].

- *Security.* The questions "who should be able to change which translation unit" and "what resources should be accessible" will become crucial for security related aspects of system design. Already today, SoC platforms include configurable firewalls to shield critical code from accesses of other software running on the same chip [ARM09; Tex14]. Likewise, the Intel SCC [How10] allows to completely sequester cores even when running in privileged mode through programming of the system lookup tables.

- *Energy Consumption.* DRAM is a major contributor to energy consumption of today's servers [SW09]. Approaching the power-wall, it might be impossible to power on all components of a system [Esm+11]. Thus, the operating system must take power constraints into consideration when allocating and managing memory.

2.6 Conclusion

This chapter provided evidence based on several real world and prototype examples that a physical address space is no longer what it used to be. The single, globally uniform physical address space is an illusion.

Multiple address spaces form a network of numerous heterogeneous processor cores, devices, co-processors, interconnects, buses and byte-addressable memory resources such as DRAM, non-volatile memory, or devices registers, all of which together forms a tightly interconnected distributed system [Bau+09b]. Address-translation units and hardware firewalls translate and manage accesses to and from hardware components. Two distinct cores and devices have different views of the system. In summary, there is no single "reference" physical address space [Ger+15].

Moreover, the chapter presented problems resulting from the mismatch between the single address space abstraction and real hardware, and the resulting implications and challenges for the design and implementation of system software. Addressing any of these challenges requires a clear and sound basis for unambiguously naming and referring to physical memory resources. This is a prerequisite allowing an operating system to reason, control and manage memory accessibility from different parts of the system, resource allocation and address translation schemes.

The remaining chapters of this thesis explores ways to find applicable techniques and architectures to design and implement operating systems which are capable of handling the large amount of address spaces of today's and future machines.

3

Related Work

The previous chapter showed that a physical address is no longer (or never has been) a stable concept. A memory request traverses multiple interconnect and buses where its address is translated several times, often based on the configuration of translation units and firewalls. Modern computer systems from rack-scale to SoCs are composed of multiple physical address spaces. The interactions between those address spaces is non-trivial: they overlap and intersect in complex, and dynamic ways.

Abstract system representations and models thereof are an important part of system software and runtime systems. They are used in identifying memory resources, as performance models for allocation and scheduling decisions, understanding and specification of memory systems and hardware platforms, and system software verification.

This section presents related work in the following areas:

- Section 3.1 describes sources of system topology information to operating systems.
- Section 3.2 shows behavioral platform descriptions including the semantics of memory accesses.
- Section 3.3 presents a survey of physical resource management in operating systems.
- Section 3.4 investigates software runtime systems and programming languages.

3.1 System Topology Descriptions

Having a clear description of the system topology is important for system software to locate particular physical resources such as memory, devices and processors. System software obtains relevant information from parsing firmware provided data structures, hardware discovery mechanisms, use information encoded in domain specific languages, or use hard coded values in a header file when building a platform-specific image.

3.1.1 Self-Describing Hardware and Firmware

Certain hardware subsystems are self-describing. Standards such as PCI Express [PCI17] and USB [USB17] define discovery and enumeration procedures to obtain the bus hierarchy and the connected devices. System software needs to correctly initialize and program the devices. This process includes assigning addresses: in the case of USB devices are operated by the host controller using a logical address within the USB hub topology, whereas in the case of PCI Express devices appear at configurable physical addresses in the system address space through PCI bridge windows. The PCI Express specification [PCI17] defines a set of

constraints (e.g. alignment, ordering, size) a valid bridge configuration must satisfy. This is a natural match for declarative programming and constraint solvers which operate on top of an encoded PCI Express topology to produce a valid configuration for all PCI Express bridges satisfying all constraints imposed by PCI standards [Sch+11; Sch+08].

Tools like LIKWID [THW10; THW12] use the self-describing features of processors (e.g. the `cpuid` instruction on x86 processors) to provide detailed information about the processor topology. This includes the mapping of hardware threads to cores, and information about the cache hierarchy like the different sizes, associativity and how they are shared among the processor cores of the system.

During the boot process, lower-level firmware can provide information about the system topology through well-defines data structures or services: UEFI [UEF17] provides system services to allocate and free memory, and to obtain a system memory map which describes the memory resources of the system and whether they are unallocated or how they are used. ACPI [UEF15] defines a set of tables that list processors, PCI Express root complexes, IOMMUs with their device assignments, memory controllers with affinity and bandwidth information.

In summary, the self-describing mechanisms of hardware in combination with low-level system firmware can provide a platform description for operating systems, which in turn provide this information to applications. However, this information is incomplete. Those mechanisms only provide a *partial* description of the platform. For instance, PCI Express describes the existence of a compliant device, but not the processing and memory resources of the device itself, ACPI lists the different IOMMUs but not how they are translating memory accesses.

3.1.2 Domain Specific Languages

The self-describing mechanisms or data structures provided by firmware only provide a partial description of a platform, or worse they may be not available at all. System software needs to use other sources of information.

One example used in production environment are DeviceTrees [dev17] adapted from the *Standard for Embedded Power Architecture Platform Requirements* [Pow11]. They provide information of devices and components including processors, memory and peripheral devices for which there is no discovery mechanism. Similar to ACPI or PCI, DeviceTrees indicate the presence of a particular device, but not the resources of the device itself. As the name suggests, a DeviceTree is a tree data structure with a single root. They are not capable of describing complex, non-uniform heterogeneous platforms. The hierarchy described by a DeviceTree is projected to the view of a single core, while other topology information such as caches, TLBs and views from other cores are ignored.

The Arm Architecture Specification Language (ASL) [Rei16] specifies the environment of an Arm platform including the instructions with its encoding and semantics, registers, stack and program counters, exceptions, system operations, and the memory translation of the processor's MMU with a page-table walk. From the ASL, architecture and register documentation can be generated. The combination of an ASL interpreter and elf-loader produces a simulator (or executable specification) of an Arm processor [Rei17a].

Similar to the ASL register specification, which describes the details of every bit field of a register, Barrelfish's Mackerel [Bar13] provides similar functionality to describe registers and hardware-defined data structures of a device. In addition, Mackerel has a notion of *address spaces* which defines how device registers are accessed (e.g. I/O, memory mapped or the Arm model specific register space). An address space defines read/write functions of specified bit widths, and translates reads/writes to the relevant instructions (e.g. I/O or loads/stores).

In contrast, the Sockeye language [Sch17; Bar17b] describes memory, interrupt, clock and power subsystems of a platform on an abstract level. Sockeye is descriptive and not behavioral. It can express the existence and static state of an MMU, however it does not define the behavior of the translation mechanism and the page-table walker, for instance. Sockeye was motivated and influenced based on the model that Chapter 4 presents and

is further used as part of the operating system implementation described in Chapter 6.

3.1.3 System Topologies Summary

In summary, hardware description standards and bus specifications provide information about the hardware and the topology of a system. However, none of the methods described is capable of representing the entire memory subsystem with enough detail and semantics to enable formal reasoning of memory addressing in a system.

3.2 Behavioral System Descriptions

The platform description languages and discovery mechanisms of the previous section provide information *where* a certain resource is located and at which address it is accessible. However, two platforms with identical memory map can provide different memory access semantics. This section surveys related work of behavioral memory models focusing on their representation of memory address translations.

3.2.1 Micro Architecture Specifications

Platform descriptions indicate the location at which physical memory resources of a machine reside. They may further provide information on the memory type. However, those specifications are not behavioral: they do not model the effect of different load and store instructions issued by the processor. In particular, out-of-order execution, memory access reordering, and the memory model of the processor are not expressed. This is an orthogonal problem to the work presented in this dissertation. Nevertheless, this section presents the related work in this area with respect to memory addressing and address translation.

The memory model of a hardware architecture defines the guarantees that hardware provides on the ordering of memory accesses. Weak memory models as used on Arm and IBM Power platforms, allow many memory accesses to be re-ordered leading to different execution traces due to write buffering or speculation. Ideally, the set of possible traces should be a subset of the set of allowable traces even in the presence of reordering or speculation [Alg+10].

Behavioral models of weak memory system (Arm and Power) express software visible semantics of memory operations [Alg12; Flu+16] and the number of fences used in programs [AHW15]. This also includes pipelining effects, or operation reordering in write buffers. However, those models stop at the last-level cache of the processor and do not capture the complexity thereafter, which this dissertation focuses on and thus complementing the weak memory models [Alg12; Flu+16]. Similarly, the total-store ordering (TSO) model [OSS09] expresses the semantics of the x86 memory model.

Each instruction the processor executes has a certain effect on the processor and memory state. The instruction set architecture (ISA) defines the semantics of the different instructions a processor can execute. Behavioral models of the ISA e.g. 32- and 64-bit Arm [FM10; Rei16; Rei17b] including a concurrency model [Flu+16], or x86_64 [Das+19] provide a framework to unambiguously write down and specify the semantics of every instruction of a micro architecture. Based on this model specification, interpreters and executable specifications provide an execution environment to run and test software, and examine the effect on the processors model state [Das+19; Rei17a]. This also includes the effects on a particular memory address with respect to store instructions and the visibility to other processor's on the platform. Serval [Nel+19] uses the Rosette [TB14] language to specify the semantics of instructions to reason about the correctness of software using symbolic execution. While modeling memory accesses and their semantics with respect to possible re-orderings in write buffers, the models stop at the processor boundaries or express the memory as a flat, byte-addressable global array.

3.2.2 Processor Models

Processors are complex pieces of hardware on their own consisting of execution pipelines, load and store units, and several layers of caches or buffers. The first verification efforts (e.g. CLI's FM9001 [HB92; BHY92; BHK94]) only targeted parts that today would just comprise the processor core without much more. The mechanized proof shows that the circuits of the FM9001 (as specified in the hardware description language) implements the instruction-level specification. Later projects such as the Verisoft VAMP [Bey+06], modeled the cache-memory interface and the processor at gate level and shown that the collection of gates implements the specification. Those types of verification show the functional correctness of the hardware at gate level with respect to its specification. Despite modeling parts of the memory subsystem, those CPU models did not attempt to capture and express its full complexity especially the PCI Express bus hierarchy, non-uniform memory access configurations of multiprocessor systems or multi-stage translation schemes despite being commonly present in modern systems of that era. This is the focus of this dissertation.

The memory system micro architecture influences the instruction-set semantics of processors, which affects the considerations on how hardware, including the instruction-sets, are modeled [Vel01; GGA05]. A behavioral specification can be defined in terms of instruction-set model e.g. the HOL4 Arm model [FM10], or machine-readable specifications e.g. Arm V8-A System Level Architecture [Rei16]. Those models provide a foundation to represent and reason about the behavior of software running on the processor. However,they do not include the complex interconnects present on, for example, modern SoCs.

Translation lookaside buffers (TLBs) caches virtual-to-physical translations defined by the in-memory page table. Ensuring consistency between the cached translations in the TLB and the in-memory page table is security critical. Operational models of a TLB [SK17; SK18] formally express the state of the TLB and the semantics of the maintenance operations. The model focuses on the virtual-to-physical address translation, including

page-table walks and the resulting memory accesses. It does not consider further translation of the local physical address.

Lastly, there seem to be no industrial projects that claim to target the area of physical memory addressing [Hun+17]. This thesis focuses on the complexity in the physical interconnection of devices.

3.2.3 Behavioral Models Summary

The micro architecture of a platform defines the behavior of instructions and memory accesses. Behavioral models of processors specify the semantics of instructions, in particular the possible re-ordering and visibility of memory writes. However, those models currently stop at the processor boundaries, and include the processor's MMU at most. Complex networks of buses and interconnect with multi-stage translation schemes, and the routing of memory accesses are not expressed in those models.

3.3 Memory Management in Operating Systems

Operating systems need an abstraction model of the memory addressing subsystem of the system hardware to manage physical resources and configure address translation hardware. Correct management of physical memory resources is one of the core tasks of system software. To accomplish this task, system software needs a way to refer to memory resources and to implement a suitable accounting facilities to keep track of allocated and free memory. This section surveys the memory management and physical resource representation of operating systems.

3.3.1 Monolithic Operating Systems

The BSD operating system [BSD19; CP99] manages a data structure (`vm_page_t`) per page of physical memory. This data structure contains the physical address of the memory page which is then used to set up page-tables and considered globally unique. Similarly, Linux [Lin19a] also uses a data structure (`struct page`) per physical memory frame which is identified by its physical frame number. The Linux kernel can be configured to use one of three different memory models which assume either a flat, contiguous physical address space or supporting different memory nodes and memory hot plugging. In its evolution, the Linux memory subsystem experienced various bugs and vulnerabilities [HQS16]. Heterogeneous Memory Management [Lin19c] attempts to unify the view between processors and devices by duplicating the processor's page-table in the device page-table. The goal is that a pointer stays valid when dereferenced from the processor and the device. (refer to Section 6.2 for a more detailed description).

The Popcorn Linux project [Bar+15a] deploys a replicated kernel operating system model on a heterogeneous hardware platform with different instruction set architectures by providing a consistent memory view across machine boundaries (single system image). The hardware model includes a global eventually consistent memory and memory areas that are exclusively accessible by processor groups. Popcorn Linux implements a distributed-shared-memory abstraction using page replication. Popcorn Linux supports migrating applications between different instruction set architectures, which requires a special compiled application image and the reconstruction of the page tables on the new platform.

Biscuit [CKM18] is an operating system written in Go [Go19]. It implements the POSIX API and is able to run unmodified Linux binaries. Biscuit makes use of the Go runtime for memory allocations and garbage collection for pages. Unlike Linux, there is no reverse mapping or NUMA (non-uniform memory access) or heterogeneous memory support.

3.3.2 Single-Address-Space Operating Systems

In contrast to the operating systems above, where each process runs isolated in its own virtual address space, a single-address-space operating system (SASOS [Wil+95]) runs all processes within in a single, global virtual address space which is sparse, flat and has a 64-bit addressing width (e.g. MIPS R4000 series [ITD95]). Consequently, all translations are global. Any virtual address to a memory object remains valid for all processes which allows implementing shared pointer-based data structures. This separates protection from translation: instead of using different translations, protection domains [KCE92] provide mechanisms for protection and isolation between processes. The TLB only stores address translations, whereas a protection lookaside buffer caches protection tags, but no address translations [KCE92]. This is not limited to TLB-based architectures. An alternative to this approach is provided by memory protection keys as available on the Intel Itanium architecture [Int10a], for instance.

Opal [Cha+92; Cha+94] uses a single virtual address space where all primary and secondary storage is mapped. Any memory residing object is named with the same unique virtual address by all processes that access them. This facilitates sharing. Opal uses protection domains to restrict access to resources (this is analogue to a Linux process). Opal uses disjoint virtual segments and capabilities naming them for access control. A protection domain can attach a segment if it holds a capability to it. Opal also supports the integration of persistent storage into the global virtual address space. This implements a single-level store [Kil+62] which enables persisting pointer-based data structures without serialization using the virtual memory abstraction to hide the disk as persistent storage.

The Nemesis operating system [Han99] provides a self-paging mechanism which makes applications responsible for all their memory faults using their own resources. The basis-virtual-memory abstractions used in Nemesis are the stretch (a range of virtual addresses, similar to Mach's regions) and a stretch driver handling faults occurring on the stretch. To handle a fault, the stretch-driver can use resources already owned by the application it belongs to. Both, stretches and physical memory are allocated centrally.

Applications can request physical frames (even with special properties) from the frame allocator. The virtual memory system allows mapping, unmapping and querying the current translation of a virtual address. Nemesis uses a concept of self-contained modules [Ros94] without unresolved symbols and mutable data. A module can be executed by any domains. This improves sharing of code and data segments using pointers between protection domains.

Mungi [Hei+98] is a single-address-space operating system that runs on standard 64-bit hardware (MIPS R4600 [ITD95]). It uses objects, a contiguous range of virtual pages, as a basic storage abstraction. The protection domain defines the set of objects a process is allowed to access. Rights on objects are represented using password capabilities [APW86]. Mungi uses a guarded page table [Lie96; Lie94] to store and look up virtual-to-physical mappings during a hardware TLB miss on a software-loaded TLB (e.g. the MIPS R4600).

The IBM AS/400 [McC98; SC97; Daw97] featured a large, virtual address space, which was emulated using custom hardware on top of processors supporting smaller address spaces. The AS/400 supported single-level stores, and provides functionality to do object-based protection using capabilities [HSH81]

In summary, single-address-space operating systems run all processes in the same address space. This allows using the virtual address as a unique identifier for objects in the single global address space.

3.3.3 Verified Operating Systems

Formal verification of operating systems (and software in general) provide correctness guarantees with respect to an abstract specification and a representation of the hardware platform. The proofs are therefore based on a particular abstraction of a machine and its memory subsystem. The abstract specification and hardware model are assumed to be correct.

The seL4 microkernel [Kle+09] externalizes physical resource allocation to user-space processes while enforcing integrity in the kernel. seL4 uses typed capabilities [EDE08] to enable user-space to securely manage physical memory which is abstracted as a flat, linear addressable array. This exposes physical memory to user-space applications and provides a strict policy-mechanism separation by exposing low-level capability operations which enable the implementation of memory management policies in user-space. seL4 assumes a single, fixed, physical address space. The model does not include any no other translation hardware besides the processor's MMU and does not provide guarantees of safety in the presence of other cores or incorrectly programmed DMA devices.

The machine model used in the CertiKOS [Gu+16] operating system uses logical memory and distinguishes between memory that is private to the processor and memory that is shared between multiple processors. This distinction eliminates synchronization for private memory. CertiKOS uses a page allocation table to manage physical memory. Each physical page has a corresponding entry in this table. The modeled memory management unit translates virtual to physical addresses using a page map.

Hyperkernel [Nel+17] uses a push-button approach to verify the correctness of xv6-based operating system [CKM06] which is another UNIX-like operating system managing physical memory using 4 KiB pages. Their memory model consists of a virtual and physical address space acknowledging that aliasing of physical memory in the kernel address space pose challenges for verification especially when the virtual-to-physical mappings can change. This is not limited to operating system verification. Models of memory consistency that explicitly verify memory addressing and translation only consider the case of virtual-to-physical translation mappings [RLS10].

The verification efforts need a hardware model, and they used one that is available at the time, or which fits their verification goals as proving the functional correctness of a real operating system like seL4 comes with significant proof effort. This array-like model of physical memory, while certainly useful to prove the correctness when running on an abstract

machine, does not reflect reality accurately enough. In particular the complex networks of modern SoCs, hierarchical bus topologies as in PCI Express, or multi-stage translation schemes are not captured.

3.3.4 Microkernel Operating Systems

In contrast to monolithic operating system kernels, microkernels provide only a minimal set of mechanisms to ensure isolation and protection, and to provide inter-process communication, while pushing many of the operating system personality into user-level services and libraries. This enables applications to safely implement their own virtual address space management policies, for instance.

The L4 microkernel [Lie95] uses address spaces as an abstraction for memory management. Initially, an address space is created empty and is then populated and managed through operations which grant, map or flush pages in an existing address space. The grant operation transfers a page between two address spaces, the map operation creates a (shared) mapping of a page in another address space, and flushing a page removes the page from all other address spaces. Address spaces are thus defined recursively with the base being sigma zero, a special address space containing a mapping to all physical memory (excluding the kernel region). All resources are referred to by an address in some address space. Flexpages [HWL96] enable address space management to operate on pages of arbitrary sizes in contrast to a fixed page size.

The CMU Mach operating system [Ras+87] manages memory resources using memory objects which are managed by a server. Memory objects can be mapped into a task's virtual address space by that object's pager (the task that created the memory object). When creating a new memory object the calling task obtains access rights to a port representing the new memory object. Physical memory serves as a cache for virtual memory objects. Mach uses physical page numbers to refer to physical frames which could only belong to at most one memory object at a time, and are allocated on demand from pools held by the kernel. Advanced Shared Virtual

Memory (ASVM) [ZTM96] extends Mach's virtual memory system to work distributed over multiple machines. ASVM extends the use of physical memory as a cache for the contents of virtual memory objects to multiple nodes and copies. Extended memory management (XMM) [Bla+98] added support for distributing memory management among multiple Mach nodes across a machine cluster. This enabled Mach address spaces to share any memory with any other address space independent of its location in the cluster of machines.

The Chorus Distributed Operating System [Roz+91] is a modular OS that consists of a small nucleus running on multiple machines. The Chorus memory model is based on global segments, a storage construct e.g. file or swap area. Segments are identified by capabilities and can be mapped into a region (virtual memory space) of a process running on Chorus. A segment server provides a read/write IPC interface reacting to read/write requests to the segment's backing store. The Chorus nucleus manages a per-segment local cache of physical pages. A segment can be shared among actors running on different machines. The segment server must keep the local caches of the segment consistent.

Exokernel [EKO95; EGK95] pushes the management of hardware resources to an untrusted library operating system. This separates resource protection (done by the kernel) and resource management (handled by the library), and in turn allows applications to use their own policies to manage resources. The Exokernel explicitly exposes allocation (of specific physical resources), physical names (to avoid a level of indirection) and revocation protocols. Physical pages are protected by capabilities. The Exokernel also exposes TLB resources to the application.

The K42 operating system [Kri+06] and its predecessor Tornado [Gam+99] is built on an object-oriented structure. K42 implements address spaces which consist of contiguous regions of virtual addresses, which conceptually map onto a special "file" representing physical memory. The assignment of page frames to files is controlled by the file cache manager, which obtains frames from the page manager implementing global paging policies. This memory structure is expressed as clustered objects connected

by references. The hardware representation of a process' address space is managed by a hardware address translator managing hardware segments (e.g. the page-tables defining the segment translation). Hardware address translators may be shared among multiple address spaces. K42 is aware of the machine topology i.e. the affinity of memory nodes to processors. It uses this information to allocate memory in proximity to the processor where it is being accessed.

The Multikernel [Bau+09a] used in the Barrelfish operating system manages physical resources using a distributed, partitioned capability system [Ger18]. Similar to seL4 [Kle+09], Barrelfish's capabilities live in their own "address space", the CSPACE, which is partitioned across the different operating system nodes. Certain capability operations (e.g retype or revocation) may require agreement of all operating system nodes and thus require a distributed protocol [Ger18]. The capability system relies on physical addresses for comparing capabilities with each other.

M3 is a microkernel-based system for heterogeneous many-cores [Asm+16]. It uses capabilities to manage physical resources of a machine. M3 runs on a network-on-chip architecture where each node in the network has a data transfer unit (DTU) which offers message-passing and memory access functionality. The DTU is the abstraction to access other cores and memory in the system. SemperOS [Hil+19] extends M3 to support a large number of heterogeneous cores by a hardware/software co-designed distributed capability system.

Helios [Nig+09] targets heterogeneous platforms. It deploys satellite kernels consisting of scheduler, memory and namespace manager, and communication modules. Every NUMA domain runs its own satellite kernel. There is a single, unified namespace where services advertise. Using an affinity metric, applications can hint the operating system where it is best to run (e.g. co-locating with a used service). Helios limits applications to a single NUMA node and strictly uses local memory.

3.3.5 Hypervisors and Simulators

Virtual machines (e.g. Xen [Bar+03], Linux KVM/Qemu [lin19], Microsoft Hyper-V [Mic18], Oracle VirtualBox [Ora19] or VMware ESX [VMw19]) provide a configurable, virtual hardware platform to the guest operating system which can run unmodified. Some hardware elements have support for virtualization acceleration e.g. processors have virtualization extensions and nested paging, and PCI Express has SR-IOV which provides virtual device functions. Arrakis [Pet+14] uses this feature to safely export devices and memory management to applications. Virtual machines also support emulating a different platform than the host machine e.g. running Arm binaries in Qemu on an x86 host. The hypervisor provides a flat, uniform guest physical address space to system software running in the virtual machine. This effectively hides details about the underlying physical machine by providing an abstract, uniform memory topology to the guest. Architectural simulators e.g Gem5 [Bin+11] and the Arm FastModels [ARM19b] allow the implementation of complex memory topologies, translation features and hardware components. Section 7.7 uses the Arm FastModels to simulate a system with heterogeneous views of memory.

3.3.6 Early Capability-Based Operating Systems

Hydra [Wul+74] manages physical and virtual resources based on a notion of objects as the abstraction and unit of protection of resources. Objects are referred to by capabilities. Hydra runs on the Carnegie-Mellon Multi-Mini-Processor, where each processor has a small amount of private memory and relocation hardware translating a virtual address to a physical address. Hydra separated protection (mechanism) from security (policy), this concept policy-mechanism separation is adapted from Nucleus [Han70].

The Cambridge CAP computer [NW77] uses capabilities to provide fine-grained access control to programs executed on the CAP. Every program should only ever have access to data it needs to correct functioning, and nothing more. Memory segments (contiguous set of memory locations) on

the CAP have two types: either data or capability. The processor contains a capability unit, which holds 64 evaluated capabilities and the addresses of their memory segment. Applications can use up to 16 capability segments. Applications address memory by specifying the triple: capability segment, index within the capability segment, and word offset into the segment referred to by the capability at then index. This triple is also called the complete CAP virtual address [Lev84]. Ultimately, the CAP virtual address gets translated to a physical address.

Built for the CM* architecture, StarOS [Jon+79] uses typed, distinct and unique objects to manage information in the system. Access to those objects is mediated and protected by capabilities naming the objects and specifying the authority on the object. The capability contains a 16 bit name which identifies a 16 byte descriptor. Based on the descriptor, a 18 bit address is calculated, which is local to the processor specified by the corresponding field in the descriptor. Objects can be mapped into a 4 KiB window of the immediate address space of a process. Processor memory references are routed through a switch and either forwarded to a local memory of the module, or to the map bus and further of another module. Memory is effectively distributed across the cluster, appearing as a single large memory.

Accent[RR81] is a network operating system kernel which uses message passing as its core abstraction. Messages are sent and received on ports which are indirectly referred to using capabilities. Memory is allocated using the notion of segments and mapped into an address space using the `ReadSegment` RPC the reply of which contains newly allocated pages which are mapped upon the reception of the message. Accent assumes a per-node flat physical address space.

KeyKOS [Har85; Bom+92] is a capability-based operating system designed to run on the IBM System/370. The ‘page’ is the simplest KeyKOS object referring to a location on a persistent storage device. A page object can be mapped into a process’ address space or a segment which is a collection of pages or other segments. Segments form address spaces. The implementation of segments in KeyKOS is based on a tree of nodes with

pages as leaves resembling hardware page-tables.

IBM System/38's capability system had "resolved address" registers which cached the resolved physical address corresponding to a particular capability [Lev84; HSH81].

3.3.7 Other Operating Systems

The Mondrian memory protection system [WCA02] splits a single address space into multiple protection domains. Mondrian provides segmentation like protection semantics by checking virtual addresses against the access rights defined in permission tables residing in main memory with a protection lookaside buffer as a cache. This is similar to single-address-space operating systems such as Opal [Kil+62; Cha+92; Cha+94] separating translation from protection. Mondrian supports protecting variable ranges and works on top of normal virtual memory without tagging pointers or other ISA capabilities. Mondrix [WRA05] applied the Mondrian memory protection inside the Linux kernel where each kernel module gets its own protection domain.

CheriBSD is a variant of the BSD operating system adapted to run on a CHERI-enabled processor [Woo+14] with a corresponding POSIX C runtime environment [Dav+19]. CHERI is a capability-based addressing extension to the instruction set architecture offering byte-granularity protection of data structures. Capabilities are implemented using fat pointers encoding base and length that can be addressed with it. CHERI capabilities enforce protection on top of transitional virtual memory systems.

The hardware model used in LegoOS [Sha+18] consists of dis-aggregated, network-attached hardware components (e.g. CPU, RAM or storage) that separate processor and memory functionalities. To manage such dis-aggregated systems, LegoOS deploys a Splitkernel which splits operating system functionality among the hardware components each running a Splitkernel monitor. LegoOS uses virtual addresses in the caching hierarchy. Only the dedicated memory components contain translation units.

GMM [Mil+00] splits memory management into local resources and a global address space which is distributed among all the nodes of the cluster. Each node is effectively capable of accessing parts of the other nodes' dedicated memory over an interconnect. In GMM, the size of the global address space exceeded the size of the virtual address spaces of the nodes.

Global Memory Service [Fee+95] unifies memory management of a workstation cluster in a distributed way. The system fetches memory pages over an ATM network and identifies nodes with their IP address.

3.3.8 Operating Systems Summary

Operating systems, from monolithic architectures over single address space operating systems to verified kernels use different ways to manage resources and enforce isolation between processes. This involves naming the resource or object (e.g. using its virtual address, physical frame number, or file name), and enforcing access control and protection (e.g. access control lists or capabilities). However, this relies on a stable concept of a physical address space and does not take multi-stage translation schemes into account. Lastly, virtual machines and simulators expose a certain machine topology to the guest operating system, the problem of managing resources and translation still exists at the hypervisor layer.

3.4 Runtime Systems and Programming Languages

The performance of large-scale workloads highly depends on data allocation and scheduling policies. Applications or libraries adapt their memory allocation and working set sizes based on the sizes of memory and caches, as well as their memory access patterns [Gau+15; Cal+17; Kae+15; Kae+16]. This requires a model of the machine or cluster they are running on. This section presents a survey of different libraries and

operating system extensions which tune memory allocation decisions to the platform.

3.4.1 Memory Topology Models

Big-memory workloads running on large machines consisting of multiple processor sockets, each having a portion of DRAM attached, experience non-uniform memory access (NUMA) effects. Memory accesses to local memory resources inhibit lower latency and higher bandwidth than accessing memory attached to another processor. Optimal allocation of data structures in such a system is a challenge [Gau+15] and depends on not only the system NUMA topology, but also on the algorithm and the resulting access pattern [Kae+15].

Topology information obtained through mechanisms outlined in Section 3.1 is used to initialize the machine topology models of schedulers, memory allocators and other runtime services used by memory intensive workloads to allocate, run and tailor workload sizes and data structures to the concrete runtime system e.g. Smelt [Kae+15] or use micro-benchmark to augment the topology with performance numbers e.g. Smelt [Kae+16].

Operating systems like Linux offer different memory allocation policies, which can be controlled on a per application basis. Libnuma [Kle08] provides application with an interface to allocate memory form a particular memory node. DVMT [Ala+17] allows applications to explicitly request physical frames with specific properties from the operating system. The system enables applications to tailor the virtual-to-physical mappings to their needs by registering a TLB miss handler for the special DVMT range. Likewise, microkernel operating systems such as Barrelfish [Bau+09a; Ger18] and Arrakis [Pet+14] let applications manage physical memory and their address space directly. This is yet again an example of policy-mechanism separation where user-space implements policy decision using mechanisms exposed and enforced by the kernel.

Systems like Carrefour [Das+13] or Linux's AutoNUMA [Red18] monitor memory accesses and try to optimize the placement of data structures

by migrating data between memory nodes, while others migrate hot and cold data pages between fast and slow memories, e.g. Thermostat [AW17]. Shoal [Kae+15] replicates data structures across NUMA sockets and black-box concurrent data structures has support for writable, replicated data structures [Cal+17]. MPI+OpenMP [Mah+12] runtimes use message passing between processor sockets and the OpenMP framework to parallelize computation within processor sockets. Those systems rely on firmware provided topology information (Section 3.1) and operating systems support to allocate memory from particular memory nodes.

Hybrid or heterogeneous memory consists of multiple types of memory with different characteristics e.g. fast and slow memory. The programmer or runtime system must make explicit data placement decisions for optimal performance [SLL16]. Data intensive workloads are sensitive to allocation policies in heterogeneous memory systems. CoMerge [DG17] prefers a sharing over a partitioning technique for co-located applications.

Rack-scale systems consisting of a collection of machines which are connected with a high-speed network which offers direct access to remote memory using RDMA [Rec+07]. FARM [Dra+14] provides a shared address space abstraction of the cluster's memory offering an interface to read, write, allocate and free objects. RDMA can be used to distribute in-memory join [Bar+15b] or data shuffling [LYB17] operators of databases across the cluster by using networking primitives offered by the network.

Scale-out NUMA [Nov+14] proposes the use of a remote memory controller integrated into the local coherence hierarchy offering a load/store interface to remote memory. Scale-out ccNUMA [Gav+18] introduces a caching layer at each node to mitigate skewed memory accesses.

The Message Passing Interface (MPI [Wal92]) is a widely used standard for parallel applications running on high-performance clusters. MPI works well for parallel workloads over multiple machines. Combining MPI with OpenMP exploits inter and intra machine parallelism [CE00; Mir+17] following the cluster topology.

In summary, the libraries and operating system extensions described in

this section try to allocate memory with certain properties (e.g. proximity to a processing unit) or access remote memory directly to write data structures to be used in the next phase of the algorithm. Central to this approach is a good representation of the system topology with its memory resources, processing units and interconnects. Current APIs use coarse grained information such as machines or memory nodes, and do not take memory channels or interconnect links into account.

3.4.2 Cache Topology Models

The memory topology does not stop at the level of memory nodes. Processors consist of a collection of cores, interconnected with a certain network topology (e.g. a ring bus or mesh), and a hierarchy of different caches of various sizes. Tools like LIKWID [THW10] provide detailed information of the cache hierarchy to applications. Despite being on the same processor, memory accesses or communication between two cores can be non-uniform and different for each new architecture. Augmentation of the topology information with actual measurements provide a good basis for allocation and scheduling decisions [Kae+16].

The size of data caches and TLBs influence the design of algorithms. Techniques such as processing data in blocks [LRW91] limit the working set size to the size of the cache to reduce cache misses. Hash joins in databases are a prime example, where the hash table size is chosen with respect to the cache size [BLP11].

Total size of the cache is important, but so is the optimization with single cache lines in mind. For instance, Masstree [MKM12] carefully layouts data structures to cache-lines and uses prefetching to populate the processor cache with relevant data strategically to optimize performance.

Scaling cache-coherence to large machines is hard [FNW15]. The result is a collection of coherency domains. This requires careful data placement and cache management to ensure correctness. The same applies to programming translation hardware or DMA-capable devices where cache

flushes ensure the device or translation unit see up-to-date data, or the data does not get overwritten by a cache-line eviction. Cache-snooping [PCI17] or direct cache access [HIT05] enable certain devices to directly read/write processor cache contents.

While caches are not a core focus of this dissertation, their presence, related memory resources, and management are an important factor in the performance and correctness of applications and system software.

3.4.3 Co-Processor Offloading

Different software frameworks provide specific ad-hoc point solutions targeting their particular use-case. Offloading computations to GPUs is a prime example: OpenGL [Khr18] or CUDA [NVI19] provide runtime support for allocating and accessing data structures on the host or the GPU, VAST [LSM14] uses compiler support to dynamically copy memory to and from the GPU and Mosaic [Aus+17] provides support for translating addresses in a shared, virtual address space with multiple page sizes.

The nVidia CUDA framework [NVI19] distinguishes between host and device memories. Memory is allocated on the host or the device explicitly. The CUDA runtime abstracts memory references using pointers, where a *host pointer* can be converted explicitly to a *device pointer* using `cudaHostGetDevicePointer` which takes a valid host pointer and returns a device pointer, or indicates failure . Unified memory (UM [NVI13]) sets up a managed memory space. This eliminates the need for explicit allocation and copy. The goal is to provide the same view of memory resources to all processing cores. This requires that each processor must have its own dedicated translation unit. Similarly, Shared Virtual Memory (SVM) as implemented by HSA [Rog16; HHG16] exposes a single, virtual address space to the host and to the devices. This enables using of pointer-based data structures. It does so by extending parts of the global memory into the host address space, and by definition, devices have access to global memory and hence the host address space. To some extent, this provides

similar functionality as with single-address-space operating systems where the virtual address uniquely identifies a memory object.

The OpenCL specification [Khr18] defines three named address spaces: private, local and global. OpenCL programs use pointers to refer to memory where a pointer on the host is not necessarily valid on the device. Pointers can be converted between the address spaces, but may not be valid between devices. The OpenCL runtime abstracts physical resource management. Memory regions (or pools) define a distinct, logical address space. Memory regions may overlap in physical memory. Global and constant memory is accessible by all kernels, where constant memory remains constant during the execution of a kernel-instance. Local memory can be shared between work-items of a particular work group, where as private memory is valid within a single work-item only.

3.4.4 Programming Languages

High-level programming languages such as Java and C# have well-defined memory models [MPA05; Dem+13] which define the semantics of variable and data structures accesses. In the case of Java, programs are compiled into an intermediate byte code representation which is the run in the Java virtual machine (JVM). Java programs are architecture independent as the JVM interprets the byte code and interfaces with the underlying operating system to provide memory resources to the application. Hera-JVM [MS09] hides the details of a heterogeneous platform in a virtual machine abstraction.

The interaction of the language-specific memory models and the underlying hardware memory models are of particular interest [ZP16; Sar+12] especially in highly parallel workloads where locks, barriers and fences are essential building blocks of programs. However, those models do not go beyond operation reordering and the required fences and hence low-level memory access routing is ignored and left for the operating system to be taken care of.

3.4.5 Runtimes and Programming Languages Summary

Understanding the memory topology, including locality and cache information, is important for application performance. Runtime libraries and programming languages provide an environment with certain semantics. Those environments are point solutions, targeting a specific use case (e.g. GPU offloading). Overall, there is an implicit notion of address spaces (e.g. allocate memory on the GPU or a specific memory node). However, memory is either assumed to be accessible at the same address, or copied in the background from and to the GPU.

3.5 Summary

The related work presented in this chapter showed that there exist various sources of hardware topology information which is used by libraries and operating system in resource allocation and scheduling decision. Runtime libraries such as libnuma provide an API to applications to request memory from a particular memory node, or even a specific machine in a cluster.

In all cases, the physical address is used as an identifier of memory resources within a machine and all memory resources are in a single, uniform and global physical address space. Virtual machines abstract the underlying machine topology to provide the exact same single-address space representation. Other formal specifications of processors and memory models focus on the semantics of instructions and possible reordering of memory accesses in the various buffers of a machine, and less about memory address translation which is what this thesis is focusing on.

4

A Formal Model for Memory Addressing

This chapter presents a new abstraction to faithfully represent memory-address decoding of modern hardware and the formalization there of: the *Decoding Net* model [Ach+17b; Ach+18] a model specification of address translation in Isabelle/HOL. The formal model is able to express the complex addressing schemes implemented by hardware designers of the different real platforms and System-on-Chip (SoC) from Chapter 2.

To show the applicability and versatility of the *Decoding Net*, the model is then further *refined* to express the translation behavior of a software-loaded, fully associative translation lookaside buffer (TLB) in great detail at the example of the MIPS R4600 TLB [ITD95].

The main objective of the work presented in this chapter is to describe a new model to faithfully capture memory-address translation characteristics of modern hardware. Formal specification of the address decoding semantics provides a sound basis for accurate system descriptions and reasoning about memory address translation in a system. This chapter consists of the following parts:

1. Section 4.2 describes the address-space model in the context of a naming problem.
2. Section 4.3 then defines the *Decoding Net*, a formal specification of the address spaces and their semantics in Isabelle/HOL.
3. Section 4.4 expresses the *current* state of an example system using the *Decoding Net* model.
4. Section 4.5 describes and specifies algorithms on top of the *Decoding Net* to perform *view preserving* transformations.
5. Section 4.6 shows how real translation hardware can be expressed with address spaces using seL4-style *refinement* methodology.

4.1 Motivation

Modern computing platforms, from high-end servers to smartphone SoCs are highly complex and diverse (Chapter 2). Those systems are effectively a heterogeneous collection of cores, interconnects, programmable memory translation units, and devices – abstractly a network of physical address spaces. Moreover, each processor core or DMA-capable device may have a different *view* of the system requiring different (local) physical addresses to be used for memory accesses.

Correct operation of system software inherently relies on the correct resource management and configuration of these interconnects and translation units. For this, operating systems must be able to

1. unambiguously name physical resources of the system,
2. correctly resolve the names of physical resources to a local address in any address space.

The second point is important to derive the local address to be issued to access the physical resource, and how to configure the local translation unit which needs the local address the resource appears in its address space.

However, current operating systems, including formally verified or certified projects such as seL4 [Kle+09] or CertiKOS [Gu+16], use a simplified, flat representation of the memory subsystem and assume a globally uniform physical address space which does not hold as shown in Chapter 3.

It is the operating system's job to correctly enumerate and name physical resources, as well as initialize and configure all address translation units and protection modules with the right values. This requires an unambiguous representation of the memory system topology. This task mostly involves reading technical manuals from hardware vendors which are often thousands of pages of plain English prose describing the hardware platform. These descriptions are not always precise enough and leave room for ambiguity [Arc19].

DeviceTrees [dev17] are the state of the art of machine-readable platform descriptions. They are widely used by the Linux kernel to describe devices, processors, memory and interrupt resources of a platform. The DeviceTree files are compiled into a binary file format, which the Linux kernel parses during boot to initialize certain fields of data structures. DeviceTrees are specified in a textual format, which is then compiled into a binary representation expected by the Linux kernel [Lin19b]. This compilation process removes any topology information from the representation. Often, accurate topology information is not present in the first place. For instance, the DeviceTree files for the Texas Instruments OMAP4 SoC [Tex14] and other platforms used in the Linux kernel acknowledge the complexity of the SoC network:

> *"XXX: Use a flat representation of the OMAP4 interconnect. The real OMAP interconnect network is quite complex. [...] Since it will not bring real advantage to represent that in DT for the moment, just use a fake OCP bus entry to represent the whole bus hierarchy."* – Linux Kernel, omap4.dtsi

Consequently, DeviceTrees are not capable of expressing the presence of multiple physical address spaces and different views as described in the previous chapter. Moreover, the lack of rich semantics and this narrow focus render DeviceTrees unusable for later reasoning about correctness.

Ultimately, to be able to make correct statements about the memory subsystem of a platform, system-software developers need a faithful and sound representation of the memory topology with well-defined semantics. This includes a mechanism to unambiguously name the physical resources of a platform and to derive the local address of any resource in any address space (if applicable).

4.2 An Accurate Model for Memory Address Translation

The previous section provides evidence that the hardware abstraction model of a single, uniform physical address space used in operating systems contradicts the reality of different hardware platforms, from server-class systems to phone SoCs. The physical address as seen from a core does not uniquely *name* a resource.

This section presents the foundations of new representation of address decoding on modern platforms [Ger+15]. The goal of the model proposed in this section is to provide:

1. A way to represent and express what a given physical address actually means in a machine.

2. A mechanism to unambiguously decode an address on any processor core or DMA capable device.

3. A logically sound foundation enabling the design and implementation of physical memory management systems for modern machines.

In other words, the model should be able to describe the resources and the *topology* of a platform. DeviceTrees [dev17] essentially provide similar functionality: they list the (undiscoverable) resources of a system such as the number of cores and devices including the addresses of their registers for SoC platforms. However, resembling a file format, the DeviceTree specification does not accurately represent the platform topology and as such is not able to capture and express different *views* of from distinct cores. Instead, the model needs to focus on how hardware decodes addresses within a system including non-hierarchical relationships.

4.2.1 Formulation as a Naming Problem

For system software to perform resource management tasks correctly, it is essential to have an unambiguous way to refer to the physical resources of a platform.

This is an instance of a classic naming problem as described in Saltzer's seminal work *Naming and Binding of Objects* [Sal78]. The paper presents two case studies on memory addressing in the Multics operating system [CV65] and on file systems. In summary, it is central to all naming problems to carefully define the *context* in which *names* are resolved. In the problem at hand, a name in Saltzer's terminology corresponds to addresses and the context is the address space within the addresses are resolved.

4.2.1.1 Terminology

- *Core.* An entity that is capable of issuing memory requests. For instance, processor core or DMA-capable devices.

- *Physical Resource.* Addressable memory cell or device register.
- *Address.* An offset into some address space.
- *Name.* An address qualified with its address space context.
- *Canonical name.* The unambiguous name of a resource. Resolving a canonical name yields the same canonical name. A resource can have many names, but has precisely one canonical name.
- *Virtual address.* An address used by software and issued by a processor, or a device. The virtual address is valid within the context of the process or device.
- *Local physical address.* In the context of a processor, the result of a virtual address translation. The emphasis is on *local*. The address is valid within the context of the processor.

4.2.1.2 The Address Space Context

The identification of every possible, distinct naming context in the system forms the starting point of writing down the addressing model. An *address space* forms the naming context within an address is resolved. Each address space is characterized by

1. a unique address space identifier (ASID),
2. a set of known address values, and
3. a function from addresses to referents.

The set of know addresses is typically a range. The number of bits that form an address typically defines this range: $[0, 2^b)$ with b being the bit width of an address. A referent is either: *i)* memory contents that are local to this address space and hence the *only* place where they can be directly accessed, or *ii)* a handle to a new address in possibly another address space. Note, that this does not rule out that the referent is a different address in the *same* address space. Figure 4.1 illustrates two address space contexts.

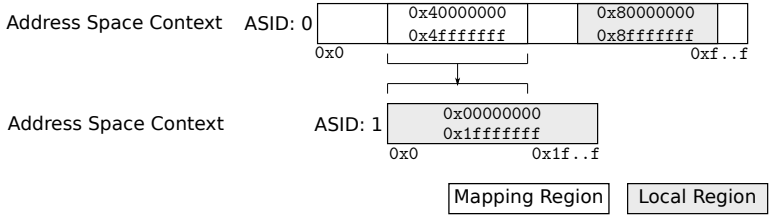


Figure 4.1: Illustration of Address Space Contexts.

4.2.1.3 Regions and Translation

An address space can be divided into a set of non-overlapping, contiguous *regions*. Each region is either:

- *local* the region only contains local referents to RAM or memory-mapped registers, or
- *mapping* the region is a function that maps addresses within the region's range to an address in the target address space context.

Regions are not mixed in the sense that they contain referents to local resources and mappings at the same time. A mapping region is effectively an "aperture" or a window into an address space. The region's translation function defines what can be seen through the aperture.

Examples for this translation function include *i)* the identity mapping, where an address range of a region is mapped one-to-one to the same address range in the target address space, *ii)* static transformations, such as adding a base address, shifting or pre-pending bits, or *iii)* dynamic translations defined by in-memory data structures (e.g. page tables) or hardware registers (e.g. segment registers).

4.2.1.4 Locality of Physical Resources

Physical resources, such as DRAM cells or memory-mapped hardware registers only exist *once* i.e. are a local region of one address space

(Invariant I4.1). At first glance, it may seem that there are resources which appears at multiple locations (regions of address spaces). However, those locations are not the true, canonical location of the resource. In fact, they are mappings to the canonical location of the resource, which exists exactly once.

Invariant I4.1 (Resource Locality) Each physical resource is local to exactly one address space.

Consequently, if a resource seems to be local in multiple address spaces, then apply Algorithm 1, which

1. Creates a new address space where the resource is local.
2. Replaces all other "local" occurrences of the resource to mappings to the region in the newly created address space.

The resource is now local to exactly one address space which satisfies Invariant I4.1.

4.2.1.5 Summary

This section defined the naming problem of address decoding. Each physical address is always relative to an address space context. Each region either resolves to a unique, real local resource, or is a mapping to a new address in some address space context. A resource is local to exactly one address space. Algorithm 1 corrects violations of Invariant I4.1.

4.2.2 Address Resolution and Memory Accesses

In modern systems, there are many agents that can issue memory accesses: processor cores execute load/store instructions, PCI transactions either

```
1  def MakeLocal(r, A):
       Data: r :: resource local to multiple address spaces,
             A :: set of address spaces
       Result: Transformed set of address spaces
2      a_l ⟵ NewAddressSpace {r};
3      A' ⟵ {a_l};
4      foreach a ∈ A do
5          if isLocal r a then
6              a' ⟵ ReplaceLocalWithMap a (r, a_l);
7          else
8              a' ⟵ a;
9          A' ⟵ A' ∪ a';
10     return A'
```

Algorithm 1: Making a Resource Local to Exactly one Address Space.

initiated from a device or the host processor, a DMA bus cycle issued by a device, or even a cache in response to a coherency protocol message.

The term *core* is used to refer to anything that is capable of issuing a memory access. Each core then has its local address space which defines what resources the core "sees" and under which addresses they appear. Each core can emit loads and stores to addresses within its local address space only. Note, even hardware threads on the same physical processor core (e.g. hyperthreading) effectively have different address spaces, as each of them has its own, memory-mapped local interrupt controller and, depending on the configuration, cache partition.

Address resolution starts at the local address space of a core. In each resolution step, the address is resolved within an address space. This can result in three possible outcomes:

1. The address is within a local region of the address space and resolution terminates.

2. The address is within a mapping region. The resolution process applies the translate-function of the address space and recurses by resolving the new address in the next address space.

3. The address is neither in a mapped/local region. This is a fault and terminates resolution.

Note, this procedure is not guaranteed to terminate: address resolution can end up in the same address space multiple times (as illustrated in the examples Section 2.2.2.6 or Figure 2.22), creating routing loops. However, this is not a problem in general: It is perfectly possible that this is fine when, for instance, the address is different each time, but may result in a bus error in the case of a true loop.

4.2.3 System Representation

The state of a complete system consists of a collection of cores and address spaces where each core has its own, local address space. In this model, only cores can issue memory accesses relative to their local address space. Recall, a *core* can refer to a processor or a DMA-capable device. Figure 4.2 illustrates an example system with three cores (two processors and a device) and four address spaces and the corresponding local and mapped regions.

One aspect to emphasize here is that there may exists resources that cannot be directly addressed by all cores of the system. For instance, the DRAM of Figure 4.2 is always accessed through a mapping, and parts of it are not accessible from the DMA device. Likewise, not all address spaces actually have local resources, and some of which do not have mappings. The novelty of this model is to express intermediate translation steps explicitly and cleanly using the address space abstraction.

Figure 4.2 shows the decomposition of a 64-bit x86 system with two processor cores, some RAM and a network card which is capable of issuing DMA transactions, but is only capable of emitting 32-bit addresses. One *possible* way to express this system in the address space model is as follows:

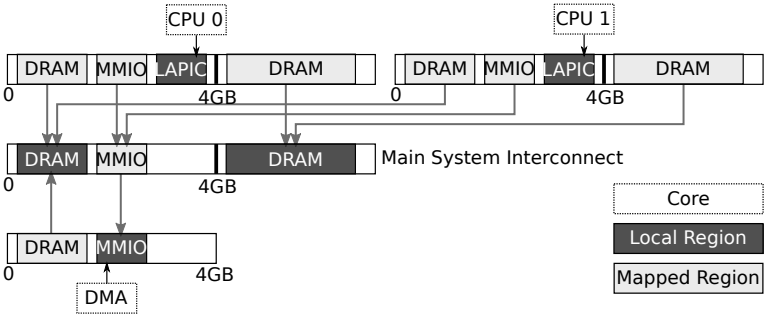


Figure 4.2: The Address Spaces in a Simplified 64-bit PC with Two Processors and a 32-bit DMA Device.

In total there are three cores (one for each processor core and one for the network card) and four address spaces (one for each core and the main system interconnect). There are two RAM regions, one in low memory (below 4GB) and one in high memory (above 4GB). Both RAM regions are local to the main system interconnect and identity mapped into the three core's address spaces. Note, the NIC supports only 32-bit addresses and therefore the NIC cannot access the high RAM region. Consequently, the RAM region above 4GB is not mapped into the network card's address space. The network card has memory-mapped registers, which are local to the NIC's address space and mapped into the system interconnect. Finally, the address spaces of the processor cores are basically an overlay of the system interconnect, but each of them include a local region for the local APIC registers. Core-local processor caches are omitted.

Note, Figure 4.2 shows *one possible* instance how to express this system in the model, an alternative representation is shown in Figure 4.3. The difference is the level of detail the memory controller is represented. Whereas in Figure 4.2, there is just DRAM, Figure 4.3 highlights which regions map to which memory channel. However, this additional information of memory channels can be useful in allocation decisions to balance the load

between the two channels. In contrast, Figure 4.2 collapses the memory controller reducing the number of resolution steps. Section 4.5.3 describes a transformation algorithm to show equivalence of the two representations, and Section 6.3 the application thereof.

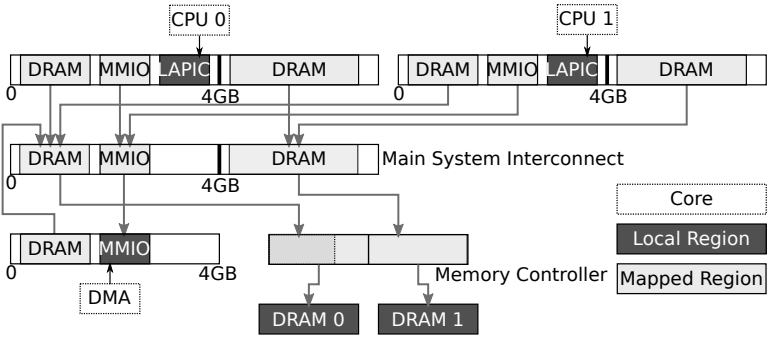


Figure 4.3: Alternative Representation of Figure 4.2 Highlighting Two Memory Channels.

This rather small example of a system nicely demonstrates the complexity of memory addressing in modern systems. The system shown in Figure 2.22 is also a simplified example of the model, where several address spaces like SMT threads or the 228 distinct address spaces of the $57 \times 4 = 228$ Xeon Phi hardware threads are omitted.

4.2.4 Discussion

Following the modeling principle of memory resources being directly accessible by and thus local to a single, unique address space (Invariant I4.1). Algorithm 1 provides a mechanism to introduce new address spaces and fix violations of the locality Invariant I4.1. Any system can be decomposed into a set of address spaces which *i)* have memory resources and *ii)* translate

addresses between themselves. Using this principle, each memory resource can be uniquely named by providing address space context and the address at which the resource appears within this context. Moreover, the "core" abstraction allows the identification of possible actors in the system.

Address spaces and cores together form a system representation which has a well-defined translation behavior that enables reasoning about static or dynamic configuration.

Despite an increasing total number of (conceptual) address spaces, any real system does not result in an exploding number of address spaces. Hardware components and modules such as hardware threads, interconnects or memory controllers typically introduce only one or two address spaces to the system, and the total number of hardware components is finite. In a large system with many devices this results in hundreds of address spaces and a number of hops between the cores and local resources below ten. This property is an important aspect for designing algorithms later on.

4.3 Model Definition

The previous section described memory address decoding as a naming problem. It further gives an *intuition* of the complexity and semantics of address resolution. However, this does not quite provide a rigorous and sound basis to express the properties and semantics of memory address decoding, a property which is desirable for correctness.

This section introduces the *Decoding Net* model [Ach+17b; Ach+18], a formalization of the address space sketch of the previous section. The formal model is capable of expressing any complex memory addressing and translation schemes of modern hardware ranging from SoCs to desktops and high-end server-class machines with custom-built hardware components.

The *Decoding Net* is specified in Isabelle/HOL.

4.3.1 Design Goals

The design goals of the formal specification of the model are as follows:

1. Express the memory subsystem of real hardware.
2. Provide an unambiguous interpretation of memory accesses.
3. Avoid pre-mature simplifications or abstractions.
4. Form a sound basis for system software verification.

Overall, the model should enable easier code engineering for real operating systems with high confidence of interoperability across different platforms. This should include system software code generation at compile time and using the model at runtime.

4.3.2 Address Space Definition

The first core definition of the model is the *qualified name* which is an address in the context of an address space i.e. its *namespace*. The address space effectively specifies where the decoding of the address starts. Each address space is uniquely identified by a natural number, the address space identifier (or ASID for short). Moreover, addresses are discrete, non-negative integers and hence are also a natural numbers. Formally, a *name* is a tuple of two natural numbers identifying the address space context and the address within.

$$name = \mathbb{N} \times \mathbb{N}$$

A memory translation unit, for example, defines an address space which is identified by n. It *translates* (or maps) names (addresses qualified with by the address space identifier n) to another address within an address space n' (e.g. the local physical address space of the processor). Generally, a name can potentially be translated to any other name including the name

itself or to no name at all. In practice, for each address space there is a set of address spaces that can be reached from it.

Formally, the translation behavior of a fixed name is given by the function:

$$\texttt{translate} : \mathbb{N} \times \mathbb{N} \rightarrow \{\mathbb{N} \times \mathbb{N}\}$$

The function maps a fully-qualified name (n, a) (i.e. an address a within an address space context n), to a set of names $\{(n', a')\}$ where address a' is in address space context n'. If there is no translation, the empty set is returned. Thus, the `translate` function is fully defined. There is no restriction on the set of possible translations.

Note, the fact that the `translate` function returns a *set* of addresses instead of a single address is required to express non-determinism (e.g. returning the universal set). This is important to express undefined behavior as required in Section 4.6. Otherwise, this would be an early simplification ruling out particular hardware designs e.g. hardware-based broadcast and multicast. Additionally, this allows using the very same model to express interrupt routing [Hum+17; Ach+17b] which makes use of broadcast and multicast of interrupts. These aspects fall out of the scope for this thesis.

The memory request should eventually *terminate* somewhere. There are two possible cases that terminate address resolution:

1. The address resolution process reaches, for example, a DRAM cell or device register.
2. Address resolution stops as a result of *undefined* behavior such as if the name is neither terminating nor translating.

Those two cases must be distinguishable: A name (n, a) is terminated at an address space n if a is in the *accept set* of the address space n:

$$\texttt{accept} : \mathbb{N} \rightarrow \{\mathbb{N}\}$$

Name resolution terminates with undefined behavior if the name does neither translate nor is accepted:

$$\texttt{undefined}\ (n, a) \leftrightarrow a \notin \texttt{accept}\ n \wedge \texttt{translate}\ (n, a) = \emptyset$$

An address space has therefore two properties: a set of addresses that are accepted by this address space and a translation function that translates local addresses to names. This is expressed as a `node` record in Isabelle/HOL. Here, nodes and address spaces are used interchangeably.

$$\begin{aligned} &\textbf{record}\ node = \\ &\quad accept\ ::\ addr\ set \\ &\quad translate\ ::\ addr \rightarrow name\ set \end{aligned}$$

Note, accept and translate are not exclusive. A node can have non-empty accept sets *and* non-empty translate sets. This is important to express behavior present in caches: some addresses are locally accepted by the cache itself (cache hit) while other addresses are forwarded to another cache or DRAM (cache miss). Expressing caches formally is out of the scope of this thesis. Another example is the Intel Xeon Phi presented in Section 2.1: the co-processor can be expressed as an address space where one part accepts addresses (corresponding to the GDDR memory) and another part is translated onto the IOMMU address space (corresponding to the system memory page table region.)

4.3.3 Decoding Net Definition

The *Decoding Net* is an association of address space identifiers (or node identifiers) to nodes. The function `net` returns the corresponding node record for a given node identifier.

$$\texttt{net} : \mathbb{N} \to \mathit{node}$$

The two primitives *accept* and *translate* together with the *Decoding Net* definition are sufficient to define the core semantics of the model. Other properties can be derived from them.

Moreover, one possible interpretation of the *Decoding Net* is a directed graph which is not guaranteed to be cycle free i.e. it is not a directed, acyclic graph (DAG). In this graph, the set of well-defined paths is given by applying the `translate` function repeatedly until it eventually ends up in an accepting set of a node (see resolution below). In other words, any well-defined path through the *Decoding Net* ends up in the *accepted names* set which is the union of all accept sets of all nodes:

$$\texttt{accepted\_names} = \{\ (n, a) \mid a \in \texttt{accept}\ n\ \}$$

The *one-step* `decodes_to` relation can be constructed by associating the all possible inputs of the `translate` function with the obtained output. This relation effectively encodes the edges of a directed graph. If this resulting graph is a directed, acyclic graph (DAG) then the decoding of all names is well-defined i.e. there are no decoding loops.

$$\texttt{decodes\_to} = \{\ ((n, a), (n', a')) \mid (n', a') \in \texttt{translate}\ (n, a)\ \}$$

Similarly, the `decodes_to` relation can be expressed using the *Decoding Net* and the node representation. Two names $(n, a), (n', a')$ are in the `decode` relation of a *Decoding Net*, if the name (n', a') is in the result set of the `translate` function of node n evaluated with address a. (This is the duality between the relational and functional specification).

$$((n, a), (n', a')) \in \texttt{decode net} \iff (n', a') \in \texttt{translate}\ (\texttt{net}\ n), a$$

The core *Decoding Net* model is able to capture and express the *static* state of the system at a fixed point in time. The subsequent chapters of this thesis will extend this to include the dynamic aspects such as configurable translations and authority.

4.3.4 Address Resolution

The `decodes` relation defined above only encodes a one-step translation. In general, the diameter of the *Decoding Net* graph is not guaranteed to be one. In fact, it is likely to be larger than one: each node in the graph corresponds to a single translation step and Chapter 2 showed that platforms consisting of multiple, fixed or configurable translation steps are not uncommon. Therefore, resolving a name requires decoding it repeatedly.

The `resolve` function returns all possible names where a given input name $n = (nd, a)$ (i.e. address a in address space nd) might be accepted. Note, in the equation below, n and n' are names i.e. node-address tuples. For memory translation this might just be the empty set or a singleton set, whereas for hardware with broadcast support (e.g. interrupts) this might as well be a set with multiple names.

$$\texttt{resolve}\ n = (\{n\} \cap \texttt{accepted\_names}) \cup \bigcup_{(n,n') \in \texttt{decodes\_to}} \texttt{resolve}\ n'$$

The `resolve` function works as follows: If the input name is already part of the accepting names set, then resolution terminates, and the name is added to the result set. Note, that this definition allows cases where accepting names are further translated. This is important to represent caches.

Otherwise, take the recursion step: for all relations where the current input name (n, a) decodes to, take the union of all recursion results starting from the name obtained by one-step decoding.

Note, the termination of the `resolve` function is not guaranteed, which is the case in the presence of loops. See section Section 4.5 for necessary and sufficient termination conditions. In this case, the resolution function is not defined for this name.

The function definition mechanism of Isabelle/HOL defines a predicate (`resolve_dom`) of the domain of incomplete functions (such as `resolve`). It asserts that the function is well-formed for the input arguments of its domain. Under the condition of the domain predicate, the `resolve` function for a name (n, a) is the intersection of the `accepted_names` of the *Decoding Net* with the transitive closure of the `decodes_to` relation.

assumes `resolve_dom` (n, a)
shows `resolve` $(n, a) =$ `accepted_names` $\cap$ (`decodes_to`$^*(n, a)$)

4.4 System Descriptions and Syntax

To achieve the design goals stated in the previous section, it is necessary to write down the hardware configuration of a system in terms of *Decoding Net* nodes which corresponds to the address spaces on the platform. Writing down every single address in the accepted set or specifying the translation function directly is a tedious endeavor. Ideally, the specification of nodes gives some correct-by-construction guarantees.

This section presents a *concrete syntax* to specify a *Decoding Net* and its address space nodes for a particular machine. It further provides examples of real systems expressed in this syntax.

4.4.1 Syntax

The following snippet shows the elements of the concrete syntax where $\langle e \rangle$ denotes an optional element e, the pipe operator $e_1|e_2$ denotes an

alternative branch of either e_1 or e_2, curly braces {..} denote set of zero or more elements and brackets [..] denote a list of zero or more elements.

$$
\begin{aligned}
block_s &:= \mathbb{N}\text{–}\mathbb{N} \\
map_s &:= block_s \ \textbf{to}\ \mathbb{N} \left\langle\ \textbf{at}\ \mathbb{N} \right\rangle \left\{\textbf{,}\ \mathbb{N} \left\langle\ \textbf{at}\ \mathbb{N} \right\rangle\right\} \\
node_s &= \left\langle \textbf{accept [}\ \left\{block_s\right\}\ \textbf{]} \right\rangle \left\langle \textbf{map [}\ \left\{map_s\right\}\ \textbf{]} \right\rangle \left\langle \textbf{over}\ \mathbb{N} \right\rangle \\
net_s &= \left\{ \mathbb{N}\ \textbf{is}\ node_s \;\middle|\; \mathbb{N}\textbf{..}\mathbb{N}\ \textbf{are}\ node_s \right\}
\end{aligned}
$$

Blocks The syntax does not operate on addresses directly. Instead, it uses contiguous blocks of addresses which include every address between a base and a limit. Naturally, a single address can be expressed by setting base equal to limit.

A single 4 KiB page at address $0x1000$ is expressed as:

$$page = \texttt{0x1000} - \texttt{0x1fff}$$

which corresponds to the set of addresses

$$page = \{\ a \mid \texttt{0x1000} \leq a \wedge a \leq \texttt{0x1fff}\ \}.$$

Maps The map construct specifies how a single block of addresses is translated. The entire input block is mapped **to** some output node **at** a possible new base address, effectively shifting all address values to this new address. The same block may be mapped to multiple nodes to support possible broadcast scenarios.

The 4 KiB page from above might be mapped to node 3 at address `0x2000`

$$map_1 := page\ \ \textbf{to}\ 3\ \ \textbf{at}\ \texttt{0x2000}$$

Nodes The node construct consists of three separate parts where all of them are optional. In the order of precedence:

1. *accept:* A set of blocks that define the addresses accepted by the node. The accept set is the union of all blocks.
2. *map:* A set of maps that specify how the node translates addresses.
3. *over:* Indicates the overlay node, a one-to-one map of all input addresses to the specified overlay node, unless the address is otherwise mapped.

The node in the following example overlays node 4, i.e. all addresses are translated one-to-one to node 4 unless they are within one of the two mapped blocks (map_1 or map_2). Recall, an address can be both, mapped and accepted.

$$node_1 = \textbf{accept [} page_1 \textbf{] map [} map_1, \ map_2 \textbf{] over } 4$$

Net Finally, the net construct assigns identifiers to nodes. It is possible to assign multiple identifiers to a single node specification. This corresponds to multiple identical nodes with different node identifiers in the *Decoding Net*. This is useful to express a multi-core processor for instance.

$$net = 3 \textbf{ is } node_1, \ 4..5 \textbf{ are } node_2$$

4.4.2 System Descriptions

Recall Figure 2.25 which depicts the simplified block diagram of the Texas Instruments OMAP4460 SoC earlier in this chapter. This section expresses parts of the memory subsystem of this chip using the syntax defined above. More system descriptions can be found in [Ach+17b] and the open source Isabelle theories [ACH19].

Figure 4.4 and highlights three interesting cases of the OMAP4460 SoC:

- RAM Access from the ARM Cortex-A9 and M3 cores.
- A loop through the MIF node of the ARM Cortex-M3 subsystem.
- Shared and private accesses to the General Purpose Timer (GPT)

The block diagram of Figure 4.4 is expressed using the model syntax in Figure 4.5

Accessing RAM The ARM Cortex-A9 cores of the SoC have a private connection to the DRAM module which shadows the path through the L3 Interconnect. DRAM appears at address `0x80000000` in the core's 'physical' address space (P_{A9}). The DSP subsystem on the other hand only has connection to DRAM through the L3 interconnect which exposes DRAM at the same address to the DSP cores. In contrast, the ARM Cortex-M3 first routes memory requests through an address splitter (labeled as "MIF" in the block diagram) and then through a second stage address translation unit before reaching the L3 interconnect. The address window of the "L2" translation unit starts at `0x0` and provides 1.5GB of address range. Consequently, the Cortex-M3 and Cortex-A9 cores never see DRAM at the same local physical address.

General-Purpose Timer Access The general-purpose timer is visible at different addresses for the ARM Cortex-A9 cores, the DSP subsystem and even at a configurable address as seen from the ARM Cortex-M3 subsystem. Therefore, there are at least three *distinct* addresses for the same registers. The Cortex-A9 uses `0x40138000`, a DSP uses `0x01D38000` and a DMA-capable device on the L3 interconnect uses `0x49038000`. Plus, the Cortex-M3 core uses the configurable mapping onto the L3 interconnect at `0x49038000`.

The MIF Loop The ARM Cortex-M3 subsystem contains a (benign) routing loop: A memory access that is forwarded to the "L2" translation unit by the MIF address splitter is translated to the address on the L3 interconnect which matches the address port back to the MIF splitter. This

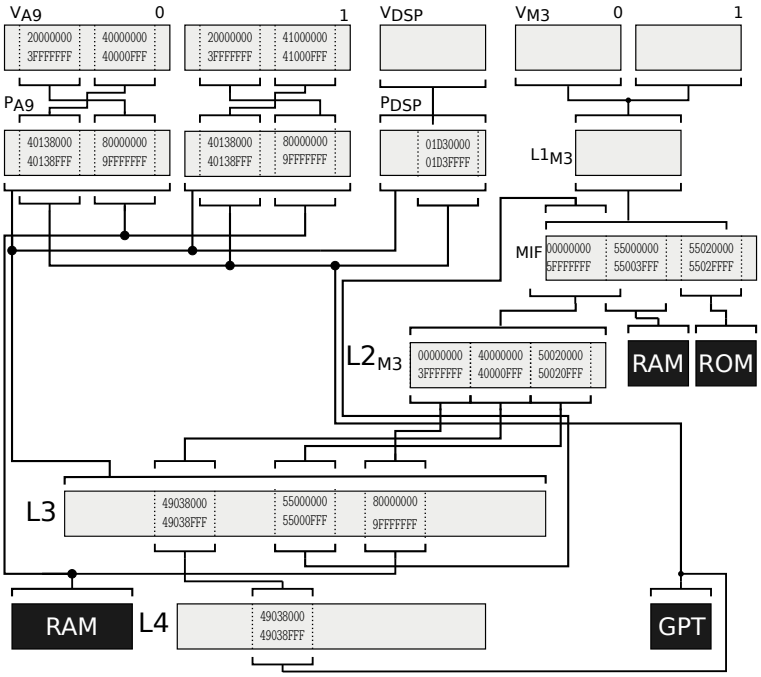


Figure 4.4: Simplified Addressing Block Diagram of the Texas Instruments OMAP 44xx SoC as Shown in [Ach+17b].

$V_{A9:0}$ **is map** [`0x20000000-0x20000fff` **to** $P_{A9:0}$ **at** `0x80000000`]
$V_{A9:1}$ **is map** [`0x20000000-0x20000fff` **to** $P_{A9:1}$ **at** `0x80000000`]
$P_{A9:0}$, $P_{A9:1}$ **are map** [`0x40138000-0x40138fff` **to** *GPT* **at** `0x0`
`0x80000000-0xbfffffff` **to** *RAM* **at** `0x0`] **over** *L3*
V_{DSP} **is over** P_{DSP}
P_{DSP} **is map** [`0x01d3e000-0x01d3efff` **to** *GPT* **at** `0x0`] **over** *L3*
$L2_{M3}$ **is map** [`0x00000000-0x3fffffff` **to** *L3* **at** `0x80000000`]
V_{M3}, V_{M3} **are over** $L1_{M3}$
$L1_{M3}$ **is map** [`0x00000000-0x0fffffff`$_{28}$ **to** *MIF*]
RAM_{M3} **is accept** [`0x55020000-0x5502ffff`]
$spcL4$ **is map** [`0x49038000-0x49038fff` **to** *GPT* **at** `0x0`]
ROM_{M3} **is accept** [`0x55000000-0x55003fff`]
GPT **is accept** [`0x00000000-0x00000fff`]
MIF **is map** [`0x00000000-0x5fffffff` **to** $L2_{M3}$,
`0x55000000-0x55003fff` **to** RAM_{M3},
`0x55020000-0x5502ffff` **to** ROM_{M3}]
L3 **is map** [`49000`$_3$/24 **to** *L4* **at** `40100`$_3$,
`55000`$_3$/12 **to** *MIF*]
`80000`$_3$/30 **to** *RAM*]
RAM **is accept** [`0x00000000-0x3fffffff`]

Figure 4.5: Describing the Texas Instruments OMAP4460 using the Concrete Syntax.

creates a loop, however the address is different the second time and will be forwarded to the RAM or ROM nodes where resolution terminates.

4.5 Algorithms

The previous sections defined the *Decoding Net* model and presented a concrete syntax that enables concisely describing the memory subsystem of real systems. With heterogeneous platforms in mind, where two distinct cores have two different views of the system in which addresses are translated multiple times before their decoding terminates, with the possibility of decoding-loops, how can one answer arising questions such as:

- 'How can two views be expressed and compared efficiently?'
- 'How can the model be efficiently manipulated at runtime?'
- 'How can the absence of decoding loops be shown?'

This section presents algorithms that run on top of the *Decoding Net* model that enable reasoning about address translation.

4.5.1 Views

The *Decoding Net* model introduced in Section 4.3 consists of a `decode` relation and the set of `accepted_names` which together encode a directed graph. While this allows specifying the address decoding behavior of hardware concisely, it does not explicitly and directly say anything about what a core (i.e. a processor or device) sees from the system.

The questions such as *"Which resources are visible in this address space?"* and *"At which address does this resource show up in that address space?"* are of particular interest for memory allocation considerations.

A core has a particular *viewpoint* which is defined by the cores' local address space. The `resolve` function links the local names (addresses

relative to the core's viewpoint) to global names i.e. the nodes may accept the name and the local address at which they will accept it.

Fixing the viewpoint and resolving addresses within therefore defines a *view* of the *Decoding Net* from a particular node, in other words, all resources that can be reached from that node. However, not all addresses can be resolved: some of them may result in a decoding loop while others simply do not translate. The view from a node n is defined as the reflexive, transitive closure of the decoding relation:

$$\lambda(n, a).\ \texttt{accepted\_names} \cap (\texttt{decodes\_to}^*(n, a))$$

Formulating this as a function, however, requires a proof of termination.

4.5.2 Termination

Decoding Nets are, by construction, directed graphs which may have loops and thus they are neither acyclic graphs nor trees. This property is necessary, as real hardware in fact does have loops e.g. the Texas Instruments OMAP4460 or the Intel Xeon Phi as illustrated on Figure 4.6 and more examples in [Ach+17b]. The presence of loops is a problem because in this case, address resolution does not terminate. This results in unspecified behavior. However, despite having loops, the two examples of Figure 4.6 can be proved to be benign i.e. they are not true loops. Instead, the address is different every time the same node is traversed. In general, whether a system is free of true loops is a property to be proved. It is not an *a priori* assumption.

The absence of loops is a desired property and ideally it should be possible to verify that system software does not configure translation units such that true loops are created. This proof obligation is not added to the model out of the blue. Instead, the absence of loops is a naturally arising property from a functional representation of address spaces: Isabelle/HOL is a

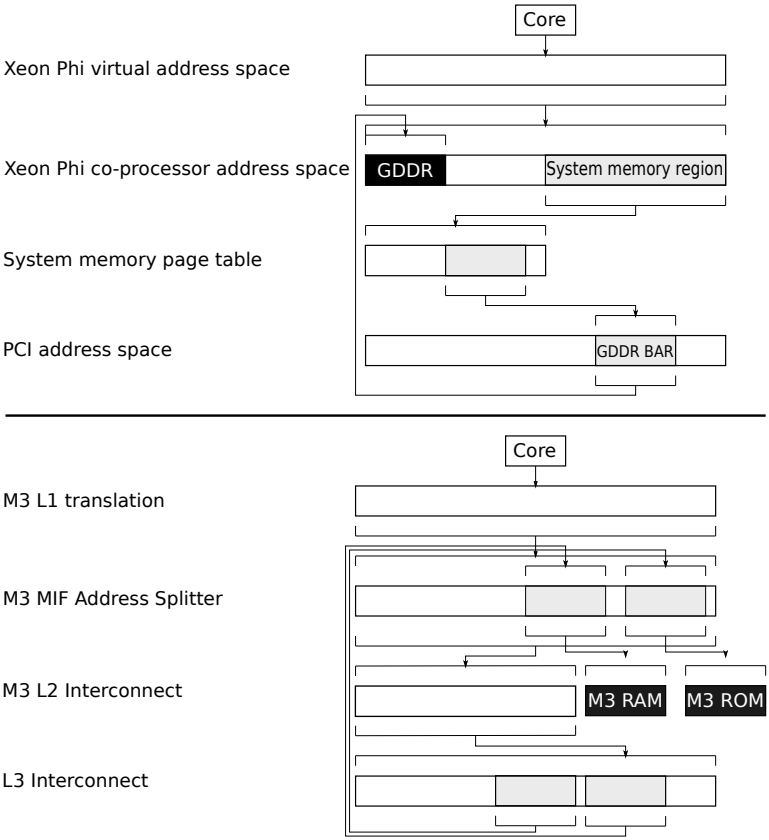


Figure 4.6: Existing Loops in Hardware. Xeon Phi on Top and OMAP 4460 on the Bottom [Ach+18].

logic of total functions which implies that the function is defined for *all* its possible input values. This is not the case in the presence of loops.

In other words, there are no loops if every path defined by the decode-relation starting at the input address is finite. Every step of the decode-relation brings the resolution process closer to the final step. Intuitively, the 'distance' to the result is strictly decreasing until it reaches 0. Formally, this is a *variant* expressed by a well-formed *ranking function*:

$$f : name \rightarrow \mathbb{N}$$

The ranking function f is well-formed for a starting name (n) and if it is strictly monotonic decreasing for every step in the decode-relation:

$$\forall x, y.\ (n, x) \in \texttt{decodes\_to} \wedge (x, y) \in \texttt{decodes\_to}^* \longrightarrow f(y) < f(x)$$

Termination follows by induction on the value of the ranking function: f is bounded below by 0 and f is strictly monotonic decreasing. Therefore, if a well-formed ranking function f exists, then the name n is in the domain of the `resolve` function ($n \in$ `resolve_dom`) and resolution terminates.

What is left to show is that the decode-relation does not grow indefinitely. In other words, each decoding step produces a *finite* number of translations for a name. This is trivially true for real hardware, as the components are finite. Also, the syntax enforces this by construction, as it requires *listing* all translations of a node. This is a necessary condition to be able to find a well-formed ranking function.

The ranking function can be constructed by induction over the `decodes_to` relation as shown in Algorithm 2. Every name n that are part of the `accepted\_names` set of the *Decoding Net* have a trivial ranking value of 0, because resolution terminates at the accepting set. Otherwise, Algorithm 2 assigns the rank as the maximum rank of all successors plus one (if a

```
1 def RankingFn(n):
      Data: Decoding Net, name
      Result: rank of name n
2     if n ∈ accepted_names then
3         return 0;
4     else
5         rank_max ⟵ 0;
6         foreach (n, x) ∈ decodes_to do
7             r ⟵ RankingFn(x) ;
8             rank_max = max(rank_max, r);
9         return rank_max + 1
```

Algorithm 2: Ranking Function Construction Algorithm.

well-formed ranking exist). Note, this is where the finiteness condition of the `translate` function is required.

Finally, this establishes the fundamental duality between the graphical and operational views of an address space by the equivalence of the relational and recursive-functional models. If the resolution is well-defined, then the result is the set of accepting names that are reachable via the decode-relation. Whenever Algorithm 2 produces a well-defined ranking function f for a name (n, a), then that name is in the domain of the resolve-function.

$$\exists f.\ \texttt{wf\_rank}\ f\ (n,a) \quad \longleftrightarrow \quad \texttt{resolve\_dom}\ (n,a)$$

4.5.3 Normal Forms and View-Preserving Transformations

For practical use in system software, an efficient representation and algorithms are needed to manipulate and query the decoding net at both, runtime and compile time. For example, generating kernel page tables

```
1 def Flatten(net, root):
      Data: Decoding Net, Root Node
      Result: Flattened Representation
2     net_s ⟵ empty_net;
3     net_a ⟵ empty_net;
4     foreach (n, node) ∈ net do
5         node_a, node_s ⟵ Split(node);
6         net_s(n) := node_s;
7         net_a(n) := node_a;
8     node_root ⟵ Merge(net_s(root));
9     return net_a + node_root
```

Algorithm 3: Flattening Operation.

for the bootstrap processor (e.g [Sch17]) requires knowledge of where in the processor's "physical" address space devices and RAM appear. This information is implicit in the *Decoding Net* model, but not easily accessible as it requires computationally and memory intensive computations.

A more efficient way to obtain the same information is to compute and materialize a flattened representation, while *preserving* the view from each core. Because of view preservation, querying the full *Decoding Net* or the flattened representation yields the same results, while the latter is more efficient. The flattened representation is the *normal form* (Figure 4.7): a single translating node (one-step transitive closure of the decode-relation) maps directly to a set of accepting nodes. Algorithm 3 transforms the *Decoding Net* into a normal form for a well-defined root using the `split` and `merge` operations:

1. *Split:* Every node in the *Decoding Net* is split such that it either accepts addresses (i.e. it is a physical resource), or it translates addresses and forwards them to other nodes, but not both. Thus, nodes can be classified into accepting and translating.

2. *Merge:* Starting from the root node, the translating nodes are merged

together by flattening multi-stage translation steps – effectively computing the one-step transitive closure of the decode-relation. The resulting, flattened node translates each input address directly to an accepting name. The `translate` function then defines the address space visible from that node.

It is important to emphasize here, that the result of normalization depends on the fixed observer, i.e. the *root*-node for which the `flatten` operation generates the view.

The question arises, whether the transformation `flatten` (Algorithm 3) is actually *correct*. This requires verification of the algorithm. The reminder of this section shows the definition and verification of *view equivalence-preserving transformations* on the semantic model. Ultimately, an algorithm that refines (or faithfully implements) the transformation produces the correct result. The reminder of the section defines view equivalence and gives formal definitions of the transformations.

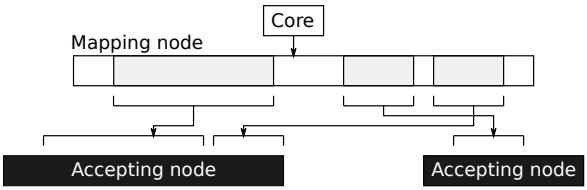


Figure 4.7: Normalform Representation – A Single Translation Step onto the Physical Resources of the System.

View Equivalence Two *Decoding Nets net* and *net′* are *view-equivalent* if all observer nodes in S in both nets have the same view. In other words, the output of the `resolve` function from the nodes S is the same modulo some renaming of accepting nodes (f and g). This implies that in both *Decoding Nets*, the same accepting resources can be reached from the nodes in S. The view equivalence is denoted as:

$$(f, net) \sim_S (g, net')$$

Node Splitting Splitting of nodes effectively doubles the number of nodes as each node is divided into an accepting and mapping part. Doing so requires associating the split nodes with new, unused node identifiers.

For this purpose, let c be greater than the label of any existing node. Then the `accept` and `translate` functions of the split decoding net are defined as follows:

$$\begin{aligned}
\texttt{accept}'_n &= \emptyset \\
\texttt{accept}'_{(n+c)} &= \texttt{accept}_n \\
\texttt{translate}'_n\ a &= \{(n+c, a) \mid a \in \texttt{accept}_n\} \cup \texttt{translate}_n\ a \\
\texttt{translate}'_{(n+c)}\ a &= \emptyset
\end{aligned}$$

Where $\text{accept}|_n$ denotes the accept-set for node n, and $\texttt{translate}_n$ the translation function of node n analogously.

Node n is therefore split into two nodes: n with populated translation function and $n + c$ with populated accept set. Note, the translation function now also includes translations to the accept-set of node $n + c$ because of the splitting.

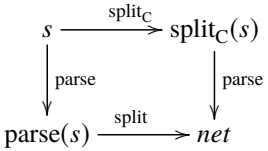
The resulting *Decoding Net* is view-equivalent to the original one. Names that were accepted at node n are now accepted at node $n + c$, and node n translates those addresses to node $n + c$. This gives the renaming function $n \mapsto n + c$ for the equivalence:

$$(n \mapsto n + c, net) \sim_S (\emptyset, \texttt{split}(net))$$

Syntactic Node Splitting Instead of splitting the node in the abstract representation as above, the same operation can be applied to the concrete syntax representation. In this case the $\texttt{split}_C$ is a simple syntactic operation:

$$nd \textbf{ is accept } A \textbf{ map } M \textbf{ over } O \mapsto [\ nd + c \textbf{ is accept } A,\ nd \textbf{ is map } M(nd \mapsto nd') \textbf{ over } O\ ]$$

Whether splitting at the syntactic level or on the abstract level produces the same outcome needs to be proven. The following commutative diagram expresses the necessary refinement proof:



The proof requires to show that the two operations (split() and $\text{split}_C()$) together with the state relation (parse()) actually produce the same result. parse() constructs the *Decoding Net* from the syntactic representation. In the end, the equivalence must hold:

$$\texttt{split}\ (\texttt{parse}\ s) = \texttt{parse}\ (\texttt{split}_C\ s)$$

The combination of the split equivalence and the refinement yields that the concrete implementation preserves the equivalence of the nets constructed by parsing.

$$(n \mapsto n + c, \texttt{parse}\ s) \sim_S (\emptyset, \texttt{parse}\ (\texttt{split}_C\ s))$$

Flattening Operation Flattening of a node does not change the number of nodes in the *Decoding Net* as the splitting operation does. Flattening targets the `translate` function of a node, or in other words the `decode` relation of the *Decoding Net*. Effectively, all names, that are translated by that node, are now directly translating to the accepting resource (`resolve`).

For a fixed node n, the flattening is expressed using the `resolve` function:

$$\begin{aligned} \texttt{accept}'_n &= \texttt{accept}_n \\ \texttt{translate}'_n a &= \lambda a.\texttt{resolve}\ (n, a) \end{aligned}$$

This transformation does not alter the set of `accepted_names`. Moreover, the equivalence of the decode-relation of the original *Decoding Net* and the flattened version can be shown:

$$(x, y) \in (\texttt{decode}\ (\texttt{flatten}\ net\ nd))^* \longleftrightarrow (x, y) \in (\texttt{decode}\ net)^*$$

Putting it all together Combining the equivalence results of the splitting and flattening operations enables the verification of equivalence between:

1. The physical resources that are reachable from the transformed model.
2. The physical resources that would be reachable when expensively traversing the original, hardware-derived model for all addresses.

Chapter 6 uses this result in operating system services and at compile time to generate low-level, platform specific operating systems code.

4.6 Modeling the MIPS R4600 TLB

Translation lookaside buffers (TLBs) of processors cache the translation of virtual addresses to (local) physical addresses defined by a data structure called page tables. TLBs are probably the single most complex translation element and are used to implement the abstraction of virtual memory. The processor core executing an application issues loads and stores to virtual addresses. Those memory requests are intercepted by the translation hardware (e.g. the memory management unit) which translates the virtual address to a local physical address, based on the state of the TLB, which defines the translation function. TLBs are small and are therefore not able to cache all translations. An attempt to translate an address for which there is no matching entry in the TLB triggers an exception or translation fault: this may either invoke a hardware page-table walker or result in a page fault to be handled by the operating system.

This section demonstrates the ability of the *Decoding Net* model to capture and express the configuration of real hardware, as well as reasoning about it at the example of the MIPS R4600 TLB [ITD95]. This is a well-understood and well-documented instance of a software-loaded TLB. The MIPS R4600 TLB explicitly exposes configuration operations to the operating system. The reminder of the section is structured as follows:

1. Define the operational model of the MIPS R4600 TLB including operations and invariants.

2. Show that the operation model refines the *Decoding Net*.

3. Define and show the equivalence of a small TLB plus fault handler and an infinitely large TLB holding all translations.

4. Use the operational model to prove the impossibility of provably-correct initialization.

4.6.1 The TLB Model

This section defines the operational model of the MIPS R4600 translation lookaside buffer (TLB) including its state and operations invoked by systems software to update it.

4.6.1.1 State Definition

The MIPS R4600 TLB consists of 48 entry-pairs, each of which is mapping two adjacent virtual pages to independent physical frames of the same size. The virtual pages are identified by their virtual page number (VPN) as specified in the `EntryHi`. The two `EntryLo0` and `EntryLo1` contain the physical frame number (PFN) which defines the target, local physical address of the translation. The mask defines which of the seven pre-defined page sizes is used for the entry. Figure 4.8 gives the layout of a single TLB entry-pair.

An entry-pair of the TLB is expressed in Isabelle/HOL with the following record type definitions:

$$\begin{aligned} \mathit{TLBEntryHi} &= (\text{region} : \mathbb{N},\ \text{vpn2} : \mathbb{N},\ \text{asid} : \mathbb{N}) \\ \mathit{TLBEntryLo} &= (\text{pfn} : \mathbb{N},\ \text{v} : \mathit{bool},\ \text{d} : \mathit{bool},\ \text{g} : \mathit{bool}) \\ \mathit{TLBEntry} &= (\text{hi} : \mathit{TLBEntryHi},\ \text{lo0} : \mathit{TLBEntryLo},\ \text{lo1} : \mathit{TLBEntryLo}) \end{aligned}$$

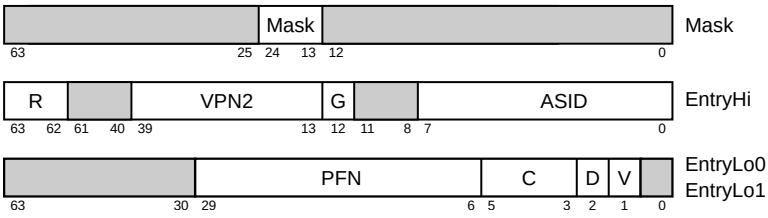


Figure 4.8: A MIPS R4600 TLB Entry with Non-Zero Fields Labeled.

As already mentioned, the MIPS R4600 TLB consists of 48 entry-pairs where each entry is indexed by a number from $0 - 47$. Besides the entry-pairs, the state of the TLB also contains the wired and random values which alter the behavior of some state modifying operations (see paragraph below). Moreover, the state also includes the (static) capacity to facilitate refinement proofs later on. In Isabelle/HOL the entire state of the TLB is expressed as the following data record:

$$\begin{aligned} \mathit{MIPSTLB} = \ & (\text{wired} : \mathbb{N},\ \ \text{capacity} : \mathbb{N},\ \ \text{random} : \mathbb{N}, \\ & \text{entries} : \mathbb{N} \rightarrow \mathit{TLBEntry}) \end{aligned}$$

4.6.1.2 Operations

The MIPS R4600 TLB is software-loaded. This means there is no hardware component that automatically walks in-memory page tables on a TLB miss. Instead, when a virtual address lookup fails, the TLB hardware triggers a fault. In response, the operating system is responsible for taking action and install an entry for the faulting address (if appropriate).

The operating system, therefore, modifies the state of the TLB explicitly and it does so using four special instructions provided by the MIPS instruction set. The operations effectively transfer the TLB entry-pair contents from and to a temporary register. The four operations are:

- *TLB Probe [tlbp]* performs an associative lookup by matching the TLB entry-pairs against the contents of the register based on the VPN, ASID, and the size mask. The result of the `tlbp` operation is either the index of the matching entry, or an miss indication.
- *TLB Read [tlbr]* reads the requested TLB entry-pair at the specified index, and copies it to the temporary register.
- *TLB Write Indexed [tlbwi]* updates the TLB entry-pair at the specified index with the contents of the temporary register.

- *TLB Write Random [tlbwr]* updates the TLB entry-pair at the index specified by the `Random` register. The contents of the `Random` can assume to some value in [wired, capacity).

Note, the value of the `Random` register is effectively decremented every processor clock tick. Therefore, the precise value of the `Random` register at the time of the random-write operation depends on the exact execution trace. Without knowing the trace, the precise value is unknown and thus model assumes a value non-deterministically chosen from [wired, capacity).

The result of the `tlbp` operation is undefined when multiple TLB entry-pairs match at the same time. This is an important correctness property. A violation of this leads to undefined behavior and may even cause permanent physical damage to the chip (see Section 4.6.2). A TLB entry-pair matches against `EntryHi` if the address falls within the range covered by the VPN and the address-space identifier (ASID) matches. An entry with the global bit set matches any ASID. The `EntryMatch` predicate expresses this in the model. Similarly, a VPN and ASID can be matched against an entry directly using the `EntryMatchVPNASID` predicate.

$$\texttt{EntryMatch} :: \mathit{TLBEntry} \Rightarrow \mathit{TLBEntry} \Rightarrow \mathit{bool}$$
$$\texttt{EntryMatch}\ e1\ e2 =$$
$$((\texttt{EntryVPNMatch}\ e1\ e2) \wedge (\texttt{EntryASIDMatch}\ e1\ e2))$$

$$\texttt{EntryMatchVPNASID} :: \mathit{VPN} \Rightarrow \mathit{ASID} \Rightarrow \mathit{TLBEntry} \Rightarrow \mathit{bool}$$
$$\texttt{EntryMatchVPNASID}\ \mathit{vpn}\ a\ e =$$
$$\texttt{EntryMatchWithVPN}\ \mathit{vpn}\ e \wedge \texttt{EntryMatchWithASID}\ a\ e$$

The state-update are then operations are expressed in Isabelle/HOL using the following signature types:

$$\begin{aligned}
&\texttt{tlbp} : \mathit{TLBENTRYHI} \to \mathit{MIPSTLB} \to \{\mathbb{N}\} \\
&\texttt{tlbr} : \mathbb{N} \to \mathit{MIPSTLB} \to \{\mathit{TLBENTRY}\} \\
&\texttt{tlbwi} : \mathbb{N} \to \mathit{TLBENTRY} \to \mathit{MIPSTLB} \to \{\mathit{MIPSTLB}\} \\
&\texttt{tlbwr} : \mathit{TLBENTRY} \to \mathit{MIPSTLB} \to \{\mathit{MIPSTLB}\}
\end{aligned}$$

All of these operations return a set of values to express undefined behavior and non-determinism. In fact, any of these operations can result in undefined behavior: For instance, the attempt to read or write a TLB entry-pair at an out-of-bounds index or probing the TLB with multiple matching entries is unpredictable. Moreover, updating an entry with one that conflicts other entries leaves the TLB in an unknown state.

Undefined outcomes are modeled as non-determinism and hence all operations return a *set* of possible outcomes where UNIV, the universal set, indicates complete under-specification. The `tlbwi` operation, for instance, returns UNIV when an out-of-bounds index access occurs i.e. $i \geq$ capacity *tlb*, and otherwise the specified entry-pair i is updated and a singleton set is returned. This is expressed in Isabelle/HOL as

$$\begin{aligned}
\texttt{tlbwi}\ i\ e\ \mathit{tlb}\ = \ &\textbf{if}\ i < \text{capacity}\ \mathit{tlb} \\
&\textbf{then}\ \{\mathit{tlb}\,(\text{entries} := (\text{entries}\ \mathit{tlb})(i := e))\}\ \textbf{else}\ \text{UNIV}
\end{aligned}$$

where the notation $f(x := y)$ denotes a function update at x with value y in Isabelle/HOL syntax.

Recall, the TLB Write Random (`tlbwr`) updates the entry-pair at the index defined by the `Random` register whose values are within the range [wired, capacity). Because the actual value of the `Random` register is updated

non-deterministically, the TLB Write Random operation is expressed as the non-deterministic choice of some indexed write. This is the union of all possible TLB states which may be observed depending on the capacity and the contents of the wired register.

$$\texttt{tlbwr}\ e\ tlb = \bigcup_{i=\text{wired}\ tlb}^{(\text{capacity}\ tlb)-1} \texttt{tlbwi}\ i\ e\ tlb$$

4.6.2 The Validity Invariant

The MIPS R4600 TLB and its predecessors famously permit the programmer to unsafely configure the TLB, leaving the TLB in an inconsistent state. This results in undefined future behavior of the processor. Worse, in earlier versions of the chip this could even lead to permanent hardware damage whereas newer versions silently turn off the TLB.

At its core, the source of the problem is how virtual addresses (or the registers when doing a probe operation) are matched against entries in the TLB. Being fully associative TLB, this is implemented by a parallel comparison against all 48 entries. This happens on the critical path of any memory access as the TLB translates every virtual address used by the processor. The implementation of such an associative lookup requires a substantial amount of logic and thus is very expensive. Unsurprisingly, this is highly optimized e.g. by making the assumption that there will be at most one entry that matches an address. Breaking with this assumption leads to two or more entries driving the same wire to different voltages resulting in inconclusive results, or worse burning the chip.

The vendor of the MIPS R4600 TLB places this burden onto the operating system programmer who needs to ensure that this assumption is never broken. It's important to emphasize here that entries still match an address despite being marked as invalid (valid bit (v)) set to zero). This is an important detail. In addition, the requirement is in fact that two entries

never can match which is a stronger requirement than that they never *do*. A possible reason for this could be that doing the associative lookup first, then check the valid bit is cheaper than doing both steps in parallel. In general, this is even more important on hardware, which does speculative execution: a memory access is triggered speculatively based on some access patterns (as in [Lip+18; Can+19; Koc+18]) which causes a TLB lookup of an address it actually never computes. Despite discarding the results afterwards, damage would have been done.

This property is expressed in Isabelle/HOL as a conflict set. The conflict set of an entry-pair is the set of indices at which there exists a matching or overlapping entry. The same applies for virtual addresses analogously.

$$\begin{aligned}&\texttt{EntryConflictSet} :: \mathit{TLBEntry} \Rightarrow \mathit{MIPSTLB} \Rightarrow \{\mathbb{N}\}\\&\texttt{EntryConflictSet}\ e\ tlb =\\&\quad \{\, i.\ i < (\texttt{capacity}\ tlb) \wedge \texttt{EntryMatch}\ (\texttt{entries}\ tlb\ i)\ e\,\}\end{aligned}$$

The correctness invariant is that the conflict set is either empty or it is a singleton set consisting solely of the index of the entry-pair itself (an entry pair always matches itself).

$$\texttt{EntryConflictSet}\ (\texttt{entries}\ tlb\ i)\ tlb \subseteq \{i\}$$

Each entry-pair also must be well-formed. This includes the mask field having a valid bit pattern e.g. the VPN or PFN not exceeding the number of supported bits, etc. Lastly, the number of wired entries must not exceed the capacity of the TLB.

Putting everything together yields the `TLBValid` invariant (Invariant I4.2):

Invariant I4.2 (TLBValid)

$$\begin{aligned}
&\texttt{TLBValid} :: \mathit{MIPSTLB} \rightarrow \mathit{bool}\\
&\texttt{TLBValid}\ \mathit{tlb} = \texttt{wired}\ \mathit{tlb} \leq \texttt{capacity}\ \mathit{tlb} \wedge\\
&\quad \big(\forall i < \texttt{capacity}\ \mathit{tlb}.\\
&\qquad \texttt{TLBEntryWellFormed}\ \mathit{tlb}\ i \wedge\\
&\qquad \texttt{EntryConflictSet}\ (\texttt{entries}\ \mathit{tlb}\ i)\ \mathit{tlb} \subseteq \{i\}\big)
\end{aligned}$$

The invariant states that all entry-pairs of the TLB are well-formed and there are no conflicting or overlapping entries in the TLB. Changes to the TLB state using the indexed or random write operations must always preserve this invariant, otherwise leaving the TLB in an undefined state. This requires a proof that both modifying operations, indexed and random write, preserve the invariant.

Indexed write preserves invariant. The first lemma states that if the TLB is in a valid state, the new entry is well-formed and does at most conflict with the entry at the index with is to be replaced and the index is within range, then the `tlbwi` operation will preserve the invariant.

Lemma L4.1 (tlbwi preserves invariant)

$$\begin{aligned}
&\textbf{assumes}\ i < \texttt{capacity}\ \mathit{tlb}\ \textbf{and}\ \texttt{EntryConflictSet}\ e\ \mathit{tlb} \subseteq \{i\}\\
&\quad \textbf{and}\ \texttt{TLBValid}\ \mathit{tlb}\quad \textbf{and}\ \texttt{TLBENTRYWellFormed}\ e\\
&\textbf{shows}\ \forall t \in \texttt{tlbwi}\ i\ e\ \mathit{tlb}.\ \texttt{TLBValid}\ t
\end{aligned}$$

Random write preserves invariant. The second lemma is analogous to the first one, however the random write operation may replace *any* entry of the TLB, therefore the precondition must be stronger: it is required that the new entry does not conflict with any existing entry in the TLB, i.e. the conflict set is empty.

Lemma L4.2 (tlbwr preserves invariant)

assumes `TLBValid` *tlb* **and** `TLBENTRYWellFormed` *e*
and `capacity` *tlb* > 0 **and** `EntryConflictSet` *e tlb* $= \{\}$
shows $\forall t \in$ `tlbwr` *e tlb*. `TLBValid` *t*

4.6.3 Modeling TLB Translations

The previous sections define the MIPS R4600 TLB operational model with the validity Invariant I4.2, which ensures that the TLB is in a well-defined state. Translation lookaside buffers typically cache a small number address translations and, as a consequence, they cannot hold all address translations of the entire address space. Nevertheless, the TLB provides the illusion to applications that they have the full address space at their disposal.

An attempt to translate an address for which there is no matching entry triggers an exception (Section 4.6.3.1). In response, software or hardware tries to replace an entry with based on a data-structure lookup (e.g. a page table Section 4.6.3.2) and then continues execution of the user-space program. This provides the illusion of an infinitely large TLB (Section 4.6.3.3) which contains all translations of the process' virtual address space. The remainder of this section describes how those aspects are expressed on top of the MIPS R4600 TLB model.

4.6.3.1 Translations and Exceptions

Addresses that do not have a matching entry in the TLB, or the entry has wrong permissions raise an exception when the processor issues that address. On the MIPS R4600, this triggers a software handler routine, whereas on other architectures, such as ARMv8 or x86, this handler routine is implemented in hardware. In both cases, the handler evaluates the translation function based on the configuration defined in some data structure (e.g. a page table).

Not all exceptions are created equal: An exception can occur when there is no matching entry, the entry is invalid, or the permissions do not allow the desired access to that memory address. Figure 4-19 in the MIPS technical reference manual [ITD95] shows the corresponding exception flowchart. The following three exceptions are defined:

- *TLB Refill.* No entry matched the given virtual address.
- *TLB Invalid.* An entry matched, but was invalid.
- *TLB Modified.* Access violation, e.g. a write to a read-only page.

Table 4.1 summarizes the possible outcomes of an address lookup.

The operation model expresses this as an datatype in Isabelle/HOL, where `EXNOK` represents a success condition.

datatype `MIPSTLBEXN` = `EXNREFILL` | `EXNINVALID` | `EXNMOD` | `EXNOK`

Recall, an entry-pair matches a certain address range defined by its virtual page number (VPN) and the address space identifier. Depending on whether the VPN is odd or even, the address is translated either using `EntryLo0` or `EntryLo1`. The result is either a singleton set in the case of successful translation or the empty set otherwise. The following function defines the translation of a single TLB entry in Isabelle/HOL.

Match	Valid	Entry writable / Memory write	VPN even	Result
No	*	*	*	TLB Refill Exception
Yes	No	*	*	TLB Invalid Exception
Yes	No	No and memory write	*	TLB Modified Exception
Yes	Yes	Yes or memory read	Yes	Translate using EntryLo0
Yes	Yes	Yes or memory read	No	Translate using EntryLo1

Table 4.1: The Outcome of the Translate Function as Shown in [Ach+18].

$$
\begin{aligned}
&\texttt{TLBENTRY\_translate} :: \mathit{Entry} \to \mathbb{N} \to \mathbb{N} \to \{\mathbb{N}\} \\
&\texttt{TLBENTRY\_translate}\ e\ \mathit{as}\ \mathit{vpn} = \\
&\quad \textbf{if}\ \texttt{EntryMatchVPNASID}\ \mathit{vpn}\ \mathit{as}\ e\ \textbf{then} \\
&\qquad \textbf{if}\ \texttt{even}\ \mathit{vpn}\ \wedge\ \texttt{EntryIsValid0}\ e \\
&\qquad\quad \textbf{then}\ \{(\texttt{pfn}\ (\mathit{lo0}\ e)) + (\mathit{vpn} - \texttt{EntryMin4KVPN}\ e)\} \\
&\qquad \textbf{else if}\ \texttt{odd}\ \mathit{vpn} \wedge \texttt{EntryIsValid1}\ e \\
&\qquad\quad \textbf{then}\ \{(\texttt{pfn}\ (\mathit{lo1}\ e)) + (\mathit{vpn} - \texttt{EntryMin4KVPN1}\ e)\} \quad \textbf{else}\ \{\} \\
&\quad \textbf{else}\ \{\}
\end{aligned}
$$

The translation behavior of the entire MIPS R4600 TLB is then given by the union of the translates of all entries:

$$
\begin{aligned}
&\texttt{MIPSTLB\_translate} :: \mathit{MIPSTLB} \to \mathbb{N} \to \mathbb{N} \to \{\mathbb{N}\} \\
&\texttt{MIPSTLB\_translate}\ \mathit{tlb}\ \mathit{vpn}\ \mathit{as} = \\
&\qquad \bigcup_{i < \texttt{capacity}\ \mathit{tlb}} \texttt{TLBENTRY\_translate}\ ((\texttt{entries}\ \mathit{tlb})\ i)\ \mathit{as}\ \mathit{vpn}
\end{aligned}
$$

Note, the TLB validity invariant (Invariant I4.2) ensures that there are no conflicting TLB entries. This implies that at most one entry will actually match and produce a non-empty, singleton result set. Therefore, the union over all entries itself is either empty or a singleton set.

Using a similar approach, the `MIPSTLB_try_translate` function can be defined which probes the TLB, but instead of translating the address, it returns whether the address translation would be successful, or if the address translation would cause an exception. Note the distinction between odd and even VPNs. The predicate `TLBHasMatchingEntry` evaluates `EntryMatchVPNASID` on all entries of the TLB.

```
MIPSTLB_try_translate :: MIPSTLB → ℕ → ℕ → TLBEXN
MIPSTLB_try_translate tlb as vpn =
  if TLBHasMatchingEntry vpn as tlb then
    if even vpn ∧ EntryIsValid0 vpn as tlb then
      EXNOK
    else if odd vpn ∧ EntryIsValid1 vpn as tlb then
      EXNOK
    else EXNINVALID
  else EXNREFILL
```

4.6.3.2 Replacement Handlers and Page Tables

As described earlier, the translation lookaside buffer caches address translations which are defined based on some hardware or software defined data structure such as a page table.

In the case of a software-loaded TLB, the operating system may implement any suitable data structures to manage and keep track of the translations. Logically, this data structure is an array of `TLBEntryLo` values indexed by address space identifier and virtual page number. The TLB model expresses this as the following function:

$$MIPSPT :: \mathbb{N} \rightarrow \mathbb{N} \rightarrow TLBENTRYLO$$

While the MIPS R4600 TLB fits well with a linear array as page table, in general n operating system or hardware implementation is free to chose other data structures than an array or function representation to save memory in sparsely populated address spaces. An example of a data structure used today is a multi-level radix trees e.g. x86 [Int19a] or Arm [ARM17]

page table. Complex data structures but require a refinement proof to the logical array used in the model.

When an exception occurs, the handler routine consults the page table to obtain the translation information, it then creates the TLB entry-pair for the faulting address. The TLB validity invariant (Invariant I4.2) requires entries to be well-formed. The following Lemma L4.3 proves that every TLB entry constructed from the page table in-memory representation using the `MIPSPT_mk_tlbentry` function results in a well-formed TLB entry-pair. This satisfies one part of Invariant I4.2.

Lemma L4.3 (well-formed entry pair construction)

assumes `MIPSPT_valid` *pt* **and** `ASIDValid` *as*
and $vpn < MIPSPT\_EntriesMax$
shows `TLBENTRYWellFormed` (`MIPSPT_mk_tlbentry` *pt as vpn*)

The TLB is always populated from the contents of a page-table representation. The combination of the page-table representation with the MIPS TLB then allows modeling a replacement handler that populates the TLB with translations defined in the page table. The Isabelle/HOL record *MipsTLBPT* expresses this combination:

$$MipsTLBPT = \left(tlb : MIPSTLB, \quad pte : MIPSPT \right)$$

By constructing TLB entry-pairs from the page table, the TLB should never have conflicting translations with the page table structure. In fact, this is what the operating system maintains using TLB invalidation when it modifies the page table. In other words, the TLB should therefore always be an "instance" of the page table holding a subset of the translations defined by the page table. Invariant I4.3 ensures that every entry in the TLB has been constructed from the page-table representation.

Invariant I4.3 (TLB instance)

```
MipsTLBPT_is_instance mt =
    ∀i < capacity (tlb mt).
      entries (tlb mt) i = MIPSPT_mk_tlbentry (pte mt)
          (asid (hi (entries (tlb mt) i)))
          (vpn2 (hi (entries (tlb mt) i)))
```

The possible translation function of the TLB is therefore always a subset of the translation function defined by the page table. The replacement handler can chose to update entries in the TLB either deterministically, e.g. based on some function of the entry's VPN or ASID, or non-deterministically using some random replacement strategy.

Deterministic Replacement Deterministic choice of an entry to update effectively implements a direct mapped cache: for each entry there is exactly one well-defined slot (index) in which it can be placed. This simplifies reasoning about possible conflicts and the TLB invariant. The `MIPSTLBIndex` is one possible way to define the index:

$$\texttt{MIPSTLBIndex}\ tlb\ entry\ =\ (\texttt{vpn2}\ (\texttt{hi}\ entry))\ mod\ (\texttt{capacity}\ tlb)$$

Two potentially conflicting entries will end up at the same location in the TLB and hence always replace potentially conflicting entries, and thus preserving Invariant I4.2.

Non-Deterministic Replacement The deterministic replacement strategy, however, is not always applicable. In particular, when dividing the entries into wired and random, or in the presence of a hardware page-table walker, which may update any entry of the TLB. Moreover, this strategy

may experience more conflict misses than other replacement strategies such as least-recently used (LRU).

This requires a non-deterministic model of a replacement-handler function, which can update any entry (more specifically any entry $wired \leq i < capacity$). Consequently, this needs a proof that this replacement handler does not break Invariant I4.2 by inserting the same entry twice into the TLB at two different locations (or conflicting with an entry in the wired part, see Figure 4.9). In other words, the TLB must only ever be updated if there is no matching entry and hence any translation attempt would trigger a refill exception, otherwise the same state is returned:

```
MipsTLBPT_fault mtlb as vpn =
    if MIPSTLB_try_translate (tlb mtlb) as vpn = EXNREFILL
      then  MipsTLBPT_update_tlb mtlb as vpn
      else  { mtlb }
```

The definition of `MipsTLBPT_fault` ensures that conflicts are never caused and the TLB invariant is preserved. Lemma L4.4 proves this statement: the lemma states that if the TLB is in a valid state, the ASID and the VPN of the address to be faulted on are valid, then updating the TLB state using the `MipsTLBPT_fault` function preserves Invariant I4.2 and Invariant I4.3.

Lemma L4.4 (Fault handler preserves Invariant)

assumes `MipsTLBPT_valid` *mpt* **and** `ASIDValid` *as*
and *vpn* < `MIPSPT_EntriesMax`
shows ∀*m* ∈ `MipsTLBPT_fault` *mpt as vpn*. `TLBPT_valid` *m*

The validity invariant used in this lemma is actually stricter than just Invariant I4.2. It also includes the well-formedness of the page tables as well as Invariant I4.3 which links the page tables and the TLB together.

$$\texttt{TLBPT\_valid}\ mt = \texttt{MIPSPT\_valid}\ (\texttt{pte}\ mt) \wedge \texttt{TLBValid}\ (\texttt{tlb}\ mt) \wedge \texttt{MipsTLBPT\_is\_instance}\ mt$$

4.6.3.3 Equivalence to Infinitely Large TLB

The combination of the models for translations, exceptions and refill handlers enables the implementation of the expected abstraction of a single address space that translates virtual to local physical addresses as defined by the contents of the page table. This is as if there is a hypothetical large TLB capable of "caching" the entire page table.

The equivalence of a small TLB plus a refill handler, and a hypothetical large TLB pre-loaded with all mappings requires a proof. The model of the hypothetical large TLB assumes it can hold all address translations at once, and consequently a refill exception can never happen. Populating the large TLB requires a deterministic placement scheme using the "extended virtual address", a combination of the ASID and the VPN. There is a well-defined correspondence between the extended virtual address and the TLB index:

$$tlb\_index \longleftrightarrow (asid, vpn)$$

Using this extended addressing scheme, which gives a unique, deterministic location for each entry, the `MipsTLBLarge_create` function pre-populates the entire TLB with the contents of the page table:

$$\texttt{MipsTLBLarge\_create} :: MIPSPT \rightarrow MIPSTLB$$
$$\texttt{MipsTLBLarge\_create}\ pt =$$
$$\big(\ capacity = \texttt{MaxEntries},\quad wired = \texttt{MaxEntries},$$
$$entries = \lambda n.\ \texttt{MIPSPT\_mk\_tlbentry}\ pt\ (\texttt{i2asid}\ n)\ (\texttt{i2vpn}\ n)\ \big)$$

A large TLB initialized in this way will never experience a refill exception when translating any valid ASID or VPN combination. Lemma L4.5 proves this statement.

Lemma L4.5 (no refill exceptions)

assumes $\texttt{MIPSPT\_valid}\ pt$ **and** $\texttt{ASIDValid}\ as$
and $vpn < \texttt{MIPSPT\_EntriesMax}$
and $mtlb < (\texttt{MipsTLBLarge\_create}\ pt)$
shows $\texttt{MIPSTLB\_try\_translate}\ mtlb\ as\ vpn \neq EXNREFILL$

Lastly, Lemma L4.6 proves that the hypothetically large TLB and the small TLB plus replacement have precisely the same translation behavior i.e. they produce the same output PFN for a given VPN-ASID combination.

Lemma L4.6 (large TLB equivalence)

assumes $vpn < MIPSPT\_EntriesMax$ **and** $as < ASIDMax$
and $\texttt{capacity}\ (\texttt{tlb}\ mpt) > 0$ **and** $\texttt{MipsTLBPT\_valid}\ mpt$
shows $\texttt{MipsTLBPT\_translate}\ mpt\ as\ vpn =$
$\texttt{MipsTLBLarge\_translate}\ (\texttt{pte}\ mpt)\ as\ vpn$

4.6.4 The TLB Refines a Decoding Net

The previous sections define the operational model of the MIPS R4600 TLB including a page-table definition, refill handlers and the equivalence to a hypothetical, large TLB. This section shows that this operational model of the MIPS R4600 TLB *refines* the *Decoding Net* model of a translate-only node under an appropriate lifting function.

Lifting Function The entries of the TLB define the translating behavior of the resulting *Decoding Net* node. In the MIPS R4600, each entry-pair contributes up to two address-range mappings to the translation function. An entry-pair matches the input virtual address *va* against the first or second half of the virtual address range covered by the page identified by the entry-pair's VPN, and translates it if the corresponding `EntryLo` is valid, otherwise resulting in the empty set. The `EntryToMap` function returns a function from address to a set of names for a given TLB entry-pair. The node ID of the local physical address space of the processor is passed as an argument. Note, this function flattens the two dimensional (ASID, VPN) representation, by extending the virtual address with the ASID.

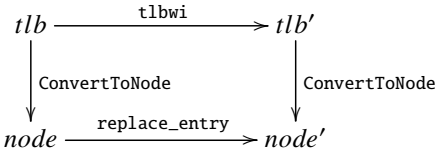
$$
\begin{aligned}
&\texttt{EntryToMap} :: \mathit{nodeid} \Rightarrow \texttt{TLBENTRY} \Rightarrow (\mathit{addr} \Rightarrow \{\mathit{name}\})\\
&\texttt{EntryToMap}\ n\ e\ va =\\
&\quad (\textbf{if}\ \texttt{EntryIsValid0}\ e \wedge va \in \texttt{EntryExtendedRange0}\ e\\
&\qquad \textbf{then}\ \{(n, \texttt{EntryPA0}\ e + (va \bmod \mathit{VASize}) - \texttt{EntryMinVA0}\ e)\}\\
&\qquad \textbf{else}\ \{\}) \cup\\
&\quad (\textbf{if}\ \texttt{EntryIsValid1}\ e \wedge\ va \in \texttt{EntryExtendedRange1}\ e\\
&\qquad \textbf{then}\ \{(n, \texttt{EntryPA1}\ e + (va \bmod \mathit{VASize}) - \texttt{EntryMinVA1}\ e)\}\\
&\qquad \textbf{else}\ \{\})
\end{aligned}
$$

The `ConvertToNode` function lifts the entire TLB representation by taking the union of all translation functions obtained by applying `EntryToMap` on

all TLB entries. The TLB itself does not have resources that the processor can address using load/store operations (recall, the entries of the TLB are accessed using special instructions). Therefore, there are no resources that may accept memory accesses and the resulting accept-set of the *Decoding Net* node lifted from the TLB model is empty.

$$\begin{aligned}&\texttt{ConvertToNode}\ n\ tlb = \\ &\quad (\ accept = \{\}, \\ &\qquad translate = \lambda a.\ \bigcup \texttt{EntryToMap}\ n\ (\texttt{entries}\ tlb\ i)\ a\ )\end{aligned}$$

Refinement Proof The following commutative diagram illustrates the schema of the refinement proof showing that updating an entry in the TLB then converting it to a node results in the same state as first converting it and then applying the corresponding operation on the abstract level.



On the abstract level, the equivalent to the `tlbwi` operation is the function `replace_entry`. This function replaces entry $e1$ with $e2$ by removing the translation region of entry $e1$ and adding region $e2$:

$$\begin{aligned}&\texttt{replace\_entry} = \texttt{translate}\ n\ a \mapsto \\ &\qquad (\texttt{translate}\ n\ a\ - \texttt{EntryToMap}\ n\ e1\ a) \\ &\qquad\quad \cup\ \texttt{EntryToMap}\ n\ e2\ a\end{aligned}$$

The following lemma proves the refinement by establishing that the indexed write operation (`tlbwi`) and the `replace_entry` commute with the lifting

function resulting in the same state under the assumption that the TLB is in a valid state and that the entry can actually be written i.e. it is well-formed and does not conflict with the rest of the entries in the TLB.

Lemma L4.7 (TLB refines *Decoding Net*)

assumes $i <$ `capacity` tlb **and** `TLBValid` tlb
and `TLBEntryWriteable` $i\ e\ tlb$
shows ((`ConvertToNode` n)$'$(`tlbwi` $i\ e\ tlb$)) =
(`replace_entry` n (`entries` $tlb\ i$) e
(`ConvertToNode` $n\ tlb$))

Refinement of the Large TLB Section 4.6.3.3 has already shown the equivalence of the large TLB and the TLB plus replacement handler under the operational model. What is left to do, is to show the equivalence of the combination of the TLB with the page table and replacement handler, and the hypothetical, large TLB under the abstract model. For this proof Lemma L4.8 compares the result of conversions to the *Decoding Net* node.

Lemma L4.8 (Equivalence under the abstract model)

assumes `capacity` (`tlb` mpt) > 0 **and** `MipsTLBPT_valid` mpt
and mlg = `MipsTLBLarge_create` (`pte` mpt)
shows `MipsTLBPT_to_node` $nd\ mpt$ =
`MipsLarge_to_node` $nd\ mlg$

Recall, Lemma L4.6 has already established the equivalence of the `translate` function of the two TLB representation. Consequently, applying the lifting

function on both TLB representation produces equivalent nodes in *Decoding Net* semantics.

4.6.5 Specification Bugs

The MIPS R4600 TLB manual defines certain states and behaviors which turn out to be problematic when attempting to formally model the TLB. This section presents two instances where, for instance, Invariant I4.2 is violated at reset, and the random write operation performs unexpectedly.

Those specification bugs are an example of an excessively cautious abstraction hiding correctness-critical details of the underlying hardware. In the case of the MIPS R4600 TLB this is most likely harmless. However, similar instances of hiding behavior behind an abstraction lead to correctness and security critical vulnerabilities as demonstrated by the Meltdown and Spectre attacks [Koc+18; Lip+18].

4.6.5.1 Invariant Violation at Power On

Lemma L4.1 and Lemma L4.2 state the necessary preconditions the operating system must satisfy when updating entries in order to preserve Invariant I4.2. This requires the TLB to be in a valid, well-known state to start with.

However, at reset (e.g. after the chip is powered on) the guarantees provided by hardware on the state of the TLB are too loose according to the specification of the power-on state. This effectively renders satisfying the invariant at all time impossible. The following quote from the MIPS R4600 manual [ITD95] describes the reset state as:

> *"The Wired register is set to 0 upon system reset. [...] The TLB may be in a random state and must not be accessed or referenced until initialized."* – MIPS R4600 manual [ITD95]

Therefore, the state of the TLB entry-pairs is undefined after reset. This would not be a problem, if the TLB could be switched off, but this is not the case. The MIPS R4600 TLB is always on. A strict reading of the invariant demands that there are no two conflicting entries in the TLB, even if they are invalid. Consequently, it cannot be guaranteed that the invariant is satisfied at any time because of the unpredictable and undefined initial state. This despite the kernel being provided a special, untranslated segment (`kseg0`) for bootstrapping.

The proof that there is in fact a state that violates the invariant while satisfying the reset condition at the same time is done by constructing a plausible initial state and evaluate the invariant and the at-reset-predicate on it. An example of such a plausible initial state has all entries zeroed out i.e. being the `null_entry`:

$$\texttt{tlb\_at\_reset} = (\text{wired} = 0,\ \text{random} = 47,\ \text{capacity} = 48,\ \text{entries} = \lambda\_.\ \texttt{null\_entry})$$

The valid bits of all entries are zero and hence this TLB does not actually translate any address. However, addresses within the first page of memory e.g. null pointer dereferences, a common programming bug, will match all entries of the TLB. Recall, the specification demands that this situation does never occur even if the addresses are never issued.

Based on experiences from practice, operating systems *do* successfully initialize and run on MIPS processors. This suggests that the obvious approach is likely also a correct one: there is no real problem as long as no two entries actually match the addresses issued by the processor. This is achieved by the operating system running in a non-translated window (`kseg0`) until the TLB is configured properly. However, an unintended memory access or dereferencing a wrong address might be enough already to cause an address translation using the invalid TLB state.

4.6.5.2 Behavior of a Fully-Wired TLB

The MIPS R4600 TLB has a concept of *wired* entries. An entry-pair that is wired cannot be overwritten by the random write operation (`tlbwr`) as stated in the manual:

> *"Wired entries are non-replaceable entries, which cannot be overwritten by a TLB write random operation."* – MIPS R4600 manual [ITD95]

The number of wired entry-pairs w can be configured, such that the lower w entries are wired. This is illustrated in Figure 4.9. The entry-pairs are divided into two distinct sets:

$$\begin{aligned} wired &= \{\, i \mid i \geq 0 \wedge i < w \,\} \\ nonwired &= \{\, i \mid i \geq w \wedge i < \texttt{capacity} \,\} \\ wired \cup nonwired &= \{\, i \mid 0 \leq i < \texttt{capacity} \,\} \end{aligned}$$

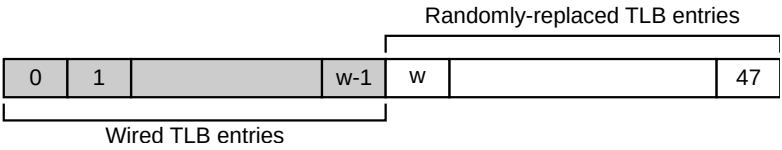


Figure 4.9: Illustration of Wired TLB entries as in [Ach+18].

The value of w indirectly defines the entry-pair to be updated using the random write operation: the value of the `Random` register selects the entry-pair to be updated. The range of values the `Random` register may assume is defined by `RandomRange` function below.

At reset, i.e. power on or updating w, hardware sets the `Random` register to the highest possible value `capacity` − 1. Whenever the processor retires

an instruction the value of the Random register is decremented until value w is reached, after which a wrap around happens and the value is set to capacity $-$ 1 again. The entries $w - 1$ down to 0 are effectively skipped. This RandomRange is defined as:

$$\texttt{RandomRange}\ tlb = \{x.\ \texttt{wired}\ tlb \leq x \wedge x < \texttt{capacity}\ tlb\}$$

Because the reset value is set to the entry-pair with the highest possible index (capacity $-$ 1), the random write operation will always succeed in updating an entry-pair, regardless of the value of w

The corner case, where all entries are wired (w = capacity or even $w \geq$ capacity which is not prevented by hardware) renders this definition problematic. The manual does not specify what happens in this case. With w = capacity the RandomRange definition yields an empty set:

$$\texttt{RandomRange}\ tlb = \{x.\ \texttt{capacity}\ tlb \leq x \wedge x < \texttt{capacity}\ tlb\} = \{\}$$

By setting w = capacity, the programmer wants all entries to be wired and none to be replaced randomly. This is in line with the empty set above.However, this conflicts with the value at reset of the Random register, which is in this case:

$$\texttt{capacity} - 1 \notin \texttt{RandomRange}\ tlb = \{\}.$$

This results in a contradiction: either the semantics of the wired entries, or the random write operation is wrongly specified in the manual. One way to work around this is to slightly adjust the definition of the RandomRange set in the operational model:

$$\begin{aligned}\texttt{RandomRange}\ tlb = \{x.\ \texttt{wired}\ tlb \leq x \wedge x < \texttt{capacity}\ tlb\} \\ \cup \{\texttt{capacity}\ tlb - 1\}\end{aligned}$$

As an alternative, one could interpret the behavior of a random write in a fully-wired TLB as undefined behavior, e.g. when the number of wired entries exceeds the capacity of the TLB. Moreover, the manual assumes the random register to always have a valid index.

Ultimately, experimentation will reveal the actual implemented behavior.

4.6.6 Comparison to an ARMv7 TLB Model

The TLB of the MIPS R4600 is software-loaded and thus its state is in control of the operating system. Architectures such as x86, ARMv7 and ARMv8 accelerate the handling of TLB misses by walking the page table using dedicated hardware. This section compares the model of the MIPS R4600 TLB presented in this chapter with a model of the ARMv7 memory management unit and its TLB [SK17; SK18]. Their model uses a state monad to express the TLB state and integrates with the Cambridge ARMv7 model [FM10] to express memory accesses originating from the hardware page-table walker. In contrast to the MIPS R4600 TLB model, there are additional sources of non-determinism, as the ARMv7 TLB can evict and replace entries with an unspecified replacement policy at any memory read or write. Using data refinement, they remove non-determinism from the model using an explicit evict-operation. The MIPS R4600 TLB model also supports deterministic replacement of entries using a predefined location based on ASID and VPN. Both models abstract away the TLB state showing equivalence to a saturated TLB. The MIPS R4600 TLB model does not take partial page table walks into account.

4.7 Conclusion

This chapter presented a sketch of the address space model, a new abstraction for resource management and address decoding in the presence of multiple address spaces following the principles of Saltzer's work on

Naming and Binding of Objects [Sal78]. Secondly, *Decoding Net* formally defines the semantics of the address space model in Isabelle/HOL. This model enables reasoning about the increasingly complex process of memory address resolution and translation in modern systems.

The *Decoding Net* model is capable of expressing the complex memory subsystems of desktop computers, rack-scale systems, phone SoCs or experimental hardware. This provides the foundation of a semantically rich system description, capable of faithfully representing the memory topology of heterogeneous platforms. The concrete syntax of the model together with the presented algorithms form the basis for system software implementations. Examples include the Sockeye description language [Bar17b; Sch17] which implements a syntax similar to the presented concrete syntax. The Sockeye compiler translates system descriptions into a Prolog representation which implements the core model and transformation algorithms providing a way to query the model during compilation and runtime of the operating system (Section 6.3).

Section 4.6 of this chapter presented a detailed, operational model of the MIPS R4600 TLB as an instance of a complex memory translation hardware. The model precisely defines the state of the TLB, its invariants and update operations. The proofs presented in this chapter show that the operational TLB model refines the abstract *Decoding Net*. Moreover, there is an equivalence between the combination of a TLB, a page-table structure and a refill handler and an hypothetical, large TLB holding all page-table entries. This equivalence is verified using the model sin Isabelle/HOL.

As a side-benefit of the modeling effort, the specification bugs revealed in this study clearly demonstrates the benefit of a rigorous formal model with accompanying proofs of the hardware semantics.

The presented model effectively extends to virtual memory systems present in today's processors. The memory management unit effectively creates a new, per-process address space with mapping regions. However, this poses another question on whether virtual memory in its current form is still a viable form for translation and protection [Ach+17a]. Moreover, the virtual memory is not free and misses in the translation lookaside buffer are

expensive [Bas+13]. This is, however, orthogonal to memory addressing and out of the scope for this thesis.

At its current form, the model is able to express a *static* configuration of the system encoding basic reachability of memory resources. However, not all memory requests are equal: properties such as read, write, cacheable, non-temporal stores etc. can result in different translation outcomes which are not expressed in the current model. Moreover, the configuration is not static, but rather re-configured at a regular basis. In the current form, the model does not express the configuration space as well as the authority needed to change the translation behavior. This is exactly what the next chapter is about.

5

Dynamic Decoding Nets

The *Decoding Net* model presented in the previous chapter statically expresses the memory decoding and address translation configuration of a system. The formally specified model provides a sound basis with well-defined semantics for reasoning about the memory address resolution. The *Decoding Net* is a *static* representation of the system configuration at a particular point in time. Hardware, on the other hand, is *configurable*: software can alter the configuration by writing to particular registers or specific in-memory data structures. Even the hardware components themselves can change. Device discovery, hot-plugging [PCI13] add or remove hardware components during the runtime of a system. Power constraints reduce the number of devices that can operate simultaneously by temporarily shutting off hardware [Esm+11].

A machine is a *configurable* network of address spaces. This chapter extends the static *Decoding Net* model with a notion of configurable address spaces and authority. Two important questions regarding changing the configuration of an address space are:

- What is the set of valid configurations of an address space?
- What is the required authority to change the configuration of an address space?

The extension of configurable address spaces and authority then forms the basis for managing memory resources and address-space configuration within an operating system for modern, heterogeneous computing platforms. Through systematic refinement and with the combination of an executable specification, the dynamic *Decoding Net* model extension provides the guidance of an implementation in operating systems.

5.1 Motivation

Recall, the physical memory abstractions highlighted in Chapter 3 and their discrepancy with reality:

Operating systems, from production-quality to fully-verified kernels, use a flat and array-like representation of the physical memory resource of a machine. A single physical address space contains RAM and devices. Physical addresses are similar to an offset into the array-like representation and serve as a unique identifier for physical resources. All processor cores and devices of a system have a uniform, homogeneous view of the physical address space. Processes run in their own virtual address space where a memory management unit translates virtual to physical addresses. This translation unit is managed by an operating system. Often, this is a monolithic kernel which manages all physical resources and configures all translation units of the system.

Network of Configurable Address Spaces Unfortunately, this view of the world does not accurately represent the hardware configuration which has diverged from, or has never been aligned with, the single-address-space model. In fact, there is a mismatch between real hardware and the assumptions made by software. This includes accounting for the network of address spaces, multi-stage configurable translations, and heterogeneous cores and devices with non-uniform views of the system resources. The collection of cores and devices issue memory requests from various locations within the address spaces network, reaching different resources depending on the address space configuration and the path taken through the decoding network. Many of the address spaces of a system are *configurable*. System software is responsible for safe and correct configuration.

Configuration Complexity System software needs to safely manage and correctly program a wide variety of memory translation units and memory resources of a system. This includes understanding of valid configurations an address space can assume. This needs to hold for a variety of different hardware components. For example, a translation unit may only be able to translate naturally-aligned 4 KiB regions of memory, while another has a 16 GiB alignment requirement. Misconfiguration of such translation units happens. For instance, misusing and wrongly programming the IOMMU or System MMU [NVD18; Mor+18; Mar+19; MMT16] or incomplete setup of translation units [NVD13a] allow devices to access resources they should not be able to. Moreover, reusing virtual memory [NVD17a], or mismanagement of TLB flushes [NVD13c; NVD13b] can lead to unintended memory accesses. Overall, about 30% of code committed to the memory manager in Linux are bug fixes [HQS16].

Authority of Configuration Changes Securely managing the complete set of hardware resources present in a system (e.g. mobile phone SoC or accelerators) seems infeasible for mainstream operating systems in particular with respect to identifying the rights and authority in the system. The operating system may know how to configure a particular translation unit, but the used authority model, where a monolithic kernel has all

the required rights to decide what memory to allocate, to whom grant access to, and how to configure translation units, is another source of security vulnerabilities. Examples include a "cross-SoC" attack on the Snapdragon SoC [Gon19], bypassing IOMMU protection from GPU malware [Zhu+17], breaking through operating system isolation domains using co-processors [SWS14], or allowing the processes to access and modify their own page tables [Che18]. A single entity in the system has too many rights and is therefore capable to perform any operation.

Features like secure co-processors and system management engines present on modern platforms do not integrate well with the concept of a centralized authority over the memory subsystem. In particular the interaction between processes, operating system, device drivers, and the secure execution environment (e.g. ARM TrustZone [ARM09] or Intel System Management Mode [Int19a]) has been a source of security vulnerabilities such as overwriting memory by passing crafted pointers leading to arbitrary code execution in ARM TrustZone [NVD19b; NVD19c], accessing memory outside of the supplied memory region [NVD17c; NVD19d] and insufficient memory protection for the system management mode [NVD19a], or not providing the right information to a secure co-processor to unambiguously reference memory [NVD17b].

Summary In summary, a modern platform consists of multiple, configurable address spaces which system software needs to correctly manage. Each of these configurable address spaces has certain constraints on how they can be configured. Lastly, each configuration change requires a certain authority. Misconfiguration and missing authority checks have led and still are leading to numerous security bugs in system software.

The work presented in this chapter has therefore two objectives:

1. represent the *configuration* of an address space, and
2. express the required authority to change it.

The chapter extends the *static Decoding Net* defined in the previous Chapter 4 with a notion of least-privilege authorization, protection and configuration which captures the richness, complexity, and diversity of modern

hardware platforms. Following the methodology outlined in Section 5.3, this chapter defines the semantics of configurable address spaces and a least-privilege model of access control in three steps:

1. Express the configuration space and the dynamic behavior of a *Decoding Net* node (Section 5.4).
2. Identify and specify the required authority for changing the configuration of an address space (Section 5.5).
3. Develop an executable specification [Hos19] of the model extension (Section 5.6).

This then forms the basis for an implementation in an operating system, which is presented in Chapter 6.

5.2 Bookkeeping and Access Control

Shared-memory programming is important as it obliterates the need to copy, update and maintain consistency of data structures, allowing different threads to directly dereference a pointer to access the data structure. However, the name used by the thread (i.e. the virtual address) is only valid local to its virtual address space: the address may be different for two processes as a virtual address. This makes the use of pointer-rich data structures impractical. Moreover, sharing is hard to support as memory mappings need to be updated which requires to unambiguously name the shared resource and implement proper access control and bookkeeping mechanisms.

Address translation, virtual memory [Den70] and virtualization [PG74] in general, are a fundamental building blocks providing isolation and protection between multiple tenants in a system. In addition, the virtual memory abstraction enables the implementation of demand paging, machine virtualization, shared memory and libraries, and other functionality. This

provides the process, or, in the case of virtual machines, even the operating system, the illusion of being the only entity running on a shared system.

A process has only access to resources that the operating system has mapped into the address space of the process. Similarly, a guest operating system has only access to resources the hypervisor has mapped into the guest physical address space. Two threads of the same process can use the same virtual addresses as pointers to data structures. For example, a dispatcher thread can simply pass the pointer to the received request to the worker thread. In this model, the two threads share the same virtual address space and hence run in the same protection domain. Using multiple processes, it is not sufficient to send a pointer over an inter-process communication channel. The sender must explicitly grant access to the resource through mechanisms provided by the operating system, e.g. files or shared memory objects. Recall an address is a name which is only valid relative to its specific context. Furthermore, the operating system may decide to map the resources at a different address making pointer sharing impracticable. To avoid serialization, SpaceJMP [El +16] provides a mechanism similar to a protected procedure call to context switch into another address space. Single address space operating systems Section 3.3.2 run all applications in one global virtual address space, which makes pointer sharing between applications trivially possible.

In contrast, CleanQ [Hae+19] sends descriptors with offsets and lengths into preregistered memory regions. VirtIO [OAS18] defines the transmitted pointers to be guest physical addresses. Runtime libraries such as OpenCL [Khr18], nVidia's CUDA [NVI13] or HSA [HSA16; HSA14] try to establish the illusion of a shared-virtual address space between programs and devices by using the processor MMU and the IOMMU and demand paging mechanisms to copy data between host and device memory [VMB15].

The concept of shared access to memory resources is important, and often required for efficient communication (e.g. zero copy transfer of bulk data). The operating system needs to ensure correct operation, in particular enforcing authority by maintaining suitable access control measures for

establishing shared memory objects, correctly enable shared access to the same resource, passing pointers between user-level processes and the operating system kernel, and switching between different address spaces. This requires the existence an *unambiguous* handle to the physical resources of a system. This is similar to what single address space operating systems and single level stores provide to applications, but with taking multi-stage address translation, and different views of memory into account. Operating system kernels use physical addresses to identify resources for bookkeeping and access control, which is problematic (Chapter 2).

The Linux kernel, for instance, maintains a data structure (`struct page`) for each physical frame of 4 KiB, identified by its physical address. This data structure serves as a unit of bookkeeping and contains necessary information about the use of that memory region. A shared memory object then corresponds to a collection of those physical frames, and an access control list that specifies which users or groups can obtain access to the shared memory object. A process requests a mapping into its address space from the kernel.

In contrast, Barrelfish [Bau+09a; Ger18] and seL4 [EKE08; Kle+09] use a capability system [DV66; Lev84] to represent and manage physical memory regions and their corresponding access rights. Processes can transfer the capability between each other and then map it in their own address space by presenting the capability as a token of authority to the kernel. However, the kernel still needs to resolve the object name correctly before using the object, and name resolution may fail.

In both cases, the task of the operating system kernel is to enforce authority by refusing any requests for which the process has not sufficient rights. In capability-based systems, the process invokes an operation on a capability which represents all the authority needed for the operation. In contrast, the Linux kernel validates the request arguments with the access control list and uses the authority it has to execute the requested operation on behalf of the process.

All of these access control and bookkeeping mechanisms must work correctly in the presence of complex, configurable address space topologies.

Recall the examples from Chapter 2: the Intel Xeon Phi co-processor [Int14b] uses a "system memory page table" to access host resources. The other direction, the programmable PCI bridges map addresses in the PCI address space to device registers or apertures. Intel's Single Chip Cloud Computer [Int10b] translates a core-local, 32-bit physical address to a 46-bit "system address" based on the configuration of a 256-entry lookup table (LUT). Secondary physical address translation is commonly used in phone SoC like the NXP iMX8 [NXP19], Texas Instruments OMAP 4460 [Tex14], and NVIDIA Parker [NVI17] processors. The use of separate processors, overlapping and intersecting address spaces, firewalls and secondary address translation schemes are a deliberate design choice: secure co-processors holding encryption keys, for instance, are shielded from the main application processors.

Wrongly configured firewalls and IOMMUs result in bugs and vulnerabilities due to failure to provide protection from malicious memory accesses [Mor+16; Mor+18; MMT16; Mar+19]. This is likely to become worse by integrating IOMMU designs into GPUs, co-processors, and intelligent NICs (e.g. [Mel17]). Consequently, the overall complexity of software increases making it harder to write correct software, especially without sound and well-defined representation of the hardware and authority model.

Emerging technologies and algorithms for "in-memory" or "near-data" processing [Pat+97] raise further questions for usable operating systems abstractions and mechanisms for resource management and authorization [Bar+17]. This likely increases the complexity of software which in turn becomes more prone to errors.

5.3 Methodology

The *Decoding Net* model presented in Chapter 4 provides a sound foundation for expressing memory address decoding in a system. Recall, the

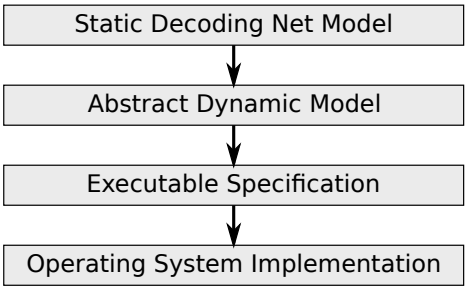


Figure 5.1: Illustration of the Refinement Steps.

Decoding Net captures a *static* configuration. Consequently, the following two important features are not defined in the *Decoding Net* model:

1. Dynamic configuration of the `translate` function of the *Decoding Net* nodes. This is important to capture the reconfiguration of real translation units.

2. The *rights and authority* which are required for software processes to change the configuration of the `translate` function.

Note, Section 4.6 presented a refinement of the *Decoding Net* model to the MIPS R4600 TLB model. This included the semantics of updating a TLB entry with respect to the configuration of the *Decoding Net* node. The MIPS R4600 TLB model does not express the possible configuration space and the authority to change a TLB entry, which is what this chapter is about.

The methodology applied in this chapter is strongly influenced by the seL4 project which combined the principle of *refinement* with the development of an *executable specification* (Figure 5.1). This allows for rapid prototyping of the model using real operations. The choice of applying a proven methodology is deliberate: it provides confidence that the resulting artifact is likely to be compatible with an seL4-style verification proof. Moreover, it

could be used as a more accurate and faithful replacement for the hardware abstraction model used in the seL4 and CertiKOS proofs.

1. Abstract Dynamic Model This refinement step introduces an abstract dynamic model on top of the *Decoding Net* model. This includes defining the dynamic state (Section 5.4) and the associated rights and authority to perform state transitions (Section 5.5). Specifying rights in a system requires the identification of all relevant *objects*, *subjects* present in the system, and which *authority* each subject can exercise over an object. For example, a pager process maps a frame to a virtual address by writing an entry into a page table. An *access-control matrix* [Lam74] is the standard representation of authority in systems [Lam74]. It expresses the relationships between subjects (rows), objects (columns) and the corresponding authorities (cells) a subject S_i has ob object O_j in cell $M[i][j]$. The access control matrix represents the high-level *security policy* or the system's integrity (Section 5.5). A correct implementation of a system must adhere to this integrity policy which requires a refinement proof all the way down to the executable binaries (as done in seL4 [Sew+11]).

2. Executable Specification Following the methodology applied in the seL4 verification project [CKS08], an executable specification of the model [Hos19] bridges the gap between the abstract dynamic model and the operating system implementation. The executable specification serves as a tool to express the behavior of state transitions and dynamic reconfiguration of the model extension. The executable specification expresses the identified subjects, objects and authority as first-class objects in the Haskell functional programming language. This in turn enables rapid prototyping while enforcing strong formal semantics amenable for formal verification at the same time. Note, the correspondence between abstract and executable models is thus far by inspection and careful construction. A rigorous formal proof is part of future work of this thesis.

3. Operating System Implementation The executable specification then serves as a guideline in the development of high-performance system software implementations. Chapter 6 describes a particular implemen-

tation based on an extension of the Barrelfish research operating system [Bau+09a] which is then evaluated in Chapter 7. This step follows the precedent set by Winwood *et al.* [Win+09] showing that the abstract model can be implemented efficiently in a real system.

5.4 Expressing Dynamic Behavior

The *Decoding Net* model represents a static snapshot of the *current* system state. In reality, the system state is inherently dynamic: System software constantly changes the configuration of MMUs, and other translation and protection units in response to requests from applications. This section introduces the notion of configurable address spaces as a layer on top of the static *Decoding Net* model.

In principle, it would be possible to extend the *Decoding Net* with the notion of dynamic behavior directly. This, however, would trigger a ripple effect resulting in the adaption and re-verification of many existing proofs. Instead, the model extension adds an abstraction layer *on top* of the *Decoding Net* model. This abstraction layer expresses configuration state as a set of dynamic *address spaces*.

Each of these address spaces corresponds to a static node in the *Decoding Net*. The `configuration` function

$$\texttt{configuration} :: \textit{address space} \rightarrow \textit{node}$$

assigns to each address space a static *Decoding Net* node which corresponds to the current, active configuration. This function effectively encodes the state of the address spaces and lifts the address space model to the *Decoding Net* model.

The Configuration Space Conceptually, an address space can have a fixed or configurable translation function, or a set of physical resources

such as memory or device registers. A *Decoding Net* node is able to capture all of this, except the dynamic aspects.

Whenever the translation behavior of an address space changes, the current configuration state transitions, and with it the *Decoding Net* node of that address space. In principle, this also applies in response to a change in the availability of physical resources. In general, hardware imposes constraints on valid translation settings or the availability of physical resources. For example, a page-table translating at 4 KiB granularity would only allow contiguous, naturally aligned mappings of 4 KiB in size, and likewise a DRAM module accepts exactly 64 GiB worth of addresses.

Each address space therefore has a well-defined *configuration space* listing all *possible*, hardware-supported configurations the address space may assume. The `config_space` function

$$\texttt{config\_space} :: \textit{address space} \rightarrow \{\textit{node}\}$$

defines the configuration space of an address space as a set of static *Decoding Net* nodes. The configuration of the system is valid, if for all address spaces the configuration is within the configuration space (Invariant I5.1).

Invariant I5.1 (Valid Configuration)

$$\forall a.\ \texttt{configuration}\ a \in \texttt{config\_space}\ a.$$

The configuration space, therefore, expresses the possible system states which in turn may be further reduced to the set of *allowable* system states that adhere to a specified security property, e.g. the state of the access control matrix (Section 5.5).

State Transitions Changing the mapping behavior of an address space then corresponds to a state transition from the current configuration C_n to the next C_{n+1}. An example for this state transition operation is `ModifyMap`, which changes the translation function of an address space:

$$\texttt{ModifyMap} :: (\mathit{name} \rightarrow \mathit{name}) \rightarrow \mathit{configuration} \rightarrow \mathit{configuration}$$

In this case, `ModifyMap` updates the current configuration of the system state by changing where a name maps to as defined by the first argument, and returns a new configuration.

Note, there is no distinction between map and unmap, these are both handled by the `ModifyMap` operation. Conceptually, unmapping is expressed by modifying the mapping to point to the *null space*, which invalidates any existing translation at that name.

Any update to the configuration that starts as valid, must remain valid (Equation 5.1). For example, all translating blocks of addresses must be naturally aligned.

$$\begin{aligned} &\textbf{assumes}\ \texttt{ConfigValid}\ C_n \\ &\quad\textbf{shows}\ \texttt{ConfigValid}\ (\texttt{ModifyMap}\ (a \rightarrow a')\ c_n). \end{aligned} \tag{5.1}$$

5.5 Authority

Recent vulnerabilities in the wireless stack of the Qualcomm SnapDragon 835 and 845 chips [NVD19e; NVD19f; NVD19g] allowed the WiFi co-processor to request arbitrary sized regions to be mapped in the System MMU [Gon19]. This results in the wireless stack gaining access to any system memory region and possibly transmitting sensitive data from the mobile phone. This is by far not an isolated instance [Mar+19; MMT16; Mor+16]. The Snapdragon example illustrates two things:

1. There is an *authority* that can change the view of an address space.
2. The WiFi co-processor succeeded in getting access to memory it should not be able to.

In this case, the Linux kernel trusted the mapping request coming from the wireless stack and performed the operation. This scenario is an instance of the confused-deputy problem [Har88].

Identification of Objects and Rights To analyze what went wrong in the SnapDragon example, consider Figure 5.2 showing a simplified setup with three address spaces. This scenario is common for a virtual machine setup with a two-stage translation scheme. It serves as a basis for investigating the involved rights to change the configuration of an address space.

Assume, a hypervisor managing the intermediate address space (or guest physical address space in Intel terminology) wants to update its configuration by mapping a region of the physical address space into the intermediate address space. This corresponds to making the region in the physical address space *accessible* from the intermediate address space.

Applying this change requires certain rights: First, the *map* right expresses the right to change the meaning of an address in the intermediate address space, i.e. update where this address maps to. Second, the *grant* right expresses the right to grant access to some region in the physical address space i.e. installing a mapping that points to this region or a subset thereof.

Right R1 (Grant)
The right to insert *this* object into *some* address space

Right R2 (Map)
The right to insert *some* object into *this* address space

Observing this scenario, there are two objects involved in this scenario: *i)* a part of the physical address space which should be mapped into the

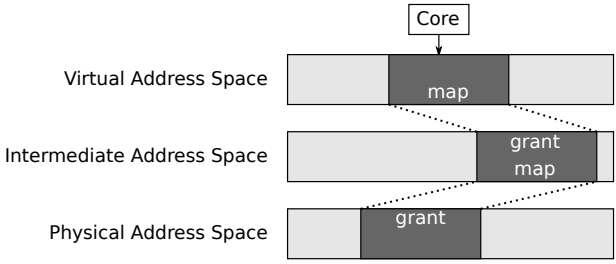


Figure 5.2: Mappings Between Address Spaces Showing Grant and Map Rights of Mapped Segments.

intermediate address space, and *ii)* a region of the intermediate address space for which its mapping configuration is to be changed.

Note, this extends to all address spaces. For instance, as shown in Figure 5.2, the virtual address space of a process can be interpreted as an address space for which *nobody* has a grant right. Likewise, *nobody* has the map right to a physical address space containing memory resources.

Real-World Example Figure 5.3 illustrates the simplified address space structure a real-world example. The system consists of a host processor, some RAM and two PCI Express devices, a network card with a DMA engine and a Xeon Phi co-processor. The co-processor uses its many cores for handling network traffic it receives on various queues (1). This requires a shared mapping of a receive-buffer between the co-processor cores and the DMA engine of the network card. This buffer is allocated in the GDDR memory of the co-processor (2).

The receiver process 'owns' the buffer in GDDR and it can control the network card's send and receive queues. Consequently, the receiver process has the right to temporarily *grant* the network card's DMA core access to the buffer. However, it does not have the right to modify the IOMMU (4) address space that translates memory accesses originating from the

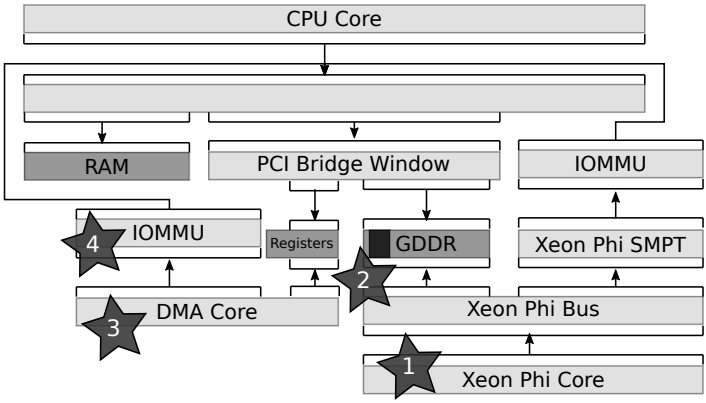


Figure 5.3: Address Spaces in a System with Two PCI Devices

DMA core (3). This is in line with the security specification: a general user-space process should not be allowed to change a translation. The receiving process does not possess the required *map* right.

To install this mapping in the IOMMU address space, there must be an agent (or *subject* in standard authority-control terminology) which possess both, the *grant* right on the buffer *object* in the GDDR and the *map* right of the IOMMU address space *object*.

The Access Control Matrix To perform the map operation of the example above, system software needs to represent those rights. The access control matrix is the standard representation of authority in systems [Lam74]. It lists the subjects (rows) and objects (columns) in the system and states the rights a subject S_i has ob object O_j in cell $M[i][j]$.

Building the access control matrix requires therefore the identification of subjects, objects and authorities of a system to populate the rows and columns of the matrix. This process follows the principle of least-privilege and fine-grained decomposition.

subject/object	DMA IOMMU	buffer
IOMMU driver	**map**	
Xeon Phi process		**grant**

Table 5.1: Access Control Matrix of the Xeon Phi Example.

Table 5.1 illustrates the access control matrix of the real-world example above. There are two subjects: the IOMMU driver which manages the configuration of the IOMMU, and the process running on the Xeon Phi co-processor which owns the receive-buffer.

The access control matrix can grow large, which many empty cells and thus implementing the full access control matrix is impractical [Lam74]. The table can be projected in two ways:

- Row-order: This represents the *capabilities* each subject has. In the example, the IOMMU driver has the *map capability* to the IOMMU address space, and the receiving process has the *grant capability* to the buffer.

- Column-order: This represents the access-control lists of the objects. The IOMMU allows mapping requests from the IOMMU driver, and the buffer records a *grant* permission for the process.

In monolithic system software architectures, the two rights (map and grant) are implicitly held by the kernel. Processes make requests which prompt the kernel to exercise those rights on behalf of the subjects. The kernel needs to validate the request and maintain accurate bookkeeping to conclude the action is safe to execute (which was not the case in the Snapdragon example). Access-control lists (ACLs) are typically used for this matter, where the ACL of an object lists the rights subjects have on it.

In microkernel-based systems, many system services such as memory manager, pager, or device drivers run in dedicated user-level processes, which need the rights to the *relevant* resources in order to provide a useful service (e.g. device registers) and address spaces (e.g. the IOMMU configuration). At the same time, the set of rights should be minimal e.g. a device driver must not interfere with the hardware registers of an unrelated device. In other words, the principle of least privilege strongly influences the design of system services. Capabilities are a natural way to express authorization in this context.

From an access control perspective, there is a duality between the two forms. However, they are not strictly equivalent. The obvious difference lies in the way they are implemented. Moreover, access control lists do not permit strict implementation of least privilege and are vulnerable to confused deputy problem [Har88].

Security Property The access control matrix above is the used to define the correct (and secure) state of a system. If the system's current configuration is consistent with the access control matrix, then it is correct (secure) *statically* – or in a secure state for short.

Likewise, the system is *dynamically* secure if it *is* in a secure state and for any possible state transition (i.e. change of the translation behavior of an address space), the system remains in a secure state.

Transfer of Rights The security property reduces the configuration space of the address spaces. Only state transitions that do not violated the security properties are permissible. In other words, without the needed rights there is no change in configuration as the target configuration does not exist in the configuration space. The IOMMU address space of the example has precisely one *Decoding Net* node in it, corresponding to the current state. By transferring the grant right from the Xeon Phi process to the IOMMU driver, the secure state of the system is updated, which alters the configuration space of the DMA IOMMU address space accordingly. Therefore, the IOMMU driver now has the needed rights to perform the mapping and the state transition remains valid. Similarly, revoking a right

removes it from a cell in the access control matrix which also removes the possible transition of the configuration.

5.6 Executable Specification

The previous section describes the extension to the *Decoding Net* model which adds semantics and authority of dynamic address spaces in the form of an access-control matrix and configuration spaces. This extension specifies the system's correctness property and the resulting, valid configurations an address space may assume. A direct implementation of the abstract model in an operating system is impractical, as it does not specify the *concrete* interactions of user-level processes with the operating system kernel and the concrete kernel state.

Recall, the operating system implementation is a refinement of the abstract model where, for example, the access control matrix is implemented as capabilities or access control lists. Applications then invoke a well-defined API to perform memory allocation and address space configuration operations which trigger state transitions. The specification of the API with its operations and their semantics is another refinement step which bridges the gap [CKS08] between the abstract model and the operational implementation (Chapter 6).

The approach taken follows the example of the seL4 verification project, which used such an *executable specification* [Der+06] to prototype the kernel prior to the implementation in C. Defining the relevant operations and their semantics is critical for the usability in the operating system.

This section describes the co-developed executable specification by Nora Hossle [Hos19] aiding rapid prototyping of the operational model and its implementation in parallel, defining its API and the model state. Moreover, it serves as an intermediate step in the refinement process from the abstract, access-control matrix representation, to the operating system implementation. The executable specification is a Haskell program which implements

a *reference monitor* [And72]. Functional languages like Haskell are well-suited as they require specifying a state monad explicitly and how a function modifies it.

The target implementation of the reference monitor is not limited to an operating system kernel, trusted user-level processes (e.g. a pager process) acting as a reference monitor are also a possibility. This thesis, however, focuses on the operating system kernels of Linux and Barrelfish as target environments. Therefore, the naming scheme of the reference monitor's operations and data structures are suggestive of an operating system kernel.

This section describes the refinement steps from the abstract model using the definitions of the executable specification [Hos19].

5.6.1 Typed Memory Objects

The contents of in-memory data structures such as page tables or device registers (e.g. Xeon Phi system memory page table or segmentation registers) define the behavior of translation units. Software changes how an address is translated by writing the corresponding bit patterns in a particular page-table entry, or device registers. Therefore, the contents of *specific* regions of memory or device registers define the translation state of the system and ultimately what resources a process can access.

Consequently, unprivileged user-level processes must not be able to change the translation configuration of an address space just by issuing a memory write to a device register, or a DRAM region holding a page table, for instance. However, the process should be able to write to memory regions and device registers that are not used to configure an address space. This implies that not all memory objects (e.g. DRAM regions or device registers) are equal.

To express this in the model, it is necessary to distinguish between objects of different types: unprivileged user-space processes can request a mapping of a normal memory object into its address space. However, the reference

monitor must refuse any request to map a *translation structure* object into the address space of an unprivileged, user-space process. Memory objects therefore have a specific *type* which defines whether they are *mappable* or *unmappable*. This makes translation structures explicitly visible.

It is important to write down the necessary integrity condition of the reference monitor using the distinction of object types as Invariant I5.2.

Invariant I5.2 (Never Accessible)
Subjects can never access unmappable objects.

To summarize up to here, regions of memory or device registers are memory objects with a well-defined type which defines whether it is mappable or unmappable. In particular, unmappable objects are never accessible from anyone else than the reference monitor itself.

Object Representation The physical memory resources of an address space are represented as a set of *objects* with a certain type. An object is a set of names, in this case all names starting from a base up to a given size. The representation in the operating system is implementation defined. Listing 5.1 shows the data type definition of some objects in the Haskell executable model. The following list gives an explanation of the types and their usage.

- *RAM.* Untyped (physical) memory object. Not mappable.
- *Translation Structure.* Memory object defining the translation of an address space. Not mappable.
- *Frame.* Mappable (physical) memory object.
- *Device Frame.* Mappable device registers.
- *Segment.* Mappable segment of a configurable address space.

Listing 5.1: Object representation in the executable specification

```
data Object
    = RAM {base :: Name, size :: Natural}
    | Frame {base :: Name, size :: Natural}
    | DeviceFrame {base :: Name, size :: Natural}
    | Segment {base :: Name, size :: Natural}
    | TranslationStructure {base :: Name,
                            size :: Natural}
```

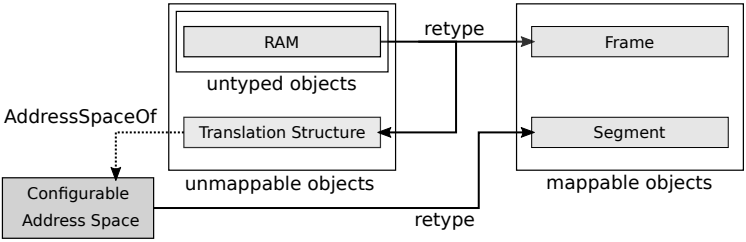


Figure 5.4: Object Type Hierarchy Indicating Possible Retypes.

Note, the details of the translation structure objects are deliberately kept opaque in this representation. In comparison, the seL4 executable specification modeled page tables explicitly. Keeping the translation structure opaque enables reasoning about their effect on the configuration of an address space without imposing restrictions on their form. Modeling hardware defined page tables is another refinement step.

Object Type Conversion Object derivation explicitly converts an object of a type T_1 into an object of type T_2. This is also known as *retyping*. Object derivation is not arbitrary: There exist strict rules on how and when objects can be derived from each other. This is important as an unmappable object must never be retyped into a mappable object. The object type-hierarchy

(Figure 5.4) strictly defines the valid object derivations using the `retype` operation. For example, a `Frame` object can be derived from a `RAM` object, which is considered a special case of untyped objects. The next section discusses the special case involving the configurable address space.

Expressing Authority In the abstract model, subject have specific rights over the objects. The executable specification expresses this as explicit authority on a specific object (Listing 5.2).

Listing 5.2: Authority in the executable specification

```
data Authority = Access Object
               | Map Object
               | Grant Authority
```

5.6.2 Address Spaces

The translation structures define how an address space translates addresses. In other words, they *span* an address space. For example, the system memory page table of the Xeon Phi co-processor spans the address space controlling the translation from the Xeon Phi to the PCI Express (or IOMMU address space). Translation structures, therefore, define a set of implementation-defined address spaces in the system. The `AddressSpaceOf` function makes this explicit.

```
AddressSpaceOf :: TranslationStructure -> AddressSpace
```

To program multi-stage translations, a subject can derive a segment from the address space and use that to create a mapping in an upstream address space. Note, when the translation structure object vanishes (e.g. through deletion or revocation [Ger18]), the derived address spaces and segments must also disappear.

5.6.3 Kernel State and API

The kernel (or reference monitor) state implements the refined model state consisting of a representation of the access control matrix which includes the subjects, objects and the corresponding authorities. Moreover, it also keeps track of the address spaces of a system. Listing 5.3 shows the encoding in the executable specification. The kernel state corresponds to an instance of the static *Decoding Net* model.

Listing 5.3: Expressing the Model State

```
data KernelState
     = KernelState (Set Dispatcher) MDB (Set AddrSpace)
     | InvalidState
```

The kernel state can assume two conditions: it is either *valid* if it adheres to the model invariants, or it is *invalid* if there is a consistency or invariant violation (`Invalidstate`). In the latter case, no statement about the effect of state transition can be made.

The valid state consists of three parts:

1. *Subjects.* The subjects in the system are represented as a set of dispatchers (a term borrowed from Barrelfish), each having a certain set of rights on objects and an address space they execute in.
2. *Mapping Database.* The mapping database (MDB) records the derivation of objects and keeps track of installed mappings them.
3. *Address Spaces.* Represents the set of active address spaces in the system.

Similar to the seL4 executable specification, subjects invoke the reference monitor through invocations of a well-defined API. The kernel state is encapsulated as a state monad (Listing 5.4) explicitly exposing operations on the state.

Listing 5.4: Operations on the Reference Monitor API

```
data Operation a = Operation (State -> (a, State))
  instance Monad (Operation) where ...
```

Any change to the system state corresponds to a sequence of low-level API calls. For example, calling the high-level function `mmap` corresponds to a `retype` operation on a RAM object to obtain a Frame object which is then `map`'ed into a translation structure object.

The kernel state can be updated with operations such as, but not limited to:

- `retype` converts an existing object into an object of a permissible subtype.
- `map` installs a mapping in a translation structure.
- `copy` copies the rights from one subject to another.

An attempt to perform an operation that would leave the kernel state inconsistent (e.g. not sufficient rights, invalid retype, etc) sets the kernel state to the invalid representation.

5.6.4 Validating Traces

A sequence of API calls is referred to as a trace. This corresponds to a sequence of observed kernel states, each of which defines a static configuration of the *Decoding Net* model. As stated above, not all kernel states are valid with respect to the access control matrix. For example, a subject may try to map a translation structure or perform an invalid retype. Thus, there is a set of *correct* traces CT within the set of *possible* traces T:

$$CT \subseteq T$$

A correct trace is a sequence of valid and consistent kernel states that adhere to the abstract access control matrix and thus satisfy the invariant. In the

case an operation violates integrity constraints, the kernel state transitions to the invalid state aborting the execution.

This enables modeling and validating high-level functions performed by the subjects (user-level applications) running on the operating system kernel. The `mmap` example above corresponds to the example trace shown in Listing 5.5.

Listing 5.5: Operations on the Reference Monitor API

```
mappingTrace :: (Operation KernelState)
mappingTrace = do
    ...
    -- retype a RAM object to a Frame
    res <- retype RAM Disp Frame Disp
    -- retype another RAM object to a translation
        structure
    res <- retype RAM2 Disp TStructure Disp
    -- map the frame into the translation structure
    mapping1 <- Model.map TStructure Frame Disp
    ...
```

5.7 Conclusion

This chapter presented an extension of the *Decoding Net* model, introducing dynamic updates to address spaces and the authority to change the configuration of an address space. The development of this extension follows the same, proven methodology used in the seL4 project to produce a rigorous model of memory management with explicit address spaces.

The authorization model applies well-known access control concepts, which define an abstract model that is amenable to implementations in both, capability-based systems (e.g. Barrelfish), as well as ACL-based systems such as Linux. The model refinement together with the co-developed executable specification [Hos19] then provides guidance for an operating systems implementation. This is the topic of the next chapter.

6

Operating System Support for Dynamic Decoding Nets

The previous Chapter 5 describes the extension of the *Decoding Net* model which supports dynamic updates, including the authority required to change the configuration of an address space. This chapter uses the co-developed executable specification [Hos19] as a guide to drive the implementation in an operating system.

1. Section 6.1 describes general considerations for an implementation of the model.
2. Section 6.2 outlines a proposal of a possible implementation in a monolithic system by describing how Linux might be extended with a subset of the model, foregoing most of the least-privilege principle.

3. Section 6.3 describes a fully functional implementation of the model's executable specification in the Barrelfish research operating system as an example of a microkernel-based system. The implementation targets 64-bit x86 and ARMv8 platforms.

6.1 General Implementation Considerations

An efficient implementation of the address-space model is not tied to a particular hardware and software architecture. The relevant abstractions and mechanisms can be implemented in micro-kernels and monolithic-kernel operating systems. This section outlines the abstractions and mechanism needed to enable an efficient implementation in system software.

The address-space model describes the concepts of "cores", address spaces with local and translating regions, configuration spaces and authority. Those concepts need to be reflected in an implementation. This includes:

1. making the address space a first-class operating system abstraction of the memory-management subsystem,
2. managing physical resources of a machine using the local resource abstraction of the address space,
3. expressing and managing the address space configuration, including the required authority, and
4. being aware of "cores" having distinct views of the system resources.

6.1.1 Explicit Address Spaces

Address spaces are no longer an implicit construct tied to a process. They are explicitly a first-class abstraction in system software. This is similar to SpaceJMP [El +16] where applications can create multiple virtual address spaces and switch between them. For instance, instead of serializing a

pointer-rich data structure in a response to a request, a service can create and populate a new address space and pass it as a response to the client providing similar benefits to a single-level store [Kil+62].

In the context of this thesis, the technique presented in SpaceJMP is not limited to purely virtual address spaces. A similar approach can be used when running on different processors with distinct views of the system. In this scenario, the page tables created for one processor cannot be used on another. For instance, during different execution phases the application can request a migration to a processor which is closer to where its data resides, or which offers particular hardware features. This requires setting up the virtual address space correctly and selecting the right one on this core. In some sense, the virtual address space is valid for a particular core, or set of cores. This is effectively a "replication" of the translation behavior on different processors, in contrast to a data-structure replication (Mitosis replicates page tables on different NUMA nodes [Ach+19b]).

In addition, the topology of the address spaces must be efficiently encoded similarly to today's use of the NUMA topology, which is used in memory allocation policies.

6.1.2 Physical Resource Management

The management of physical resources, and memory in particular, is one of the core tasks in operating systems. Applications request memory from the operating system, which tries to satisfy the request by allocating some memory based on some allocation policy, and in the end updating the relevant bookkeeping entries to keep track on which applications use memory resources. Operating systems typically manage memory using contiguous regions which may have a fixed size (e.g. 4KiB pages in Linux or CertiKOS [Gu+16]), or are variable sized (e.g. memory descriptors or capabilities [Lev84; Har85; SSF99; Kle+09; Ger18]). This matches naturally the regions of the address space model of Chapter 4.

Recall that certain regions may not be reachable from all processors or devices. Therefore, it is necessary to request memory from a specific address

space. To some extent, this is similar to NUMA-aware memory allocators (e.g. Linux' NUMA library (`libbnuma`) [Kle08]) where applications can request memory from a specific NUMA node, zone-based allocators supporting heterogeneous memory management [Lin19c], or Barrelfish's memory server which supports allocations from a specific address range. While those mechanisms and APIs offer applications to request memory with certain properties, but those do not include address space identifiers to request the allocation of memory from a specific address space.

Having address space aware memory allocators, requires that the data structures used for bookkeeping include information about the address space to such a resource belongs. Recall Invariant I4.1, which states that each physical resource is *local* in exactly one address space. Therefore, it is sufficient to tag the data structures (e.g. Linux's `struct page`, memory descriptors or capabilities) with an address space identifier, e.g. an integer.

6.1.3 Managing Address Translation

Changing the configuration of an address translation unit is equivalent to altering the referent where the region is mapped onto. To map a non-local region into an address space, the destination referent must be *resolved* first. This involves querying a representation of the address space topology and converting the global name of the resource to a local address within the address space. There are three possible outcomes of the resolution process for a non-location region:

1. *Static translation:* Address resolution is fixed and pre-defined by the hardware. A non-local region can therefore *always* be translated to the corresponding region in the local address space using the static translation function. This function can be pre-generated based on the address space topology information of the target platform. Note, the identity function serves as special case, effectively overlaying one address space onto another, e.g. the core-local address space mostly overlays the system-wide address space, or a memory address is expanded from 32-bits to 64-bits.

2. *Dynamic translation:* Address resolution is dynamic and defined by hardware registers or in-memory translation structures, such as page tables. In this case, the resolution process needs to consult the actual state of the translation hardware. If there is currently no corresponding translation set up, the resolution process can indicate an error, or program it accordingly. Exhausting the translation resources (e.g. the 32 entries of the Xeon Phi system memory page table) renders the translation of that region impossible.

3. *Impossible translation:* Address resolution and derivation of a local address is not feasible because there is no direct path from the core to the destination resource, e.g. a hop over a network link. Note that holding a capability (or a descriptor to a memory mapped file) to this region is still useful: this allows authorization for, and decentralized allocation of, remote buffer memory, for example.

The dynamic case is the most complex as it needs hardware programming, which in turn requires access to memory regions, either physical memory or device registers, which are potentially a non-local region again. Translation resources (e.g. entries in a table, or device registers defining the translation) are yet another instance of scarce resources. Moreover, a single change in an address space's translation configuration may trigger many other related changes to make a region accessible. For instance, setting up subsequent translations in a multi-stage translation scheme, or unmapping the region in other address spaces because its mapping has changed and now refers to a different resource. Consequently, a region may actually be reachable by setting up a dynamic translation. However, the local core may not be able to change the translation itself. For instance, a device cannot change its own IOMMU translation.

In the worst case, resolving the remote region in the local address space requires knowledge of the memory subsystem topology of the entire machine. In addition, where the translation structures are not reachable from the local core, requests need to be sent to the cores that can manage the translation configuration and poke the translation hardware of the corresponding address spaces. A possible way to reduce the overhead is to cache

parts of the translations locally for future resolution steps, at the cost of maintaining an invalidation protocol similar to TLB shoot-downs in which the operating system notifies the cores, which may have cached the page table entry in their TLB, to invalidate the corresponding TLB entry.

Reclaiming previously allocated memory, the operating system must ensure that this memory is no-longer used anywhere. This is important and in turn requires finding all references to that resource. Linux, for instance, maintains a reverse mapping data structure to find where the page has been mapped, whereas Barrelfish uses the capability system for this purpose. When using capabilities, all derived capabilities for that region must be invalidated, a process called revocation [Ger18; EDE07; EKE08]. The revocation process for static or in-accessible translations is straight-forward. In the case of dynamic translations, the corresponding translation configuration needs to be updated to reflect the new state after the revocation operation has completed.

6.1.4 Address Space Aware Cores

Each core in the system must be aware of the local address space it resides in. This is important when configuring translation units and allocating memory. Monolithic system software typically supports core-local data structures, virtual machines have the concept of vCPUs, and Barrelfish's Multikernel architecture runs an independent cpudriver per core. Information about the local address space including pre-processed topology information for fast address lookups can be added to those data structures.

6.2 Proposal of a Possible Implementation in a Monolithic Kernel

An implementation in a monolithic kernel centralizes authority in the kernel and therefore an implementation could not take advantage of fine-grained privilege separation. This section describes a *proposal* outlining how one could obtain a *possible* implementation of the address space model in a monolithic kernel at the example of a paper study with the Linux kernel. In this operating system, the kernel is responsible for managing all the physical resources, address translation hardware, and cores of a system.

6.2.1 Reference Monitor

The Linux kernel is the privileged entity in the system and as such assumes the role of the reference monitor. It implicitly has all the rights on all address spaces in the system. Consequently, it can change how address spaces translate memory requests, and grant or revoke access to memory resources at will, including to itself.

However, those changes are mostly applied on behalf of user-space processes issuing requests to the kernel in form of system calls (e.g. `mmap()` to map a memory resource into the caller's address space), or the result of policy decisions inside the Linux kernel (e.g. demand paging, page-cache management, NUMA balancing, or handling out-of-memory situations).

6.2.1.1 Privilege Separation

Despite being a monolithic kernel, privilege separation is possible to achieve with a varying degree of success. Virtual machines provide vertical separation, but no horizontal compartmentalization within the monolithic kernel. The next paragraphs describe techniques that could be used to implement privilege separation with varying degree of enforcing capabilities.

Kernel Modules. Linux supports dynamic extension of the kernel functionality with so-called kernel modules. Similar to libraries, a kernel module can be statically linked into the kernel or loaded dynamically at runtime. The kernel module then uses the exposed interfaces (the API) of other subsystems to allocate memory or configure their device. In some sense, the kernel module is "unprivileged" and the core kernel is the "reference monitor". There is, however, no enforcement of isolation. The kernel module can, in principle, access all memory that is accessible to the kernel as a whole and modify data structures directly.

Privilege Separation with Para-Virtualization. Para-virtualization provides a way to achieve separation cooperatively between the kernel (subject) and the hypervisor (reference monitor). In this case, the para-virtualization subsystem (PV-Ops) is configured such that calls to update translation tables are diverted to the hypervisor. While this gives a clean separation whenever a translation configuration is updated, without proper virtualization strict enforcement is hard to achieve. The nested kernel [Dau+15] works similar by integrating a small privileged kernel inside the monolithic kernel which interposes all updates to translation tables.

Other Approaches. Intel's MPX [Ole+18] provides protection from accessing memory which is not intended by the programmer (e.g. out-of-bounds access or buffer overflows). ERIM [Vah+19] uses MPX to implement isolation within the same protection domain. Likewise, using Intel SGX to isolate operating system components [RGM16] i.e. running the reference monitor inside the enclave.

Hardware capabilities such as CHERI [Woo+14] enforces pointer integrity on top of virtual memory. A pointer acts as a CHERI capability containing the base and length of a memory region that can be accessed by dereferencing the pointer e.g. prohibiting accesses to critical data structures (e.g. the page tables) outside the reference monitor.

Hilps [Kwo+19] implements separation by using the virtual address space size configuration of ARM. By shrinking the size of the kernel virtual

address space certain mappings become (temporarily) inaccessible. This can isolate reference monitor data structures from the remaining kernel.

Hypernel [Kwo+18] adds a memory bus monitor, a hardware module which monitors and intercepts suspicious memory accesses from the kernel. Privtrans [BS04] tries to partition monolithic programs such that privilege separation can be implemented.

6.2.2 Authority with Access Control Lists

Memory resources in Linux are either file backed, i.e. they have a name (this includes shared memory objects or segments), or they are backed by so-called anonymous memory. This distinction restricts how user-space processes may grant access to those memory resources to other processes. This section describes the two and outlines methods one could use to implement access control and passing of rights between processes.

Linux uses access control lists (ACLs) to express authority over its file-based resources. This means that for each object in the system (resource) there exists a list of subjects plus the permissions those subjects have over the object. Practically, the standard UNIX ACLs include rights for the owner, the user group, and everyone else.

6.2.2.1 File Backed Memory

When a user-space process creates a shared-memory object or segment, a special file is created which resides in a `ramfs` or `tmpfs` mount point. In other words, there exists a path which uniquely names that object. The Linux kernel enforces authority over accessing this file using standard UNIX file permissions, which are represented as access control lists. This implies that every process belonging to a matching user or group can access the file, open it to obtain a file descriptor, and then call `mmap()` on the file descriptor to get access to the file's backing memory. This is similar to files on disk. Moreover, open file descriptors can be sent to other processes

using UNIX domain sockets. This explicitly transfers a grant right to the memory resource to the receiving process.

6.2.2.2 Anonymous Memory

In contrast to file-backed memory, there exists no handle (or file) to anonymous memory. This implies that a process cannot pass a grant right to the anonymous memory region to another user-space process explicitly. The kernel grants access to frames of anonymous memory by mapping it into a process' address space. The process has the access right to the anonymous memory region by virtue of running in its virtual address space.

Applications request anonymous memory mappings using calls to `mmap()` or `sbrk()`. The actual grant of the access right may only happen in response to a page fault when the process first accesses the memory region. User-space may supply hints such as NUMA node, huge-page mapping, permissions, or desired virtual address. Ultimately, the Linux kernel decides what memory to allocate and at which address to map it.

The only mechanism, which a process can *implicitly* grant access to its anonymous memory resources, is by calling `fork()`. This mechanism creates a new child process which inherits open file descriptors and the access right to mapped physical frames from the parent. However, depending on the permissions, this access right is diminished and may only allow read accesses. Attempts to write may trigger a copy-on-write operation which allocates a new physical frame, copies the contents of the original frame, and re-maps the page to the newly allocated one. This results in a loss of access rights to the original page in either the child or the parent.

In response to policy decisions (e.g. AutoNUMA migrating a page) or gathering access statistics, the Linux kernel may decide to swap the access right to one page with another, or temporarily unmap the page. This effectively removes the access right from the process, which is restored when the process accesses the memory page the next time. The backing memory frame, however, may not be the same. For instance, demand

paging reads the memory contents from the swap file and copies it into a new physical frame, which is then mapped into the process' address space.

6.2.3 Physical Resource Management

This section outlines a proposal to adapt the existing infrastructure of the Linux memory models, physical frame numbers and page flags to implement globally unique names and types objects in Linux.

The Linux kernel implements different methods for allocating and managing virtual and physical memory resources, including swapping strategies and various page-caches [Gor04; BC05; HQS16]. At its core, Linux manages physical memory at the granularity of frames, a 4 KiB contiguous region of physical memory. For each memory frame, there is a corresponding data structure, the page struct (`struct page`), which stores various information about the use of this frame, including the virtual address where this frame is accessible from the kernel (if configured).

Linux maintains multiple memory managers for physical memory frames. Core-local pools of free frames avoid synchronization, node-based allocators allow requesting memory from a particular NUMA node, and zone-allocators enable the allocation zones of memory with different properties (e.g DMA memory, realmode-accessible memory, highmem above 4 GiB).

Frames are identified by a globally-unique physical frame number (PFN). Together with an offset into the physical frame, the PFN can be seen as the global name of the physical resource. In other words, the tuple (PFN, va), where $va < 4096$, uniquely identifies a byte of memory. The currently active *memory model* defines the relation between

$$PFN \leftrightarrow \texttt{struct page}.$$

Linux Memory Models. Linux supports three different memory models, *flat*, *discontinuous* and *sparse*, that change how Linux manages memory internally, including how the struct page is located given its PFN.

- *Flat Model.* The flat memory model assumes a single flat physical address space with a fixed mapping between the PFN and the page struct. The PFN is used as an index into a global array of page structs. This model does not support changes in the amount of physical resources, such as hot-plugging or PCI Express attached resources. Consequently, it is not possible to implement support for multiple physical address spaces cleanly.

- *Discontinuous Model.* The discontinuous memory model is able to express holes in the physical address space by maintaining a set of memory nodes, each of which has its own identifier. A memory node covers a range of PFNs. This model allows expressing address spaces as nodes. However, it is not possible to have holes in the nodes themselves, which also renders this memory model unsuitable.

- *Sparse Model.* The sparse memory model adds support for multiple memory nodes which may have holes within them. A memory node consists of a set of frames forming a section. Each section has a map from the section local PFN to the page struct. The sparse model adds support for changes in the amount of memory in response to discovery or hot-plug events.

In summary, the sparse memory model [Whi19] allows encoding different, non-contiguous address spaces using sections. The PFN-map of the section data structure holds the current mapping between a PFN and the underlying data structure representing the physical resource, i.e. the page struct. This allows for a flexible, configurable relation between the *PFN* $\leftrightarrow$ `struct page`.

Encoding object types. The Linux kernel already distinguishes between memory objects used by the kernel or user-space. The supplied "get-free-pages" (GFP) flags during allocation specify its intended use. This

implicitly converts a free memory frame to a user-space accessible object, a page table object or a kernel data structure object. To support additional memory object types, the page struct could be augmented to include additional type information to implement different memory object types. Note, that the underlying fixed size of a memory frame restricts the possible object sizes to the same unit of account.

Accessing the page structs. To allocate and use a physical frame, the kernel needs access to the corresponding data structure. Each page struct resides in exactly one memory location which may not be accessible from all cores managed by the Linux kernel. In addition, it may appear at a different, core-local address. The first case renders the frame unusable on the core, whereas in the second case, a page table replication technique (e.g. [Ach+19b]) could ensure a consistent view from all cores.

6.2.4 Explicit Address Spaces

Recall, the physical frame number (PFN) uniquely identifies the underlying page struct representing a contiguous 4 KiB region of memory. However, hardware uses core-local, physical addresses to refer to memory resources, and does not directly know about concept of PFNs used in the Linux kernel.

Local Addresses and Canonical Names. After the allocation of a new physical frame, the corresponding PFN (i.e. the canonical name of the memory resource) is converted into a physical address.

$$\textit{Physical Address} \leftrightarrow \textit{PFN}$$

To do this conversion, the Linux source code already provides pre-processor macros that simply shift the values by the number of bits representing the page size (e.g. 12 for 4 KiB pages). This is problematic because the same PFN will appear at the same, fixed local physical address on all cores. In an actual implementation, the relation would need to be changed to include

the processor core's or device's local address space where the PFN should be resolved in.

$$(Core, Physical\ Address) \leftrightarrow PFN$$

The resulting implementation of this relation in the Linux kernel would provide the conversion between the globally unique, canonical name of a physical resource to a core-local address. To avoid ambiguity, the Linux kernel would need to be modified to strictly use the PFN to share pointers between different cores and devices. Kernel modules and core functions convert the PFN into a core-local address prior its use, e.g. updating a page-table entry. An actual implementation may use code-generation, lookup tables, or other data structures for this conversion.

User-Space Access to Physical Resources. The Linux kernel does not expose physical memory directly to user-space processes. Instead, it abstracts physical memory resources and presents user-space processes with their virtual address space plus file handles. Processes may hint the kernel to allocate physical memory from a specific set of NUMA nodes, or with a specific property. File descriptors provide means to request a mapping of a specific, previously allocated memory resource, but not necessarily a particular region of physical memory. The exception is the special file `/dev/mem`, which represents the entire physical memory of the system. Accessing it requires root privileges. Instead of a single file, an actual implementation might use multiple files to expose physical memory to user space, each representing an address space. This makes address spaces explicit.

6.2.5 Managing Address Translation

The Linux kernel has the authority over all address spaces, including the grant right on the physical resources and the map rights on the address spaces, and acts as the reference monitor. Therefore, the Linux kernel can

(in principle) allocate some memory, and map it into some address space at will. Privilege separation as outlined in Section 6.2.1 may provide means compartmentalizing and separating certain subsystems.

Therefore, the Linux kernel already has access to all relevant data structures needed to derive the local address of a resource which is needed to set up a translation. This provides the functionality for resolving an address within an address space. For example, the kernel already walks the page tables to find where a virtual address maps to.

Likewise, Linux already maintains a data structure (`rmap`) to store reverse mappings of pages. This enables finding all places where a page is mapped. Using this data structure, an actual implementation could track all locations where a particular page is used. This is important when a frame needs to be unmapped from all address spaces it is currently mapped in response of reconfiguration or hot-plug event.

6.2.6 Address-Space Aware Cores

Linux maintains kernel data structures for representing devices, processes, tasks and cores. For example, upon initialization a device driver module obtains a pointer to the generic PCI device data structure. Kernel threads access processor-private data structures to obtain information about the core they are currently executing on. On x86, this is achieved with the help of segmentation (the GS register). One way to make the cores address-space aware is to augment the corresponding data structures with address-space information by either adding a pointer to the address-space representation or store the local address-space identifier.

Running a process concurrently on multiple cores in the system requires selecting the right configuration of the MMU for each core. An actual implementation could use the address space identifier of the core-local data structure in the scheduler to select the correct copy of the page table, for instance. This is similar to the page-table replication technique used in Mitosis [Ach+19b].

6.2.7 Conclusion

An implementation of the address-space model in a monolithic kernel seems possible, but due to the monolithic architecture (i.e. the kernel has maximum authority) an implementation cannot take advantage of the fine-grained privilege separation outline in Chapter 5. The Linux kernel acts as a central authority holding the grant and map right to all address spaces and its resources. It is possible to modify the existing data structures, translation and conversion functions, and scheduling algorithms used in the kernel to include and use information about address spaces.

In theory, to some degree privilege separation of the Linux kernel seems possible but invasive. In practice, strict enforcement might require additional hardware support. However, a possible separation is doable on the vertical axis. For example, using virtual machines, system monitors (e.g. Arm TrustZone), or running operating system services in user-space together with a small kernel enforcing isolation and acting as reference monitor. This is exactly the setup the next section talks about.

6.3 Implementation in *Barrelfish/MAS*

This section describes the implementation of the address space model using the principle of least-privilege. The executable specification of Section 5.6 serves as a guide to extend the open-source Barrelfish OS [Bau+09a] with support for **m**ultiple **a**ddress **s**paces (*Barrelfish/MAS*).

Barrelfish manages authorization and physical resources using a capability system similar to seL4 [Kle+09], but distributed and partitioned among processor cores. *Barrelfish/MAS* builds on and extends Barrelfish's capability system as described in [Ger18].

Despite its lack of formal verification, Barrelfish is *currently* a more suitable evaluation platform tan seL4 due to the support for multiprocessing,

heterogeneous hardware including drivers for IOMMUs and Xeon Phi co-processors. *Barrelfish/MAS* runs on real hardware and is able to manage protection rights on a variety of real and simulated hardware platforms.

6.3.1 Reference Monitor

Barrelfish is based on the Multikernel architecture [Bau+09a], where most of the operating system personality runs as user-level processes or exists in libraries. The kernel, or *cpudriver* as it is called in Barrelfish terminology, runs in privileged mode and is responsible for protection and isolation of process domains, inter-process communication and scheduling.

The Barrelfish cpudriver does not do any memory allocations. Instead, it provides mechanisms to user-space applications to safely manage physical resources or virtual address spaces using explicit capability operations. The cpudriver acts as the reference monitor and validates the rights represented by the capabilities. It either executes the requested capability operation or refuses and returns an error.

This gives a clear separation between unprivileged user-space processes and the reference monitor (privileged cpudriver).

6.3.2 Background on Capabilities in Barrelfish

The least-privilege model operates on continuous ranges of addresses within an address space, which is identified by an address space identifier (ASID). This is a natural match for *memory descriptors* or capabilities [Lev84; Har85; SSF99] in particular. Capabilities serve as secure, non-forgeable handles to regions of physical resources including ones which are not directly accessible. This section provides a brief introduction to the capabilities used in Barrelfish.

Barrelfish's capability system [Ger18] is derived from seL4 [Kle+09; EKE08]. Consequently, there are many similarities: Regions of physical

memory are represented using typed capabilities. A capability gives the holder a specified set of rights (authority) on the object that the capability references. In Barrelfish, the all different capability types are defined using Hamlet, a domain specific language designed for this purpose.

Typed Objects. Every object has a *type* which is reflected in the capability that refers to the object. The type defines the set of valid operations and how the memory object they refer to can be used. For example, a `Frame` can be mapped, whereas raw `RAM` cannot. Recall Section 5.6, a `RAM` object represents untyped, physical memory, corresponding to seL4's untyped type. Moreover, retype rules dictate valid type conversions. For instance, retyping `RAM` to a `Frame` is permitted, a `Frame` to `RAM` is not.

Access Rights Besides its type, each capability has a set of rights defining the authority the holder has on the object. The rights cannot be changed. However, the holder can derive a new capability with lesser rights. In its current state, the Barrelfish capability system does not fully implement the rights e.g. a read-only `Frame` can only be mapped read-only, but the canonical ordering does not take the rights into account (Listing 6.1).

Capability Space. The capabilities themselves live in the capability space, or CSPACE for short, where each capability occupies an entry in the capability table. In Barrelfish, this capability table is organized as a two-level structure consisting of *cnodes* (Figure 6.1). The root (or L1) cnode contains capabilities pointing to fixed-sized L2 cnodes. Those memory-resident objects are *inaccessible* from applications to prevent user-space processes from forging capabilities. Only the reference monitor is able to access the CSPACE directly. Besides storing the capability representation, the cache-line-sized capability-table entry (cnode slot) holds other bookkeeping information required to track capability relations.

Capability Invocation. Each user-space process has its CSPACE containing the capabilities it owns. Programs refer to a particular capability in their CSPACE using capability references indicating the location of the cnode and the slot within. A process can exercise the rights provided by a

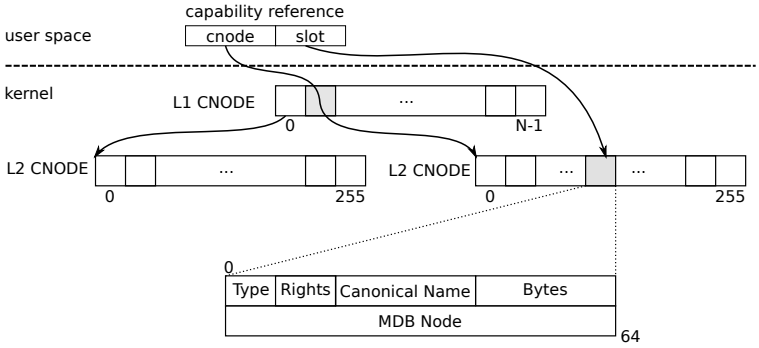


Figure 6.1: Barrelfish's Two-Level CSPACE Layout.

capability in its CSPACE by a *capability invocation*. This corresponds to calling the reference monitor API through the system call interface. The reference monitor then tries to locate the capability using the provided capability reference, and checks whether the type and rights match the requested operation.

Canonical Ordering. The algorithms and data structures for looking up a capability rely on a well-defined ordering of capabilities. The Barrelfish capability system defines a canonical order of the capabilities based on the base address, size and type of the memory object they refer to: *i)* smaller base addresses first, *ii)* then larger objects first, *iii)* then smaller type first (types higher up in the type hierarchy appear first). Listing 6.1 shows the implementation of `capcomp(c1, c2)` returning -1 if c1 comes before c2, $+1$ if c2 comes before c1, and 0 if they are equal.

Listing 6.1: Canonical Ordering of Capabilities

```
int capcomp(struct capability *c1,
            struct capability *c2)
{
  if (c1->Addr != c2->Addr)
    return c1->Addr < c2->Addr ? -1 : 1;
  if (c1->Size != c2->Size)
    return c1->Size > c2->Size ? -1 : 1;
  if (c1->Type != c2->Type)
    return c1->Type < c2->Type ? -1 : 1;
  return 0;
}
```

Descendant Relation. The canonical ordering is important. Barrelfish does not store explicit pointers to ancestors or descendants. Using the canonical ordering, the *descendant* relation is defined as follows:

$$\texttt{descendant}\ c_1\ c_2 \leftrightarrow c_1 \cap c_2 = c_2 \wedge c_1.type \leq c_2.type$$

This means that capability c_2 is a descendant of capability c_1 if c_2 is fully contained within c_1 and the type of c_1 is smaller than the type of c_2. Barrelfish's current implementation only considers the object covered by the capability, but not the associated authority the capability represents over the object. In other words, retyping c_1 to c_2 is permitted and retyping can only make the capability refer to a smaller object. In contrast, minting a capability creates a new descendant with diminished rights.

Mapping Database. Barrelfish maintains a *mapping database*, a balanced, tree-based data structure which stores capabilities in their canonical ordering. This enables efficient capability lookups and range query operators (overlap and contains) based on the object's base address and size. Using the canonical ordering, the mapping database can be traversed efficiently to find the descendants of a capability (successors) and ancestors

(predecessors) efficiently. This is important for finding all affected capabilities in the event of revocation or deletion. There is one mapping database per operating system node (i.e. cpudriver/kernel) each of which keeps track of a partition of all capabilities in the system. Distributed protocols ensure consistency among operating systems nodes and their mapping data bases [Ger18].

Revocation and Deletion. Capabilities can be deleted. This invalidates and frees up a slot in the CSPACE. A simple delete does not remove descendants. Using the mapping database, the reference monitor knows when the last copy of a capability is deleted. If this is the case, additional actions are performed to ensure no-one has access to that object anymore, and there are no memory leaks. This recurses when the deleted capability was a cnode. Revocation deletes all copies and descendants of the revoked capability, without deleting the capability itself. The revoked capability therefore remains as the last copy. Barrelfish's capability system already implements the required revocation protocols in a distributed way while adhering to the same semantics as in seL4 [EDE07; EKE08].

6.3.3 Physical Resource Management

In Barrelfish, physical resources are not managed by the kernel. Instead, dedicated user-level processes manage physical resources directly. This is safe, because the capability system enforces integrity: only well specified operations can be executed, and capabilities cannot be forged. This section provides a high-level summary of the process in Barrelfish [Ger18].

At boot time, the reference monitor creates the initial set of capabilities covering all physical resources of the system minus firmware and kernel regions, which are excluded so user-space cannot allocate and use them otherwise. Each other region of memory is therefore covered by at least one capability. Those capabilities are handed over to the first user-space process. A subset of them correspond to "untyped" RAM objects which are passed to the memory manager.

Other processes request physical memory from the memory manager using an RPC interface. Upon request, the memory manager allocates memory by performing capability operations which derive a new capability from an existing one satisfying the request. The new capability is then transferred to the requesting process. This is effectively a copy from the CSPACE of the memory manager to the CSPACE of the requesting process.

All policy decisions and allocations happen in user space. The kernel's task is to validate and execute the capability operations, e.g. deriving a new capability from an existing one, or performing the copy from one CSPACE to the other. In addition, the kernel maintains the mapping database which is required to validate whether a capability operation is permitted. For instance, it is not allowed to retype the same region of an object twice.

In *Barrelfish/MAS*, the underlying principles remain the same. To support dynamic discovery of address spaces and physical resources within, *Barrelfish/MAS* introduces a new capability type (`PhysAddrSpace`) that corresponds to the *entire* address space. It has an address space identifier, and a size in bits. Using retype operations, new RAM objects or device register objects can be derived within this address space. This is important, as only the driver of a PCI Express attached device knows in detail about the resources on the device. For example, the driver of the Xeon Phi receives the `PhysAddrSpace` to the Xeon Phi co-processor's local physical address space. It then retypes one part of it to `RAM` (either 6 GiB, 8 GiB, or 16 GiB depending on the model) and another part to `DevFrame` representing the memory mapped registers. This process also adheres to Barrelfish's existing retype semantics e.g. the same part of an address space can only be retyped if there are no overlapping descendants.

6.3.4 Explicit Address Spaces

The multiple address space extension, *Barrelfish/MAS*, replaces the use of addresses in the capability system with the object's *canonical name*. By doing so, all addresses used by the capability system are qualified with the identifier of the address space their resource belongs to.

Listing 6.2: Canonical Name Representation

```
/* ASID - address space identifier */
typedef uint32_t asid_t;

/* the genaddr is used within an address space */
typedef uint64_t addr_t;

/* expanded canonical name variant */
typedef struct {
    addr_t  addr;
    asid_t  asid;
} __attribute__ ((packed)) cname_t;

/* compressed canonical name variant */
typedef uint64_t cname_t
```

6.3.4.1 Canonical Names and Capability Comparison

The capability structure encodes canonical names as a struct with two fields: the address space identifier (ASID) and the address as an offset into this address space. This representation supports address-space sizes up to 64-bit and up to $2^{32} - 1$ address spaces. Optimizing for space, both values can be packed into a single 64-bit integer which is the same size of a pointer on 64-bit x86 or ARMv8 systems. This provides support for a 16-bit address space identifier and a 48-bit address, which is sufficiently large to address all virtual memory on current 64-bit x86 or ARMv8. Listing 6.2 shows the corresponding C type definitions.

The mapping database implements range queries such as overlap or contains. Those operations need to remain efficient with the new canonical names. Recall, physical resources are *local* to exactly one address space. Therefore, objects do not span more than one address space. This fact can be used when comparing two capabilities: if the address space identifier is not matching, the capabilities cannot overlap.

Listing 6.3: Canonical Ordering of Capabilities on *Barrelfish/MAS*

```
int capcomp(struct capability *c1,
            struct capability *c2)
{
  if (c1->Asid != c2->Asid)
    return c1->Asid < c2->Asid ? -1 : 1;
  if (c1->Addr != c2->Addr)
    return c1->Addr < c2->Addr ? -1 : 1;
  if (c1->Size != c2->Size)
    return c1->Size > c2->Size ? -1 : 1;
  if (c1->Type != c2->Type)
    return c1->Type < c2->Type ? -1 : 1;
  return 0;
}
```

The canonical ordering is adapted to use canonical names, where the expanded form reads *i)* smaller address space identifier first *ii)* smaller base addresses first, *iii)* then larger objects first, *iv)* then smaller type first. The updated comparison function is given in Listing 6.3. The modification of the compare function, integrates canonical names with existing infrastructure of the mapping database and capability operations.

6.3.4.2 Address Space Representation

Barrelfish/MAS promotes address spaces to first-class objects, and consequently introduces corresponding capabilities to manage them. Consider the following example.

Motivating Example. During PCI discovery, the PCI driver finds and initializes a new device. In response to the discovery event, the device manager spawns the device driver to take care of further configuration. Only the driver knows that there are 57 cores and 8 GiB of memory on that PCI-attached device which happens to be a Xeon Phi co-processor. The

reference monitor does not know about the co-processor at boot time and hence does not create capabilities for the memory on the co-processor.

Address Space Identifiers. Like physical memory, address space identifiers are a limited resource. This requires their allocation to be managed to prevent ASID exhaustion. Barrelfish already uses capabilities for a similar problem: the allocation of interrupt vectors. *Barrelfish/MAS* adds a new capability type representing ranges of address space identifiers. Processes can retype this capability to allocate a new ASID from the range.

Physical Address Spaces. The driver receives a capability representing the address space containing the physical memory resources on the co-processor (e.g. RAM and device registers). Using the `retype` operation provided by the capability system, the driver can derive new RAM and device-register capabilities from the address space capability. This is safe, because the driver can only derive new capabilities that refer to the address space on the co-processor, and not to other existing resources. Finally, the driver can hand over the capabilities to the physical memory on the co-processor to the memory manager for allocations by other processes.

Intermediate Address Spaces. Memory accesses from the co-processor to the host RAM are mediated through a multi-stage translation scheme involving a system memory page table (SMPT) and an IOMMU. The configuration of the SMPT is defined by the contents of 32 memory mapped registers, each controlling a 16 GiB region. The IOMMU translates memory requests using memory-resident page tables. Both, the SMPT and the IOMMU, define a configurable address space, and both have a corresponding capability type that represent those translation resources. *Barrelfish/MAS* adds a new derivation rule to obtain an intermediate address space capability from a translation table capability. This represents the *input address space* of the translation unit. Further, retypes of the intermediate address space produce segments corresponding to regions of the address space which can be used for mappings in upstream address spaces.

Destroying Address Spaces. An address space is destroyed when the last copy of the address space capability is deleted. In contrast to a simple delete, destroying an address space can be seen as revoking plus deleting the address space capability. This recursively deletes all descendants, which is a required behavior since there must be no capabilities referring to objects in non-existing address spaces.

The same applies for translation tables: when the capability system recognizes that the last copy of the translation table is being deleted, the protocol also revokes the derived address space. This ensures that upon deletion of the page table, the address space including all *segments* within it are deleted. This is equivalent to *revoking* all descendants of the address space capability and then deleting it.

6.3.5 Managing Address Translation

Barrelfish's capability system allows for safe and sound construction of hardware-defined translation tables from user space. A process can modify a translation table through capability invocations. This is safe, because the reference monitor only allows operations resulting in correct-by-construction page tables, and processes can only map resources for which they hold a capability with the grant right to it. The reference monitor enforces the required rights and types based on the presented capabilities. A process owning a translation table capability has the right to change the mapping. There is a matching capability for each hardware-defied translation table type. This enforces safe construction of multi-level page tables, for instance.

Dealing with address spaces in translations. Recall, translation structures span an address space. They define a mapping from an input address space to an output address space. *Barrelfish/MAS* uses this information when changing an entry of a translation table. This is important, as the canonical name of the resource needs to be resolved to the local address to perform the mapping. The reference monitor refuses to install the mapping

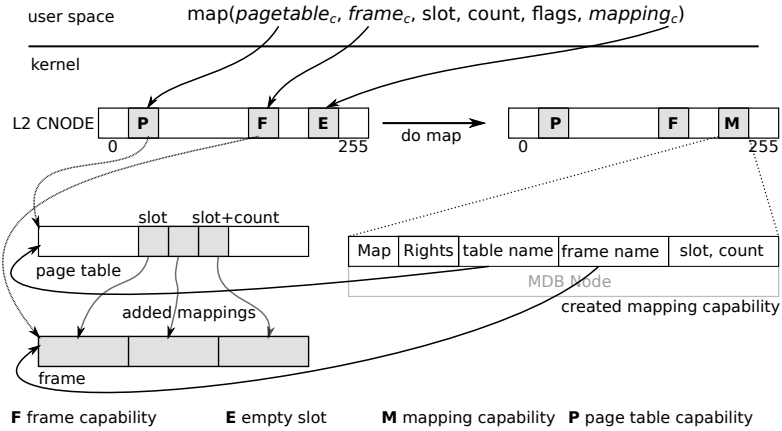


Figure 6.2: Mapping Capabilities in Barrelfish.

if it cannot verify at which address the object appears in the local address space, or when it is not (directly) reachable. In the latter case, a capability from the intermediate address space is required for the mapping. For example, the Xeon Phi co-processor driver uses the address space capability derived from the translation table of the IOMMU context to derive a segment which in turn is mappable into the system memory address space of the co-processor.

Tracking mapped capabilities. When a process revokes a capability or deletes the last copy of it, the deletion or revocation protocol needs to find and invalidate all instances where this memory object was mapped into an address space. This requires additional book-keeping information as the capability table entries are fixed sized and a memory object might be mapped many times into different address spaces.

This book-keeping information is safety critical. Barrelfish uses the capability system to keep track of all existing, valid mappings. For each capability type referring to a mappable object or an address space, there

exists a corresponding *mapping* capability which is a descendant of the mapped object:

`descendant` *mappable_object mapping*

Because of the descendant relation, all mapping capabilities of a mappable object can be found efficiently by traversing the mapping database, and all mapping capabilities are deleted by the revocation protocol. When the last mapping capability is deleted, the translation table entry is invalidated.

Figure 6.2 shows the creation of a mapping capability as an effect of the mapping operation. In the example, a user-space software provides provide three capability references to *i)* a page table capability (P), *ii)* a frame to be mapped (F), and *iii)* an empty slot in its CSPACE (M). The first two identify existing objects in memory, the latter stores the newly created mapping capability. *Barrelfish/MAS* stores the canonical names of the frame and the page table objects in the mapping capability, as opposed to vanilla Barrelfish which stores a pointer to a *specific* capability. The mapping capability also stores the page table entry range which this mapping covers. In the end, the mapping capability is stored in the empty cnode entry.

In addition to using canonical names in the mapping capability, *Barrelfish/-MAS* extends this mechanism to include the new address space and segment types. When parts of an intermediate address space are mapped, a mapping capability is created. When the intermediate address space disappears, the capability system can find all related segments and mapping capabilities using the `descendant` relation.

6.3.6 Address-Space Aware Cores

Barrelfish/MAS extends the data structures of the cpudriver (the kernel) and the device drivers with information about their local address space. This also includes the input and output address spaces of translation hardware

such as the IOMMU or the SMPT of the Xeon Phi. The minimum information required is the local address space identifier. The reference monitor includes a fully generated conversion function translating canonical names of memory resources to local addresses (Listing 6.4). User-space applications obtain further information about address decoding and the memory topology by querying the runtime representation of the model by the local address space identifier.

6.3.7 Runtime Support

The capability system of *Barrelfish/MAS* provides the mechanisms for managing physical resources and configure translation hardware securely. Higher-level operations, such as mapping a region of memory, is broken down into a sequence of capability operations. Depending on the system topology, this involves configuring multiple translation units, some of them may be only discovered at runtime. This section describes a way to find the required sequence of operations. Figure 6.3 shows the high-level architecture of *Barrelfish/MAS* highlighting the policy mechanism separation, the executable model, and the capability system integration.

In Barrelfish, information about the current system topology, cores, devices and memory is stored in the system knowledge base (SKB) [Sch+11]. The SKB is a Prolog engine. Various system services insert pieces of information (e.g. the presence of a PCI device) by asserting them as facts. Inference rules and constraint solvers evaluate queries and produce additional views of those facts. Allocation polices base their decisions on the output of the queries. *Barrelfish/MAS* makes use of the SKB to store the runtime representation of the address space model.

Static Information. The state of the system knowledge base contains predefined, static information. This includes the device data base, for instance, which maps vendor and device identifiers to device drivers. Moreover, it contains non-discoverable platform specific facts such as the number

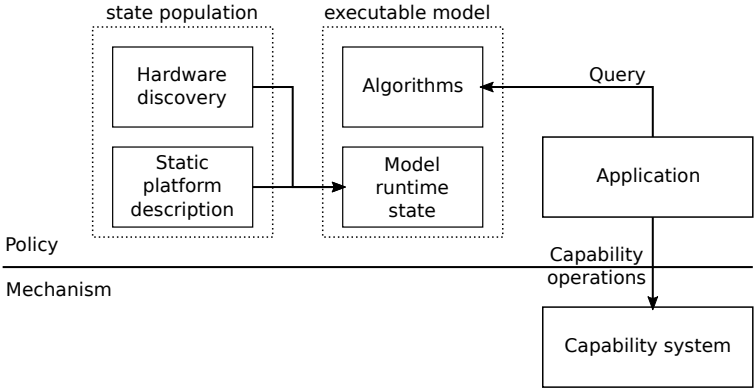


Figure 6.3: Runtime Architecture of *Barrelfish/MAS*

of cores and devices on SoC platforms without hardware discovery. *Barrelfish/MAS* adds additional information such as the address spaces of a discovered device or even the entire set of address spaces for SoCs.

Dynamic Information. The hardware topology of a system is often partially unknown and hence cannot be statically asserted. Consequently, the number and configuration of address spaces may be incomplete and requires dynamic discovery at runtime. Examples include the number of processor cores, the cache topology of the processors, memory affinity, the presence of PCI root complexes and IOMMUs. System software obtains this information by parsing tables provided by UEFI or ACPI, walking the PCI configuration space, executing specific instructions, or reading hardware registers. For example, the PCI driver discovers the Xeon Phi co-processor, but the number of cores and memory resources on the Xeon Phi depends on the exact model, which is only known to the driver. In summary, the machine topology is discovered bit-by-bit based on information from multiple sources.

Asserting Facts. New facts are added to the System Knowledge Base by asserting – the Prolog term for inserting information into the data base. This happens in two steps:

1. During initialization, the device manager asserts static facts based on the current platform (e.g. x86_64 PC or ARMv7 Texas Instruments OMAP4460). This includes non-discoverable information.
2. Depending on the added static facts, the device manager starts hardware discovery services such as the ACPI table parser, to obtain more facts about the system. This in turn may trigger other discovery services.

Barrelfish/MAS uses the same mechanism to add information about the address space topology and their connections to the System Knowledge Base, either by explicit assertion or through inference rules that describe an entire hardware module (e.g. the Xeon Phi co-processor with its cores, memory and the system memory page table.)

Model Queries. Device drivers and other services need to answer the following questions to properly initialize and configure the devices and system services:

- *Allocation.* Where do I allocate memory from, such that it is accessible from those address spaces?
- *Configuration.* Which address spaces need to be configured, such that the memory region is actually accessible from this address space?
- *Backward Resolution.* At which address does the resource appear in this other address space?
- *Forward Resolution.* Where does this address in the local address space resolve to?

The model representation in the System Knowledge Base provides answers to those questions. Common to all of them is the fact that they require

knowledge about the sequence of address spaces that are being traversed from the source to the destination address space. Recall, the *Decoding Net* model is a directed graph and therefore this is a natural match for a shortest-path algorithm returning all the (configurable) address spaces between source and destination node.

Prolog finds a solution for which the posted query evaluates to true. Therefore, by fixing either the source or destination address space, clients can obtain all reachable resources from a particular address space, or obtain all address spaces which can reach a specific resource.

The result of the query serves as a blueprint for the client indicating what steps should be executed. For example, the query responds with a list of address spaces that the client needs to configure. In return, the client converts this list into a sequence of capability operations to allocate memory, setup translation structures and perform the relevant mappings. This is evaluated in Section 7.4.

Query Optimizations Invoking the Prolog engine of the System Knowledge Base is a rather heavy-weight operation (see Section 7.5). Running the query on the full address space topology needs to consider many decode steps which are static. There are two levels of optimization possible:

1. *Flatten.* The address spaces with static translations can be flattened and the result saved as a view. The resulting topology consists of cores, accepting regions and configurable address spaces. Changes in the topology trigger re-computation of the flattened representation.

2. *Caching.* A client which performs multiple queries, can cache either the full, or flattened representation in a C library outside of the SKB.

Note, operating on invalid caches is safe, because the capability system enforces the integrity of the system. A process cannot obtain access to a resources without adequate rights. An attempt to create an invalid mapping results in an error. Depending on the error (e.g. trying to allocate from an

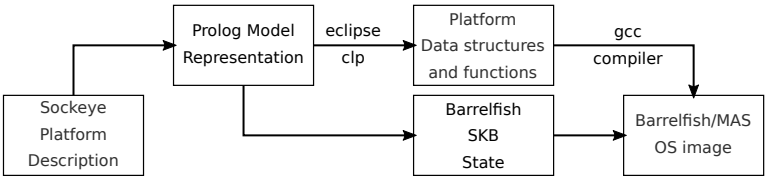


Figure 6.4: *Barrelfish/MAS* Toolchain.

address space which does no longer exist, or trying to map an object which is not directly reachable), the application can then update its cache by invoking the Prolog engine containing the full topology, and then retry the operation. This happens when the number of address spaces has changed.

Address Resolution The ground truth of how an address space translates addresses lies with the configuration of the corresponding translation unit. The SKB does not store the actual configuration of the address spaces. Instead, the query above returns the list of configurable address spaces, each of which are managed by a reference monitor. Application processes can call the API of the reference monitor to resolve an address within the address space it manages.

6.3.8 Compile Time Support

Building an operating system image for a specific platform can make use of the fact that all relevant facts about the hardware configuration are known and complete at compile time [Sch17]. For instance, the hardware topology of SoC platforms (e.g. Texas Instruments OMAP4460), or co-processors (e.g Intel Xeon Phi) can be extracted from the hardware manuals. Figure 6.4 shows a high-level illustration of the toolchain.

Therefore, a system software developer (or hardware vendor) can precisely write down the hardware configuration of those platforms. *Barrelfish/MAS*

adapts the infrastructure provided by the Sockeye [Sch17; Bar17b] system for this purpose. The Sockeye language lets programmers write specifications about hardware platforms naturally using the *map* and *accept* constructs of the address space model.

The Sockeye platform description is the compiled into the Prolog runtime representation of the model. The toolchain populates the System Knowledge Base with the Sockeye generated facts about the address spaces of the target platform. This enables the enumeration and pre-computation of all address spaces and translations of the platform. Using this output, the toolchain *generates* platform specific data structures and low-level functions of the reference monitor for a specific address space (e.g. the cpudriver for a core). Examples include:

- *Capabilities.* The initial set of capabilities referring to the physical resource of the platform.
- *Translate Functions.* Translation functions that convert between core-local addresses and canonical names. Listing 6.4 shows an example of an automatically generated function, which translates canonical names to local physical addresses valid on a particular core.
- *Page Tables.* The kernel page tables, memory maps and device locations. This completely removes the page-table setup from the kernel.

The generated C code is then compiled and linked into the *Barrelfish/MAS* system image. Section 7.7 evaluates this scenario.

Listing 6.4: Canonical name to local address conversion

```
lpaddr_t canonical_name_to_local_phys(cname_t arg)
{
    addr_t ad = cname_extract_addr(arg);
    asid_t asid = cname_extract_asid(arg);

    if(asid == 1035 && 0x0UL <= ad && ad <= 0x1fffUL)
        return ((0x1c000000UL - 0x0UL) + ad);
    if(asid == 1035 && 0x0UL <= ad && ad <= 0x1fffUL)
       return ((0x1c000000UL - 0x0UL) + ad);
    if(asid == 1038 && 0x0UL <= ad && ad <= 0x3fffffffUL)
        return ((0x40000000UL - 0x0UL) + ad);
    if(asid == 1038 && 0x0UL <= ad && ad <= 0x3fffffffUL)
        return ((0x40000000UL - 0x0UL) + ad);
    if(asid == 1039 && 0x0UL <= ad && ad <= 0x3fffffffUL)
        return ((0x80000000UL - 0x0UL) + ad);
    if(asid == 1039 && 0x0UL <= ad && ad <= 0x3fffffffUL)
        return ((0x80000000UL - 0x0UL) + ad);
    if(asid == 1040 && 0x0UL <= ad && ad <= 0xffffffffUL)
        return ((0x0UL - 0x0UL) + ad);
    if(asid == 1040 && 0x0UL <= ad && ad <= 0xffffffffUL)
        return ((0x0UL - 0x0UL) + ad);
    if(asid == 1057 && 0x0UL <= ad && ad <= 0xfffffUL)
        return ((0x2f000000UL - 0x0UL) + ad);
    if(asid == 1057 && 0x0UL <= ad && ad <= 0xfffffUL)
        return ((0x2f000000UL - 0x0UL) + ad);
    if(asid == 1058 && 0x0UL <= ad && ad <= 0xfffffUL)
        return ((0x2f100000UL - 0x0UL) + ad);
    if(asid == 1058 && 0x0UL <= ad && ad <= 0xfffffUL)
        return ((0x2f100000UL - 0x0UL) + ad);

    return LPADDR_INVALID;
}
```

6.4 Conclusion

This chapter demonstrates that it is possible to implement the least-privilege model in a real operating system. *Barrelfish/MAS* uses capabilities to enforce authorization and to manage physical resources. With the help of the Prolog implementation, clients can query the state of the system to obtain a sequence of capability operations to allocate memory and configure address spaces. This is safe, because the capability system enforces the integrity. Based on platform descriptions in Sockeye, the toolchain is able to generate parts of the *Barrelfish/MAS* image such as page tables and translation functions for a particular core.

7

Evaluation

This chapter evaluates the quantitative and qualitative performance of *Barrelfish/MAS* which implements the address space model and the least-privilege authority model based on the executable specification. The goal of this chapter is to quantify the resulting overheads of implementing these models in an operating system, show that these overheads are small, and in addition to demonstrate that the resulting implementation is capable of handling unusual memory topologies.

The evaluation is structured as follows:

- Section 7.2 demonstrates that *Barrelfish/MAS* is able to achieve comparable performance to vanilla Barrelfish and the Linux kernel for the standard virtual memory operations `map`, `protect` and `unmap`.

- Section 7.3 compares the performance of *Barrelfish/MAS* with vanilla Barrelfish and Linux using the Appel-Li benchmark for non-paging related virtual memory operations.
- Section 7.4 assesses the overheads resulting from the least-privilege principle and model evaluations using a real-world scenario involving the configuration of multiple translation units.
- Section 7.5 evaluates the scaling behavior of the runtime representation and shows that the native implementation is able to scale well for a reasonable sized system.
- Section 7.6 calculates storage requirements of the capability system and the cost of a capability lookup in the mapping database.
- Section 7.7 qualitatively demonstrates that *Barrelfish/MAS* is able to handle pathological system topologies using simulators.

This evaluation provides support for the hypothesis that it is possible to efficiently implement the address space model in an operating system, and that the resulting implementation is capable of handling complex memory topologies.

7.1 Evaluation Platform

All experiments in this chapter are executed on the same platform with the following hardware and software configuration:

Hardware. As evaluation platform serves a dual-socket server consisting of two Intel Xeon E5-2670 v2 processors (*Ivy-Bridge* micro-architecture) with 10 cores each, totaling at 20 cores clocked at 2.5GHz. The machine has a grand total of 256 GiB of DDR3 main memory split equally into two NUMA nodes.

In all experiments, the system is set to run in "*performance mode*" by enabling the corresponding setting in the BIOS. In addition, the following processor features have been *disabled*:

- Simultaneous Multi-threading (SMT) / Hyper-Threading
- Intel TurboBoost technology
- Intel Speed Stepping

The reason for disabling these features is to ensure more consistent and stable results by reducing resource sharing and dynamic changes in the processor's clock speed. With SMT or Hyper-Threading, two hardware threads share the resources of a single core. This can cause contention and interference. TurboBoost and speed stepping can change the processor frequency, and the selected clock rate can also depend on the temperature and the clock frequencies of other processor cores.

There are a total of 40 PCI Express 3.0 lanes per processor, each having an Intel Xeon Phi co-processor 31S1 attached as a PCI Express device. The co-processors are of the "*Knights Corner*" generation and have 57 cores with four hardware threads per core. There is a total of 8 GiB GDDR memory per co-processor. Memory accesses from PCI Express attached devices are translated by Intel VT-d [Int19b] (IOMMU).

Software. All experiments involving Linux are done using Ubuntu 18.04 LTS which uses Linux kernel version 4.15. This includes the latest patches for Spectre/Meltdown mitigation which are *disabled* for fair comparison as neither vanilla Barrelfish, nor *Barrelfish/MAS* do have this kind of patches.

Linux is configured using the default configuration provided by Ubuntu. Barrelfish and *Barrelfish/MAS* are using the compiler optimization levels (`-O2`) and the `NDEBUG` flag which disables assertions in the code.

System Call Cost Table 7.1 shows the latency of a no-op system call on Linux, Barrelfish and *Barrelfish/MAS*. For the latter, the table also includes the latency of a no-op capability invocation, which includes a capability lookup in the process' CSPACE. All values are the median out of 1000 runs, with the standard deviation in brackets. Compared to vanilla Barrelfish, *Barrelfish/MAS* experiences a statistically significant slowdown. During the invocation, the kernel looks up the capability in the

Operating System	Cycles		Nanoseconds	
Linux (Sys)	216	(± 2.3)	86	(± 0.9)
Barrelfish (Sys)	172	(± 2.2)	68	(± 0.9)
Barrelfish/MAS (Sys)	172	(± 2.1)	68	(± 0.8)
Barrelfish (Cap)	236	(± 8.9)	94	(± 3.6)
Barrelfish/MAS (Cap)	260	(± 7.5)	104	(± 3.0)

(Cap) indicates a no-op capability invocation
(Sys) indicates a no-op system call.

Table 7.1: System Call Cost on Linux, Barrelfish and *Barrelfish/MAS*.

process' CSPACE, which requires converting the canonical name stored in the capability to a local kernel virtual address twice. This is accomplished using inline functions in vanilla Barrelfish, which add a constant to the physical address stored in the capability to obtain the kernel virtual address. On *Barrelfish/MAS*, the canonical name stored in the capability is first converted to a local physical address, and subsequently to a kernel virtual address. This requires extracting the address space identifiers and addresses from the canonical name resulting in additional instructions and function calls, which add about 24 cycles of latency to the operation.

7.2 Virtual Memory Operations - Map/Protect/Unmap

This section evaluates the performance of virtual memory operations `map`, `protect` and `unmap` by comparing *Barrelfish/MAS* with vanilla Barrelfish and Linux as a frame of reference, using an increasing buffer size from 4 KiB up to 64 GiB and different page sizes.

Benchmark Methodology. The benchmark consists of three phases, each of which is measured separately:

1. `map`: Allocates a free region of virtual memory and faults on it to ensure that it is backed by physical memory,
2. `protect`: write-protects the allocated region of virtual memory,
3. `unmap`: unmaps and frees the virtual memory region.

All three operations manipulate one or more page-table entries in bulk using a single high-level operation. Where possible, the backing memory is pre-allocated. All three supported page-sizes are evaluated separately with regions up to 64 GiB where the page size defines the minimum region size.

- Page size: 4 KiB (base), 2 MiB (large), 1 GiB (huge)
- Region size: page size up to 64 GiB

	Operation		
Configuration	`map`	`protect`	`unmap`
Linux (mmap)	`mmap`	`mprotect`	`munmap`
Linux (shmat)	`shmat`	`shmdt` + `shmat`	`shmdt`
Linux (shmfd)	`shm_open` + `mmap`	`mprotect`	`munmap`
Barrelfish	`vspace_map`	`memobj_protect`	`vspace_unmap`
Barrelfish/MAS	`vspace_map`	`memobj_protect`	`vspace_unmap`

Table 7.2: Evaluated Virtual Memory Operations on Linux, Barrelfish and *Barrelfish/MAS*.

Linux supports different memory management operations: mapping of anonymous memory (mmap), attaching a shared memory segment (shmat) and using a file descriptor to a shared memory object (shmfd). Table 7.2

shows the memory operations used on Linux, vanilla Barrelfish and *Barrelfish/MAS*. All Linux configurations are benchmarked without Spectre/Meltdown mitigation.

Where possible, the allocation of physical memory itself is not measured because this is dominated by clearing the memory pages. On Linux, using the `MAP_POPULATE` flag where possible, or touching the memory after allocation ensures that the page table entries are updated with the mapping information. Barrelfish's operation are all eagerly updating the page table. In all cases, the measurement includes updating the page table and bookkeeping data structures.

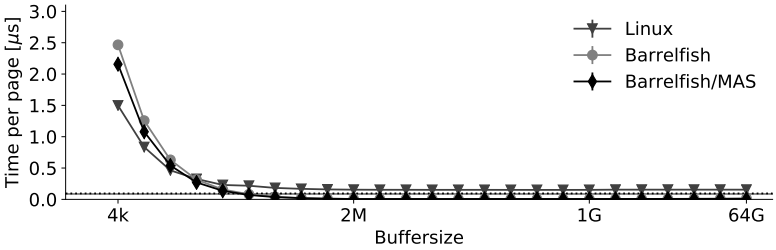
Results. Figures 7.3, 7.3, and 7.3 show the median execution time per affected page for the three operations `map`, `protect` and `unmap` respectively. The figures show performance for all operations with the three different page sizes and an increasing memory region. The Linux results are based on the *best* of the three options for each operation. Table 7.3 shows the best configuration for each page size and operation.

	`map`	`protect`	`unmap`
4 KiB page	Linux-shmfd	Linux-shmfd	Linux-shmat
2 MiB large page	Linux-shmat	Linux-mmap	Linux-shmat
1 GiB huge page	Linux-shmat	Linux-shmat	Linux-shmat

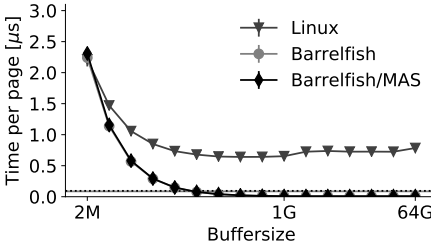
Table 7.3: The Best Configuration of the Linux Virtual Memory Operations.

The graphs show the median execution time (lower is better) of the operation per page (modified page-table entry) on the y-axis and an increasing region size on the x-axis. Recall, the numbers do not include the allocation of backing memory. From the plots we can make the following observations:

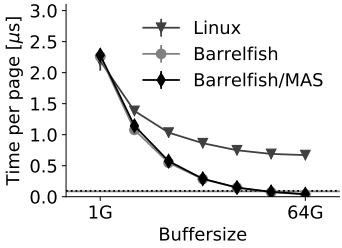
- *Amortization:* Overall the pattern for all configuration looks similar: the cost per page is going down with increasing memory region



(a) 4 KiB Page Size



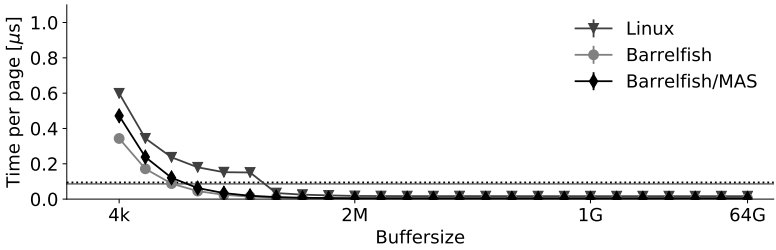
(b) 2 MiB Page Size



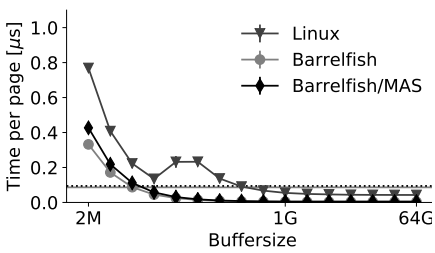
(c) 1 GiB Page Size

The plots show the execution time (lower is better) of the `map()` operation on the entire buffer divided by the number of affected pages in μs. The memory region sizes range from 4 KiB to 64 GiB.

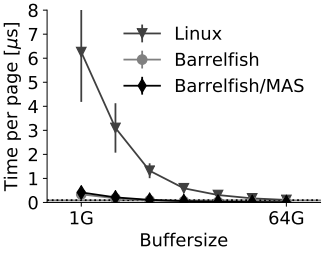
Figure 7.1: Comparison of the Virtual Memory Operation `map()` with Increasing Region Size on *Barrelfish/MAS*, Barrelfish and Linux.



(a) 4 KiB Page Size



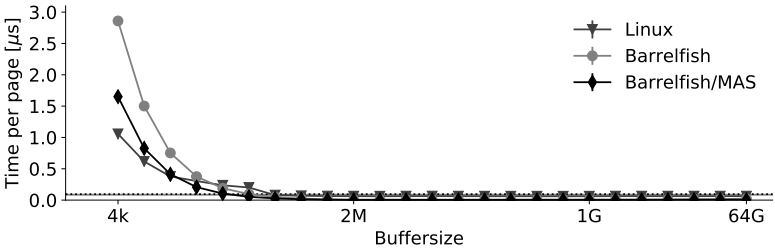
(b) 2 MiB Page Size



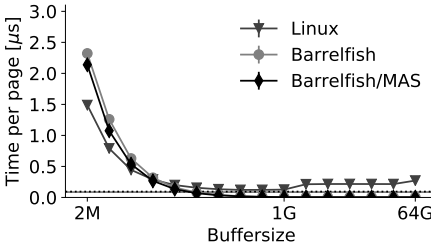
(c) 1 GiB Page Size

The plots show the execution time (lower is better) of the `protect()` operation on the entire buffer divided by the number of affected pages in μs. The memory region sizes range from 2 MiB to 64 GiB.

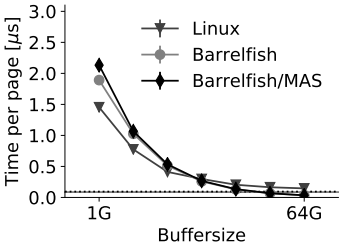
Figure 7.2: Comparison of the Virtual Memory Operation `protect()` with Increasing Region Size on *Barrelfish/MAS*, Barrelfish and Linux.



(a) 4 KiB Page Size



(b) 2 MiB Page Size



(c) 1 GiB Page Size

The plots show the execution time (lower is better) of the `unmap()` operation on the entire buffer divided by the number of affected pages in μs. The memory region sizes range from 1 GiB to 64 GiB.

Figure 7.3: Comparison of the Virtual Memory Operation `unmap()` with Increasing Region Size on *Barrelfish/MAS*, Barrelfish and Linux.

sizes affecting more pages. On Linux this includes a system call, looking up the relevant data structures, walk the page table and finally update one or more page table entries. On Barrelfish and *Barrelfish/MAS*, this corresponds to a lookup of the memory region in user space, invoke the capabilities associated with the region, the kernel then locates the capabilities in the CSPACE, and performs one or more updates to the page table the capability references. In both cases, locating the page table entry is most of the work, which is amortized when multiple, consecutive entries are modified, which is the case for larger regions.

- *Map:* Recall, this maps previously allocated memory in an address space. *Barrelfish/MAS* and Barrelfish exhibit both predictable performance per page regardless of the used page size. This is because the capability operation to update a page table entry is independent of the used page size, in fact it is the same operation just at another level of the page table tree. In contrast, Linux performs worse with large/huge pages compared to base pages for the same number of affected pages. This is because Linux allocates and deposits a page of memory to hold a page table in case the huge/large page mapping has to be broken up and replaced with a smaller mapping granularity later. This introduces additional memory management overheads, and memory writes. Writing a 4 KiB page holding a page table can add up to 0.71us, which corresponds to the difference shown in the graph. Linux is faster than Barrelfish for up to two base pages. This is because Barrelfish creates a new mapping capability and inserts it into the mapping database, which requires finding the right spot in the derivation tree. In direct comparison with Barrelfish, *Barrelfish/MAS* is able to match the performance for the `map` operation for all page sizes. Moreover, using Linux as the reference frame, *Barrelfish/MAS* is able to achieve comparable performance.

- *Protect:* Again, *Barrelfish/MAS* has predictable performance regardless of the used page size. Compared to Barrelfish, *Barrelfish/MAS* has very similar performance characteristics, showing a slight overhead due to a required lookup of the page table from the mapping

capability, whereas Barrelfish includes an explicit pointer. Both, Barrelfish and *Barrelfish/MAS* use capability invocations on the mapping capability, which contains information to efficiently locate the page table entry. Linux needs to locate the page table entries based on the virtual address. This is equivalent to a page table walk. *Barrelfish/MAS* is consistently better than Linux.

- *Unmap:* The performance characteristics per affected page are more predictable for all tested operating systems and configurations. Linux has the best performance for a few pages, whereas Barrelfish and *Barrelfish/MAS* have an advantage for unmapping larger regions. Note, that Barrelfish and *Barrelfish/MAS* do a full TLB flush on an unmap operation, whereas Linux does a selective TLB flush up to 33 pages, resulting in lower overall TLB misses. Moreover, Barrelfish and *Barrelfish/MAS* do need to remove the mapping capability from the MDB, which is an additional lookup, and is amortized over 512 affected page table entries. (Recall, there is at least one mapping cap per page table, which may cover all entries.) *Barrelfish/MAS* is able to match vanilla Barrelfish in all cases.

Discussion. The results for the virtual memory operations `map`, `protect` and `unmap` demonstrate that *Barrelfish/MAS* is able to match the performance of the same operation on vanilla Barrelfish. In addition, both operating systems comparable to Linux in all cases. Linux has an advantage when mapping and unmapping a small number of pages, whereas it needs to lookup page table structures based on virtual address. Furthermore, Linux maintains additional page tables for huge or large page mappings, which introduces overhead. In contrast, Barrelfish and *Barrelfish/MAS* use capabilities which efficiently reference the page table structure. However, Barrelfish and *Barrelfish/MAS* need to update the mapping database (insert or remove the mapping capability), which is amortized over multiple pages.

Therefore, one can implement a fine-grained, least-privilege access control model in an operating system while still having competitive memory management performance to an optimized, monolithic operating system

kernel such as Linux. Moreover, *Barrelfish/MAS* matches the performance of vanilla Barrelfish.

7.3 Virtual Memory Operations - Appel-Li Benchmark

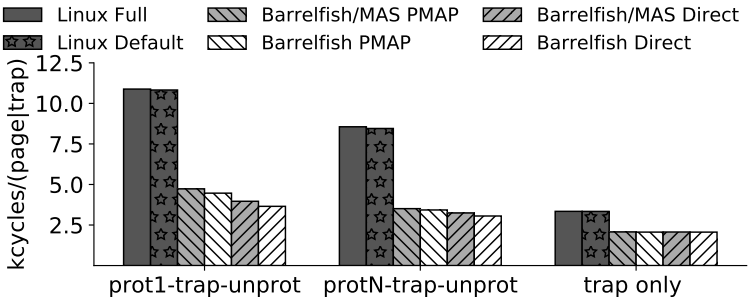
This section evaluates the performance of virtual memory operations which are relevant to tasks such as garbage collection using the Appel-Li benchmark [AL91].

Benchmark Methodology. The Appel-Li benchmark measures the time it takes to protect a page of memory, take a trap into the operating system kernel by writing to the protected page, and then unprotect the page again. It does not evaluate other paging-related operations such as `mmap` and `munmap`. This may be used to track page modifications.

The Appel-Li benchmark is executed in three configurations:

1. *prot1-trap-unprot.* Randomly picks a page of memory, write-protects the page, writes to it, takes a trap, unprotects the page, continue with next page.
2. *protN-trap-unprot.* Write-protects 512 pages of memory, writes to a page of memory, takes a trap, unprotects the page, continues with next page. This amortizes the cost of protecting the pages by using a single system call.
3. *trap only.* Picks a page, tries to write to it, takes the trap, continues with next page without changing any permissions.

Linux runs the Appel-Li benchmark with its default TLB flush strategy (Linux Default), which does a selective TLB flush up to 33 affected pages, and full TLB flush thereafter, and always using full TLB flushes (Linux Full), *Barrelfish/MAS* and Barrelfish always use full TLB flushes, and are evaluated using direct capability invocations, and using the higher-level memory management functions of `libbarrelfish`.



Latency per page handled or trap taken on the y-axis in 1000 cycles.

Figure 7.4: Results of the Appel-Li Benchmark on *Barrelfish/MAS* and Linux.

Results. Figure 7.4 shows the benchmark results for the three configurations. The y-axis shows the time per page (or trap for the trap-only case). The standard error is less than 0.5%. For all configurations, *Barrelfish/MAS* is able to outperform Linux.

- *TLB Flush Strategy.* The differences between the default and full TLB flush strategies are insignificant on Linux.
- *Barrelfish vs.* Barrelfish/MAS. The performance of both systems are comparable, but *Barrelfish/MAS* is 100-300 cycles (or <5%) slower than vanilla Barrelfish.
- *PMAP vs. Direct Invoke.* Barrelfish and *Barrelfish/MAS* maintains a user-space library to keep track of a process' virtual address space. This library accounts for about 10-15% overhead compared to direct capability invocations.
- *Batching.* Protecting all 512 pages in one go (protN-trap-unprot), reduces the amount of traps taken by 511 which amortizes the time per page reducing the latency by 600-2000 cycles.

- *Trap Performance* Barrelfish and *Barrelfish/MAS* have the exact same trap handing latency, which is about 1300 cycles faster than Linux. The trap handlers on Linux and Barrelfish/*Barrelfish/MAS* are identical. Barrelfish simply invokes the registered user-space trap handler on the event of a page fault, whereas Linux performs more operations such as checking whether the fault address has been allocated [Bar00].

Discussion. *Barrelfish/MAS* is able to match Barrelfish with a small 5% overhead, despite support for multiple address spaces and fine-grained protection mechanisms. The slight overhead originates from *i)* a different mapping capability which does not store a direct pointer to the page-table capability, and *ii)* the use of canonical names, which the kernel need to convert to a kernel virtual address to access the page table. This requires additional address calculation operations to extract the name of the page table and convert it to a core-local address.

Both, Barrelfish and *Barrelfish/MAS* are able to outperform Linux due to its efficient and lightweight system call and trap handling. This is also because the capability operations only allow mappings that do not require multiple translations. This check can be done using a local comparison of the address spaces of the involved capabilities.

7.4 Dynamic Updates of Translation Tables

This evaluation investigates the overheads resulting from the least-privilege implementation and model consultation at a real-world scenario of allocating memory and making it available to software running on a co-processor. This could be, for instance, setting up a descriptor ring for communication between a driver and the co-processor. To accomplish this, the following steps need to be executed:

1. Allocate a region of memory such that it is accessible from the driver *and* the co-processor.

2. Obtain the set of translation units to be configured to make the allocated memory accessible from the device and the driver.

3. Perform the necessary instructions to configure a variable number of translation units to make the memory accessible.

Driver software obtains the information of step one and two by consulting the runtime representation of the model. The model response is then translated into a sequence of capability operations and RPCs to perform the necessary memory allocations and address space mappings.

Background: Device Drivers in Barrelfish. This paragraph briefly describes the mechanisms used by device drivers in *Barrelfish/MAS* (and Barrelfish), and how they are started and initialized. The Barrelfish Technical Note 19 [Bar17a] describes the process in detail.

Device drivers run as user-space processes. Kaluga (the Barrelfish device manager) starts a new device driver as a response to a hardware change event, e.g. the PCI Express driver discovers a PCI Express device. As part of their start arguments, device drivers receive the initial set of capabilities required to operate the device. For example, a PCI Express device driver gets a capability to memory mapped registers of the device, the PCI Express configuration space, and the IOMMU input address space (some of which may be RPC endpoints). The latter is important to configure the device's memory translation unit.

Barrelfish/MAS adds another capability to this set: device drivers also receive an address-space capability, which allows managing the physical resources on the device. This is important, because only the device driver knows about memory resources present on the device. For instance, depending on the exact model, the Xeon Phi co-processor comes with 6, 8, or 16 GiB of GDDR memory. The driver uses the address space capability to derive the corresponding RAM capabilities.

Benchmark Methodology. This benchmark profiles the sequence of steps initializing and booting the Intel Xeon Phi co-processor. This consists

of setting up a shared memory region for communication between the driver and the bootstrap code on the Xeon Phi co-processor, to make the co-processor operating system image accessible to the bootstrap code. For this, the Barrelfish Xeon Phi co-processor driver allocates memory and configures the necessary translation units. Tracing points in the driver code measure and profile the allocation and configuration steps.

The Xeon Phi co-processors accesses a memory region in host DRAM through the following chain of translation steps, which the device driver needs to configure correctly:

$$\textit{Co-processor Core MMU} \rightarrow \textit{SMPT} \rightarrow \textit{IOMMU} \rightarrow \textit{System Bus}$$

Note, the IOMMU can be either enabled or disabled. The driver obtains this information by consulting the model representation at runtime.

In this evaluation, the following steps are profiled:

1. *Model Query:* The first step is to find memory resources that are accessible from the Xeon Phi co-processor and the device driver managing it. This is effectively a reachability query on the runtime representation of the model. Moreover, to make the memory resources accessible, the driver may need to configure intermediate address spaces. This corresponds to running shortest path on the runtime representation of the model between the address spaces of the co-processor core and the memory resource.

2. *Memory Allocation and Mapping:* Based on the results from the model query, the driver allocates memory resources. This involves an RPC to the memory manager of the address space indicated by the model. For the evaluation, the driver requests a memory region of 8 MiB in size and maps the received frame into its virtual address space using capability operations. Note, that this operation is similar to `mmap()` on Linux.

3. *IOMMU Programming:* If the IOMMU is enabled, this step configures the required translation. To map the allocated memory region, the driver uses the obtained capability of the memory region as a token of authority to perform the mapping in the co-processor's IOMMU address space. The evaluation compares two alternatives to accomplish this:

   - *RPC:* The driver does an RPC to the IOMMU reference monitor which manages the IOMMU address space and authenticates the capability presented and ultimately performs the mapping. This is implemented on top of the capability system. One possible reason for this setup could be that the cpudriver does not know the details about a particular IOMMU.
   - *Capability Invocation* The device driver uses the capability system which exposes the IOMMU translation structures. It installs the mapping with a direct capability invocation. This is safe because the capability system enforces that only valid mappings can be installed that way.

   In both cases, the device driver obtains a capability to the input address space of the IOMMU. This carries the grant right which is needed to create a mapping from the SMPT address space to the IOMMU address space.

4. *SMPT Programming:* In the last step, the driver uses the capability obtained in the previous step to set up a mapping in the Xeon Phi system memory page table. As before, the driver obtains a corresponding capability of the mapped region in the input address space of the SMPT.

The capability obtained in the last step enables the mapping into the virtual address space of the processor cores on the Xeon Phi co-processor. The driver can send it, or parts of it, to software running on the co-processor.

To provide a frame of reference on the duration of those operations, the evaluation compares the steps above with the latency of the `mmap()` operation

for anonymous memory on Linux with the exact same size, and a user-space `memset()` operation using streaming instructions.

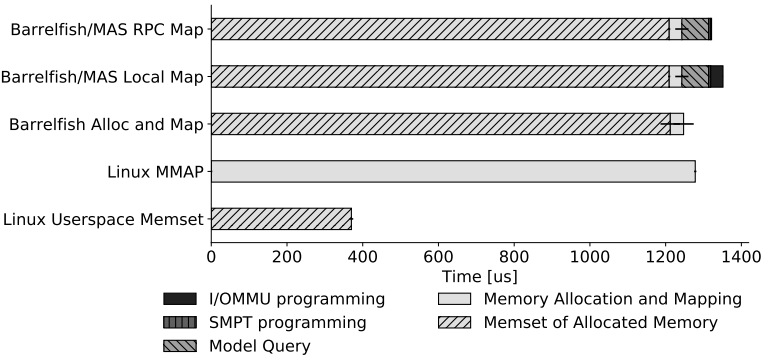


Figure 7.5: Breakdown of Memory Allocation and Mapping Latency.

Results. Figure 7.5 shows the profiling breakdown of the four operations explained in the previous paragraph. In addition, it highlights the time used for clearing the allocated memory. The x-axis shows the latency in micro seconds for the three evaluated configurations on the y-axis. The latency of a `mmap()` call on Linux requesting a mapping of an anonymous memory region of the same size serves as a frame of reference.

- *Memory Allocation and Mapping.* The allocation of the memory resources and setting up the mappings is the dominant factor in this benchmark. In fact, zeroing the memory page in the kernel is the main contributor. Comparing with "Linux User" `memset` shows that there is room for improvement in *Barrelfish/MAS*. Both, Linux and *Barrelfish/MAS* perform the allocation, clearing and mapping of memory in similar times. On Linux, this accounts for the latency until the `mmap` call returns. Note, in contrast to Figures 7.1, 7.2, and 7.3 the memory allocation is also accounted

for in this evaluation. Compared to Barrelfish, *Barrelfish/MAS* achieves the same performance.

- *Model Query.* The model query determines the address space to allocate the memory region from and the address spaces which need configuration. This accounts for 71μsec, or about 5%, to the overall latency.

- *SMPT Configuration.* This corresponds to a capability invocation and a device register write, which accounts for 5μsec or 0.3% of the runtime.

- *IOMMU Programming.* Using direct capability invocations ("Local Map") to update the in-memory IOMMU translation tables. This adds 2μsec to the runtime. In contrast, the "RPC Map" configuration performs the same operations by invoking the IPC endpoint capability of a server which manages the IOMMU. The process presents two capabilities to the IOMMU server, one that conveys the right to change a mapping, and one that provides the grant right to the memory resource. These are the same capabilities needed for the local mapping, but now they are transferred over RPC to the IOMMU server, which then performs the mapping operation. Note, that this requires multiple round trips of the underlying messaging protocol, as Barrelfish's IPC system can only transfer a single capability per message. This adds 30μsec more to the local case.

Overall, the overhead for the additional address space configurations and the model query account for 5.7% overhead compared to Linux and 8.3% of the overall evaluation. There is no significant overhead for the memory allocation and mapping of *Barrelfish/MAS* compared to Barrelfish.

Discussion. This evaluation shows that it is possible to efficiently implement the model queries and the address space configuration following the least-privilege principle. For subsequent memory allocations and address space configurations, the results of the model query can be cached which reduces the overheads even further. This is safe, as the model query merely

indicates what operations need to be done, but the integrity thereof is enforced by the capability system. The overall performance is dominated by the allocation of memory, in particular zeroing the newly allocated memory in the kernel.

7.5 Scaling and Caching Performance

In the evaluation above, the model query consults the runtime representation of the model to obtain the address space to allocate memory from, and the address spaces to configure to make the allocated memory resource accessible to the co-processor. This corresponds finding the shortest path from the source node (e.g. a processor core) to the destination (e.g. a DRAM cell).

This yields the sequence of address spaces in between. The graph grows with the number of *configurable* address spaces and possible connections between them. This evaluation explores the scaling behavior of the model queries for both, the Prolog implementation and the cached graph in C.

Graph Properties. Modern hardware platforms have an ever-increasing number of cores and DMA-capable devices. Memory accesses traverse multiple *configurable* and fixed interconnects and translation units resulting in multi-stage address translations schemes. The address spaces are either:

1. a possible source of a memory access (e.g. processor core or device),
2. a possible destination of a memory access (e.g. DRAM cell or device register),
3. an address space with a configurable translation scheme, or
4. an address space with a fixed translation function.

The model transformations convert the graph by flattening the fixed address spaces. The resulting graph consists of *configurable* address spaces,

memory resources and cores. This representation is persisted and used to run queries on.

The resulting graph has a low diameter, i.e. the number of nodes to traverse from a processor core or device to a memory resource is small. The systems encountered so far have a diameter of less than ten. On the other hand, the fan-out of the nodes may be large, e.g. an IOMMU can translate memory requests from hundreds of PCI devices.

Benchmark Methodology. This benchmark is synthetic. It simulates the address space topology of a hypothetical system based on the one described in Section 7.1. The system is configured with an increasing number of Xeon Phi co-processors, each of which having its own (local) translation unit and an independent IOMMU configuration. The runtime representation of the model grows linearly in the number of Xeon Phi co-processors added to the simulated system.

The benchmark measures the time it takes to determine the set of *configurable* address spaces between a co-processor core and the host DRAM – a common scenario explained in the previous Section 7.4. This corresponds to executing a shortest-path algorithm on the graph. In each round of the benchmark, the number of PCI Express-attached Xeon Phi co-processors is doubled, until 256 co-processors are present in the system, resulting in a few thousand of address spaces.

The evaluation compares the following two implementations, which both are based on Dijkstra's shortest path algorithm and running on *Barrelfish/MAS*.

1. *Prolog.* Calculating the shortest path directly in the Prolog representation of the model running in EclipseCLP on *Barrelfish/MAS*.
2. *C.* Running shortest path on a native C implementation using a graph encoding as an adjacency matrix.

Results. Table 7.4 shows the results of this evaluation. Each row provides the runtime and standard deviation of the runtime for the native C

Number of Devices	Native C		Prolog EclipseCLP	
1	68	(± 0)	131	(± 38)
2	68	(± 0)	149	(± 49)
4	68	(± 0)	144	(± 44)
8	68	(± 0)	166	(± 69)
16	68	(± 0)	187	(± 54)
32	69	(± 0)	257	(± 60)
64	69	(± 0)	427	(± 91)
128	71	(± 0)	723	(± 83)
256	72	(± 0)	1504	(± 2)

Times in Microseconds, Standard Deviation in Brackets.

Table 7.4: Scaling Behavior of Determining the Configurable Address Spaces in a Synthetic, Increasingly Large System.

implementation and the Prolog variant in EclipseCLP. The number of devices indicate added co-processors to the system. The two evaluated implementations differ in their scalability:

1. *Prolog.* The runtime of the Prolog shortest path algorithm in EclipseCLP scales linearly with the number of devices. The Prolog implementation has at least a factor of two overhead.

2. *C.* Evaluating the shortest path in the C graph representation exhibits an almost constant runtime for the number devices measured. Initialization of data structures and consulting the adjacency-matrix results in a slight linear increase of the runtime.

Discussion. The cost for determining the set of configurable address spaces which need to be dynamically configured to enable memory access from a processor node is almost static for efficiently represented graphs with low diameter and a few thousands of address spaces.

In summary, the native C implementation makes it possible to consult the address space model representation at runtime to determine which address spaces require configuration.

7.6 Space and Time Complexity

This evaluation analyzes the space and time complexity of the address space aware capability system used in *Barrelfish/MAS*. An efficient implementation is important for a usable deployment in an operating system. Barrelfish and *Barrelfish/MAS* share most of the implementation, with the delta of address space capabilities.

Evaluation Methodology. The analysis consists of two parts:

1. *Space Complexity* estimates the space requirements to store the capabilities in *Barrelfish/MAS*. This includes the space overhead per page of memory, and bookkeeping information of mappings.

2. *Runtime Complexity* analyses the asymptotic behavior of capability lookups during invocations and mapping database queries.

In both cases, the Linux implementation serves as a frame of reference.

Space Overheads. *Barrelfish/MAS* stores capabilities in the CSPACE of an application where they occupy a slot in a CNODE (or a capability table entry). All CNODE slots are 64 bytes in size. This is enough to store the following two data structures:

1. *Capability Representation.* This stores the type of the object this capability refers to including its name (qualified address), size and associated rights that can be exercised.

2. *MDB Node.* This holds pointers and bookkeeping information needed to insert the capability into the mapping data base.

There is no additional overhead for implementing the mapping database as the MDB node in the CNODE slot is sufficient. In theory, a single capability is enough to manage all physical resources of one address space. To refer to smaller objects, or non-contiguous resources, new capabilities can be derived. The data, stack, and code segments of a program can be represented using three capabilities. This is independent of the respective segment sizes.

To manage and keep track of physical resources, *Barrelfish/MAS* needs at least one capability per address space. Overall, the number of capabilities grows linearly in the number of contiguous regions of physical resources in the system. There is, however, no upper bound in how many *copies* of the same capability can be created. This is limited by the amount of physical memory available as each copy uses 64 bytes.

Assuming one capability per physical 4 KiB frame results in 64 bytes overhead per 4 KiB of memory which is 1.5%. As not all objects need to be exactly 4 KiB in size, the use of larger frames further reduces the overhead. In comparison, the page struct that Linux manages is up to 80 bytes in size and exists for each 4 KiB frame of memory (or about 2%).

Barrelfish/MAS uses mapping capabilities to track of mappings. Setting one or more page-table entries creates a mapping capability each time. Mappings spanning multiple page-table entries require only a single mapping capability per page table spanned. In other words, there are at least one and up to 512 mapping capabilities for the valid entries of a x86_64 page table. This is similar to Linux maintaining the `rmap` data structures for each page of memory that is mapped. The memory requirements for bookkeeping of mappings grows linearly in the number of mapped frames. Using larger mappings, that set multiple page-table entries at once further reduces the bookkeeping overhead.

Runtime Overheads. The CSPACE stores the capabilities of a process in a two-level structure where a capability reference identifies the slot within this structure. Because the depth of the tree-like structure is fixed, lookups are therefore constant time operations: locate the slot in the L1 CNODE, obtain the L2 CNODE and locate the slot within.

The mapping database is a balanced tree data structure which supports range queries to find all capabilities between a base and a limit. The mapping database stores the capabilities in their canonical order which, allows efficient lookup of successors and predecessors. There is a mapping database per reference monitor. Lookups are in the mapping database are logarithmic in the total number of capabilities in this partition.

The ability to efficiently traverse the mapping database is essential to find the related capabilities when a capability is revoked (or the last copy of an address space capability is deleted). This corresponds to a range query plus traversing the MDB and removing the capability.

Linux does not expose physical resources directly to user space. Within the kernel, direct pointers to the page struct can be passed around. Finding the page struct that corresponds to a physical address depends on the memory model. First, the physical address is converted to a physical frame number. Then, in the simplest case (flat memory model), the physical frame number is used as an index into an array, or passed to a more complex function to find the page struct in the worst case.

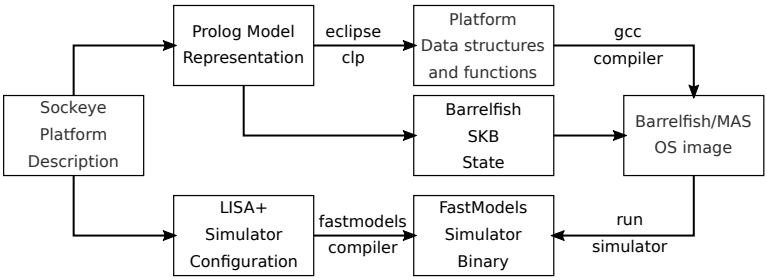


Figure 7.6: Running *Barrelfish/MAS* on an ARM FastModels [ARM19b] Platform Based on a Sockeye Description.

Discussion. In summary, the minimum space required to track resources with capabilities grows linearly in the number of contiguous regions of physical memory. In contrast, Linux maintains a page struct per 4 KiB frame of memory resulting in 2% overhead compared to $< 1.5\%$ for a capability per 4 KiB frame of memory. In *Barrelfish/MAS*, the number of capabilities is limited by the amount of memory resources available to create CNODEs for storing them. Overall, bookkeeping overhead is comparable between *Barrelfish/MAS* and Linux. The tree-based mapping database implementation enables efficient lookups of capabilities by name and allows finding predecessors and successors efficiently.

7.7 Correctness on Simulated Platforms

This evaluation is qualitative. It demonstrates the integration of the address space model into the operating system toolchain to *generate* platform-specific operating systems code. This enables *Barrelfish/MAS* platforms with unusual memory topologies. This stress-tests the model application in the operating system.

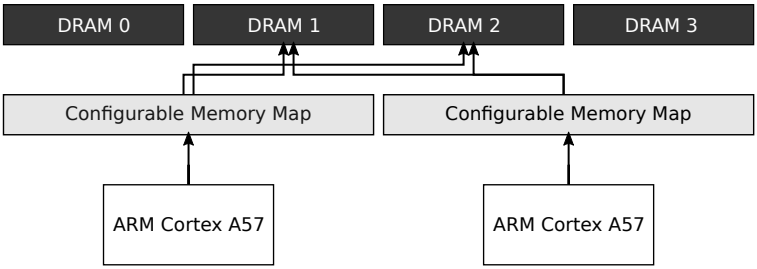


Figure 7.7: FastModels Simulator Configuration

Evaluation Methodology. Figure 7.6 illustrates the schema of this evaluation. First, the platform configuration (including the memory subsystem) is specified in a Sockeye [Sch17; Bar17b] file. This describes the components and the memory subsystem of a simulator platform. From the Sockeye description, the Sockeye compiler produces two outputs:

1. *Simulator Configuration.* The LISA+ hardware description that configures the ARM FastModels simulator [ARM19b], and

2. *Operating System Platform Code.* Low-level operating systems code such as page tables and address translation functions based on the address space model extracted from the Sockeye description.

The generated operating-systems code is then linked into the *Barrelfish/-MAS* kernel at compile time. This creates platform specific bootdrivers, page tables, and operating systems kernels.

The simulated platform consists of two ARM Cortex A57 processor clusters, each having its own memory bus which defines the address translation from the processor cluster to the rest of the system. Figure 7.7 shows an illustration of the four tested configurations:

1. *Uniform* Both processor clusters have an identical memory map.

2. *Swapped* DRAM is split in two halves where on one processor cluster, the first half appears before the second, and vice versa on the other i.e. the address ranges of the two halves are swapped.

3. *Private* There are three memory regions, one of which is shared at between both processors, and there is a private memory region per processor cluster.

4. *Private Swapped* This is the combination of the swapped and private configuration: each processor cluster has its own private memory, plus the shared memory is split in half and is mapped with swapped address ranges.

Results. *Barrelfish/MAS* is able to boot and run successfully on all tested configurations including memory management tasks and shared-memory message passing between two cores. This worked just out of the box, no programmer effort was required. All platform specific data structures, page tables and translation functions were generated based on the Sockeye platform description.

Discussion. This evaluation showed that *Barrelfish/MAS* together with the address space model is capable on running correctly on platforms with complex memory systems. This worked because capabilities strictly use canonical names which are converted to a local address prior using it (if possible). Other operating systems only support case 1 (e.g. Linux, seL4, Barrelfish, Popcorn Linux [Bar+15a]) or provide limited support for case 3 with private memory resources (Barrelfish, Popcorn Linux)

7.8 Conclusion

This chapter demonstrated that it is possible to efficiently implement the detailed and faithful address space model following the least-privilege principle in an operating system. The implementation in *Barrelfish/MAS* is able to match the virtual memory operation performance of Linux.

Additional model consultation and translation hardware configuration add less than 10% overhead to implement least-privilege. Capabilities are a natural match for fine-grained authorization, while at the same time allowing for time and space efficient resource management.

8

Conclusion and Future Directions

This chapter concludes the thesis and presents future directions of the research presented in this thesis.

8.1 Conclusion

This dissertation surveyed different memory address translation schemes and hardware platforms which have been implemented in real hardware or which have been proposed in the architecture community. Based on this survey, the thesis demonstrated how the memory abstractions currently

used in operating systems violate the actual hardware characteristics which in turn has resulted in various bugs and security vulnerabilities in operating systems.

Based on the implications drawn from the analysis of memory translation schemes and platforms, the dissertation made the case to make the address space as a first class operating system abstraction to represent and capture the characteristics of memory address translation, from virtual address spaces to multi-stage address translation schemes and interconnect networks, of any platforms. The semantics of the address space abstraction were then formally defined in the *Decoding Net*, a representation of the model in Isabelle/HOL. This provides a sound basis to reason about address decoding and algorithms operating on top of the model. Moreover, the thesis shows that existing hardware at the example of a software loaded TLB can be expressed as a refinement of the *Decoding Net* model.

The *Decoding Net* model is able to capture the static state of a platform. This state, however, is hardly static: processes and devices require updates to translation units to access memory, or new hardware is discovered, turned on or off, or even hot-plugged in a machine. To express this dynamic behavior, the *Decoding Net* is extended by a layer expressing this dynamic configuration. Each dynamic address space then gets its configuration space from which it can select its current configuration state from.

The configuration space of an address space defines what address translation settings are supported by the hardware itself. To change the configuration of an address space, the subject requesting the change needs to have sufficient authority to do so. The dissertation presented a fine-grained, least-privilege separation of the involved subjects, objects and authorities which is expressed as an access control matrix. Based on this matrix, the configuration space of each address space is further reduced to only allow transitions for which the subject has sufficient authority.

The dissertation adopts the proven methodology of the seL4 verification project to obtain a rigorous model of memory management in an operating system. The development process of the model and its semantics are guided by an executable specification which allows rapid prototyping. The

resulting model and its refinement are the amenable to implementations in both, capability-based systems (e.g. Barrelfish), and ACL-based systems such as Linux.

The thesis further describes the implementation of *Barrelfish/MAS* demonstrating the feasibility of an efficient operating system implementation while at the same time providing safe and clean way to deal with the complexity problem of memory allocation, enforcement and configuration. *Barrelfish/MAS* leverages and extends the distributed, partitioned capability system of Barrelfish, which is a natural match for an implementation following the principle of least-privilege.

The evaluation of *Barrelfish/MAS* provides evidence that it is indeed feasible to efficiently implement the full complexity of multiple, dynamic address spaces in an operating system in a least-privilege fashion. Not only the performance and scalability aspects are good, *Barrelfish/MAS* is able to boot inherently complex and even pathological systems with heterogeneous memory layouts.

8.2 Future Work

The work presented in this thesis provides an end-to-end solution, including a new abstraction of a machine's memory subsystem as seen by software, and an efficient operating system implementation with corresponding toolchain. This section describes are areas that have room for improvement or where there is opportunity for future projects.

8.2.1 Model Improvements

The *Decoding Net* model presented in this thesis purely encodes how an address is being decoded by hardware. This, however, does not capture the entire picture: even if two memory resources are reachable from a particular core, their access characteristics may be different, depending on

their affinity or proximity to the accessing core, or the type of memory. Moreover, an access may be a load or a store.

8.2.1.1 Memory Properties

There are different types of memory resources, each of which having different characteristics: DRAM is volatile and byte addressable, multiple writes can be combined, and accesses may be cached. Non-volatile memory has similar characteristics but is persistent. Device registers can also be byte addressable, but accesses should not be cached or writes combined. Some memory resources may be cache-coherent with a few other cores, while others are not.

This information is important when allocating memory resources and configuring translation units. For example, I/O regions should be mapped non-cachable or as strongly-ordered memory in a processor's MMU, or the GDDR memory on the Xeon Phi co-processor is cache coherent with the co-processor cores, but not with the cores on the host accessing it through the PCI Express subsystem.

It is, however, not sufficient to just add some properties along the decode-relation in the model. There can be multiple resolution paths through the graph, resulting in memory accesses with different characteristics. Properties can be added or removed along the decode-path. The rules and semantics of those alterations need to be well specified.

8.2.1.2 Access Properties

Related to the memory properties above are the characteristics of a memory access themselves. For example, the MMU supports different memory attributes or memory protection keys per page of memory which affects the properties of the memory access. Moreover, a memory resource can be read or written to whereas not all of them support those two operations. The processors have different operating modes such as user/supervisor or secure/non-secure, which do change the access properties.

A memory access has a certain width, e.g. memory is fetched from the DRAM in cache-line granularity whereas a processor may load a single byte from a cache line into a register. Likewise, 64-bit device registers must be accessed using two 32-bit reads or writes.

Some of these access properties can be modeled as separate address spaces (e.g. secure/non-secure). While others (e.g. access width) cannot, or not naturally. Like the memory properties, the semantics of the access properties need to be well-defined.

8.2.1.3 Performance Characteristics

The *Decoding Net* model encodes how many decode steps it takes from a processor or device doing a memory access until it reaches the memory resource. This is some indication about the affinity and proximity of the access. By adding more performance characteristics, such as latency and bandwidth, to the translate-function of an address space, a detailed performance model of a machine can be built.

Using this performance information, allocation and scheduling decisions can be made in much finer granularity as possible with libnuma today. Runtime systems such as Smelt [Kae+16] or Shoal [Kae+15] can then base their scheduling and allocation policies on a detailed performance model of the machine.

8.2.1.4 Caches

The *Decoding Net* model describes how a name is resolved in the system. However, doing so on real hardware can change the state of the system: data is fetched from a cache or loaded into the cache as a result of the memory access, misses in the TLB are triggering page-table walks which in turn issue more memory reads.

It is the ultimate goal of the model to exactly express this kind of highly dynamic aspects of caches in the system with sufficient detail. The interaction with the dynamic extension to the model and the least-privilege model

seems interesting in this regard: having the access right to a memory resource seems to be tied with the right to modify the cache's address space.

8.2.2 Implementation in a Monolithic OS Kernel

The dissertation sketched a possible implementation inside a monolithic kernel at the example of Linux. While the sketch hints at the feasibility of a possible implementation, a real prototype may still require invasive changes to the memory management subsystem of the Linux kernel itself.

There are two aspects to an implementation inside a monolithic operating system kernel:

1. The use of canonical names throughout the kernel when referring to physical resources and name resolution in an address space.
2. Applying (or even enforcing) the principle of least-privilege to address space management functions.

8.2.3 Towards Correct-by-Construction Address Spaces

At its current state, the Sockeye language and toolchain are designed to support the memory and interrupt subsystem, power domains and clock trees. This enabled its use in the work presented in this dissertation to express the memory topology of a platform and generate operating systems code. This section describes a road map for obtaining a correct-by-construction address space management system.

8.2.3.1 Algorithm Verification

The model representation (in Prolog) produced by the Sockeye compiler is similar to the *Decoding Net* model defined in Isabelle/HOL. Likewise, the

transformations that run on top of the Sockeye-generated executable model more or less follow the formally specified counterparts in Isabelle/HOL.

The executable model representation and the algorithms seem to produce the right results. To be sure about the actual correctness of any software requires a formal proof. There are three aspects to this:

1. *Sockeye to Prolog.* The Sockeye compiler takes an hardware description file as input and produces a Prolog representation encoding all address spaces and their translations or memory resources. This transformation process is not verified.

2. *Prolog Representation.* Assuming that the Sockeye compiler correctly produces Prolog code (previous point), the next step is to show that the Prolog representation and its semantics are a refinement of the *Decoding Net* model. In particular, address resolution produces the same result.

3. *Prolog Algorithms.* The previous two steps ensure that the generated Prolog representation of the *Decoding Net* model is correct and has well-defined semantics. The last step is to show the transformation algorithms and queries written in Prolog produce a correct result.

Following these steps ensures that a Sockeye description of a hardware platform can be transformed into a well-founded Prolog representation usable during compilation and runtime.

To accomplish this, a proof framework for Prolog programs is required.

8.2.3.2 Expressing Configuration Spaces

The work presented in this dissertation added the notion of a configuration space to the dynamic address spaces in a system. Conceptually, hardware defines the possible configurations an address space can assume, while the capability system (or the ACLs) reduce this to the set of allowed configurations.

The goal of this line of future work is to express the set of possible configurations of an address space in Sockeye, e.g. an MMU translates 4 KiB- and 2 MiB-aligned regions of memory. Based on these descriptions in Sockeye, operating system code can be generated that check, whether the arguments of a configuration step are valid, e.g. the memory region has the right alignment and size.

8.2.3.3 Configuration Steps Generation

The current implementation is able to produce a list of address spaces that need to be configured to make a resource accessible in the target address space. This separates the address space to be configured from the process of applying the configuration.

Using the capability system, the latter could be abstracted as a sequence of capability invocations that performs a mapping in an address space. For example, the subject presents a capability to the input address space and a capability to the destination resource. The generated code then verifies that it matches the constraints of the underlying hardware, then performs a sequence of capability operations that performs the mapping. The generated code roughly corresponds to the PMAP code currently present in Barrelfish. The CapDL framework [seL18] could possible used for this.

8.2.3.4 Device Representation

Barrelfish already has Mackerel [Bar13] another domain specific language that expresses the register and data structure layout defined by hardware. Combining Sockeye with Mackerel can provide the basis for referring to a specific register and obtain its location in the machine. This would somewhat represent a device. The information that this memory resource are the memory mapped registers described by this Mackerel file can be used to bootstrap and start device drivers.

8.2.3.5 API Generation

Translation structures are hardware defined: a page table on x86_64 is different than on ARMv8, for instance. If Sockeye is able to express the configuration space as explained above, this information can also include the corresponding translation structure and how they are managed.

Using this information, an API modifying the translation structure can be generated and, in the case of Barrelfish, integrated into the capability system. The sequence of configuration steps obtained from above combined with the generated translation structure modification API then yields a correct-by-construction address space management infrastructure.

8.2.4 Integration with Other Models

Expressing the semantics of memory accesses and representing the memory subsystem of a platform are used in proofs about system software and runtime systems. The *Decoding Net* model and its extensions can serve as a basis for those proofs faithfully describing the hardware.

8.2.4.1 Integration into System Software Verification

Verified system software, such as seL4 and CertiKOS, use an over-simplified abstraction of physical memory to base their proofs on. The model presented in this dissertation can replace this simple abstraction to provide a sound foundation for those fully verified systems.

This integration leads to new challenges regarding the correctness proofs with respect to dynamic address spaces and multi-stage memory translation schemes, especially when a single memory resource is accessible under different names from different cores with independent translation units.

8.2.4.2 Memory Models and ISA Semantics

Memory requests can get reordered by the processor. The extent this can happen is defined by the memory model, e.g. total store ordering (TSO)

on x86 or weak memory models such as used in ARM or Power. The semantics of instructions with respect to the memory model have been formally specified. This is orthogonal to the *Decoding Net*. Combining the semantics of possible memory access reordering with the formalism of address decoding could lead to a rich and expressive model of the memory subsystem of any platform.

8.2.5 Applications to Other Areas of Operating Systems

The memory subsystem is not the only aspect of a hardware platform where the formal modeling approach and least-privilege principle can be applied.

8.2.5.1 Interrupts, Power and Clocks

There is a duality between memory addressing and interrupts [Ach+17b] and the same model can be used to express both – in fact some interrupts are delivered as memory writes (e.g. MSI-X). Interrupts also traverse multiple translation steps from its source to destination and those interrupt controllers need to be configured [Hum+17].

Similarly, power and clocks are routed through a network of dividers and voltage regulators. This seems to form a directed graph which indicates the power sources and clock domains that need to be enabled. Like with memory or interrupts, those clock dividers and voltage regulators need to be configured. The question at hand is whether the abstraction of addresses can be used to represent voltage or clocks in this case.

8.2.5.2 Least Privilege Device Drivers

Device drivers require access to a broad set of resources: the device registers to configure the device, interrupt vectors to set up interrupt delivery, and IOMMU configuration mechanisms. Managing devices following the principle of least-privilege seems reasonable: each driver should only have the right to the resources it needs to operate the device.

A similar decomposition on a higher level can be done: identification of the subjects (device drivers and system services), objects (device registers, IPC endpoints) and the authority.

8.2.6 Application to Network Configuration

The *Decoding Net* expresses how addresses are routed through the memory subsystem of a platform. There seems to be an overlap in the area of networking where switches forward network packets based on their configuration. Similar to the memory subsystem, the switches also need to be configured, which requires a well-founded understanding of the network topology. Techniques such as correct-by-construction configurations [Ryz+17] may be also applicable to the context of memory addressing.

List of Tables

List of Figures

Bibliography

[Abu+19] Ahmed Abulila, Vikram Sharma Mailthody, Zaid Qureshi, Jian Huang, Nam Sung Kim, Jinjun Xiong, and Wen-mei Hwu. "FlatFlash: Exploiting the Byte-Accessibility of SSDs Within a Unified Memory-Storage Hierarchy". In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '19. Providence, RI, USA: ACM, 2019, pp. 971–985. ISBN: 978-1-4503-6240-5. DOI: 10.1145/3297858.3304061 (cit. on p. 40).

[Ach+17a] Reto Achermann, Chris Dalton, Paolo Faraboschi, Moritz Hoffmann, Dejan Milojicic, Geoffrey Ndu, Alexander Richardson, Timothy Roscoe, Adrian L. Shaw, and Robert N. M. Watson. "Separating Translation from Protection in Address Spaces with Dynamic Remapping". In: *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*. HotOS '17. Whistler, BC, Canada: ACM, 2017, pp. 118–124. ISBN: 978-1-4503-5068-6. DOI: 10.1145/3102980.3103000 (cit. on pp. 39, 56, 151).

[Ach+17b] Reto Achermann, Lukas Humbel, David Cock, and Timothy Roscoe. "Formalizing Memory Accesses and Interrupts". In: *Proceedings of the 2nd Workshop on Models for Formal Analysis of Real Systems*. MARS 2017. Upsala, Sweden, 2017, pp. 66–116. DOI: 10.4204/EPTCS.244.4 (cit. on pp. 8, 9, 91, 103, 105, 111, 113, 116, 254).

[Ach+18] Reto Achermann, Lukas Humbel, David Cock, and Timothy Roscoe. "Physical Addressing on Real Hardware in Isabelle/HOL". In: *Proceedings of the 9th International Conference on Interactive Theorem Proving*. ITP'18. Oxford, United Kingdom: Springer International Publishing, 2018, pp. 1–19. ISBN: 978-3-319-94821-8 (cit. on pp. 8, 9, 91, 103, 117, 135, 148).

[Ach+19a] Reto Achermann, Nora Hossle, Lukas Humbel, Daniel Schwyn, David Cock, and Timothy Roscoe. *A Least-Privilege Memory Protection Model for Modern Hardware*. 2019. arXiv: 1908.08707 [cs.OS] (cit. on pp. 8, 9).

[Ach+19b] Reto Achermann, Ashish Panwar, Abhishek Bhattacharjee, Timothy Roscoe, and Jayneel Gandhi. "Mitosis: Transparently Self-Replicating Page-Tables for Large-Memory Machines". In: 2019. arXiv: 1910.05398 [cs.OS] (cit. on pp. 181, 191, 193).

[Ach14] Reto Achermann. "Message Passing and Bulk Transport on Heterogenous Multiprocessors". Master's Thesis. Switzerland.: Department of Computer Science, ETH Zurich, 2014. DOI: 10.3929/ethz-a-010262232 (cit. on pp. 8, 9, 48).

[ACH19] Reto Achermann, David Cock, and Lukas Humbel. *Hardware Models in Isabelle/HOL*. Online. Code Repository. Accessed: 16. December 2019. 2019. URL: https://github.com/BarrelfishOS/Isabelle-hardware-models (cit. on p. 111).

[AHW15] Hagit Attiya, Danny Hendler, and Philipp Woelfel. "Trading Fences with RMRs and Separating Memory Models". In: *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*. PODC '15. Donostia-San Sebastián, Spain: ACM, 2015, pp. 173–182. ISBN: 978-1-4503-3617-8. DOI: 10.1145/2767386.2767427 (cit. on p. 70).

[AL91] Andrew W. Appel and Kai Li. "Virtual Memory Primitives for User Programs". In: *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS IV. Santa Clara, California, USA: ACM, 1991, pp. 96–107. ISBN: 0-89791-380-9. DOI: 10.1145/106972.106984 (cit. on p. 226).

[Ala+17] Hanna Alam, Tianhao Zhang, Mattan Erez, and Yoav Etsion. "Do-It-Yourself Virtual Memory Translation". In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ISCA '17. Toronto, ON, Canada: ACM, 2017, pp. 457–468. ISBN: 978-1-4503-4892-8. DOI: 10.1145/3079856.3080209 (cit. on pp. 38, 84).

[Alg+10] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. "Fences in Weak Memory Models". In: *Proceedings of the 22nd International Conference on Computer Aided Verification*. CAV'10. Edinburgh, UK: Springer-Verlag, 2010, pp. 258–272. DOI: 10.1007/978-3-642-14295-6_25 (cit. on p. 70).

[Alg12] Jade Alglave. "A Formal Hierarchy of Weak Memory Models". In: *Form. Methods Syst. Des.* 41.2 (Oct. 2012), pp. 178–210. ISSN: 0925-9856. DOI: 10.1007/s10703-012-0161-5 (cit. on p. 70).

[AMD16] AMD. *AMD I/O Virtualization Technology (IOMMU) Specification*. Version 3.0. Advanced Micro Devices, Inc. Dec. 2016 (cit. on pp. 16, 22, 32, 48).

[AMD19] AMD. *AMD64 Architecture Programmer's Manual - Volume 2: System Programming*. 24593, Revision 3.31. Advanced Micro Devices, Inc. July 2019 (cit. on pp. 16, 18, 21, 22, 26).

[And72] James P. Anderson. *Computer Security Technology Planning Study*. Tech. rep. ESD-TR-73-51, Vol. I, AD-758 206. L. G. Hanscom Field, Bedford, Massachusetts 01730, USA: Electronic Systems Division, Deputy for Command and Management Systems HQ Electronic Systems Division (AFSC), Oct. 1972 (cit. on p. 172).

[APW86] M. Anderson, R. D. Pose, and C. S. Wallace. "A Password-Capability System". In: *The Computer Journal* 29.1 (Jan. 1986), pp. 1–8. ISSN: 0010-4620. DOI: 10.1093/comjnl/29.1.1. eprint: https://academic.oup.com/comjnl/article-pdf/29/1/1/1042091/290001.pdf. URL: https://doi.org/10.1093/comjnl/29.1.1 (cit. on p. 75).

[Arc19] Giuseppe Arcuti. "Formally Modelling Hardware Standards". Bachelor's Thesis. Switzerland.: Department of Computer Science, ETH Zurich, 2019 (cit. on pp. 34, 93).

[ARM09] ARM Ltd. *ARM Security Technology - Building a Secure System using TrustZone Technology*. PRD29-GENC-009492C. Apr. 2009 (cit. on pp. 17, 34, 46, 53, 62, 156).

[ARM11] ARM Ldt. *Cortex-A15 Technical Reference Manual*. Technical Reference Manual. Revision: r2p1, (ID122011). 2011 (cit. on p. 55).

[ARM16] ARM Ltd. *ARM System Memory Management - Unit Architecture Specification*. Version 2.0, DDI IHI0062D. June 2016 (cit. on pp. 17, 22, 32, 48).

[ARM17] ARM Ldt. *ARMv8-A Address Translation*. ARM 100940_0100_en, version 1.0. 2017 (cit. on pp. 17, 22, 23, 26, 34, 46, 137).

[ARM19a] ARM Ltd. *ARM Architecture Reference Manual for Armv8-A architecture profile*. DDI 0487E.a. July 2019 (cit. on pp. 16, 22, 23).

[ARM19b] ARM Ltd. *Development Tools and Software: Fast Models*. https://www.arm.com/products/development-tools/simulation/fast-models. Aug. 2019 (cit. on pp. 80, 240, 241).

[Asm+16] Nils Asmussen, Marcus Völp, Benedikt Nöthen, Hermann Härtig, and Gerhard Fettweis. "M3: A Hardware/Operating-System Co-Design to Tame Heterogeneous Manycores". In: *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '16. Atlanta, Georgia, USA: ACM, 2016, pp. 189–203. ISBN: 978-1-4503-4091-5. DOI: 10.1145/2872362.2872371 (cit. on p. 79).

[Aus+17] Rachata Ausavarungnirun, Joshua Landgraf, Vance Miller, Saugata Ghose, Jayneel Gandhi, Christopher J. Rossbach, and Onur Mutlu. "Mosaic: A GPU Memory Manager with Application-transparent Support for Multiple Page Sizes". In: *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-50 '17. Cambridge, Massachusetts: ACM, 2017, pp. 136–150. ISBN: 978-1-4503-4952-9. DOI: 10.1145/3123939.3123975 (cit. on p. 87).

[AW17] Neha Agarwal and Thomas F. Wenisch. "Thermostat: Application-transparent Page Management for Two-tiered Main Memory". In: *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '17. Xi'an, China: ACM, 2017, pp. 631–644. ISBN: 978-1-4503-4465-4. DOI: 10.1145/3037697.3037706 (cit. on p. 85).

[Azr+19] Leonid Azriel, Lukas Humbel, Reto Achermann, Alex Richardson, Moritz Hoffmann, Avi Mendelson, Timothy Roscoe, Robert N. M. Watson, Paolo Faraboschi, and Dejan Milojicic. "Memory-Side Protection With a Capability Enforcement Co-Processor". In: *ACM Trans. Archit. Code Optim.* 16.1 (Mar. 2019), 5:1–5:26. ISSN: 1544-3566. DOI: 10.1145/3302257 (cit. on pp. 39, 42, 56).

[Bai+11] Katelin Bailey, Luis Ceze, Steven D. Gribble, and Henry M. Levy. "Operating System Implications of Fast, Cheap, Non-volatile Memory". In: *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*. HotOS'13. Napa, California: USENIX Association, 2011, pp. 2–2 (cit. on p. 56).

[Bar+03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. "Xen and the Art of Virtualization". In: *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*. SOSP '03. Bolton Landing, NY, USA: ACM, 2003, pp. 164–177. ISBN: 1-58113-757-5. DOI: 10.1145/945445.945462 (cit. on p. 80).

[Bar+15a] Antonio Barbalace, Marina Sadini, Saif Ansary, Christopher Jelesnianski, Akshay Ravichandran, Cagil Kendir, Alastair Murray, and Binoy Ravindran. "Popcorn: Bridging the Programmability Gap in heterogeneous-ISA Platforms". In: *Proceedings of the Tenth European Conference on Computer Systems*. EuroSys '15. Bordeaux, France: ACM, 2015, 29:1–29:16. ISBN: 978-1-4503-3238-5. DOI: 10.1145/2741948.2741962 (cit. on pp. 73, 242).

[Bar+15b] Claude Barthels, Simon Loesing, Gustavo Alonso, and Donald Kossmann. "Rack-Scale In-Memory Join Processing Using RDMA". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD '15. Melbourne, Victoria, Australia: ACM, 2015, pp. 1463–1475. ISBN: 978-1-4503-2758-9. DOI: 10.1145/2723372.2750547. URL: http://doi.acm.org/10.1145/2723372.2750547 (cit. on p. 85).

[Bar+17] Antonio Barbalace, Anthony Iliopoulos, Holm Rauchfuss, and Goetz Brasche. "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*. HotOS '17. Whistler, BC, Canada: ACM, 2017, pp. 56–61. ISBN: 978-1-4503-5068-6. DOI: 10.1145/3102980.3102990 (cit. on pp. 43, 160).

[Bar00] Moshe Bar. "The Linux Signals Handling Model". In: *Linux Journal* (May 2000). https://www.linuxjournal.com/article/3985 (cit. on p. 228).

[Bar13] Barrelfish Project. *Mackerel User Guide*. Barrelfish Technical Note 2. Mar. 2013 (cit. on pp. 68, 252).

[Bar17a] Barrelfish Project. *Device Drivers in Barrelfish*. Barrelfish Technical Note 19. May 2017 (cit. on p. 229).

[Bar17b] Barrelfish Project. *Sockeye in Barrelfish*. Barrelfish Technical Note 025. Aug. 2017 (cit. on pp. 68, 151, 212, 241).

[Bas+13] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. "Efficient Virtual Memory for Big Memory Servers". In: *Proceedings of the 40th Annual International Symposium on Computer Architecture*. ISCA '13. Tel-Aviv, Israel: ACM, 2013, pp. 237–248. ISBN: 978-1-4503-2079-5. DOI: 10.1145/2485922.2485943 (cit. on pp. 26, 36, 51, 152).

[Bau+09a] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. "The Multikernel: A New OS Architecture for Scalable Multicore Systems". In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. SOSP '09. Big Sky, Montana, USA: ACM, 2009, pp. 29–44. ISBN: 978-1-60558-752-3. DOI: 10.1145/1629575.1629579 (cit. on pp. 9, 79, 84, 159, 163, 194, 195).

[Bau+09b] Andrew Baumann, Simon Peter, Adrian Schüpbach, Akhilesh Singhania, Timothy Roscoe, Paul Barham, and Rebecca Isaacs. "Your Computer is Already a Distributed System. Why isn't Your OS?" In: *Proceedings of the 12th Conference on Hot Topics in Operating Systems*. HotOS'09. Monte Verità, Switzerland: USENIX Association, 2009, pp. 12–12 (cit. on p. 63).

[BC05] Daniel Bovet and Marco Cesati. *Understanding The Linux Kernel*. Oreilly & Associates Inc, 2005. ISBN: 0596005652 (cit. on p. 189).

[Bey+06] Sven Beyer, Christian Jacobi, Daniel Kröning, Dirk Leinenbach, and Wolfgang J. Paul. "Putting It All Together — Formal Verification of the VAMP". In: *Int. J. Softw. Tools Technol. Transf.* 8.4-5 (Aug. 2006), pp. 411–430. ISSN: 1433-2779. DOI: 10.1007/s10009-006-0204-6 (cit. on p. 71).

[Bha17] Abhishek Bhattacharjee. "Translation-Triggered Prefetching". In: *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '17. Xi'an, China: ACM, 2017, pp. 63–76. ISBN: 978-1-4503-4465-4. DOI: 10.1145/3037697.3037705. URL: http://doi.acm.org/10.1145/3037697.3037705 (cit. on p. 36).

[BHK94] Bishop C. Brock, Warren A. Hunt, and Matt Kaufmann. *The FM9001 Microprocessor Proof*. Tech. rep. 86. Computational Logic, Inc., 1994 (cit. on p. 71).

[BHY92] Bishop Brock, Warren A. Hunt, and William D. Young. "Introduction to a Formally Defined Hardware Description Language". In: *Proceedings of the IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*. NLD: North-Holland Publishing Co., 1992, pp. 3–35. ISBN: 0444896864 (cit. on p. 71).

[Bin+11] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. "The Gem5 Simulator". In: *SIGARCH Comput. Archit. News* 39.2 (Aug. 2011), pp. 1–7. ISSN: 0163-5964. DOI: 10.1145/2024716.2024718 (cit. on p. 80).

[Bla+98] David L. Black, Dejan S. Milojičić, Randall W. Dean, Michelle Dominijanni, Alan Langerman, and Steven J. Sears. "Extended Memory Management (XMM): Lessons Learned". In: *Softw. Pract. Exper.* 28.9 (July 1998), pp. 1011–1031. ISSN: 0038-0644. DOI: 10.1002/(SICI)1097-024X(19980725)28:9<1011::AID-SPE180>3.3.CO;2-I (cit. on p. 78).

[BLP11] Spyros Blanas, Yinan Li, and Jignesh M. Patel. "Design and Evaluation of Main Memory Hash Join Algorithms for Multi-core CPUs". In: *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*. SIGMOD '11. Athens, Greece: ACM, 2011, pp. 37–48. ISBN: 978-1-4503-0661-4. DOI: 10.1145/1989323.1989328 (cit. on p. 86).

[BO15] Randal E. Bryant and David R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. 3rd. Pearson, 2015. ISBN: 9780134092669 (cit. on p. 4).

[Bom+92] Allen C. Bomberger, William S. Frantz, Ann C. Hardy, Norman Hardy, Charles R. Landau, and Jonathan S. Shapiro. "The KeyKOS Nanokernel Architecture". In: *Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures*. Berkeley, CA, USA: USENIX Association, 1992, pp. 95–112. ISBN: 1-880446-42-1 (cit. on p. 81).

[BPH15] Andrew Baumann, Marcus Peinado, and Galen Hunt. "Shielding Applications from an Untrusted Cloud with Haven". In: *ACM Trans. Comput. Syst.* 33.3 (Aug. 2015), 8:1–8:26. ISSN: 0734-2071. DOI: 10.1145/2799647 (cit. on p. 34).

[Bre+19] K. M. Bresniker, P. Faraboschi, A. Mendelson, D. Milojicic, T. Roscoe, and R. N. M. Watson. "Rack-Scale Capabilities: Fine-Grained Protection for Large-Scale Memories". In: *Computer* 52.2 (Feb. 2019), pp. 52–62. ISSN: 0018-9162. DOI: 10.1109/MC.2018.2888769 (cit. on pp. 13, 39, 42, 56, 62).

[BS04] David Brumley and Dawn Song. “Privtrans: Automatically Partitioning Programs for Privilege Separation”. In: *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*. SSYM’04. San Diego, CA: USENIX Association, 2004, pp. 5–5 (cit. on p. 187).

[BSD19] BSD. *FreeBSD Architecture Handbook*. Online. Accessed: 27. December 2019. The FreeBSD Documentation Project, 2019. URL: https://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/index.html (cit. on p. 73).

[Cal+17] Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K. Aguilera. “Black-box Concurrent Data Structures for NUMA Architectures”. In: *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’17. Xi’an, China: ACM, 2017, pp. 207–221. ISBN: 978-1-4503-4465-4. DOI: 10.1145/3037697.3037721 (cit. on pp. 83, 85).

[Can+19] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. “A Systematic Evaluation of Transient Execution Attacks and Defenses”. In: *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 249–266. ISBN: 978-1-939133-06-9 (cit. on p. 131).

[Car+99] J. Carter, W. Hsieh, L. Stoller, M. Swanson, L. Zhang, E. Brunvand, A. Davis, C.-C. Kuo, R. Kuramkote, M. Parker, L. Schaelicke, and T. Tateyama. “Impulse: Building a Smarter Memory Controller”. In: *Proceedings of the 5th International Symposium on High Performance Computer Architecture*. HPCA ’99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 70–. ISBN: 0-7695-0004-8 (cit. on p. 40).

[CE00] Franck Cappello and Daniel Etiemble. “MPI Versus MPI+OpenMP on IBM SP for the NAS Benchmarks”. In: *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*. SC ’00. Dallas, Texas, USA: IEEE Computer Society, 2000. ISBN: 0-7803-9802-5 (cit. on p. 85).

[Cha+92] Jeff Chase, Miche Baker-Harvey, Hank Levy, and Ed Lazowska. “Opal: A Single Address Space System for 64-Bit Architectures”. In: *SIGOPS Oper. Syst. Rev.* 26.2 (Apr. 1992), p. 9. ISSN: 0163-5980. DOI: 10.1145/142111.964562 (cit. on pp. 74, 82).

[Cha+94] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. “Sharing and Protection in a Single-address-space Operating System”. In: *ACM Trans. Comput. Syst.* 12.4 (Nov. 1994), pp. 271–307. ISSN: 0734-2071. DOI: 10.1145/195792.195795 (cit. on pp. 40, 74, 82).

[Che18] Adam Chester. *Exploiting CVE-2018-1038 - Total Meltdown*. Online. https://blog.xpnsec.com/total-meltdown-cve-2018-1038/. Apr. 2018 (cit. on p. 156).

[CKM06] Russ Cox, Frans Kaashoek, and Robert Morris. *Xv6, a simple Unix-like teaching operating system*. Online. Accessed: 27. December 2019. 2006. URL: https://pdos.csail.mit.edu/6.828/2019/xv6.html (cit. on p. 76).

[CKM18] Cody Cutler, M. Frans Kaashoek, and Robert T. Morris. "The Benefits and Costs of Writing a POSIX Kernel in a High-level Language". In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. OSDI'18. Carlsbad, CA, USA: USENIX Association, 2018, pp. 89–105. ISBN: 978-1-931971-47-8 (cit. on p. 73).

[CKS08] David Cock, Gerwin Klein, and Thomas Sewell. "Secure Microkernels, State Monads and Scalable Refinement". In: *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics*. TPHOLs '08. Montreal, P.Q., Canada: Springer-Verlag, 2008, pp. 167–182. ISBN: 978-3-540-71065-3. DOI: 10.1007/978-3-540-71067-7_16 (cit. on pp. 162, 171).

[CKZ12] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. "Scalable Address Spaces Using RCU Balanced Trees". In: *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XVII. London, England, UK: ACM, 2012, pp. 199–210. ISBN: 978-1-4503-0759-8. DOI: 10.1145/2150976.2150998 (cit. on p. 44).

[CKZ13] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. "RadixVM: Scalable Address Spaces for Multithreaded Applications". In: *Proceedings of the 8th ACM European Conference on Computer Systems*. EuroSys '13. Prague, Czech Republic: ACM, 2013, pp. 211–224. ISBN: 978-1-4503-1994-2. DOI: 10.1145/2465351.2465373 (cit. on p. 44).

[CP99] Charles D. Cranor and Gurudatta M. Parulkar. "The UVM Virtual Memory System". In: *Proceedings of the USENIX Annual Technical Conference*. USENIXATC '99. Monterey, California: USENIX Association, 1999, pp. 9–9 (cit. on p. 73).

[Cut19] Ian Cutress. *Intel's Enterprise Extravaganza 2019: Launching Cascade Lake, Optane DCPMM, Agilex FPGAs, 100G Ethernet, and Xeon D-1600*. https://www.anandtech.com/show/14155/intels-enterprise-extravaganza-2019-roundup. Apr. 2019 (cit. on p. 62).

[CV65] F. J. Corbató and V. A. Vyssotsky. "Introduction and Overview of the Multics System". In: *Proceedings of the November 30–December 1, 1965, Fall Joint Computer Conference, Part I*. AFIPS '65 (Fall, part I). Las Vegas, Nevada: ACM, 1965, pp. 185–196. DOI: 10.1145/1463891.1463912 (cit. on p. 95).

[Das+13] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. "Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems". In: *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '13. Houston, Texas, USA: ACM, 2013, pp. 381–394. ISBN: 978-1-4503-1870-9. DOI: 10.1145/2451116.2451157 (cit. on p. 84).

[Das+19] Sandeep Dasgupta, Daejun Park, Theodoros Kasampalis, Vikram S. Adve, and Grigore Roşu. "A Complete Formal Semantics of x86-64 User-level Instruction Set Architecture". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2019. Phoenix, AZ, USA: ACM, 2019, pp. 1133–1148. ISBN: 978-1-4503-6712-7. DOI: 10.1145/3314221.3314601 (cit. on p. 70).

[Dau+15] Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell, and Vikram Adve. "Nested Kernel: An Operating System Architecture for Intra-Kernel Privilege Separation". In: *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '15. Istanbul, Turkey: ACM, 2015, pp. 191–206. ISBN: 978-1-4503-2835-7. DOI: 10.1145/2694344.2694386 (cit. on p. 186).

[Dav+19] Brooks Davis, Robert N. M. Watson, Alexander Richardson, Peter G. Neumann, Simon W. Moore, John Baldwin, David Chisnall, James Clarke, Nathaniel Wesley Filardo, Khilan Gudka, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, J. Edward Maste, Alfredo Mazzinghi, Edward Tomasz Napierala, Robert M. Norton, Michael Roe, Peter Sewell, Stacey Son, and Jonathan Woodruff. "CheriABI: Enforcing Valid Pointer Provenance and Minimizing Pointer Privilege in the POSIX C Run-time Environment". In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '19. Providence, RI, USA: ACM, 2019, pp. 379–393. ISBN: 978-1-4503-6240-5. DOI: 10.1145/3297858.3304042 (cit. on p. 82).

[Daw97] Mike Dawson. *The AS/400 Owner's Manual*. MC Press, LLC, 1997. ISBN: 1883884403 (cit. on p. 75).

[Dem+13] Delphine Demange, Vincent Laporte, Lei Zhao, Suresh Jagannathan, David Pichardie, and Jan Vitek. "Plan B: A Buffered Memory Model for Java". In: *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium*

on Principles of Programming Languages. POPL '13. Rome, Italy: ACM, 2013, pp. 329–342. ISBN: 978-1-4503-1832-7. DOI: 10.1145/2429069.2429110 (cit. on p. 88).

[Den70] Peter J. Denning. "Virtual Memory". In: *ACM Comput. Surv.* 2.3 (Sept. 1970), pp. 153–189. ISSN: 0360-0300. DOI: 10.1145/356571.356573 (cit. on pp. 11, 12, 157).

[Der+06] Philip Derrin, Kevin Elphinstone, Gerwin Klein, David Cock, and Manuel M. T. Chakravarty. "Running the Manual: An Approach to High-assurance Microkernel Development". In: *Proceedings of the 2006 ACM SIGPLAN Workshop on Haskell*. Haskell '06. Portland, Oregon, USA: ACM, 2006, pp. 60–71. ISBN: 1-59593-489-8. DOI: 10.1145/1159842.1159850 (cit. on p. 171).

[dev17] devicetree.org. *Devicetree Specification*. Release v0.2. Dec. 2017 (cit. on pp. 3, 68, 93, 95).

[DG17] Thaleia Dimitra Doudali and Ada Gavrilovska. "CoMerge: Toward Efficient Data Placement in Shared Heterogeneous Memory Systems". In: *Proceedings of the International Symposium on Memory Systems*. MEMSYS '17. Alexandria, Virginia: ACM, 2017, pp. 251–261. ISBN: 978-1-4503-5335-9. DOI: 10.1145/3132402.3132418 (cit. on p. 85).

[Dra+14] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. "FaRM: Fast Remote Memory". In: *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*. NSDI'14. Seattle, WA: USENIX Association, 2014, pp. 401–414. ISBN: 978-1-931971-09-6. URL: http://dl.acm.org/citation.cfm?id=2616448.2616486 (cit. on p. 85).

[DV66] Jack B. Dennis and Earl C. Van Horn. "Programming Semantics for Multiprogrammed Computations". In: *Commun. ACM* 9.3 (Mar. 1966), pp. 143–155. ISSN: 0001-0782. DOI: 10.1145/365230.365252 (cit. on p. 159).

[EDE07] Dhammika Elkaduwe, Philip Derrin, and Kevin Elphinstone. "A memory allocation model for an embedded microkernel". In: *Proceedings of the 1st International Workshop on Microkernels for Embedded Systems*. 2007, pp. 28–34 (cit. on pp. 184, 199).

[EDE08] Dhammika Elkaduwe, Philip Derrin, and Kevin Elphinstone. "Kernel Design for Isolation and Assurance of Physical Memory". In: *Proceedings of the 1st Workshop on Isolation and Integration in Embedded Systems*. IIES '08. Glasgow, Scotland: ACM, 2008, pp. 35–40. ISBN: 978-1-60558-126-2. DOI: 10.1145/1435458.1435465 (cit. on p. 76).

[EGK95] D. R. Engler, S. K. Gupta, and M. F. Kaashoek. "AVM: Application-level Virtual Memory". In: *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems*. HOTOS'95. Washington, DC, USA: IEEE Computer Society, 1995, pp. 72–. ISBN: 0-8186-7081-9. URL: http://dl.acm.org/citation.cfm?id=822074.822385 (cit. on p. 78).

[EKE08] Dhammika Elkaduwe, Gerwin Klein, and Kevin Elphinstone. "Verified Protection Model of the seL4 Microkernel". In: *Proceedings of the 2nd International Conference on Verified Software: Theories, Tools, Experiments*. VSTTE '08. Toronto, Canada: Springer-Verlag, 2008, pp. 99–114. ISBN: 978-3-540-87872-8. DOI: 10.1007/978-3-540-87873-5_11 (cit. on pp. 159, 184, 195, 199).

[EKO95] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr. "Exokernel: An Operating System Architecture for Application-level Resource Management". In: *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*. SOSP '95. Copper Mountain, Colorado, USA: ACM, 1995, pp. 251–266. ISBN: 0-89791-715-4. DOI: 10.1145/224056.224076 (cit. on p. 78).

[El +16] Izzat El Hajj, Alexander Merritt, Gerd Zellweger, Dejan Milojicic, Reto Achermann, Paolo Faraboschi, Wen-mei Hwu, Timothy Roscoe, and Karsten Schwan. "SpaceJMP: Programming with Multiple Virtual Address Spaces". In: *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '16. Atlanta, Georgia, USA: ACM, 2016, pp. 353–368. ISBN: 978-1-4503-4091-5. DOI: 10.1145/2872362.2872366 (cit. on pp. 55, 158, 180).

[Esm+11] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. "Dark Silicon and the End of Multicore Scaling". In: *Proceedings of the 38th Annual International Symposium on Computer Architecture*. ISCA '11. San Jose, California, USA: ACM, 2011, pp. 365–376. ISBN: 978-1-4503-0472-6. DOI: 10.1145/2000064.2000108 (cit. on pp. 50, 62, 153).

[Far+15] Paolo Faraboschi, Kimberly Keeton, Tim Marsland, and Dejan Milojicic. "Beyond Processor-centric Operating Systems". In: *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems*. HOTOS'15. Switzerland: USENIX Association, 2015, pp. 17–17 (cit. on pp. 13, 42, 55, 61).

[Fee+95] M. J. Feeley, W. E. Morgan, E. P. Pighin, A. R. Karlin, H. M. Levy, and C. A. Thekkath. "Implementing Global Memory Management in a Workstation Cluster". In: *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*. SOSP '95. Copper Mountain, Colorado, USA: ACM, 1995, pp. 201–212. ISBN: 0-89791-715-4. DOI: 10.1145/224056.224072 (cit. on p. 83).

[Fer+17] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. "Komodo: Using Verification to Disentangle Secure-enclave Hardware from Software". In: *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP '17. Shanghai, China: ACM, 2017, pp. 287–305. ISBN: 978-1-4503-5085-3. DOI: 10.1145/3132747.3132782 (cit. on p. 34).

[Flu+16] Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. "Modelling the ARMv8 Architecture, Operationally: Concurrency and ISA". In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL'16. St. Petersburg, FL, USA: ACM, 2016, pp. 608–621. ISBN: 978-1-4503-3549-2. DOI: 10.1145/2837614.2837615 (cit. on p. 70).

[FM10] Anthony Fox and Magnus O. Myreen. "A Trustworthy Monadic Formalization of the ARMv7 Instruction Set Architecture". In: *Proceedings of the First International Conference on Interactive Theorem Proving*. ITP'10. Edinburgh, UK: Springer-Verlag, 2010, pp. 243–258. ISBN: 978-3-642-14051-8. DOI: 10.1007/978-3-642-14052-5_18 (cit. on pp. 70, 71, 150).

[FNW15] Yaosheng Fu, Tri M. Nguyen, and David Wentzlaff. "Coherence Domain Restriction on Large Scale Systems". In: *Proceedings of the 48th International Symposium on Microarchitecture*. MICRO-48. Waikiki, Hawaii: ACM, 2015, pp. 686–698. ISBN: 978-1-4503-4034-2. DOI: 10.1145/2830772.2830832 (cit. on p. 86).

[Fot61] John Fotheringham. "Dynamic Storage Allocation in the Atlas Computer, Including an Automatic Use of a Backing Store". In: *Commun. ACM* 4.10 (Oct. 1961), pp. 435–436. ISSN: 0001-0782. DOI: 10.1145/366786.366800 (cit. on p. 13).

[Gam+99] Ben Gamsa, Orran Krieger, Jonathan Appavoo, and Michael Stumm. "Tornado: Maximizing Locality and Concurrency in a Shared Memory Multiprocessor Operating System". In: *Proceedings of the Third Symposium on Operating Systems Design and Implementation*. OSDI '99. New Orleans, Louisiana, USA: USENIX Association, 1999, pp. 87–100. ISBN: 1-880446-39-1 (cit. on p. 78).

[Gan+14] Jayneel Gandhi, Arkaprava Basu, Mark D. Hill, and Michael M. Swift. "Efficient Memory Virtualization: Reducing Dimensionality of Nested Page Walks". In: *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-47. Cambridge, United Kingdom: IEEE Computer Society, 2014, pp. 178–189. ISBN: 978-1-4799-6998-2. DOI: 10.1109/MICRO.2014.37 (cit. on p. 38).

[Gan+16] Jayneel Gandhi, Vasileios Karakostas, Furkan Ayar, Adrian Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman S. Ünsal. "Range Translations for Fast Virtual Memory". In: *IEEE Micro* 36.3 (May 2016), pp. 118–126. DOI: 10.1109/MM.2016.10 (cit. on p. 38).

[Gau+14] Fabien Gaud, Baptiste Lepers, Jeremie Decouchant, Justin Funston, Alexandra Fedorova, and Vivien Quéma. "Large Pages May Be Harmful on NUMA Systems". In: *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*. USENIX ATC'14. Philadelphia, PA: USENIX Association, 2014, pp. 231–242. ISBN: 978-1-931971-10-2 (cit. on p. 38).

[Gau+15] Fabien Gaud, Baptiste Lepers, Justin Funston, Mohammad Dashti, Alexandra Fedorova, Vivien Quéma, Renaud Lachaize, and Mark Roth. "Challenges of Memory Management on Modern NUMA Systems". In: *Commun. ACM* 58.12 (Nov. 2015), pp. 59–66. ISSN: 0001-0782. DOI: 10.1145/2814328 (cit. on pp. 29, 83, 84).

[Gav+18] Vasilis Gavrielatos, Antonios Katsarakis, Arpit Joshi, Nicolai Oswald, Boris Grot, and Vijay Nagarajan. "Scale-out ccNUMA: Exploiting Skew with Strongly Consistent Caching". In: *Proceedings of the Thirteenth EuroSys Conference*. EuroSys '18. Porto, Portugal: ACM, 2018, 21:1–21:15. ISBN: 978-1-4503-5584-1. DOI: 10.1145/3190508.3190550 (cit. on p. 85).

[Gen18] Gen-Z Consortium. *Gen-Z Core Specification*. Version 1.0. Feb. 2018 (cit. on p. 42).

[Ger+15] Simon Gerber, Gerd Zellweger, Reto Achermann, Kornilios Kourtis, Timothy Roscoe, and Dejan Milojicic. "Not Your Parents' Physical Address Space". In: *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems*. HOTOS'15. Switzerland: USENIX Association, 2015, pp. 16–16 (cit. on pp. 8, 9, 63, 94).

[Ger18] Simon Gerber. "Authorization, Protection, and Allocation of Memory in a Large System". en. PhD thesis. ETH Zurich, 2018. DOI: 10.3929/ethz-b-000296835 (cit. on pp. 10, 79, 84, 159, 175, 181, 184, 194, 195, 199).

[GGA05] Malay K. Ganai, Aarti Gupta, and Pranav Ashar. "Verification of Embedded Memory Systems Using Efficient Memory Modeling". In: *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 2*. DATE '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 1096–1101. ISBN: 0-7695-2288-2. DOI: 10.1109/DATE.2005.325 (cit. on p. 71).

[GHS16] Jayneel Gandhi, Mark D. Hill, and Michael M. Swift. "Agile Paging: Exceeding the Best of Nested and Shadow Paging". In: *Proceedings of the 43rd International Symposium on Computer Architecture*. ISCA '16. Seoul, Republic of Korea: IEEE Press, 2016, pp. 707–718. ISBN: 978-1-4673-8947-1. DOI: 10.1109/ISCA.2016.67 (cit. on p. 38).

[GKG08] Chunyang Gou, Georgi K. Kuzmanov, and Georgi N. Gaydadjiev. "Sams: Single-affiliation Multiple-stride Parallel Memory Scheme". In: *Proceedings of the 2008 Workshop on Memory Access on Future Processors: A Solved Problem?* MAW '08. Ischia, Italy: ACM, 2008, pp. 350–368. ISBN: 978-1-60558-091-3. DOI: 10.1145/1366219.1366220 (cit. on p. 41).

[GKG10] Chunyang Gou, Georgi Kuzmanov, and Georgi N. Gaydadjiev. "SAMS Multi-layout Memory: Providing Multiple Views of Data to Boost SIMD Performance". In: *Proceedings of the 24th ACM International Conference on Supercomputing*. ICS '10. Tsukuba, Ibaraki, Japan: ACM, 2010, pp. 179–188. ISBN: 978-1-4503-0018-6. DOI: 10.1145/1810085.1810111 (cit. on p. 41).

[Go19] Go. *The Go Programming Langugage*. https://golang.org/. Accessed: 27. December 2019. The Go Project, Dec. 2019 (cit. on p. 73).

[Gol73] R. P. Goldberg. "Architecture of Virtual Machines". In: *Proceedings of the Workshop on Virtual Computer Systems*. Cambridge, Massachusetts, USA: ACM, 1973, pp. 74–112. ISBN: 978-1-4503-7427-9. DOI: 10.1145/800122.803950. URL: http://doi.acm.org/10.1145/800122.803950 (cit. on p. 23).

[Gon19] Xiling Gong. "Exploiting Qualcomm WLAN and Modem Over the Air". In: *Proceedings of the BlackHat USA 2019*. Las Vegas, USA, 2019. URL: https://i.blackhat.com/USA-19/Thursday/us-19-Pi-Exploiting-Qualcomm-WLAN-And-Modem-Over-The-Air-wp.pdf (cit. on pp. 58, 156, 165).

[Gor04] Mel Gorman. *Understanding the Linux Virtual Memory Manager*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2004. ISBN: 0131453483 (cit. on pp. 56, 189).

[Gu+16] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. "CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels". In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. OSDI'16. Savannah, GA, USA: USENIX Association, 2016, pp. 653–669. ISBN: 978-1-931971-33-1 (cit. on pp. 5, 76, 93, 181).

[Hae+19] Roni Haecki, Lukas Humbel, Reto Achermann, David Cock, Daniel Schwyn, and Timothy Roscoe. *CleanQ: a lightweight, uniform, formally specified interface for intra-machine data transfer*. 2019. arXiv: 1911.08773 [cs.OS] (cit. on p. 158).

[Han70] Per Brinch Hansen. "The Nucleus of a Multiprogramming System". In: *Commun. ACM* 13.4 (Apr. 1970), pp. 238–241. ISSN: 0001-0782. DOI: 10.1145/362258.362278 (cit. on p. 80).

[Han99] Steven M. Hand. "Self-paging in the Nemesis Operating System". In: *Proceedings of the Third Symposium on Operating Systems Design and Implementation*. OSDI '99. New Orleans, Louisiana, USA: USENIX Association, 1999, pp. 73–86. ISBN: 1-880446-39-1 (cit. on p. 74).

[Har85] Norman Hardy. "KeyKOS Architecture". In: *SIGOPS Oper. Syst. Rev.* 19.4 (Oct. 1985), pp. 8–25. ISSN: 0163-5980. DOI: 10.1145/858336.858337 (cit. on pp. 81, 181, 195).

[Har88] Norm Hardy. "The Confused Deputy: (or Why Capabilities Might Have Been Invented)". In: *SIGOPS Oper. Syst. Rev.* 22.4 (Oct. 1988), pp. 36–38. ISSN: 0163-5980. DOI: 10.1145/54289.871709 (cit. on pp. 166, 170).

[HB92] Warren Hunt and Bishop Brock. "A Formal HDL and its Use in the FM9001 Verification". In: *Philosophical Transactions of The Royal Society B: Biological Sciences* 339 (Apr. 1992), pp. 35–47. DOI: 10.1098/rsta.1992.0024 (cit. on p. 71).

[Hei+98] Gernot Heiser, Kevin Elphinstone, Jerry Vochteloo, Stephen Russell, and Jochen Liedtke. "The Mungi Single-Address-Space Operating System". In: *Software: Practice and Experience* 28.9 (1998), pp. 901–928. DOI: 10.1002/(SICI)1097-024X(19980725)28:9<901::AID-SPE181>3.0.CO;2-7 (cit. on p. 75).

[HHG16] L. Howes, D. Hower, and B.R. Gaster. "Chapter 5 - HSA Memory Model". In: *Heterogeneous System Architecture*. Ed. by Wen-mei W. Hwu. Boston: Morgan Kaufmann, 2016, pp. 53–75. ISBN: 978-0-12-800386-2. DOI: https://doi.org/10.1016/B978-0-12-800386-2.00004-3 (cit. on p. 87).

[HHS18] Swapnil Haria, Mark D. Hill, and Michael M. Swift. "Devirtualizing Memory in Heterogeneous Systems". In: *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '18. Williamsburg, VA, USA: ACM, 2018, pp. 637–650. ISBN: 978-1-4503-4911-6. DOI: 10.1145/3173162.3173194 (cit. on p. 38).

[Hil+17] Marius Hillenbrand, Mathias Gottschlag, Jens Kehne, and Frank Bellosa. "Multiple Physical Mappings: Dynamic DRAM Channel Sharing and Partitioning". In: *Proceedings of the 8th Asia-Pacific Workshop on Systems*. APSys '17. Mumbai, India, 2017, 21:1–21:9. ISBN: 978-1-4503-5197-3. DOI: 10.1145/3124680.3124742 (cit. on pp. 30, 41, 52).

[Hil+19] Matthias Hille, Nils Asmussen, Pramod Bhatotia, and Hermann Härtig. "SemperOS: A Distributed Capability System". In: *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*. USENIX ATC '19. Renton, WA, USA: USENIX Association, 2019, pp. 709–722. ISBN: 9781939133038 (cit. on p. 79).

[Hil17] Marius Hillenbrand. *Physical Address Decoding in Intel Xeon v3/v4 CPUs: A Supplemental Datasheet*. Technical Report. Karlsruhe Institute of Technology, Sept. 2017 (cit. on pp. 30, 41).

[HIT05] Ram Huggahalli, Ravi Iyer, and Scott Tetrick. "Direct Cache Access for High Bandwidth Network I/O". In: *Proceedings of the 32nd Annual International Symposium on Computer Architecture*. ISCA '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 50–59. ISBN: 0-7695-2270-X. DOI: 10.1109/ISCA.2005.23 (cit. on p. 87).

[Hos19] Nora Hossle. "Multiple Address Spaces in a Distributed Capability System". Master's Thesis. Switzerland.: Department of Computer Science, ETH Zurich, 2019 (cit. on pp. 8, 10, 157, 162, 171, 172, 178, 179).

[How10] Jason Howard. *Rock Creek System Address Look up Table & Configuration Registers*. External-architecture Specification (EAS) Revision 0.1. Intel Microprocessor Technology Laboratories, Jan. 2010 (cit. on pp. 56, 62).

[HQS16] Jian Huang, Moinuddin K. Qureshi, and Karsten Schwan. "An Evolutionary Study of Linux Memory Management for Fun and Profit". In: *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*. USENIX ATC '16. Denver, CO, USA: USENIX Association, 2016, pp. 465–478. ISBN: 978-1-931971-30-0 (cit. on pp. 12, 56, 73, 155, 189).

[HS01] Mary W. Hall and Craig S. Steele. "Memory Management in a PIM-Based Architecture". In: *Revised Papers from the Second International Workshop on Intelligent Memory Systems*. IMS '00. London, UK, UK: Springer-Verlag, 2001, pp. 104–121. ISBN: 3-540-42328-1 (cit. on p. 43).

[HSA14] HSA Foundation. *HSA Platform System Architecture Specification*. Version 2.3. Provisional 1.0. Apr. 18, 2014 (cit. on pp. 50, 158).

[HSA16] HSA Foundation. *HSA Runtime Programmer's Reference Manual*. Version: 1.1.4. Oct. 2016 (cit. on p. 158).

[HSH81] Merle E. Houdek, Frank G. Soltis, and Roy L. Hoffman. "IBM System/38 Support for Capability-Based Addressing". In: *Proceedings of the 8th Annual Symposium on Computer Architecture*. ISCA '81. Minneapolis, Minnesota, USA: IEEE Computer Society Press, 1981, pp. 341–348 (cit. on pp. 75, 82).

[Hua+15] Jian Huang, Anirudh Badam, Moinuddin K. Qureshi, and Karsten Schwan. "Unified Address Translation for Memory-mapped SSDs with FlashMap". In: *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. ISCA '15. Portland, Oregon: ACM, 2015, pp. 580–591. ISBN: 978-1-4503-3402-0. DOI: 10.1145/2749469.2750420 (cit. on p. 40).

[Hum+17] Lukas Humbel, Reto Achermann, David Cock, and Timothy Roscoe. "Towards Correct-by-Construction Interrupt Routing on Real Hardware". In: *Proceedings of the 9th Workshop on Programming Languages and Operating Systems*. PLOS'17. Shanghai, China: ACM, 2017, pp. 8–14. ISBN: 978-1-4503-5153-9. DOI: 10.1145/3144555.3144557 (cit. on pp. 105, 254).

[Hun+17] Warren A. Hunt, Matt Kaufmann, J Strother Moore, and Anna Slobodova. "Industrial hardware and software verification with ACL2". In: *Phil. Trans. R. Soc. A* 375.2104 (2017). ISSN: 1364-503X. DOI: 10.1098/rsta.2015.0399 (cit. on p. 72).

[HWL96] H. Hartig, J. Wolter, and J. Liedtke. "Flexible-sized Page-objects". In: *Proceedings of the 5th International Workshop on Object Orientation in Operating Systems (IWOOOS '96)*. IWOOOS '96. Washington, DC, USA: IEEE Computer Society, 1996, pp. 102–. ISBN: 0-8186-7692-2 (cit. on p. 77).

[Hyp08] HyperTransport Consortium. *HTX3 Specification*. HTC20080701-00030-0001. June 2008 (cit. on p. 51).

[IBM17] IBM Corporation. *POWER ISA - Book III*. Version 3.0B. Mar. 2017 (cit. on p. 17).

[IBM18] IBM Corporation. *POWER9 Processor User's Manual - OpenPOWER*. Version 2.0. Apr. 2018 (cit. on pp. 17, 18, 22).

[Int09] Intel Corporation. *An Introduction to the Intel QuickPath Interconnect*. 320412-001US. Jan. 2009 (cit. on p. 51).

[Int10a] Intel Corporation. *Intel Itanium Architecture Software Developers Manaul*. 245318, Revision 2.3. May 2010 (cit. on pp. 15, 23, 74).

[Int10b] Intel Corporation. *SCC External Architecture Specification(EAS)*. Revision 1.1. Nov. 2010 (cit. on pp. 26, 36, 160).

[Int13] Intel Corporation. *Intel C600 Series Chipset and Intel x79 Express Chipset*. 326514-002. Apr. 2013 (cit. on p. 48).

[Int14a] Intel Corporation. *Intel Manycore Platform Software Stack (Intel MPSS)*. Revision 3.3. Version v3.3. July 2014 (cit. on p. 26).

[Int14b] Intel Corporation. *Intel Xeon Phi Coprocessor System Software Developers Guide*. System Software Developers Guide. Version SKU 328207-003EN. Mar. 2014 (cit. on pp. 48, 56, 160).

[Int17] Intel Corporation. *Fast memcpy with SPDK and Intel I/OAT DMA Engine*. Apr. 2017 (cit. on p. 48).

[Int19a] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*. 325462-069US. Jan. 2019 (cit. on pp. 12, 15, 18–22, 24–26, 34, 50, 53, 55, 137, 156).

[Int19b] Intel Corporation. *Intel Virtualization Technology for Directed I/O - Architecture Specification*. D51397-011, Revision 3.1. June 2019 (cit. on pp. 15, 22, 32, 33, 48, 217).

[ITD94] ITD. *IDT R30xx Family Software Reference Manual*. Revision 1.0. Integrated Device Technology, Inc. 1994 (cit. on p. 28).

[ITD95] ITD. *IDT79R4600 TM and IDT79R4700 TM RISC Processor Hardware User's Manual*. Revision 2.0. Integrated Device Technology, Inc. Apr. 1995 (cit. on pp. 23, 28, 74, 75, 91, 125, 134, 146, 148).

[Jon+79] Anita K. Jones, Robert J. Chansler Jr., Ivor Durham, Karsten Schwan, and Steven R. Vegdahl. "StarOS, a Multiprocessor Operating System for the Support of Task Forces". In: *Proceedings of the Seventh ACM Symposium on Operating Systems Principles*. SOSP '79. Pacific Grove, California, USA: ACM, 1979, pp. 117–127. ISBN: 0-89791-009-5. DOI: 10.1145/800215.806579 (cit. on p. 81).

[KA18] Mahmood Jasim Khalsan and Michael Opoku Agyeman. "An Overview of Prevention/Mitigation Against Memory Corruption Attack". In: *Proceedings of the 2nd International Symposium on Computer Science and Intelligent Control*. ISCSIC '18. Stockholm, Sweden: ACM, 2018, 42:1–42:6. ISBN: 978-1-4503-6628-1. DOI: 10.1145/3284557.3284564 (cit. on p. 58).

[Kae+15] Stefan Kaestle, Reto Achermann, Timothy Roscoe, and Tim Harris. "Shoal: Smart Allocation and Replication of Memory for Parallel Programs". In: *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*. USENIX ATC '15. Santa Clara, CA: USENIX Association, 2015, pp. 263–276. ISBN: 978-1-931971-225 (cit. on pp. 29, 46, 83–85, 249).

[Kae+16] Stefan Kaestle, Reto Achermann, Roni Haecki, Moritz Hoffmann, Sabela Ramos, and Timothy Roscoe. "Machine-aware Atomic Broadcast Trees for Multicores". In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. OSDI'16. Savannah, GA, USA: USENIX Association, 2016, pp. 33–48. ISBN: 978-1-931971-33-1 (cit. on pp. 83, 84, 86, 249).

[Kar+15] Vasileios Karakostas, Jayneel Gandhi, Furkan Ayar, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman Ünsal. "Redundant Memory Mappings for Fast Access to Large Memories". In: *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. ISCA '15. Portland, Oregon: ACM, 2015, pp. 66–78. ISBN: 978-1-4503-3402-0. DOI: 10.1145/2749469.2749471 (cit. on p. 38).

[KCE92] Eric J. Koldinger, Jeffrey S. Chase, and Susan J. Eggers. "Architecture Support for Single Address Space Operating Systems". In: *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS V. Boston, Massachusetts, USA: ACM, 1992, pp. 175–186. ISBN: 0-89791-534-8. DOI: 10.1145/143365.143508 (cit. on pp. 38, 40, 74).

[Kee15] Kimberly Keeton. "The Machine: An Architecture for Memory-centric Computing". In: *Proceedings of the 5th International Workshop on Runtime and Operating Systems for Supercomputers*. ROSS '15. Portland, OR, USA: ACM, 2015, 1:1–1:1. ISBN: 978-1-4503-3606-2. DOI: 10.1145/2768405.2768406 (cit. on pp. 42, 55, 61).

[Kha+17] Samira Khan, Chris Wilkerson, Zhe Wang, Alaa R. Alameldeen, Donghyuk Lee, and Onur Mutlu. "Detecting and Mitigating Data-dependent DRAM Failures by Exploiting Current Memory Content". In: *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-50 '17. Cambridge, Massachusetts: ACM, 2017, pp. 27–40. ISBN: 978-1-4503-4952-9. DOI: 10.1145/3123939.3123945 (cit. on p. 62).

[Khr18] Khronos OpenCL Working Group. *The OpenCL Specification*. Version: 2.1, Document Revision: 24. Feb. 2018 (cit. on pp. 50, 87, 88, 158).

[Kil+62] T. Kilburn, D. B. G. Edwards, M. J. Lanigan, and F. H. Sumner. "One-Level Storage System". In: *IRE Transactions on Electronic Computers* EC-11.2 (Apr. 1962), pp. 223–235. ISSN: 0367-9950. DOI: 10.1109/TEC.1962.5219356 (cit. on pp. 74, 82, 181).

[Kle+09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. "seL4: Formal Verification of an OS Kernel". In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. SOSP '09. Big Sky, Montana, USA: ACM, 2009, pp. 207–220. ISBN: 978-1-60558-752-3. DOI: 10.1145/1629575.1629596 (cit. on pp. 5, 76, 79, 93, 159, 181, 194, 195).

[Kle08] Andi Kleen. *numa - NUMA policy library*. Version 2.0. https://linux.die.net/man/3/numa. Accessed 09.01.2020. 2008 (cit. on pp. 84, 182).

[Koc+18] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. "Spectre Attacks: Exploiting Speculative Execution". In: *arXiv preprint arXiv:1801.01203* (Jan. 2018) (cit. on pp. 131, 146).

[Kri+06] Orran Krieger, Marc Auslander, Bryan Rosenburg, Robert W. Wisniewski, Jimi Xenidis, Dilma Da Silva, Michal Ostrowski, Jonathan Appavoo, Maria Butrico, Mark Mergen, Amos Waterland, and Volkmar Uhlig. "K42: Building a Complete Operating System". In: *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*. EuroSys '06. Leuven, Belgium: ACM, 2006, pp. 133–145. ISBN: 1-59593-322-0. DOI: 10.1145/1217935.1217949 (cit. on p. 78).

[Kwo+18] Donghyun Kwon, Kuenwhee Oh, Junmo Park, Seungyong Yang, Yeongpil Cho, Brent Byunghoon Kang, and Yunheung Paek. "Hypernel: A Hardware-assisted Framework for Kernel Protection Without Nested Paging". In: *Proceedings of the 55th Annual Design Automation Conference*. DAC '18. San Francisco, California: ACM, 2018, 34:1–34:6. ISBN: 978-1-4503-5700-5. DOI: 10.1145/3195970.3196061 (cit. on pp. 40, 187).

[Kwo+19] Donghyun Kwon, Hayoon Yi, Yeongpil Cho, and Yunheung Paek. "Safe and Efficient Implementation of a Security System on ARM Using Intra-level Privilege Separation". In: *ACM Trans. Priv. Secur.* 22.2 (Feb. 2019), 10:1–10:30. ISSN: 2471-2566. DOI: 10.1145/3309698. URL: http://doi.acm.org/10.1145/3309698 (cit. on p. 186).

[Lam13] Christoph Lameter. "NUMA (Non-Uniform Memory Access): An Overview". In: *Queue* 11.7 (July 2013), 40:40–40:51. ISSN: 1542-7730. DOI: 10.1145/2508834.2513149 (cit. on pp. 29, 46, 51).

[Lam74] Butler W Lampson. "Protection". In: *ACM SIGOPS Operating Systems Review* 8.1 (1974), pp. 18–24 (cit. on pp. 162, 168, 169).

[Lev84] Henry M. Levy. *Capability-Based Computer Systems*. Newton, MA, USA: Butterworth-Heinemann, 1984. ISBN: 0932376223 (cit. on pp. 40, 81, 82, 159, 181, 195).

[Lie94] Jochen Liedtke. "Address Space Sparsity and Fine Granularity". In: *Proceedings of the 6th Workshop on ACM SIGOPS European Workshop: Matching Operating Systems to Application Needs*. EW 6. Wadern, Germany: Association for Computing Machinery, 1994, pp. 78–81. ISBN: 9781450373388. DOI: 10.1145/504390.504411. URL: https://doi.org/10.1145/504390.504411 (cit. on p. 75).

[Lie95] Jochen Liedtke. "On Micro-kernel Construction". In: *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*. SOSP '95. Copper Mountain, Colorado, USA: ACM, 1995, pp. 237–250. ISBN: 0-89791-715-4. DOI: 10.1145/224056.224075 (cit. on p. 77).

[Lie96] Jochen Liedtke. “On the realization of huge sparsely occupied and fine grained address spaces”. PhD thesis. TU Berlin, 1996. ISBN: 3-486-24185-0. URL: http://d-nb.info/949253553 (cit. on p. 75).

[Lin19a] Linux. *The Linux Kernel documentation*. Online. Accessed: 27. December 2019. The Kernel Development Community, 2019. URL: https://www.kernel.org/doc/html/latest/ (cit. on p. 73).

[Lin19b] Linux Kernel Documentation. *Device Tree Source Format*. Version 1.0. Online. https://git.kernel.org/pub/scm/utils/dtc/dtc.git/plain/Documentation/dts-format.txt. Accessed 30. December 2019. Dec. 2019 (cit. on p. 93).

[Lin19c] Linux Kernel Documentation. *Heterogeneous Memory Management (HMM)*. version 5.0. Apr. 2019 (cit. on pp. 50, 73, 182).

[lin19] linux-kvm.org. *Kernel Virtual Machine (KVM)*. Online. https://www.linux-kvm.org/page/Main_Page. Accessed 28. December 2019. Dec. 2019 (cit. on p. 80).

[Lip+18] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. “Meltdown: Reading Kernel Memory from User Space”. In: *Proceedings of the 27th USENIX Conference on Security Symposium*. SEC’18. Baltimore, MD, USA: USENIX Association, 2018, pp. 973–990. ISBN: 978-1-931971-46-1 (cit. on pp. 131, 146).

[LRW91] Monica D. Lam, Edward E. Rothberg, and Michael E. Wolf. “The Cache Performance and Optimizations of Blocked Algorithms”. In: *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS IV. Santa Clara, California, USA: ACM, 1991, pp. 63–74. ISBN: 0-89791-380-9. DOI: 10.1145/106972.106981 (cit. on p. 86).

[LSM14] Janghaeng Lee, Mehrzad Samadi, and Scott Mahlke. “VAST: The Illusion of a Large Memory Space for GPUs”. In: *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*. PACT ’14. Edmonton, AB, Canada: ACM, 2014, pp. 443–454. ISBN: 978-1-4503-2809-8. DOI: 10.1145/2628071.2628075 (cit. on p. 87).

[LYB17] Feilong Liu, Lingyan Yin, and Spyros Blanas. “Design and Evaluation of an RDMA-aware Data Shuffling Operator for Parallel Database Systems”. In: *Proceedings of the Twelfth European Conference on Computer Systems*. EuroSys ’17. Belgrade, Serbia: ACM, 2017, pp. 48–63. ISBN: 978-1-4503-4938-3. DOI: 10.1145/3064176.3064202 (cit. on p. 85).

[Mah+12] Aurèle Mahéo, Souad Koliaï, Patrick Carribault, Marc Pérache, and William Jalby. "Adaptive OpenMP for Large NUMA Nodes". In: *Proceedings of the 8th International Conference on OpenMP in a Heterogeneous World*. IWOMP'12. Rome, Italy: Springer-Verlag, 2012, pp. 254–257. ISBN: 978-3-642-30960-1. DOI: 10.1007/978-3-642-30961-8_20 (cit. on p. 85).

[Mar+19] A Theodore Markettos, Colin Rothwell, Brett F Gutstein, Allison Pearce, Peter G Neumann, Simon W Moore, and Robert NM Watson. "Thunderclap: Exploring Vulnerabilities in Operating System IOMMU Protection via DMA from Untrustworthy Peripherals". In: *NDSS*. 2019 (cit. on pp. 48, 58, 155, 160, 165).

[Mat+10] Timothy G. Mattson, Michael Riepen, Thomas Lehnig, Paul Brett, Werner Haas, Patrick Kennedy, Jason Howard, Sriram Vangal, Nitin Borkar, Greg Ruhl, and Saurabh Dighe. "The 48-core SCC Processor: The Programmer's View". In: *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11. ISBN: 978-1-4244-7559-9. DOI: 10.1109/SC.2010.53 (cit. on p. 56).

[Mat+14] Rivalino Matias, Marcela Prince, Lúcio Borges, Claudio Sousa, and Luan Henrique. "An Empirical Exploratory Study on Operating System Reliability". In: *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. SAC '14. Gyeongju, Republic of Korea: ACM, 2014, pp. 1523–1528. ISBN: 978-1-4503-2469-4. DOI: 10.1145/2554850.2555021 (cit. on p. 58).

[McC98] Mark McCall. *The AS/400 Programmer's Handbook: A Toolbox of Examples for Every AS/400 Programmer*. 1st. MC Press, LLC, 1998. ISBN: 1883884489 (cit. on p. 75).

[Mel17] Mellanox Technologies. *ConnectX-3 Pro Ethernet Single and Dual QSFP+ Port Adapter Card User Manual*. MCX313A-BCCT, MCX314A-BCCT, Rev 1.7. Jan. 2017 (cit. on p. 160).

[Mic18] Microsoft. *Hyper-V Architecture*. Online. https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/reference/hyper-v-architecture. Accessed 28. December 2019. Jan. 2018 (cit. on p. 80).

[Mil+00] Dejan Milojicic, Steve Hoyle, Alan Messer, Albert Munoz, Lance Russell, Tom Wylegala, Vivekanand Vellanki, and Stephen Childs. "Global Memory Management for a Multi Computer System". In: *Proceedings of the 4th Conference on USENIX Windows Systems Symposium - Volume 4*. WSS'00. Seattle, Washington: USENIX Association, 2000, pp. 12–12 (cit. on p. 83).

[Mir+17] Vladimir Mironov, Yuri Alexeev, Kristopher Keipert, Michael D'mello, Alexander Moskovsky, and Mark S. Gordon. "An Efficient MPI/openMP Parallelization of the Hartree-Fock Method for the Second Generation of Intel&Reg; Xeon Phi&Trade; Processor". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '17. Denver, Colorado: ACM, 2017, 39:1–39:12. ISBN: 978-1-4503-5114-0. DOI: 10.1145/3126908.3126956 (cit. on p. 85).

[MKM12] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. "Cache Craftiness for Fast Multicore Key-value Storage". In: *Proceedings of the 7th ACM European Conference on Computer Systems*. EuroSys '12. Bern, Switzerland: ACM, 2012, pp. 183–196. ISBN: 978-1-4503-1223-3. DOI: 10.1145/2168836.2168855 (cit. on p. 86).

[MMT16] Alex Markuze, Adam Morrison, and Dan Tsafrir. "True IOMMU Protection from DMA Attacks: When Copy is Faster Than Zero Copy". In: *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '16. Atlanta, Georgia, USA: ACM, 2016, pp. 249–262. ISBN: 978-1-4503-4091-5. DOI: 10.1145/2872362.2872379 (cit. on pp. 48, 155, 160, 165).

[Mor+16] Benot Morgan, Eric Alata, Vincent Nicomette, and Mohamed Kaaniche. "Bypassing IOMMU Protection against I/O Attacks". In: *2016 Seventh Latin-American Symposium on Dependable Computing (LADC)*. Oct. 2016, pp. 145–150. DOI: 10.1109/LADC.2016.31 (cit. on pp. 160, 165).

[Mor+18] Benot Morgan, Eric Alata, Vincent Nicomette, and Mohamed Kaaniche. "IOMMU Protection Against I/O Attacks: A Vulnerability and a Proof of Concept". In: *Journal of the Brazilian Computer Society* 24.1 (Jan. 2018), p. 2. ISSN: 1678-4804. DOI: 10.1186/s13173-017-0066-7 (cit. on pp. 155, 160).

[MPA05] Jeremy Manson, William Pugh, and Sarita V. Adve. "The Java Memory Model". In: *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '05. Long Beach, California, USA: ACM, 2005, pp. 378–391. ISBN: 1-58113-830-X. DOI: 10.1145/1040305.1040336 (cit. on p. 88).

[MS09] Ross McIlroy and Joe Sventek. "Hera-JVM: Abstracting Processor Heterogeneity Behind a Virtual Machine". In: *Proceedings of the 12th Conference on Hot Topics in Operating Systems*. HotOS'09. Monte Verità, Switzerland: USENIX Association, 2009, pp. 15–15. URL: http://dl.acm.org/citation.cfm?id=1855568.1855583 (cit. on p. 88).

[Nel+17] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. "Hyperkernel: Push-Button Verification of an OS Kernel". In: *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP '17. Shanghai, China: ACM, 2017, pp. 252–269. ISBN: 978-1-4503-5085-3. DOI: 10.1145/3132747.3132748 (cit. on p. 76).

[Nel+19] Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. "Scaling Symbolic Evaluation for Automated Verification of Systems Code with Serval". In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. SOSP '19. Huntsville, Ontario, Canada: ACM, 2019, pp. 225–242. ISBN: 978-1-4503-6873-5. DOI: 10.1145/3341301.3359641 (cit. on p. 70).

[Nig+09] Edmund B. Nightingale, Orion Hodson, Ross McIlroy, Chris Hawblitzel, and Galen Hunt. "Helios: Heterogeneous Multiprocessing with Satellite Kernels". In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. SOSP '09. Big Sky, Montana, USA: ACM, 2009, pp. 221–234. ISBN: 978-1-60558-752-3. DOI: 10.1145/1629575.1629597 (cit. on p. 79).

[Nov+14] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. "Scale-out NUMA". In: *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '14. Salt Lake City, Utah, USA: ACM, 2014, pp. 3–18. ISBN: 978-1-4503-2305-5. DOI: 10.1145/2541940.2541965 (cit. on pp. 46, 85).

[NVD13a] NATIONAL VULNERABILITY DATABASE - NVD. *CVE-2013-4329*. Online. National Institute of Standards and Technology, Sept. 2013. URL: https://nvd.nist.gov/vuln/detail/CVE-2013-4329 (cit. on p. 155).

[NVD13b] NATIONAL VULNERABILITY DATABASE - NVD. *CVE-2013-6375*. Online. National Institute of Standards and Technology, Nov. 2013. URL: https://nvd.nist.gov/vuln/detail/CVE-2013-6375 (cit. on p. 155).

[NVD13c] NATIONAL VULNERABILITY DATABASE - NVD. *CVE-2013-6400*. Online. National Institute of Standards and Technology, Dec. 2013. URL: https://nvd.nist.gov/vuln/detail/CVE-2013-6400 (cit. on p. 155).

[NVD17a] NATIONAL VULNERABILITY DATABASE - NVD. *CVE-2015-6994*. Online. National Institute of Standards and Technology, Jan. 2017. URL: https://nvd.nist.gov/vuln/detail/CVE-2015-6994 (cit. on p. 155).

[NVD17b] NATIONAL VULNERABILITY DATABASE - NVD. *CVE-2016-5349*. Online. National Institute of Standards and Technology, Apr. 2017. URL: https://nvd.nist.gov/vuln/detail/CVE-2016-5349/ (cit. on p. 156).

[NVD17c] NATIONAL VULNERABILITY DATABASE - NVD. *CVE-2017-6295*. Online. National Institute of Standards and Technology, May 2017. URL: https://nvd.nist.gov/vuln/detail/CVE-2017-6295 (cit. on p. 156).

[NVD18] NATIONAL VULNERABILITY DATABASE - NVD. *CVE-2018-11994*. Online. National Institute of Standards and Technology, Nov. 2018. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-11994 (cit. on p. 155).

[NVD19a] NATIONAL VULNERABILITY DATABASE - NVD. Online. National Institute of Standards and Technology, May 2019. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-0152 (cit. on p. 156).

[NVD19b] NATIONAL VULNERABILITY DATABASE - NVD. *CVE-2015-4421*. Online. National Institute of Standards and Technology, May 2019. URL: https://nvd.nist.gov/vuln/detail/CVE-2015-4421 (cit. on p. 156).

[NVD19c] NATIONAL VULNERABILITY DATABASE - NVD. *CVE-2015-4422*. Online. National Institute of Standards and Technology, May 2019. URL: https://nvd.nist.gov/vuln/detail/CVE-2015-4422 (cit. on p. 156).

[NVD19d] NATIONAL VULNERABILITY DATABASE - NVD. *CVE-2019-1010298*. Online. National Institute of Standards and Technology, May 2019. URL: https://nvd.nist.gov/vuln/detail/CVE-2019-1010298 (cit. on p. 156).

[NVD19e] NATIONAL VULNERABILITY DATABASE - NVD. *CVE-2019-10538 - Modem into Linux Kernel issue*. Online. National Institute of Standards and Technology, Aug. 2019. URL: https://nvd.nist.gov/vuln/detail/CVE-2019-10538 (cit. on p. 165).

[NVD19f] NATIONAL VULNERABILITY DATABASE - NVD. *CVE-2019-10539 - Compromise WLAN Issue*. Online. National Institute of Standards and Technology, Aug. 2019. URL: https://nvd.nist.gov/vuln/detail/CVE-2019-10539 (cit. on p. 165).

[NVD19g] NATIONAL VULNERABILITY DATABASE - NVD. *CVE-2019-10540 - WLAN into Modem issue*. Online. National Institute of Standards and Technology, Aug. 2019. URL: https://nvd.nist.gov/vuln/detail/CVE-2019-10540 (cit. on p. 165).

[NVI13] NVIDIA Corporation. *Unified Memory in CUDA 6*. Nov. 2013 (cit. on pp. 50, 87, 158).

[NVI17] NVIDIA Corporation. *NVIDIA Parker Series SoC Technical Reference Manual*. v.1.0p. June 2017 (cit. on pp. 52, 160).

[NVI19] NVIDIA Corporation. *CUDA C PROGRAMMING GUIDE*. PG-02829-001, Version v10.1. Aug. 2019 (cit. on p. 87).

[NW77] R. M. Needham and R. D.H. Walker. "The Cambridge CAP Computer and Its Protection System". In: *Proceedings of the Sixth ACM Symposium on Operating Systems Principles*. SOSP '77. West Lafayette, Indiana, USA: ACM, 1977, pp. 1–10. DOI: 10.1145/800214.806541 (cit. on pp. 40, 80).

[NXP19] NXP. *i.MX 8DualXPlus/8QuadXPlus Applications Processor Reference Manual*. REV 1, www.nxp.com/docs/en/user-guide/IMX8QXPMEKHUG.pdf. Jan. 2019 (cit. on p. 160).

[OAS18] OASIS Open. *Virtual I/O Device (VIRTIO) Specification*. Version 1.1. Accessed: 13. December 2019. 2018. URL: https://docs.oasis-open.org/virtio/virtio/v1.1/csprd01/virtio-v1.1-csprd01.pdf (cit. on p. 158).

[Ole+18] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. "Intel MPX Explained: A Cross-layer Analysis of the Intel MPX System Stack". In: *Proc. ACM Meas. Anal. Comput. Syst.* 2.2 (June 2018), 28:1–28:30. ISSN: 2476-1249. DOI: 10.1145/3224423 (cit. on p. 186).

[Ora19] Oracle Corporation. *Oracle VM VirtualBox User Manual*. Version 6.1.0. 2019 (cit. on p. 80).

[ORS13] Jason Orcutt, Rajeev Ram, and Vladimir Stojanovic. "CMOS Photonics for High Performance Interconnects". In: *Optical Fiber Telecommunications (Sixth Edition)*. Ed. by Ivan P. Kaminow, Tingye Li, and Alan E. Willner. Sixth Edition. Optics and Photonics. Boston: Academic Press, 2013, pp. 419–460. DOI: https://doi.org/10.1016/B978-0-12-396958-3.00012-3 (cit. on p. 42).

[OSS09] Scott Owens, Susmit Sarkar, and Peter Sewell. "A Better x86 Memory Model: X86-TSO". In: *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*. TPHOLs '09. Munich, Germany: Springer-Verlag, 2009, pp. 391–407. ISBN: 978-3-642-03358-2. DOI: 10.1007/978-3-642-03359-9_27. URL: http://dx.doi.org/10.1007/978-3-642-03359-9_27 (cit. on p. 70).

[Pat+97] David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. "A Case for Intelligent RAM". In: *IEEE Micro* 17.2 (Mar. 1997), pp. 34–44. ISSN: 0272-1732. DOI: 10.1109/40.592312 (cit. on p. 160).

[PCI13] PCI-SIG. *PCIe Hot Plug ECN*. Revision 3.1. Oct. 2013 (cit. on pp. 50, 153).

[PCI17] PCI-SIG. *PCI Express Base Specification*. Revision 4.0, Version 1.0. Oct. 2017 (cit. on pp. 47, 66, 87).

[Pet+14] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. "Arrakis: The Operating System is the Control Plane". In: *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*. OSDI'14. Broomfield, CO: USENIX Association, 2014, pp. 1–16. ISBN: 978-1-931971-16-4 (cit. on pp. 80, 84).

[PG74] Gerald J. Popek and Robert P. Goldberg. "Formal Requirements for Virtualizable Third Generation Architectures". In: *Commun. ACM* 17.7 (July 1974), pp. 412–421. ISSN: 0001-0782. DOI: 10.1145/361011.361073 (cit. on pp. 12, 23, 157).

[Pow11] Power.org. *Standard for Embedded Power Architecture Platform Requirements(ePAPR)*. Version 1.1. Apr. 2011 (cit. on p. 68).

[Qua18] Qualcomm Technologies, Inc. *Qualcomm Snapdragon 845 Mobile Platform*. Product Brief. https://www.qualcomm.com/media/documents/files/snapdragon-845-mobile-platform-product-brief.pdf. 2018 (cit. on p. 53).

[Ras+87] Richard Rashid, Avadis Tevanian, Michael Young, David Golub, Robert Baron, David Black, William Bolosky, and Jonathan Chew. "Machine-independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures". In: *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems*. ASPLOS II. Palo Alto, California, USA: IEEE Computer Society Press, 1987, pp. 31–39. ISBN: 0-8186-0805-6. DOI: 10.1145/36206.36181 (cit. on p. 77).

[Rec+07] R. Recio, B. Metzler, P. Culley, J. Hilland, and D. Garcia. *RDMA Protocol Specification*. RFC 5040. Oct. 2007, pp. 1–65. URL: http://www.rfc-editor.org/rfc/rfc5040.txt (cit. on pp. 45, 85).

[Red18] RedHat. *Red Hat Enterprise Linux 7 - Virtualization Tuning and Optimization Guide*. Version 2018-10-25. Oct. 2018 (cit. on p. 84).

[Rei16] Alastair Reid. "Trustworthy Specifications of ARM V8-A and V8-M System Level Architecture". In: *Proceedings of the 16th Conference on Formal Methods in Computer-Aided Design*. FMCAD '16. Mountain View, California: FMCAD Inc, 2016, pp. 161–168. ISBN: 978-0-9835678-6-8 (cit. on pp. 68, 70, 71).

[Rei17a] Alastair Reid. *ARM Releases Machine Readable Architecture Specification*. https://alastairreid.github.io/ARM-v8a-xml-release/. Accessed: 27. December 2019. Apr. 2017 (cit. on pp. 68, 70).

[Rei17b] Alastair Reid. "Who Guards the Guards? Formal Validation of the Arm V8-m Architecture Specification". In: *Proc. ACM Program. Lang.* 1.OOPSLA (Oct. 2017), 88:1–88:24. ISSN: 2475-1421. DOI: 10.1145/3133912 (cit. on p. 70).

[RGM16] Lars Richter, Johannes Götzfried, and Tilo Müller. "Isolating Operating System Components with Intel SGX". In: *Proceedings of the 1st Workshop on System Software for Trusted Execution*. SysTEX '16. Trento, Italy: ACM, 2016, 8:1–8:6. ISBN: 978-1-4503-4670-2. DOI: 10.1145/3007788.3007796 (cit. on p. 186).

[RK68] B. Randell and C. J. Kuehner. "Dynamic Storage Allocation Systems". In: *Commun. ACM* 11.5 (May 1968), pp. 297–306. ISSN: 0001-0782. DOI: 10.1145/363095.363138 (cit. on p. 18).

[RLS10] Bogdan F. Romanescu, Alvin R. Lebeck, and Daniel J. Sorin. "Specifying and Dynamically Verifying Address Translation-aware Memory Consistency". In: *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XV. Pittsburgh, Pennsylvania, USA: ACM, 2010, pp. 323–334. ISBN: 978-1-60558-839-1. DOI: 10.1145/1736020.1736057 (cit. on p. 76).

[Rog16] P. Rogers. "Chapter 2 - HSA Overview". In: *Heterogeneous System Architecture*. Ed. by Wen-mei W. Hwu. Boston: Morgan Kaufmann, 2016, pp. 7–18. ISBN: 978-0-12-800386-2. DOI: https://doi.org/10.1016/B978-0-12-800386-2.00001-8 (cit. on p. 87).

[Ros94] Timothy Roscoe. "Linkage in the Nemesis Single Address Space Operating System". In: *SIGOPS Oper. Syst. Rev.* 28.4 (Oct. 1994), pp. 48–55. ISSN: 0163-5980. DOI: 10.1145/191525.191537 (cit. on p. 75).

[Roz+91] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser. "Overview of the CHORUS Distributed Operating Systems". In: *Computing Systems* 1 (1991), pp. 39–69 (cit. on p. 78).

[RR81] Richard F. Rashid and George G. Robertson. “Accent: A Communication Oriented Network Operating System Kernel”. In: *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*. SOSP ’81. Pacific Grove, California, USA: ACM, 1981, pp. 64–75. ISBN: 0-89791-062-1. DOI: 10.1145/800216.806593 (cit. on p. 81).

[Ryz+17] Leonid Ryzhyk, Nikolaj Bjørner, Marco Canini, Jean-Baptiste Jeannin, Cole Schlesinger, Douglas B. Terry, and George Varghese. “Correct by Construction Networks Using Stepwise Refinement”. In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, Mar. 2017, pp. 683–698. ISBN: 978-1-931971-37-9 (cit. on p. 255).

[Sal78] Jerome H. Saltzer. “Naming and Binding of Objects”. In: *Operating Systems, An Advanced Course*. London, UK, UK: Springer-Verlag, 1978, pp. 99–208. ISBN: 3-540-08755-9 (cit. on pp. 95, 151).

[Sar+12] Susmit Sarkar, Kayvan Memarian, Scott Owens, Mark Batty, Peter Sewell, Luc Maranget, Jade Alglave, and Derek Williams. “Synchronising C/C++ and POWER”. In: *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’12. Beijing, China: ACM, 2012, pp. 311–322. ISBN: 978-1-4503-1205-9. DOI: 10.1145/2254064.2254102 (cit. on p. 88).

[SB13] Patrick Stewin and Iurii Bystrov. “Understanding DMA Malware”. In: *Proceedings of the 9th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. DIMVA’12. Heraklion, Crete, Greece: Springer-Verlag, 2013, pp. 21–41. ISBN: 978-3-642-37299-5. DOI: 10.1007/978-3-642-37300-8_2 (cit. on p. 58).

[SBL03] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. “Improving the Reliability of Commodity Operating Systems”. In: *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*. SOSP ’03. Bolton Landing, NY, USA: ACM, 2003, pp. 207–222. ISBN: 1-58113-757-5. DOI: 10.1145/945445.945466. URL: http://doi.acm.org/10.1145/945445.945466 (cit. on p. 58).

[SC97] Frank G. Soltis and Paul Conte. *Inside the AS/400: Featuring the AS/400e Series*. 2nd. 29th Street Press/NEWS/400 Books, 1997. ISBN: 1882419669 (cit. on p. 75).

[Sch+08] Adrian Schüpbach, Simon Peter, Andrew Baumann, Timothy Roscoe, Paul Barham, Tim Harris, and Rebecca Isaacs. “Embracing diversity in the Barrelfish manycore operating system.” In: *Proceedings of the Workshop on Managed Many-Core Systems*. MMCS. Boston, MA, USA, June 2008 (cit. on p. 67).

[Sch+11] Adrian Schüpbach, Andrew Baumann, Timothy Roscoe, and Simon Peter. "A Declarative Language Approach to Device Configuration". In: *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XVI. Newport Beach, California, USA: ACM, 2011, pp. 119–132. ISBN: 978-1-4503-0266-1. DOI: 10.1145/1950365.1950382 (cit. on pp. 67, 207).

[Sch17] Daniel Schwyn. "Hardware Configuration With Dynamically-Queried Formal Models". Master's Thesis. Switzerland.: Department of Computer Science, ETH Zurich, 2017. DOI: 10.3929/ethz-b-000203075 (cit. on pp. 10, 68, 120, 151, 211, 212, 241).

[Sel16] Surenthar Selvaraj. *Overview of an Intel Software Guard Extensions Enclave Life Cycle*. https://software.intel.com/en-us/blogs/2016/12/20/overview-of-an-intel-software-guard-extensions-enclave-life-cycle. Dec. 2016 (cit. on p. 34).

[seL18] seL4 docs. *CapDL*. Online. Accessed: 07. January 2020. 2018. URL: https://docs.sel4.systems/CapDL.html (cit. on p. 252).

[Ses+15] Vivek Seshadri, Gennady Pekhimenko, Olatunji Ruwase, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, Todd C. Mowry, and Trishul Chilimbi. "Page Overlays: An Enhanced Virtual Memory Framework to Enable Fine-grained Memory Management". In: *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. ISCA '15. Portland, Oregon: ACM, 2015, pp. 79–91. ISBN: 978-1-4503-3402-0. DOI: 10.1145/2749469.2750379 (cit. on p. 41).

[Sew+11] Thomas Sewell, Simon Winwood, Peter Gammie, Toby Murray, June Andronick, and Gerwin Klein. "seL4 Enforces Integrity". In: *Interactive Theorem Proving*. Ed. by Markovan Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 325–340. ISBN: 978-3-642-22863-6 (cit. on p. 162).

[Sha+18] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. "LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation". In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. OSDI'18. Carlsbad, CA, USA: USENIX Association, 2018, pp. 69–87. ISBN: 978-1-931971-47-8 (cit. on p. 82).

[Sim+14] Jaewoong Sim, Alaa R. Alameldeen, Zeshan Chishti, Chris Wilkerson, and Hyesoon Kim. "Transparent Hardware Management of Stacked DRAM As Part of Memory". In: *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-47. Cambridge, United Kingdom: IEEE Computer Society, 2014, pp. 13–24. ISBN: 978-1-4799-6998-2. DOI: 10.1109/MICRO.2014.56 (cit. on pp. 40, 41).

[SK17] Hira Taqdees Syeda and Gerwin Klein. "Reasoning about Translation Lookaside Buffers". In: *Proceedings of the 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning*. Maun, Botswana, May 2017, pp. 490–508 (cit. on pp. 71, 150).

[SK18] Hira Taqdees Syeda and Gerwin Klein. "Program Verification in the Presence of Cached Address Translation". In: *International Conference on Interactive Theorem Proving*. Vol. 10895. Oxford, UK: Lecture Notes in Computer Science, July 2018, pp. 542–559. DOI: https://doi.org/10.1007/978-3-319-94821-8_32 (cit. on pp. 71, 150).

[SLL16] Du Shen, Xu Liu, and Felix Xiaozhu Lin. "Characterizing Emerging Heterogeneous Memory". In: *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management*. ISMM 2016. Santa Barbara, CA, USA: ACM, 2016, pp. 13–23. ISBN: 978-1-4503-4317-6. DOI: 10.1145/2926697.2926702. URL: http://doi.acm.org/10.1145/2926697.2926702 (cit. on p. 85).

[SSF99] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. "EROS: A Fast Capability System". In: *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*. SOSP '99. Charleston, South Carolina, USA: ACM, 1999, pp. 170–185. ISBN: 1-58113-140-2. DOI: 10.1145/319151.319163 (cit. on pp. 181, 195).

[SW09] Daniel Schmidt and Norbert Wehn. "DRAM Power Management and Energy Consumption: A Critical Assessment". In: *Proceedings of the 22nd Annual Symposium on Integrated Circuits and System Design: Chip on the Dunes*. SBCCI '09. Natal, Brazil: ACM, 2009, 32:1–32:5. ISBN: 978-1-60558-705-9. DOI: 10.1145/1601896.1601937 (cit. on p. 62).

[SWS14] Pierre Schnarz, Joachim Wietzke, and Ingo Stengel. "Towards Attacks on Restricted Memory Areas Through Co-processors in Embedded multi-OS Environments via Malicious Firmware Injection". In: *Proceedings of the First Workshop on Cryptography and Security in Computing Systems*. CS2 '14. Vienna, Austria: ACM, 2014, pp. 25–30. ISBN: 978-1-4503-2484-7. DOI: 10.1145/2556315.2556318 (cit. on p. 156).

[TB14] Emina Torlak and Rastislav Bodik. "A Lightweight Symbolic Virtual Machine for Solver-aided Host Languages". In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '14. Edinburgh, United Kingdom: ACM, 2014, pp. 530–541. ISBN: 978-1-4503-2784-8. DOI: 10.1145/2594291.2594340 (cit. on p. 70).

[Tex14] Texas Instruments. *OMAP44xx Multimedia Device Technical Reference Manual*. Version AB, Literature Number SWPU235AB. Apr. 2014 (cit. on pp. 29, 30, 32, 52–54, 62, 93, 160).

[THW10] Jan Treibig, Georg Hager, and Gerhard Wellein. "LIKWID: A Lightweight Performance-Oriented Tool Suite for x86 Multicore Environments". In: *Proceedings of the 2010 39th International Conference on Parallel Processing Workshops*. ICPPW '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 207–216. ISBN: 978-0-7695-4157-0. DOI: 10.1109/ICPPW.2010.38 (cit. on pp. 67, 86).

[THW12] Jan Treibig, Georg Hager, and Gerhard Wellein. "likwid-bench: An Extensible Microbenchmarking Platform for x86 Multicore Compute Nodes". In: *Tools for High Performance Computing 2011*. Ed. by Holger Brunst, Matthias S. Müller, Wolfgang E. Nagel, and Michael M. Resch. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 27–36. ISBN: 978-3-642-31476-6 (cit. on p. 67).

[Tia19] TianoCore. *OpenSource UEFI Implementation*. EDK Version II. 2019 (cit. on p. 3).

[UEF15] UEFI. *Advanced Configuration and Power Interface Specification*. Version 6.0. Unified Extensible Firmware Interface Forum. Apr. 2015 (cit. on pp. 3, 67).

[UEF17] UEFI. *Unified Extensible Firmware Interface (UEFI) Specification*. Version 2.7. Unified Extensible Firmware Interface Forum. Aug. 2017 (cit. on pp. 3, 67).

[Uhl+05] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. M. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, and L. Smith. "Intel virtualization technology". In: *Computer* 38.5 (May 2005), pp. 48–56. ISSN: 1558-0814. DOI: 10.1109/MC.2005.163 (cit. on p. 23).

[USB17] USB Implementers Forum. *Universal Serial Bus Specification 3.2 Specification*. Sept. 2017 (cit. on p. 66).

[Vah+19] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. "ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK)". In: *Proceedings of the 28th USENIX Conference on Security Symposium*. SEC'19. Santa Clara, CA, USA: USENIX Association, 2019, pp. 1221–1238. ISBN: 978-1-939133-06-9 (cit. on p. 186).

[Van+18] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-order Execution". In: *Proceedings of the 27th USENIX Conference on Security Symposium*. SEC'18. Baltimore, MD, USA: USENIX Association, 2018, pp. 991–1008. ISBN: 978-1-931971-46-1 (cit. on p. 34).

[Vel01] Miroslav N. Velev. “Automatic Abstraction of Memories in the Formal Verification of Superscalar Microprocessors”. In: *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. TACAS 2001. London, UK, UK: Springer-Verlag, 2001, pp. 252–267. ISBN: 3-540-41865-2 (cit. on p. 71).

[Ver+16] Erik Vermij, Christoph Hagleitner, Leandro Fiorin, Rik Jongerius, Jan van Lunteren, and Koen Bertels. “An Architecture for Near-data Processing Systems”. In: *Proceedings of the ACM International Conference on Computing Frontiers*. CF ’16. Como, Italy: ACM, 2016, pp. 357–360. ISBN: 978-1-4503-4128-8. DOI: 10.1145/2903150.2903478 (cit. on p. 43).

[Ver+17] Erik Vermij, Leandro Fiorin, Rik Jongerius, Christoph Hagleitner, Jan Van Lunteren, and Koen Bertels. “An Architecture for Integrated Near-Data Processors”. In: *ACM Trans. Archit. Code Optim.* 14.3 (Sept. 2017), 30:1–30:25. ISSN: 1544-3566. DOI: 10.1145/3127069 (cit. on p. 43).

[Vil+14] Lluis Vilanova, Muli Ben-Yehuda, Nacho Navarro, Yoav Etsion, and Mateo Valero. “CODOMs: Protecting Software with Code-centric Memory Domains”. In: *Proceeding of the 41st Annual International Symposium on Computer Architecuture*. ISCA ’14. Minneapolis, Minnesota, USA: IEEE Press, 2014, pp. 469–480. ISBN: 978-1-4799-4394-4 (cit. on p. 39).

[VMB15] Pirmin Vogel, Andrea Marongiu, and Luca Benini. “Lightweight Virtual Memory Support for Many-core Accelerators in Heterogeneous Embedded SoCs”. In: *Proceedings of the 10th International Conference on Hardware/Software Codesign and System Synthesis*. CODES ’15. Amsterdam, The Netherlands: IEEE Press, 2015, pp. 45–54. ISBN: 978-1-4673-8321-9 (cit. on p. 158).

[VMw19] VMware. *VMware vSphere Documentation*. Version 6.7. Online. https://docs.vmware.com/en/VMware-vSphere/index.html. Accessed 28. December 2019. 2019 (cit. on p. 80).

[Wal92] D.W. Walker. “Standards for message-passing in a distributed memory environment”. In: *1. Center for Research on Parallel Computing (CRPC) workshop on standards for message passing in a distributed memory environment* (Aug. 1992) (cit. on p. 85).

[WCA02] Emmett Witchel, Josh Cates, and Krste Asanovic. “Mondrian Memory Protection”. In: *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS X. San Jose, California: ACM, 2002, pp. 304–316. ISBN: 1-58113-574-2. DOI: 10.1145/605397.605429 (cit. on pp. 38, 56, 82).

[Whi19] Andy Whitcroft. *Sparsemem Memory Model*. https://lwn.net/Articles/134804/. Aug. 2019 (cit. on p. 190).

[Wil+95] Tim Wilkinson, Kevin Murray, Stephen Russell, Gernot Heiser, and Jochen Liedtke. *Single Address Space Operating Systems*. Tech. rep. UNSW-CSE-TR-95. Sydney, Australia: School of Computer Science and Engineering, University of New South Wales, Nov. 1995 (cit. on p. 74).

[Win+09] Simon Winwood, Gerwin Klein, Thomas Sewell, June Andronick, David Cock, and Michael Norrish. "Mind the Gap". In: *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*. TPHOLs '09. Munich, Germany: Springer-Verlag, 2009, pp. 500–515. ISBN: 978-3-642-03358-2. DOI: 10.1007/978-3-642-03359-9_34 (cit. on p. 163).

[Woo+14] Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. "The CHERI Capability Model: Revisiting RISC in an Age of Risk". In: *Proceeding of the 41st Annual International Symposium on Computer Architecuture*. ISCA '14. Minneapolis, Minnesota, USA: IEEE Press, 2014, pp. 457–468. ISBN: 978-1-4799-4394-4 (cit. on pp. 39, 51, 82, 186).

[WRA05] Emmett Witchel, Junghwan Rhee, and Krste Asanovic. "Mondrix: Memory Isolation for Linux Using Mondriaan Memory Protection". In: *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*. SOSP '05. Brighton, United Kingdom: ACM, 2005, pp. 31–44. ISBN: 1-59593-079-5. DOI: 10.1145/1095810.1095814 (cit. on p. 82).

[Wul+74] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. "HYDRA: The Kernel of a Multiprocessor Operating System". In: *Commun. ACM* 17.6 (June 1974), pp. 337–345. ISSN: 0001-0782. DOI: 10.1145/355616.364017 (cit. on p. 80).

[Zha+01] Lixin Zhang, Zhen Fang, Mide Parker, Binu K. Mathew, Lambert Schaelicke, John B. Carter, Wilson C. Hsieh, and Sally A. McKee. "The Impulse Memory Controller". In: *IEEE Trans. Comput.* 50.11 (Nov. 2001), pp. 1117–1132. ISSN: 0018-9340. DOI: 10.1109/12.966490 (cit. on p. 40).

[Zhu+17] Zhiting Zhu, Sangman Kim, Yuri Rozhanski, Yige Hu, Emmett Witchel, and Mark Silberstein. "Understanding The Security of Discrete GPUs". In: *Proceedings of the General Purpose GPUs*. GPGPU-10. Austin, TX, USA: ACM, 2017, pp. 1–11. ISBN: 978-1-4503-4915-4. DOI: 10.1145/3038228.3038233 (cit. on p. 156).

[ZP16] Foivos S. Zakkak and Polyvios Pratikakis. "DiSquawk: 512 Cores, 512 Memories, 1 JVM". In: *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*. PPPJ '16. Lugano, Switzerland: ACM, 2016, 2:1–2:12. ISBN: 978-1-4503-4135-6. DOI: 10.1145/2972206.2972212 (cit. on p. 88).

[ZPC06] Lixin Zhang, Mike Parker, and John Carter. “Efficient Address Remapping in Distributed Shared-memory Systems”. In: *ACM Trans. Archit. Code Optim.* 3.2 (June 2006), pp. 209–229. ISSN: 1544-3566. DOI: 10.1145/1138035.1138039 (cit. on p. 40).

[ZTM96] Stephan Zeisset, Stefan Tritscher, and Martin Mairandres. “A New Approach to Distributed Memory Management in the Mach Microkernel”. In: *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference*. ATEC ’96. San Diego, CA: USENIX Association, 1996, pp. 17–17 (cit. on p. 78).