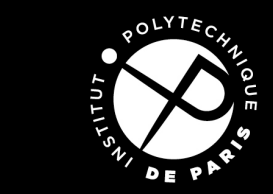
Memory management for operating systems and runtimes

Alexis Lescouet



INSTITUT POLYTECHNIQUE DE PARIS

NNT : 2021IPPAS008

Thèse de doctorat

TELECOM SudParis

IP PARIS

Memory management for operating systems and runtimes

Thèse de doctorat de l'Institut Polytechnique de Paris
préparée à Télécom SudParis

École doctorale n°626 École doctorale de l'Institut Polytechnique de Paris (EDIPP)
Spécialité de doctorat : Informatique

Thèse présentée et soutenue à Palaiseau, le 12 Juillet 2021, par

ALEXIS LESCOUET

Composition du Jury :

Amel BOUZEGHOUB Professeure, Télécom SudParis (SAMOVAR)	Président
Laurent RÉVEILLÈRE Professeur, Université de Bordeaux (Labri)	Rapporteur
Michaël HAUSPIE Maître de Conférence HDR, Université de Lille (CRIStAL)	Rapporteur
Samuel THIBAULT Maître de Conférence HDR, Université de Bordeaux (Labri)	Examinateur
Gaël THOMAS Professeur, Télécom SudParis (SAMOVAR)	Directeur de thèse
Élisabeth BRUNET Maître de Conférence, Télécom SudParis (SAMOVAR)	Co-directeur de thèse

Abstract

During the last decade, the need for computational power has increased due to the emergence and fast evolution of fields such as data analysis or artificial intelligence. This tendency is also reinforced by the growing number of services and end-user devices. Due to physical constraints, the trend for new hardware has shifted from an increase in processor frequency to an increase in the number of cores per machine.

This new paradigm requires software to adapt, making the ability to manage such a parallelism the cornerstone of many parts of the software stack.

Directly concerned by this change, operating systems have evolved to include complex rules each pertaining to different hardware configurations. However, more often than not, resources management units are responsible for one specific resource and make a decision in isolation. Moreover, because of the complexity and fast evolution rate of hardware, operating systems, not designed to use a generic approach have trouble keeping up. Given the advance of virtualization technology, we propose a new approach to resource management in complex topologies using virtualization to add a small software layer dedicated to resources placement in between the hardware and a standard operating system.

Similarly, in user space applications, parallelism is an important lever to attain high performances, which is why high performance computing runtimes, such as MPI, are built to increase parallelism in applications. The recent changes in modern architectures combined with fast networks have made overlapping CPU-bound computation and network communication a key part of parallel applications. While some degree of overlap might be attained manually, this is often a complex and error prone procedure. Our proposal automatically transforms blocking communications into nonblocking ones to increase the overlapping potential. To this end, we use a separate communication thread responsible for handling communications and a memory protection mechanism to track memory accesses in communication buffers. This guarantees both progress for these communications and the largest window during which communication and computation can be processed in parallel.

Acknowledgements

This document concludes four years of work that would not have been possible without the support of many people.

First and foremost, I would like to thank my advisors, Gaël Thomas and Élisabeth Brunet for their guidance during these years full of unforeseen developments and their unfaltering proofreading during the last months. I would like to extend special thanks to François Trahay, for being my unofficial advisor, and helping me decipher the MPI specification.

I would like to thank Laurent Réveillère and Michaël Hauspie for taking the time to read and evaluate this document as well as being part of my jury. I also thank Amel Bouzeghoub and Samuel Thibault for their participation in my jury.

These four years in the PDS Group, former HP2, would have been very different if not for the great atmosphere cultivated by all professors and students. Thanks to Denis Conan and Amina Guermouch for their counseling and the conversations we had over a cup of tea.

I am grateful to Gauthier Voron, for bearing with me while I discovered virtualization. I would also like to express my gratitude to the PhD students of room B312 and affiliated, which are always available for a coffee or two.

Alexis Colin, our C++ expert, for all the metaprogramming and template conversations. Tuanir Franca-Rezende for sharing with me the ups and downs of the daily PhD student life. Anatole Lefort for convincing me of the benefits of vertical screens. Damien Thenot for sharing Scalevisor with me.

Special thanks to Yohan Pipereau, my reference for kernel issues, and Subashiny Tanigassalame, for their great musical tastes.

Finally, I would like to thank my friends and family for their unfailing support.

My grand-parents and my sister for always asking me about my work. Thanks to Vivien for its great proofreading. Thanks to Sylvie for finding the right words. Deep thanks to Laure for being there when it mattered the most. And to my mother, thank you for always believing in me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Recently, the computation power and network throughput have reached unprecedented heights. While this much needed power helps to develop new fields of research such as data analysis or artificial intelligence, it also requires new techniques to fully exploit its potential.

This growth of computation power and network throughput has only been possible through the use of complex internal hardware structures for the new machines supporting them. First of all, with the increased frequency of the processor clocks, the rate at which instructions are executed has far exceeded the rate at which the memory controller can answer queries. This led to the introduction of smaller but faster memory banks, called caches, to decrease the visible latency of memory requests. Moreover, as all the cores of the machine shared the bandwidth of a unique memory controller, increasing the number of cores led to an overwhelming pressure on the memory controller. As it became impossible to further extend machines featuring a single memory controller, multiple memory controllers, distributed inside the machine, have been added. Because the access time of a given CPU to a given memory location depends on the proximity of the CPU and the memory controller owning this specific location, this architecture is named Non-Uniform Memory Access (NUMA).

This new topology, made of an important pool of processing, memory and network resources, requires to efficiently manage parallelism in order to leverage the full machine potential. In order to increase parallelism and simplify the use of NUMA architecture, this document proposes an approach in two parts.

Commmama The first obstacle to parallelism is to overlap network communication with computation. For this part, we focus on user space level with the MPI (Message Passing Interface) runtime. MPI provides two types of communication primitives, blocking ones, which are simple to use but cannot overlap communication with computation, and nonblocking ones which are more complex to use but have some overlapping capabilities. This forces developers to choose between the simplicity of blocking communications and the efficiency of nonblocking

communications.

In order to avoid trading simplicity for efficiency, we propose *Commmama*, which automatically transforms blocking MPI primitives into nonblocking ones at runtime. Moreover, we show that two main factors contribute to increasing the overlapping ratio. First, background progress which guarantees communications are processed while computation is running. Second, the size of the window during which both communication and computation can be run in parallel. Commmama addresses both these factors.

Scalevisor The second obstacle to parallelism is the contention on shared hardware resources. As explained above, the complexity of multicore machines has greatly increased to answer the need for computational power. As core frequencies rose, the performance gap with the memory controller has increased. This is commonly known as the *memory wall*. To avoid modern processors being blocked by the memory subsystem, small but fast memory caches have been added on the memory path to mask the latency of the slow main memory. Some of these caches are shared among cores, making them a contented resource. Moreover, as the number of cores per machine increases, so does the pressure on the memory subsystem. To overcome this issue, NUMA architectures feature multiple memory controllers, each managing a part of the machine's memory. While the multiplicity of memory controllers helps dealing with the contention on a given memory controller, it introduces another issue, the congestion on the network linking the different parts of the machine. This congestion is aggravated by the caches synchronization protocol which also uses this same internal network. Finally, each new addition to the already complex machine topology forces the overlying software to take into account the changes. Current operating systems already incorporate different resource management policies trying to deal with the different topologies, including NUMA architectures. However, modifying and maintaining efficient heuristics in code bases with millions of lines of code is a very difficult task.

In order to mask the complexity of the hardware while managing memory efficiently, the second part of our approach is a multicore resource management driver that uses virtualization technology, *Scalevisor*. Scalevisor is a small layer placed between the hardware and the operating system, which masks the real topology of the machine and exposes a simpler abstract topology for the overlying operating system. Virtualization technology thus allows Scalevisor to manage resources below the operating system transparently. Moreover, as Scalevisor tends to remain simple, focusing only on resource management, implementing or extending policies should be easier than changing kernel code.

This document is divided in multiple chapters. Chapter 2 describes the challenges of multicore architectures and more specifically of NUMA architectures. Chapter 3 describes the design and implementation of Commmama. Chapter 4 presents the design and implementation of Scalevisor. Chapter 5 concludes this document and presents future work. These chapters are organized as follows.

Background and motivation

Chapter 2 addresses the challenges of resources management on multicore machines. First, this chapter describes the internals of multicore machines, and more specifically of NUMA machines, in detail. This entails explaining the hardware mechanisms responsible for the performance issues from which multicore architectures suffer. Because Scalevisor masks the hardware topology to the overlying operating system, these mechanisms need to be understood in order to be correctly dissimulated. Second, we present a commonly used technique to manage memory: paging. This technique is important for both Commmama and Scalevisor as it is used in both cases to manage resources, either with the user space interface in Commmama, or by directly programming the hardware in Scalevisor. We present specifics of the x86 hardware mechanism that impact implementation of both Commmama and Scalevisor. Finally, this chapter describes solutions proposed in related work to solve the challenges of multicore architectures. We show that the main factors leading to non optimal performances are hardware resource contention and the increased latency of memory accesses, due to wrong memory placement.

Commmama, motivation and design

Chapter 3 presents a specific aspect of resource management: overlapping network communication with CPU-bound communication in the scope of the MPI user space runtime. First, this chapter describes the semantics of MPI communication primitives. The chapter then shows that two key factors influence how well primitives can overlap communication with computation, background progress and the duration during which the overlapping can occur. Studies targeted at improving either of these factors are presented. The chapter then introduces Commmama, which transforms blocking communications into nonblocking ones at runtime to improve the overlapping window while providing efficient background progress. Second, this chapter details the internal design of Commmama, composed of three different layers, the interception layer, offload layer, and protection layer. We present how each of these layers works. Finally, we present an evaluation of Commmama for communication with different message sizes and computation time. This final section shows that Commmama is able to overlap almost all the communication with computation when enough computation time is provided. This leads to a speedup of up to 73% for medium sized messages.

A multicore resource management driver

Chapter 4 presents Scalevisor, our approach to resource management using virtualization techniques. First, it presents virtualization techniques, both software and hardware assisted to emulate parts of the machine behavior. This section details the issues related to virtualization in general. It also presents studies related to the impact of virtualization on performances in general and on NUMA architectures in particular. Second, this chapter describes the design

of Scalevisor. The chapter shows how Scalevisor uses virtualization techniques to abstract the machine topology and to mask complex hardware resources, such as CPU caches, but also the distributed nature of NUMA architectures. The section then details the implementation of Scalevisor. The implementation of Scalevisor is incomplete. Scalevisor is able to boot a complete Linux operating system on an Intel machine, but, because of a lack of time, we have not implemented the interface to access hard drives. For this reason, we don't have evaluations that highlight the performance impact of Scalevisor. Instead, the chapter discusses our finding regarding the implementation of a new operating system, and especially reflects on the difficulty to create a new system given the complexity of current hardware without a full-time dedicated team.

Conclusion and future work

Chapter 5 concludes this thesis. It sums up the different contributions and teachings about managing resources to increase parallelism on complex topologies for both Commmama at the user space level and Scalevisor, at the hypervisor level. This chapter then discusses future work on both proposals. For Commmama, the main axis of evolution should be decreasing the overhead of the protection mechanism, with a secondary axis being the evaluation of MPI collective primitives and larger applications. For Scalevisor, the first task should be to finish the implementation of necessary parts of the system, and furthermore, developing and studying resource management heuristics.

Chapter 2

Background and motivation

As presented in chapter 1, both Commmama and Scalevisor aim to improve resource management. Commmama increases the parallelism of network communication and CPU computation using memory management techniques (paging and memory protection). Scalevisor uses virtualization techniques, among which virtualized paging to manage resources below the operating system, while exposing an abstract topology. In both cases, understanding how memory and paging work is important. In the case of Scalevisor, masking the topology requires an even better understanding of specifics of the hardware and the related issues.

This chapter thus addresses the challenge of managing resources in machines exhibiting complex topologies, known as NUMA architectures: as multicore architectures are composed of an important number of computation units, reaching good parallelism is important to fully exploit a machine potential. However, the distributed topology of NUMA machines induces new performance issues. We show that topology awareness is key in avoiding these issues and favoring scalability.

First, in order to better understand the aforementioned challenges, this chapter presents the architectural details of multicore machines in general and the specificities of NUMA machines in particular. We describe the hardware mechanism and protocols responsible for smooth operation of the architecture and the associated issues.

Then, we explain how operating systems and applications in the user space can leverage this hardware to manage memory through a commonly used data structure, the page table, and associated mechanisms.

Finally, we present solutions proposed in related work to increase efficiency of operating systems and applications on such architectures. We show that the main factors leading to non optimal performances are the contention of hardware resources or the increased latency of access, both due to wrong memory placement.

2.1 An overview of multicore architectures

Previously, Moore's law, which implied a doubling of the number of transistors per processing unit every two years, had the direct consequence of raising the frequency of processors. As evidenced by the number of transistors in recent processors, Moore's law is still valid as of today, but due to physical constraints, such as thermal dissipation, these transistors cannot contribute to the frequency factor as much as they used to. Since the introduction of IBM's POWER4 processor in 2001, the increasing number of transistors serves the different purpose of adding more computing nodes in a single chip, marking the debut of a change in paradigm for computer software, from sequential to parallel programs.

At its core, a computer is composed of three different hardware units, a processing unit or CPU, some memory accessed using a memory controller and some I/O devices accessed through an I/O controller. These three components communicate with one another using the system bus.

This layout was later extended to use multiple CPUs on the system bus to increase the computational power in ways that were unattainable by simply increasing the core frequency. This architecture, composed of multiple CPUs, a memory controller and an I/O controller is commonly known as *Symmetric Multiprocessing* (SMP) and was already in service decades ago in mainframes.

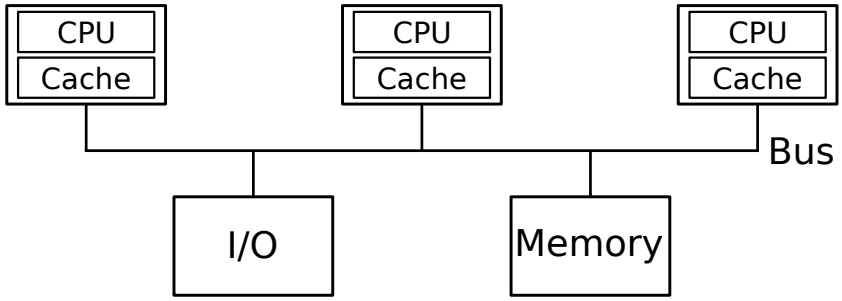


Figure 2.1: SMP architecture

2.1.1 SMP architectures

In SMP architectures, represented in Figure 2.1, the cost of accessing a memory location is the same for all processors as they are all connected through the single bus to the memory controller.

A standard multicore processor is composed of several computation units called *cores*, capable of executing multiple instructions at the same time. These cores are packed into a

single chip package, generally referred to as *CPU* or *processor*, which, when connected to a system bus featuring a memory controller and an I/O controller, acts as an SMP system.

To fully leverage the power of an SMP machine, the different processors must be used in parallel which in turn implies the need for the software run on such an architecture to leverage parallelism. As with many parallel systems, performance limitations lie in shared states and synchronization. For SMP machine, whether built from a multicore CPU or multiple processor, there are multiple hardware parts which are shared or need synchronization.

The first of these shared devices is the memory controller. In an SMP system, all the processing units are running in parallel and interact with the memory controller simultaneously. With the evolution of processor frequency, most modern multicores are now executing instruction at a very fast pace compared to the latency at which the memory controller can respond. This effect, commonly called *memory wall* pushes multicore CPUs to include fast memory, used as a cache, inside the CPU package to mask the latency of the main memory.

In modern CPUs, there are multiple such caches organized as a hierarchy with smaller and faster memory being used first, while bigger, slower caches are used when previous caches did not contain the wanted memory. Current hierarchies of caches generally contain three different levels of caches. The first one on the path to main memory called L1, is composed of two separate caches, one for data fetches called L1d and one for instruction fetches called L1i. These L1 caches are per core and thus replicated for each core of a CPU package. The next cache level, L2 is also private to each core, the Last Level Cache (LLC), L3 is the biggest cache and shared by all the cores of the CPU package. Common sizes for these caches are respectively 32 KB each for L1i and L1d, between 256 KB (laptop) and 1024 KB (high performance server) for the L2 cache. Because it is shared by all the cores, the size of the L3 cache varies more, with approximately 2 MB per core for modern CPUs, leading to 4 MB on a laptop computer and around 20 MB for server class CPUs. Figure 2.2 shows an example hierarchy of caches on a dual-core processor with three cache levels.

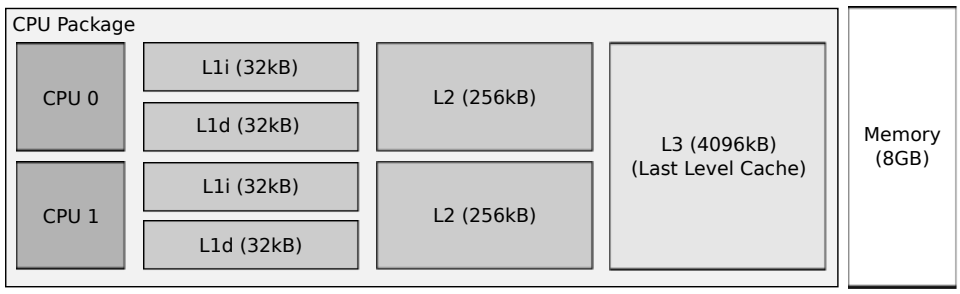


Figure 2.2: Example cache hierarchy

The first level of cache is separated into two distinct caches, L1i and L1d, for two reasons. First, separating caches multiplies the throughput allowing instruction fetches and data fetches

to be independent. Second, separating the caches allows to specialize their behaviors. In the case of the instruction cache, it allows to encode the cache content differently than random data.

While these caches may participate in increasing the overall performances by masking most of the latency of instruction and data fetches compared to the main memory, caches private to core must be synchronized in order to present a correct, uniform view of the sequence of load and stores on all cores. Different procedures have been developed to keep processor reads and writes consistent with the states of the different caches and main memory.

Cache coherency protocols for SMP architectures Caches are said to be coherent if all loads and stores of all cores to a given memory location appear to execute in a total order that respects the program order. In order to guarantee the sequential consistency of the memory, two properties must be satisfied.

First, *write propagation*, if a processor P1 reads from a shared memory location X after a write by another processor P2 to the same memory location X, P1 must read the value written by P2 and not the old value. Second, *serialization*, if two values A and B are written to location X in this order from two different processors, no processor may read B then A.

To ensure caches are coherent in an SMP system, multiple protocols have been designed, starting with the MSI protocol and its derivatives, such as the MESI and MOESI protocols. In these protocols, the blocks of data, called cache lines, which reside in a cache are tagged with a state. These states represent the state of the memory in regards to both the state of the memory (modified or not) and the state of the same location across the different caches (exclusive or shared). The protocol names are derived from the names of the different possible states of a memory location in the cache, M stands for *Modified*, S for *Shared*, I for *Invalid*, E for *Exclusive* and O for *Owned*.

The original MSI protocol operates as a reader-writer lock (single writer, multiple readers). The *Modified* state represents exclusive ownership as a writer, the *Shared* state represents shared ownership as a reader while *Invalid* means neither reader nor writer. Traditionally with reader-writer locks, acquiring the lock to write means no reader currently holds the lock. For cache coherency protocol, transitioning to the *Modified* state forces all other caches to forgo their *Shared* state and invalidate the location. Alternatively, when a cache requests to load a location, if another cache holds this location in the *Modified* state, meaning the value stored in memory is not up to date, the modified value is first committed to main memory transitioning both caches to the *Shared* state.

While this protocol is theoretically correct, it is not optimal in regard to the number of bus messages. When a cache is the only one to contain the value for a given location and writes to this location, it must broadcast a message on the bus to force other caches to invalidate their entries, which could be avoided in this case. To solve this issue and avoid unneeded messages on the bus, the MESI protocol was introduced by Papamarcos et al. [43]. According to Intel's

Software Developer Manual [28, chap. 11.4], the MESI protocol is still in use in current Intel processors.

The MESI protocol contains an additional state, *Exclusive* which means a value at a given location is valid but not present in any other caches. When a memory location is read by the core and tagged as *Invalid*, the core broadcasts a message on the bus to request the value, if at least one other cache contains the wanted value they both transition to the *Shared* state. This is true for both the MSI and MESI protocols, but in the MESI protocol, if all responses are negative, the requester cache loads the value from memory and transition to the *Exclusive* state. Once in this state, any write to the same location causes a transition to the *Modified* state without the need to issue a message on the bus invalidating this location for other caches.

When using the MESI protocol, in case of a load request directly followed by a write, if the requesting core is the only one using the data it saves one bus message. This is especially important on sequential parts of an application as only one core will access and modify a given location at a time, resulting in improved performances over the MSI protocol.

The MESI protocol can be implemented using two core-side requests and three bus-side requests. Core-side requests, for cache transactions initiated by the core are *PrRd* to get access to the value of a cache line and *PrWr* to write a value to a cache line. Bus-side requests are *BusRd* to get the value of a cache line (emitted after a read miss), *BusRdX* to get ownership of a cache line for writing or notify of a write (emitted after a write miss), *Flush* to commit a block to main memory, generally in response to a *BusRd* or *BusRdX* request. A complete state transition diagram presenting the interaction of these different messages and the evolution of a cache line state in response is presented in Figure 2.3.

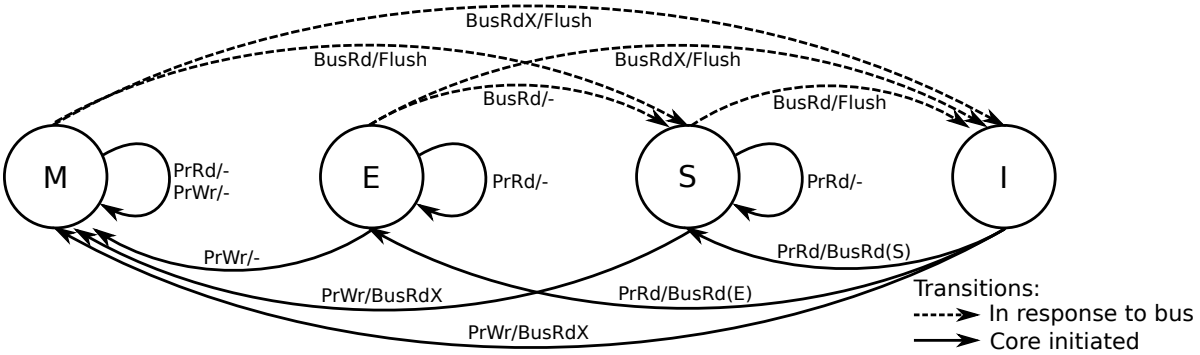


Figure 2.3: MESI State diagram

The issue with MSI and less importantly in MESI is the repeated use of the system bus to convey different cache protocol messages and data.

A standard implementation of a cache coherency protocol such as MESI operates using two types of communications as mentioned previously, local to core (core to cache) and through the

bus linking the different cores and the main memory. The two local operations are a request from the core either to read from the cache, or write to the cache. Depending on the state of the cache location targeted by these operation, the core may need to issue messages on the bus to either request the wanted value, or request the value and notify it intends to modify it, or inform other components of a modification of an already cached value. Because of the distributed nature of the caches and the wanted coherency property, all these messages are broadcast on the bus and decrease the available bandwidth for other operations.

Moreover, when a memory location is frequently used by multiple cores, it is still flushed to the main memory store[1] which is shared by all cores.

The SMP architecture, while providing more processing power with parallelism is limited by the strain it incurs on the shared components, the system bus, and both the memory and I/O controllers. This architecture is thus viable for a reasonable number of cores but cannot scale efficiently.

2.1.2 Non Uniform Memory Access architectures

The cache hierarchy of SMP architectures does reduce the amount of requests reaching the memory controller. However, when there is an important number of cores and amount of shared data, the SMP model with its unique shared memory controller does not scale well. In order to reduce the load on the memory controller, a new layout featuring multiple memory controllers distributed in different nodes is used, *Non Uniform Memory Access* (NUMA) architectures. These architectures feature multiple memory controllers distributed among the cores, which alleviates the pressure on a given controller and allows for better memory performances.

As represented in Figure 2.4, this kind of machines is composed of different nodes, each with its own memory region and cores which are connected with an inter-socket network called an *interconnect*. The topology of the communication network varies from one machine to another. Common topologies for NUMA machines comprising four sockets are either ring (all sockets have two peers) or crossbar (complete graph). For machines comprised of eight sockets, there are more variations, commonly, no more than two hops from any given socket are necessary to reach any other socket.

While having completely separate memory controllers, NUMA machines still provide a global address space in which every memory zone in the system is accessible by other hardware with its unique physical address.

Given this new topology, NUMA machines have two different kinds of memory accesses, local ones, when the targeted memory is managed by the controller located on the same node as the CPU and remote ones, when a CPU is accessing another node's memory.

By adding more memory controllers, NUMA machines can make use of more cores without overloading the memory subsystem as SMP would. However, the distributed topology exhibits

[1] This can be the LLC when it is inclusive, which is the case for most L3 caches in Intel processors.

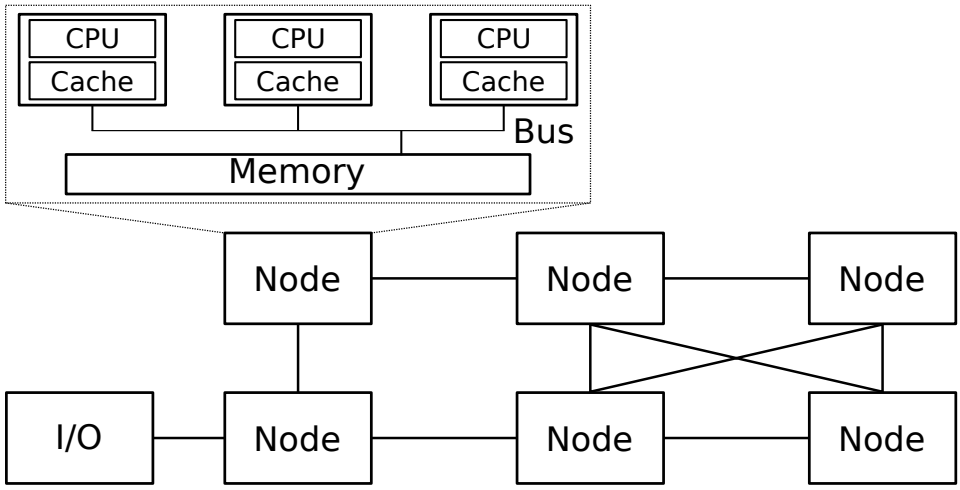


Figure 2.4: NUMA architecture

different kinds of performance issues. On one hand, memory accesses through the interconnect are slower than local ones, making locality of memory accesses important, on the other hand, the interconnect has a limited bandwidth which can lead to congestion in the event too many distant accesses would occur.

Moreover, as explained in Section 2.1.1, for performance reasons, modern CPUs have a cache hierarchy used to reduce the number of memory accesses reaching the high latency main memory. This cache hierarchy is also present in NUMA architectures and creates two subtypes of NUMA machines, the ones in which the cache coherency is local to a given node and the ones for which cache coherency is ensured between nodes. The latter are called cache coherent NUMA or ccNUMA.

While non-cache-coherent NUMA machines have some use cases, the lack of a global cache coherence breaks the usual shared memory abstraction, making these suffer from a lack of programmability as described in Yunheung Paek et al. [42]. Thus, most currently used NUMA machines are cache coherent and according to Martin et al. [35] this is not going to change in the foreseeable future for mainstream machines. They argue there still is some design space to make cache coherent hardware scale, but more importantly, forcing software to explicitly manage coherence would simply shift the complexity of cache coherency protocols from hardware to software.

However, while cache-coherency is very important to ease programming of such machines, it comes with its drawbacks as well. Using the MESI cache protocol for NUMA machines would force communications between the cache controllers of different nodes, which was already an expensive solution for SMP machines with a far smaller number of cores. To avoid overloading the interconnect network with cache protocol messages, multiple modifications to the cache

coherency system are made.

Cache coherency for NUMA machines Independently of the protocol used, cache coherency implementations can be realized using messages directly broadcast on the bus as explained earlier but the resulting traffic on the interconnect network would scale proportionally with the number of cores in the system. This approach called bus snooping is thus inappropriate for large NUMA machines which can contain hundreds of cores.

In snoop-based implementations of the cache coherency protocol, the LLC is responsible for broadcasting the relevant messages on the system bus. In directory-based implementations, the so called *directory* maintains the information of the state of cache lines and which nodes, if any, are sharing that line. This system avoids the need to broadcast the coherency protocol messages on the bus, allowing a given requester cache to directly target the right cache through the use of the directory.

The directory system can be implemented using multiple possible methods, for example using a different component, acting as the bookkeeper or with each cache tracking the potential other copies of a given memory address using a range of bits co-located with the state tag in the cache line metadata.

Intel machines use their own form of directory-based protocol for their NUMA servers. Recent hardware from the manufacturer has been using a Network-on-Chip design, in which cores, caches and memory controller are linked via a specific on-chip network. The interconnect network then connects chips. Two iterations of this network have been in use in modern servers, QuickPath Interconnect (QPI) before 2017 and UltraPath Interconnect (UPI) with the introduction of the new Skylake Scalable (Skylake-SP) microarchitecture.

For Intel's platform, the part responsible for handling the interface between private caches (L1 and L2) and the LLC is called the *caching agent*. This caching agent is thus responsible for starting caches transactions, as well as sending and receiving the bus messages generated by these transactions. There is one caching agent per core.

The caching agent communicates with the private cache levels through a basic snoop-based cache coherency protocol which is local to the core. For outgoing communications, all caching agents are linked using a ring topology to the QuickPath Interconnect and communicate using a directory-based approach with a *home agent* as represented in Figure 2.5.

There is one home agent per memory controller, each responsible for its range of memory addresses. When a cache miss occurs in the caching agent of a given core, the caching agent sends a request to the home agent responsible for the wanted memory location. This home agent then uses the information of its directory to send a snoop request only to the caching agents of the cores which may have a cached copy of the wanted memory. One of the agents[2] with a copy of the data sends a copy to the requester caching agent and an acknowledgment to

[2]The one with the F (Forward) state from the MESIF protocol explained below.

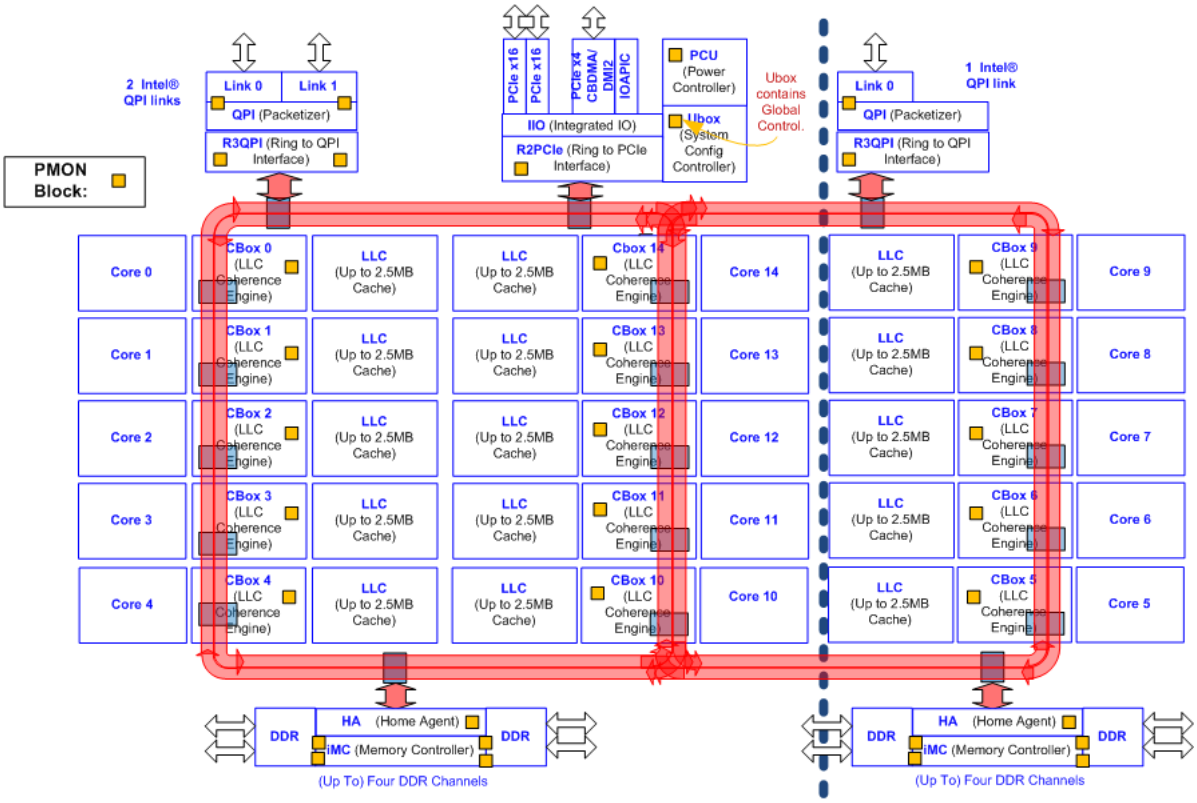


Figure 2.5: Intel Xeon E5 ring architecture (Intel documentation [27, chap. 1.1])

the home agent. The home agent concludes the transaction. These four steps are illustrated in Figure 2.6.

As the number of message increases with the number of caching agent involved, avoiding to broadcast these messages reduces the consumption of the already limited interconnect bandwidth. This methods thus reduces interconnect bandwidth consumption in exchange for an increased latency of cache coherency transactions.

In addition to the directory information used by the home agent which is an implementation of directory-based cache coherency, Intel's cache agents also use a modified version of the MESI protocol, called MESIF, initially proposed by Goodman et al. [22], [23]. The MESIF protocol introduces an additional specialized form of the *Shared* state, the *Forward* state which is a read-only (cache line has not been modified) state attributed to at most one cache agent for a given cache line. It designates the given cache agent as the primary responder for requests concerning the given cache line. This additional state allows direct cache-to-cache transfers,

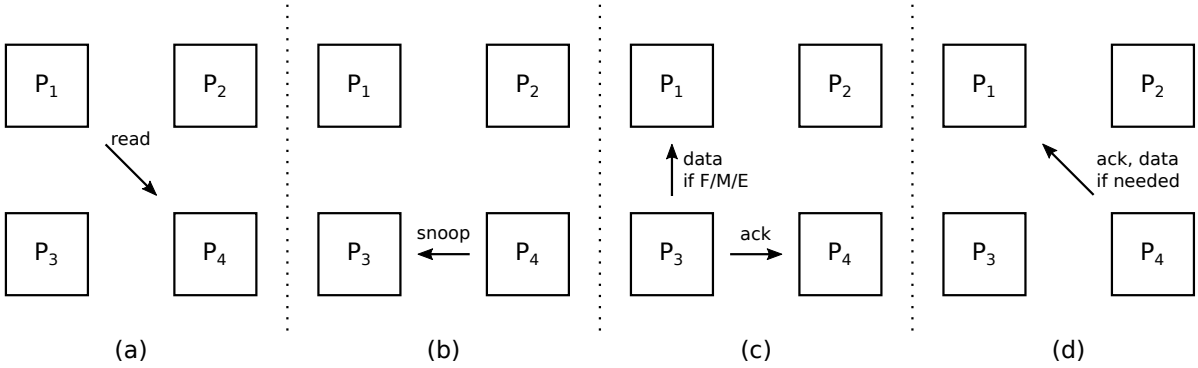


Figure 2.6: Intel's directory-based coherency example

thus avoiding either the home agent having to fetch data from the slow main memory or all peers in the *Shared* state responding at the same time. The *Forward* state is given to a cache agent after a read miss, either by the previous *Forward* owner responding to the request or because no cache agent currently has the *Forward* state for given cache line.

However, even with the multiple enhancements brought by the QPI network and cache coherency system, the increase in the number of cores per CPU in the Skylake-SP microarchitecture forced a new on-chip network to be designed as well as a new interconnect network.

The new Skylake Scalable CPU aims to further increase the number of cores per CPU leading to NUMA machines with even higher number of cores. The new UltraPath Interconnect developed for Skylake-SP machines has several improvements over its QuickPath Interconnect predecessor, among these an updated cache coherency protocol, and is integrated more tightly in the new Skylake-SP chips due to different modifications of the socket internal topology.

First, the network topology has been changed from a ring in old Xeon processors to a mesh in new Skylake-SPs reducing the number of hops from a given caching agent to another. This aims to improve the performances by decreasing latency of the cache coherency system. Moreover, the home agent previously located between the on-chip network and the memory controller is now fused with the caching agent at a rate of one per core to further increase scalability. This new architecture is represented in Figure 2.7. Each home agent has a cache for directory lookup used when performing a cache coherence transaction.

The new mesh topology and the distributed Caching-Home Agent (CHA) greatly improve sockets performances for cache coherency transactions and memory loads. However, our experiments show that NUMA effects, while reduced are still present, thus increasing the latency of memory accesses over the interconnect network.

The still existing limitations of NUMA architectures Our experiments are run on a quad socket Skylake Scalable Platform machine with four Xeon Gold 6130 processors.

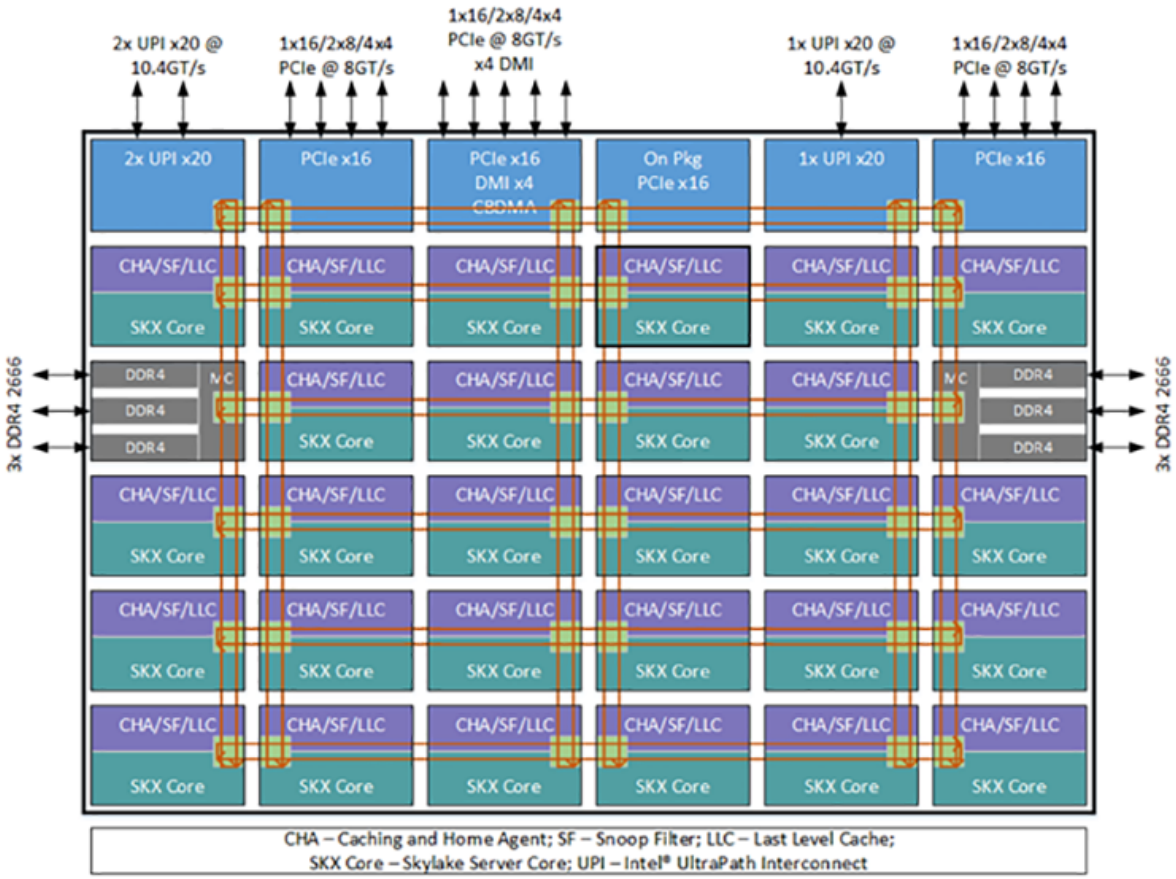


Figure 2.7: Intel Xeon Scalable family mesh architecture (Intel documentation [26])

These processors are organized as follows. The CPU package (the whole processor container) is composed of one die which is a single continuous piece of silicon. This die features a number of cores, in our case 16 cores in the only die, physical cores expose two logical cores due to Intel's Hyper-threading.

Our experiments consist in measuring the time of a read operation done by a reading core r of a memory location which is allocated by an allocator core a for different couples (r, a). Figure 2.8 shows the results as a heat map, for all core combinations on the left and only one die on the right.

On the left, featuring all the processors, the figure shows a latency around 220 cycles for node local reads and around 350 cycles for non-local reads. This still represents a 59% increase for distant loads.

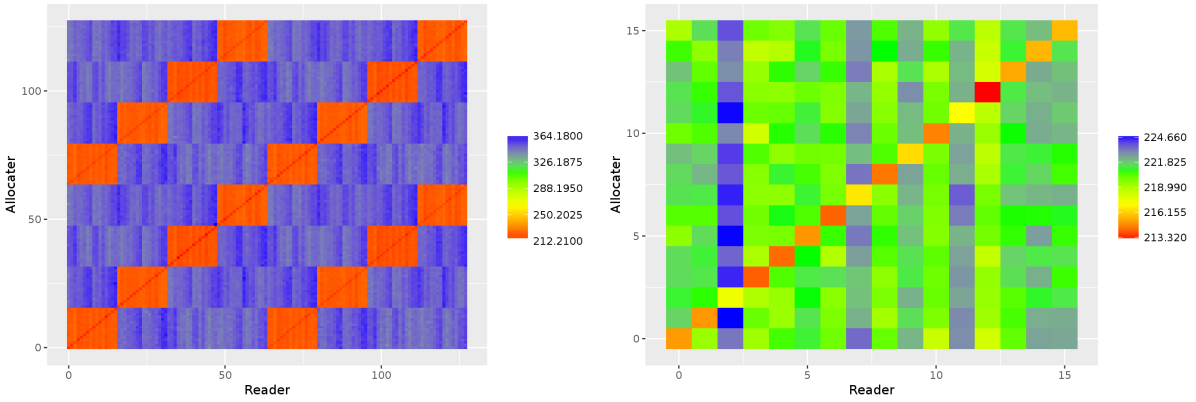


Figure 2.8: Latency (CPU cycles), for the Skylake Scalable machine (left) and one die (right)

Moreover, while previous explanations about the cache coherency mechanism and architecture of modern Intel servers are not identical to the mechanism used in machines manufactured by AMD, the global theory can be transposed. The same experiments run on a dual socket AMD EPYC server shows similar results as shown on Figure 2.9.

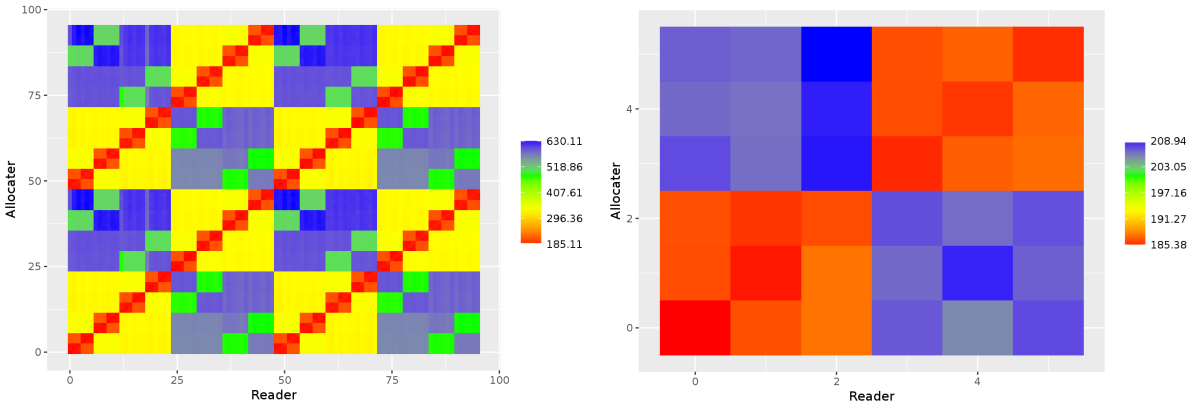


Figure 2.9: Latency (CPU cycles), for the AMD EPYC machine (left) and one die (right)

AMD EPYC processors contain four distinct dies per physical package, each containing multiple cores grouped by *Compute clusters* (two by die). Each compute cluster contains 4 cores and a shared last level cache. Once again, memory accesses inside a die are between 180 and 200 cycles, but due to the hierarchical nature of the AMD machine, we observe NUMA

effects in two different types of accesses. The first NUMA effect is visible between a die local access, at around 200 cycles, and a socket-local access on another die, at around 300 cycles. The second NUMA effect can be seen when accessing memory of another socket, with up to 600 cycles.

These measurements show that the placement of memory even in modern NUMA architectures is very important to avoid degrading the performances of running software. As forcing the software to manage memory placement using only physical memory would be limiting, the hardware supplies a number of facilities to ease the task of memory management as well as different mechanisms to increase efficiency.

2.2 Memory management in the operating system

There are multiple possible techniques that can be used in order to manage memory. Common techniques which are supported by hardware include memory segmentation and paging. Segmentation uses a segment base, and an offset to represent linear addresses, this technique was in use in early x86 processors and still exists in modern processors in compatibility modes such as real mode, protected mode in order to keep some backward compatibility with old software. While segmentation is not available anymore in 64-bit mode for modern x86_64 processors, its simplicity and efficiency lead to its consideration as an alternative to paging by Taebe et al. [52].

Paging memory is the most common technique existing at the moment in modern computers and has many advantages. This mechanism and its implementation are developed in the next subsection.

2.2.1 Paging and the page table structure

The paging mechanism allows the operating system to use an abstract view of the memory. This is done by using a translation mechanism to resolve memory addresses in executed code, called virtual addresses as physical addresses which are direct identifiers for real memory zones.

The paging mechanism is beneficial in a number of ways as it allows virtual memory addresses to exist independently of the resident storage on which are located the real data.

One of the most important feature of virtual memory is to isolate different memory spaces. This is used to isolate the memory of processes from one another, but also to isolate the memory from the processes and the kernel. The memory of processes is separated by only mapping physical memory of a process in the virtual space of its owner. The kernel memory is separated from the rest using a different mechanism provided by the hardware. Memory pages can be marked available for kernel only or for everyone. This distinction allows to isolate the memory of the kernel from the rest of the system. These two distinct memory spaces are commonly called *kernel space* and *user space* (or userland).

Virtual memory is also at the base of the swap system in which some data are offloaded to another storage type, mostly hard drive, to alleviate the strain on RAM. When some memory is accessed rarely, the content of the corresponding pages is written to disk and the physical memory used to store them is made available for other use. When the virtual addresses of such memory are accessed, the access is detected by the operating system which in turns loads the content back from disk transparently.

By the same token, virtual memory can be used to reserve memory without any resident storage, processes can thus allocate more virtual memory than physically available memory in a given machine. This mechanism, called memory overcommitment is commonly used by operating systems and virtualization systems when processes or virtual machine are requesting more memory than they need to maximize the number of such processes or virtual machines on a given hardware unit. In the case where the full amount of requested memory would be required by the processes, the system can use swap to fulfill requests albeit with worse performances.

Additionally to the above system mechanisms, paging is can also be used to share physical memory between processes (shared memory segments), to optimize process creation by duplicating the pages of a parent process on the first write, or to optimize file access with on-demand paging.

The page table structure In modern computers, the translation mechanism is done by specialized hardware, called the Memory Management Unit (MMU) which is now a part of the CPU. The MMU uses an in-memory structure to store the different mappings between physical and virtual address space. This structure is called a page table.

The format of a page table is hardware dependent as it is determined by the MMU which expects a specific layout. Modern x86 CPUs use a multi-level page table (four level in current hardware, five in upcoming Intel processors) which takes the form of an n-ary tree, more precisely a tree of arity 512 for x86 processors. The root of the tree, which is needed as the staring point for traversals, is available to the MMU through a CPU register called `cr3` in which the system needs to write the physical address of the page table root.

As presented by Figure 2.10, this tree structure is traversed by splitting a requested virtual address in five part (for a four level paging structure). The first 12 bits represent the offset in the 4 KB page, the four next parts (9 bits each) represent an offset in the levels of the page table tree structure. The Page Map Level 4 (PML4) entry at the offset given by the bit 39-47 of the virtual address contains the address to a Page Directory Pointer Table (PDPT). The pointer at the offset given by bits 30-38 of the virtual address gives access to the Page Directory (PD) and the rest of the bits to the Page Table (PT) level and finally the index inside the PT level which contains the physical address of the four kilobyte page corresponding to the given virtual address. The offset in the page is obtained by adding the 12 least significant bits to this four kilobytes aligned physical page address (12 last bits are null).

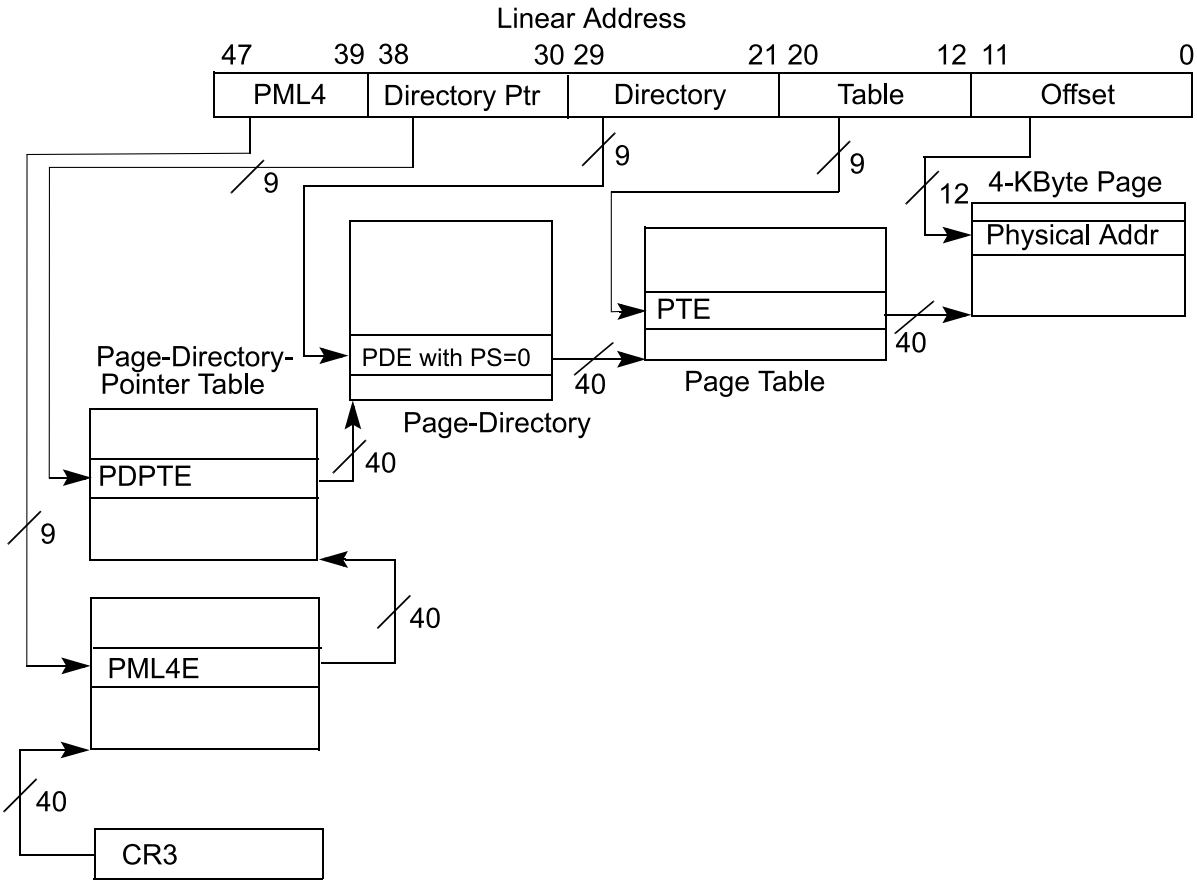
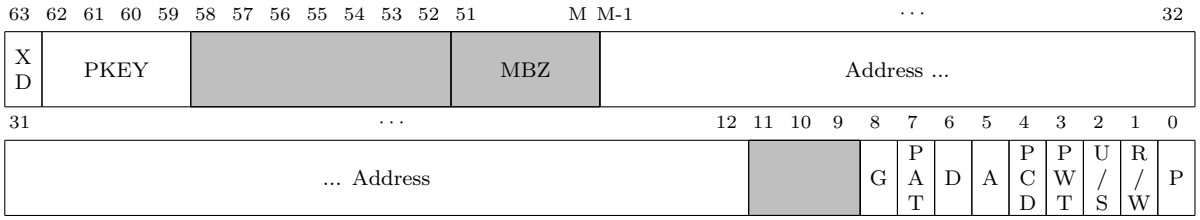


Figure 2.10: Translation mechanism (Intel documentation [28, chap. 4.5])

As the entries of each level in the page table nodes are physical addresses of page frames in physical memory, these addresses are four kilobytes aligned, meaning the twelve least significant bits of such an address must be zero. The page table mechanism can thus use these bits to store some metadata about the pointed memory. The format of a last level page table entry is given in Table 2.1.

Access rights and page fault Some metadata stored in the least significant bits of the page table entries are related to access rights, these are the bits "Present (P)", "Read/write (R/W)", "User/Supervisor (U/S)". When an access occurs to a virtual address belonging to a given page frame, the page table entry describing this page frame is loaded by the MMU while traversing the page table. If the combination of bits P, R/W or U/S does not allow the access, the processor emits a "page fault" exception.



Offset	Name	Description
63	XD	If IA32_EFER.NXE = 1, execute-disable (if 1, instruction fetches are not allowed from the 4-KByte page controlled by this entry); otherwise, reserved (must be 0)
62:59	PKEY	Protection key; if CR4.PKE = 1, determines the protection key of the page; ignored otherwise
58:52		Ignored
51:M	MBZ	Reserved (must be 0)
(M-1):12	Address	Physical address of the 4-KByte page referenced by this entry
11:9		Ignored
8	G	Global; if CR4.PGE = 1, determines whether the translation is global; ignored otherwise
7	PAT	Indirectly determines the memory type used to access the 4-KByte page referenced by this entry
6	D	Dirty; indicates whether software has written to the 4-KByte page referenced by this entry
5	A	Accessed; indicates whether software has accessed the 4-KByte page referenced by this entry
4	PCD	Page-level cache disable; indirectly determines the memory type used to access the 4-KByte page referenced by this entry
3	PWT	Page-level write-through; indirectly determines the memory type used to access the 4-KByte page referenced by this entry
2	U/S	User/supervisor; if 0, user-mode accesses are not allowed to the 4-KByte page referenced by this entry
1	R/W	Read/write; if 0, writes may not be allowed to the 4-KByte page referenced by this entry
0	P	Present; must be 1 to map a 4-KByte page

Table 2.1: Format of a last level page table entry

This exception can (and must) be intercepted by the operating system in order to correct the situation before continuing the execution of the faulting context. This mechanism is at the foundation of the multiple use cases of paging.

The operating system can give a process an address range with a bit P set to zero to load data only when an access to the given page frame would cause a page fault, this is called lazy mapping and is used by the swap and memory overcommitment mechanisms mentioned before, but also lazy file access for file with an arbitrary big size.

Similarly, copy-on-write (COW) semantics for memory are obtained by giving a range of addresses without write access. While accessible for read accesses, the memory will cause a page fault to be emitted when an attempted write operation occurs, thus giving the system the opportunity to copy the content of the faulting page frame to another location in physical memory, updating the page table mapping to point to this new location and continue the

execution. From there, the write operation will occur in the new location, effectively modifying a copy of the old read-only data. Copy-on-write is used in the `fork` system call to avoid copying all the parent process data to the offspring process while unneeded.

Finally, the U/S bit is used to differentiate between user-accessible pages and kernel-only pages when running a process in user mode and is the base of kernel and process memory isolation, and thus, of the userland concept mentioned earlier.

Extending access rights for user page: protection keys While the three bits described above are used to determine access rights for an access regarding a given page frame, changing the permissions for a large number of pages is costly as it forces to change the permission bits of each page. To avoid this overhead, another mechanism called *protection keys* has been introduced for processors Intel Skylake and newer.

This mechanism assigns a key to any number of pages, thus creating a group of pages all tagged with given key. The permissions of this group of pages are then determined by the permissions affected to the key. Therefore, when using this mechanism, there is no further need to modify the page table after setting the key for all the wanted pages.

Protection keys use the aforementioned PKEY zone in a page table entry to store a number between 0 and 15 as well as a new 32-bit CPU register called "Protection Key Right for User access (PKRU)".

When the protection key system is enabled, any access to a memory range causes the protection key system to be checked. The key stored in the page table entry PKEY slot is retrieved and used as an offset in the PKRU register. For $0 \leq key \leq 15$, the bit PKRU$[2 \times key]$ controls the access right and PKRU$[2 \times key + 1]$ controls the write right to the memory zone with PKEY set to key.

Modifying the PKRU register is done using the two `RDPKRU` and `WRPKRU` instructions to respectively read or write to the register. Setting the bit PKRU$[2 \times key]$ disables all data accesses to the memory location, this does not include instruction fetches, setting bit PKRU$[2 \times key + 1]$ disables write accesses to the memory location.

Such a system may seem redundant with the already existing permissions enforced by the P and R/W bits, but has multiple advantages. Firstly, as mentioned earlier, after setting the PKEY value of the page only the PKRU register needs to be modified. Secondly, while clearing bit P leads to a page being totally inaccessible even for instructions fetches, setting bit P and a PKEY with all accesses disabled results in an execute-only mapping, such permissions cannot be achieved with only the legacy permission bits.

The Translation Look-aside Buffer and paging structure caches As shown, the paging mechanism adds multiple convenient behaviors, allowing for techniques like swap or on-demand loading of frames to be used. At the same time it inevitably adds some overhead due

to the translation process. In order to reduce this overhead, modern MMUs possess caching capabilities through the use of the Translation Look-aside Buffer (TLB).

The TLB stores individual translation information by associating the page number to the physical address of the associated page in the page table. The page number is the part of the virtual address which is used in the page table to identify a given page frame and thus depends on the size of the page. It is composed respectively of bits 47:12, 47:21, or 47:30 for pages of respectively four kilobytes, two megabytes or one gigabyte. Other important information is cached with this page number to physical address mapping, such as the permission for the page represented by the logical-AND of the R/W flags of all the page table entries leading to the page frame, the logical-AND of the U/S flags of these same pages and the PKEY field if the protection key feature is enabled.

The CPU can also cache translation information about the top three levels of paging structure, PML4 entries, PDPT entries and PD entries. Such that in trying to translate an address, if the TLB does not contain a corresponding cached translation for the page number, the CPU will search for cached information in the PDE cache, if none is found, search in the PDPT cache, and finally in the PML4 cache. If the TLB contains the cached translation, the CPU uses it, else it finishes the translation using the page table in memory from the last level obtained from the paging structure cache.

Because the cached information may become false at some point when mappings are changed in the page table, cached information must be invalidated. Different actions naturally invalidate translation caches, this includes loading a new page table root in the `cr3` register (`MOV` instruction with `cr3` as a target), a context transition between guest and host when using virtualization extensions. The `INVLPG` instruction is used to manually invalidate translations with a finer grain.

However, invalidating the whole TLB at every context transition when scheduling another process is costly. To avoid this cost, especially when switching between the same processes multiple times, hardware manufacturers added a feature called Process-Context Identifiers (PCIDs) which are used to tag a given entry in the translation caches with the process identifier allowing invalidation to target a smaller portion of the TLB.

When PCIDs are enabled, switching page tables when scheduling another process does not force an invalidation of the whole TLB. An instruction, `INVPCID` was introduced to allow finer grained control over which entries of the TLB are invalidated, for example invalidating all the entries tagged with a given PCID.

The TLB and paging-structure caches are important additions to current MMUs as they speed up the translation process by avoiding a complete traversal of the page table tree for each virtual address translation and are used not only for data accesses but also for instruction fetches and prefetches in case of speculative execution.

The multiple benefits of paging, such as isolation or memory reservation, make the paging a preeminent part of current memory management systems. This is reinforced by the TLB cache which decreases the overhead of paging translation and thus of paging in general.

2.2.2 Paging and memory management in user space

While the page table structure is maintained by the kernel, its benefits are also available in user space both indirectly: isolation of kernel and user space, isolation between processes; and directly, through the use of different system calls.

The paging mechanism is activated per CPU and so is activated for both the kernel and user space. Depending on the kernel model, the page table could be common for all execution contexts, but using a different page table enforces some isolation between processes being executed in user space.

The Linux kernel uses a different page table for each process in user space, with several consequences. On one hand, the memory spaces of processes are separated, forbidding access for a process to the memory of another. On another hand, Linux allows mappings to be reused across different processes in order to create shared memory segments.

This ability is important to avoid a change of page table when switching to kernel mode, as each process needs the kernel to be mapped in its address space for system calls. The kernel uses a clever trick to avoid having to change every process page table when modifying kernel mappings by reserving one top level entry in the PML4 for kernel usage when a process page table is instantiated. This top level entry is the same for every process allowing the kernel to change mappings in subsequent entries without having to modify the page table of each process.

Moreover, using copy-on-write semantics described previously, shared library containing mutable data can be shared without inducing a big memory footprint as the memory is shared until a mutable part is modified in which case copy-on-write is used to copy only the modified page frame.

Creating a mapping `mmap` is the most preeminent system call for user space memory low level management. It enables the programmer to create a new mapping between virtual and physical memory space. Depending on the parameters (`MAP_ANONYMOUS` or not) the physical memory is either empty memory or file backed memory. In case the memory is file-backed, this system call exhibits the different advantages mentioned in Section 2.2.2. Using the `MAP_PRIVATE` option creates a copy-on-write mapping. As `mmap` mappings are page table mappings, they only manipulate virtual memory, backing physical memory is only allocated when needed (first access).

Changing access rights `mprotect` is a primitive used to modify permissions on a given memory mapping previously obtained by using `mmap`. As explained in Section 2.2.1, the page table entries contain permission bits determining which types of access are allowed for a given zone of memory. The `mprotect` system call defines several boolean flags among which `PROT_NONE`, `PROT_READ`, `PROT_WRITE` and `PROT_EXEC` which can be combined to define if a page can be read, written or executed.

While these flags are defined by the POSIX standard and thus used in the Linux system call, their presence in the system call interface does not imply every combination is actually supported by the underlying hardware.

The bits from an x86 page table entry used to determine access rights are bit "Present (P)", "Read/write (R/W)", "User/Supervisor (U/S)", from Table 2.1. Because the bit U/S is always activated for page meant to be accessed in user space, bits P and R/W are the only ones used to determine permissions. However, as putting the present bit to 0 causes a page fault when trying to access the page either by reading or writing, this means no combination of bits can fulfill a `PROT_WRITE` only permission set. This is summed up in Table 2.2. Moreover, x86 processors have a bit to disable execution on a given page but for the same reasons as above, the `PROT_EXEC` flag is equivalent to the `PROT_READ` flag in Linux.

Table 2.2: Correspondence between x86 page table bits and `mprotect` flags

`mprotect` flags	Page table bits
`PROT_NONE`	$\neg P$
`PROT_READ`	$P \mid \neg R/W$
`PROT_WRITE`	not possible
`PROT_READ` | `PROT_WRITE`	$P \mid R/W$

Working with protection keys In kernel version starting from 4.9, the protection key feature introduced by Intel is also available in user space memory management through the `pkey_alloc`, `pkey_free`, and `pkey_mprotect` system calls and the `pkey_set` utility function. The protection key mechanism allows a developer to further restrict accesses to a memory zone by disabling write or all kinds of accesses (read and write) to the given zone.

This is done by first using `pkey_alloc` to allocate a protection key with the right set of permissions, either `PKEY_DISABLE_ACCESS` or `PKEY_DISABLE_WRITE`, and then use `pkey_mprotect` to atomically change both regular protection flags available to `mprotect` and set a protection key to the given memory range.

Thanks to this interface, the advantages of protection keys described previously are available in user space, it is thus possible to create an execute-only mapping by combining the `PROT_EXEC` (equivalent to `PROT_READ`) flag with a protection key allocated with `PKEY_DISABLE_ACCESS`. This method also reduces the cost of changing permissions on pages, as it requires neither changing all the permission bits as explained before nor switching to kernel space as the `RDPKRU` and `WRPKRU` instructions are available in user space and usable through the wrapper `pkey_set`. Only the initial setup of the PKEY using `pkey_alloc` and `pkey_mprotect` is switching to kernel space.

To summarize, this section presents the hardware mechanisms and tools built on them which are commonly used to manage memory, either in kernel space or user space. As explained, changing mappings have a cost, both directly, and indirectly by forcing the TLB cache to be flushed. Therefore their efficiency relies on the ability of the system to properly use them.

2.3 State of the art: resources management and NUMA

Resources management is a vast subject, which has been studied since the inception of computer science. With new topologies being born, new problems and new solutions are discussed. SMP architectures and even more so NUMA architectures both provide shared memory and use parallelism as a mean to increase performances. Thus, contention of shared resources, either hardware or software, is the main cause of performance degradation.

The effect of different resource management schemes on the performances of NUMA architectures, either in user space or inside the operating system has been extensively studied and is an ongoing work. Among the different solutions proposed to manage multicore architectures, some solutions try to enhance application performances by proposing better user space resource management, others consider a more global approach by modifying the operating system or even replacing it. This section is organized as follows. Section 2.3.1 presents user space related solutions, Section 2.3.2 introduces runtimes or kernel modifications, and Section 2.3.3 describes new operating system designs built to tackle multicore architectures.

2.3.1 User space solutions

User space solutions include several techniques, from scalable memory allocation to specific profiler for NUMA architectures. All these solutions share the common characteristic of managing memory in multicore systems. They proceed using different methods, either aware of topology or not, to maximize performances by reducing contention on shared hardware resources or data structures. Works described below are summed up in Table 2.3.

Table 2.3: Related work on NUMA: user space solutions

Proposal	Issue(s)	Approach(es)
Hoard	false sharing (cache contention)	load balancing
Streamflow	lock contention + TLB contention	load balancing + lockless
MemProf	locality + interconnect contention	monitoring
NAPS	locality + interconnect contention + lock contention	load balancing + lockless

Hoard: A Scalable Memory Allocator for Multithreaded Applications As explained in Section 2.1.1 and 2.1.2, processor caches carry at the same time an important increase in terms of memory access latency but also introduce different issues related to the cache coherency mechanism. *False sharing* is the name given to one such issue that arises when different processors are accessing data from two different locations that are close and thus share a cache line. As cache controllers load and store data at the granularity of a cache line, when two different processors write to different data in the same cache line they force each other to invalidate the cache line even though there is no logical reason to do so, greatly degrading performances.

Berger et al. [4], propose Hoard, a memory allocator which avoids false sharing when possible.

The root causes of false sharing situations are numerous, those studied in their proposal are classified in three categories. *Program induced*, when the program creates a false sharing situation not related to the allocator, such as giving away memory to another processor. *Actively induced by the allocator* when the allocator is the cause of the false sharing situation such as satisfying requests from different processors using part of the same cache line. Finally, *passively induced* when the program is the cause of the false sharing situation but the allocator does not solve the issue when it could. This happens when a processor P1 gives some of its memory to another processor P2, this memory is freed by P2, and the allocator reuses this freed memory to satisfy allocation requests from P2. In this case, the memory should have been reclaimed by the core which initially allocated the memory for P1.

Hoard uses per processor heaps and a global heap. Each heap owns different superblocks which are allocated by requesting virtual memory from the operating system (using `mmap`). When multiple threads allocate memory simultaneously, they allocate from different superblocks, avoiding actively induced false sharing. Deallocated blocks are always returned to their original superblock, avoiding passively-induced false sharing.

Moreover, Berger et al. improves multiprocessor scalability by bounding the *blowup* of their allocator, which they define for a given allocator as "its worst-case memory consumption divided by the ideal worst-case memory consumption for a serial memory allocator". They prove that Hoard worst case memory consumption does not grow with the memory required by the program.

However, Hoard does not solve contention due to synchronization between processors accessing the same heap. Berger et al. consider that an application with a bad access pattern is itself not scalable, they consider the producer-consumer pattern to be the worst acceptable scenario. They evaluate Hoard behavior for this pattern to a twofold slowdown.

To sum up, Hoard is a memory allocator designed to tackle false sharing, a type of cache contention, by balancing allocation across the different cores.

Scalable Locality-Conscious Multithreaded Memory Allocation – Streamflow Similarly to several thread-safe memory allocators, including Hoard, Streamflow, proposed by Schneider et al. [48], uses thread-private heaps to manage memory in order to reduce contention between threads.

In order to increase scalibility while reducing cache and paging related bottlenecks, their primary contribution is to decouple local operations from their remote counterpart. This enables local operations for their allocator to be synchronization-free, avoiding both lock contention and the latency of atomic instructions. The remote deallocation mechanism uses a lock-free list in which freed blocks are pushed, letting the local thread reclaim these blocks when needed, without disrupting local operations.

Moreover, as explained in Section 2.2.1, the page table traversal can constitute a bottleneck when accessing memory as it contains multiple levels and is the reason why a cache for page table translation, the TLB, was introduced. However, the TLB, as any other type of cache, suffers from cache-related issues such as cache pollution when numerous different pages are accessed. To avoid TLB pollution by diminishing the number of stored TLB entries, Streamflow uses huge pages as the backing memory for their thread-local allocators. Finally, as recent CPUs often implement cache prefetching, the contiguous nature of these large allocations takes advantage of the larger private CPU caches.

In the continuity of Streamflow, Marotta et al. [34] focus on the improvement of the scalibility of the back-end allocator. While Streamflow uses hugepages in a centralized fashion in the back-end for contiguous allocation, Marotta et al. tackle the issue of the back-end scalabilty through a lockless approach.

The memory allocation strategy presented above tries to increase scalability by avoiding TLB pollution (page cache contention) and reducing lock contention through balancing memory accesses across cores and using a lockless approach.

MemProf: A Memory Profiler for NUMA Multicore Systems Works presented above often favor locality as a mean to avoid concurrent accesses to allocators themselves, and thus contention, by balancing allocation requests. However, while favoring locality at allocation time solves part of the problem, good memory placement during the whole application life is key to satisfying performances.

In order to improve memory placement during the whole application life, Lachaize et al. [31] propose to apply small scale application-level optimizations to the source code. In order to detect the patterns of memory accesses and threads interactions, they introduce MemProf, a profiler targeted at detecting these patterns on NUMA architectures. It consists of an user library and a kernel module which allows the profiler to monitor memory accesses using *Instruction-Based Sampling*, a hardware feature from the processor.

They consider three different classes of access patterns that can negatively impact performances on NUMA machines. First, remote usage of a memory location previously allocated

on another node, which is solved either by changing the original allocation node or if the latter cannot be determined statically, migrating the memory at runtime. The second pattern consists of alternate, non-concurrent remote accesses to a memory location by two different nodes, which can be solved either by collocating both accessing threads on the node owning the memory or if applicable, migrating the thread over time on the node owning the memory. Third, concurrent remote accesses to the same location can be treated either by collocating threads on the owning node or by duplicating the object and synchronizing the copies over time (this is recommended solely for read-only or mostly-read objects). Finally, when too much locality causes the memory controller to be saturated, the authors suggest to balance the most used objects over several nodes.

To conclude, Lachaize et al. propose to reduce the overhead of NUMA architectures by increasing locality and reducing interconnect contention through a profiler using monitoring to detect bad access patterns.

A Study of the Scalability of Stop-the-world Garbage Collectors on Multicores – NAPS As Lachaize et al. suggest, the lack of NUMA-awareness in applications is the cause of severe performance bottlenecks. Gidra et al. [20] focus on garbage collectors as an example of such applications. They present and evaluate different approaches to alleviate the load on memory controllers and increase memory locality.

The algorithm of Parallel Scavenge, the garbage collector of OpenJDK7, classifies objects according to their age, into three different categories, *young* generation composed of newly created objects, *old* generation representing long-lived objects and *permanent* generation containing the Java classes.

Regularly, the garbage collector stops the program execution and collects the memory in order to free objects unreachable by the application. When an object from the young generation survives the first collection it is moved to a transition space, after a new collection, all surviving objects from the transition space are promoted to the old generation space.

Gidra et al. introduce a garbage collector called NAPS, which implements different policies. The first approach presented, called *Interleaved Spaces* allocates the pages for the different memory spaces using a round robin policy, which reduces imbalance. The second approach *Fragmented spaces* divides the spaces in multiple fragments each physically belonging to a given node. Under this policy a thread allocates only from a fragment belonging to the node on which it executes. This improves locality as a thread tends to access recently allocated objects and thread migration is rare. However, this second approach can increase imbalance when a specific thread allocates objects for others. The last approach *Segregated space* uses the memory layout of fragmented space but additionally forbids access to remote objects for all garbage collector threads. This forces perfect memory locality at the cost of exchanging messages between nodes.

The evaluation shows that balancing memory accesses among the nodes greatly increases

performances by reducing the strain on a given memory controller. The evaluation also shows that improving locality becomes important when reaching a certain number of cores, to avoid saturating the interconnect network.

Gidra et al. also tackle the issue of lock contention. The garbage collector of OpenJDK7 uses locks to synchronize accesses to internal data structures. In NAPS, the lock contention is removed either by changing these data structures for lock-free alternatives or changing the synchronization mechanism to remove locks.

A second study of Gidra et al. [21] on garbage collection with a different design increases memory locality not only for newly allocated objects (young generation) but also older objects, further improving performances on NUMA architectures.

NAPS increases the performances of the garbage collection for Java by increasing locality, reducing interconnect contention and removing lock contention through a combination of two techniques: balancing memory accesses and lockless synchronization.

2.3.2 Runtimes and kernel policies

While careful resources placement for a given program improves performances for one use case, it is a complex task as evidenced by the need for specific tools such as MemProf [31]. Another approach to better handle NUMA architectures is through runtimes or generic policies integrated in the operating system.

Works presented thereafter tackles NUMA related issues such as locality, or contention using different techniques, without directly modifying applications. These approaches are summed up in Table 2.4 and described in more details below.

Table 2.4: Related work on NUMA: runtime and kernel approaches

Proposal	Issue(s)	Approach(es)
MCTOP	locality + code portability	monitoring + generic policies
Carrefour Carrefour-LP	contention (memory + interconnect)	monitoring + load balancing
AsymSched	interconnect contention	monitoring + load balancing

Abstracting Multi-Core Topologies with MCTOP While applications-specific optimizations can lead to improved performances on NUMA architectures as shown above, these modifications are too often linked to a specific topology or operating system specific topology discovery. Chatzopoulos et al. [10] propose a library which allows developers to define resources placement for their application in a portable manner, using an inferred topology.

MCTOP defines an abstraction of multicore topology using generic concepts, they thus define *hw_context*, which is a hardware thread if the machine supports hardware multithreading

(SMT) or a core, *hwc_group* which can be a core (group of hardware thread) or a group of core, *node* a memory node, *socket* which contains a hwc_group with additional information about memory nodes and *interconnect* which represents the connection between sockets and include information such as the communication latencies.

Using these abstractions, MCTOP infers the topology of ccNUMA machines using latency measurements. They proceed as follows, first they collect hw_context to hw_context latency measurements, then group components together depending on both the communication latency between each other and the communication latencies to other groups. Finally, they assign roles to the components, thus creating a multicore representation.

Chatzopoulos et al. propose multiple standard policies built on top of their deduced topologies, such as "Topology-Aware Work Stealing", "Topology-Aware Reduction Trees" (which can be used to implement MapReduce paradigm) or "Educated Backoffs".

For portability reasons, this abstraction avoids the operating system representation of the topology. It uses monitoring techniques to determine the topology and apply generic policies to increase both locality and code portability.

Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems – Carrefour Unlike MCTOP, Dashti et al. [13] propose to introduce a new placement algorithm, called *Carrefour* in the Linux kernel.

Carrefour specifically targets congestion on the memory controller and interconnect network. It operates using four mechanisms. *Page collocation* relocates a physical page close to the thread accessing it, which is efficient when the memory is accessed by a single thread or collocated threads. *Page interleaving* balances some pages among the different nodes. *Page replication* consists in replicating the memory on different nodes and keeping it synchronized with a coherency protocol. Finally, *Thread clustering* collocates threads sharing some memory.

In order to decide which mechanism is adapted to a given situation, Carrefour first gathers information about memory and cache accesses using hardware instruction sampling. Using this information, Carrefour takes two decisions. First whether or not it generates enough memory traffic to justify the use of Carrefour. Second, if this application would benefit from page replication, interleaving or collocation, multiple mechanisms can be enabled for a given application. Finally, when a page is sufficiently accessed during the application lifetime, Carrefour decides for this given page which of the globally enabled mechanisms is the most appropriate.

Considering that memory bandwidth is more susceptible to cause performances issues in newer NUMA architectures than remote accesses latency, this approach from Dashti et al. addresses the congestion of both the memory controllers and the interconnect.

A followup work by Gaud et al. [19] extends the original Carrefour algorithm to large pages. As large pages are known to reduce both the overhead of paging translation and TLB miss rate, they are used in different memory management mechanisms as testified by Schneider et al. [48]. However, Gaud et al. [19] reveal that they may also increase the imbalance in the

distribution of memory controller requests and reduce accesses locality, the two main factors of performances on NUMA architectures.

In their work, they identify two new issues related to large pages and NUMA. First, large pages cause the number of pages susceptible to be migrated to decrease. In some cases, this can even lead to fewer very accessed large pages than NUMA nodes, making balancing impossible. Second, as the size of individual pages grows, they are more susceptible to contain individual blocks of data used by different threads. They refer to this problem as *page-level false sharing* because of the similarity with the false sharing problem in cache lines.

To correct these issues, the extension of the Carrefour algorithm, called *Carrefour-LP* combines two different approaches, the *reactive* approach which monitors large pages and splits them into normal sized pages before applying the standard Carrefour algorithm, and the *conservative* approach which, to the contrary, merges small pages into large ones when better performances are expected. These decisions are based on the same performance counters (instruction sampling) than the original Carrefour algorithm.

Both Carrefour and Carrefour-LP use monitoring to detect memory access patterns and balance the load across NUMA nodes. This approach aims to reduce the contention on the memory controller and interconnect network.

Thread and Memory Placement on NUMA Systems: Asymmetry matters – AsymSched The different heuristics presented above try to solve performance issues on NUMA architectures by collocating, or conversely, spreading the resources on different nodes depending on the access patterns. However, the nature of the connection between these nodes can play an important role in deciding of the resources placement.

Dasthi et al. [13] consider that the congestion on the interconnect is more important than the latency of remote accesses. Similarly, Lepers et al. [32] study the impact of the interconnect asymmetry on placement decisions.

In particular, they show that it is best to prioritize the path with the maximal bandwidth rather than with the minimal number of hops. They propose an algorithm to place threads and memory such that the most intensive CPU-to-CPU or CPU-to-memory communications occur on a link with the best possible bandwidth.

Similarly to Carrefour, their placement algorithm, *AsymSched* uses hardware counters to gather information. However, the lack of CPU-to-CPU or CPU-to-RAM counters, makes the number of accesses from a given CPU to a node, which encompasses both of the preceding cases, the only available metric.

Because of the lack of specific CPU-to-CPU and CPU-to-RAM counters, they consider that any CPU with a high recorded level of access to a given node might be communicating with any CPU on the accessed node. Moreover, they cannot determine the access patterns of a CPU accessing its local node, and thus consider it might be communicating with all the CPUs of its local node. Using the gathered information, AsymSched computes the possible placements and

chooses the configuration that both maximizes the overall communication bandwidth across all applications and minimizes the number of page migrations.

This work adds a third criterion, the asymmetry in bandwidth of the interconnect, to the usual two criteria which characterize performances on a NUMA machine, the congestion of the interconnect and memory controllers on one hand, the latency of remote accesses on another hand.

2.3.3 Operating system solutions

With different approaches ranging from per application optimizations to OS-level scheduling and placement policies, proposals described before all use a standard operating system, not one specifically designed to manage NUMA architectures. Works presented thereafter all share the common idea that current operating systems are not correctly managing large multicores. Therefore, they propose entirely new operating system designs. Table 2.5 sums up the studied proposals.

Table 2.5: Related work on NUMA: New operating system designs

Proposal	Issue(s)	Approach(es)
Corey	OS-level structures contention	avoid sharing (processes decide what is shared)
The Multikernel	OS-level structures contention	message passing (one kernel per core)
fos	contention + portability	message passing (processes and OS do not share cores)

Corey: An Operating System for Many Cores Because classic operating systems share data structures across the whole machine, they introduce an overhead which cannot be removed using only user space mechanisms or OS-level resources placement. To solve this issue, Boyd-Wickizer et al. [6] propose a new operating system design to avoid the bottleneck of shared OS-level data structures.

This work is guided by the principle that applications should control sharing of operating system data structures and propose three abstractions demonstrating this principle. *Address ranges* allow applications to control which parts of the address space are private per core and which are shared, *Kernel cores* allow applications to dedicate cores to run specific kernel functions and *Shares* control which kernel object identifiers are visible to other cores.

Address ranges control how virtual memory mapping are shared by cores. As each processor usually contains its own page table pointer and corresponding structures, sharing and updating said structures for all cores has a cost, which address ranges aim to control. A core with a non-shared address range can update it without requiring synchronization or TLB shootdowns. A

shared address range will incur the cost of synchronization. Kernel cores allow an application to dedicate a core to kernel functionalities such as networking, reducing the contention for driver data structures and related synchronization mechanisms. Finally, Shares allow applications to keep some common kernel identifiers such as process identifier or file descriptors private.

Because typical kernel structures are shared and rely on expensive mechanisms such as cache-coherency, avoiding implicit sharing is one method to greatly increase performances on multicore architectures.

The Multikernel: A new OS architecture for scalable multicore systems In the same manner as Corey, Baumann et al. [3] propose a new kernel design reducing implicit sharing of kernel data structures. Their proposed design, the *multikernel*, treats the machine as a network of independent cores and uses message passing for inter-core communication. To demonstrate the new multikernel design, they produce an implementation called *Barrelfish*.

This design relies on three main principles, all inter-core communication are explicit, OS structure is independent of hardware and OS state is replicated, not shared.

For the same reasons as Corey, using explicit communications and replicated states avoids the cost of synchronization when updating shared data structures. It also favors locality of accesses to kernel structures. The distributed architecture of the multikernel simplifies the reasoning of programmers by making the kernel structure modular but also makes it easier to preserve OS structure while hardware evolves.

Factored Operating Systems (fos): The Case for a Scalable Operating System for Multicores While the multikernel design does not use shared memory directly, the message passing mechanism, used to explicitly communicate between cores, still uses some form of shared memory. In contrast, Wentzlaff et al. [59] propose an operating system design which does not rely on shared memory or cache coherence, *Factored Operating System* (fos).

Considering that future machines will feature a very important number of cores (hundreds), fos proposes to exploit this specificity by using a client/server architecture for both the operating system and applications. Similarly to the multikernel design, servers communicate only via message passing, which is delegated to hardware.

The goal of fos is to leverage standard Internet servers optimizations such as caching, replication or spatial distribution. To this end, the operating system functionalities themselves are split into different services which are distributed on several cores. The components of fos architectures are the following. A thin microkernel which provides the communication API and acts as a name server to contact other cores. This microkernel executes on every core. An OS layer, which provides standard operating systems services such as resources management (cores, memory), I/O functions (file system, networking). These OS functionalities are implemented as servers. They run on top of the common microkernel and are distributed across the machine.

Distributing applications and OS services on different cores avoids implicitly sharing per core resources such as data caches or TLB, leading to improved performances. Moreover, avoiding shared memory nullifies the impact of cache coherency on performances and as with the multikernel, eases to preserve the new design while hardware evolves.

2.4 Conclusion

This chapter presents different multicore architectures and more specifically the NUMA architecture. Nowadays, NUMA architecture has become the most common architecture for servers, with hundreds of cores distributed in multiple nodes. While the high density of resources, both compute or memory, allows very important computation power, this can be done only by correctly managing the complex topology, caches, memory controller, interconnect network.

In this chapter, we presented NUMA machines, their components and how resources placement is the main cause of performance degradation. Then we explained how to modify resources placement through the use of memory management techniques, both in kernel space, with the page table, and in user space with tools derived from the kernel page table. Finally, we presented some solutions proposed in related work to remedy aforementioned performance-degrading patterns. These solutions can be classified into three categories: first, user space solutions which target a specific application or class of applications; second, runtimes and changes to the kernel placement policy, more generic approaches; third, complete redesigns of the operating system layer.

The first part of our approach to resources placement falls in the second category, runtimes solutions. As increasing parallelism is important to better leverage these new architectures, our approach, *Commmama* uses memory management techniques to overlap communication with computation in the Message Passing Interface (MPI) runtime used by many high performance computing applications.

The second part, Scalevisor, falls between the second and the third categories as it proposes to use virtualization techniques to manage resources in a layer below the operating system. Scalevisor avoids the hassle of modifying the code of legacy operating systems while proposing a new design to memory management in complex architectures.

Chapter 3

Commmama, motivation and design

With the evolution of computer architectures, especially the growing number of cores and fast networks, one of the keys to performance improvement is maximizing parallelism by correctly managing compute, memory, and network resources. At the user space level with message passing in applications, this means focusing on the ability to overlap network communications with CPU-bound computation. This chapter thus addresses the challenge of maximizing the parallelism in applications using both computation and network.

MPI is a widely used library for HPC applications. It offers a message-based interface to communicate between nodes, with either two nodes involved (point-to-point operations) or more than two nodes (collective operations). The MPI specification features an important number of communication primitives, each offering different semantics. These semantics range from blocking senders and receivers until the communication is completed to letting the senders and receivers overlap communication with computation. The data provided by the application to the MPI runtime cannot be modified until the communication is completed. In blocking communications, this is transparent to the user as blocking communications block until the communication is completed. However, for nonblocking communication, the application must not modify the communication buffers containing the data of an ongoing communication.

Implementing an application that efficiently uses the nonblocking primitives is complex. As nonblocking primitives do not block until the communication is completed, the application can overlap communication with computation while the nonblocking communication is underway. However, the application must track which buffers are involved in ongoing communications and explicitly wait for these communications to end if they need to access said buffers. Choosing the right point in the program to wait for the completion of an operation may be difficult when nested functions may access the memory or not. This difficulty leads developers to take a pessimistic approach in which the communication completion is awaited earlier than necessary, thus losing some overlapping potential. In the worst scenario, the developer will use blocking communication to ensure the program correctness, thus trading efficiency for

simplicity of design.

In order to improve the performance of MPI applications we propose Commmama. Commmama automatically transforms blocking primitives into nonblocking ones at runtime. To this end, Commmama uses a separate communication thread responsible for communications and a memory protection mechanism to track memory accesses in communication buffers. This allows communications transformed by Commmama to progress in parallel until their data are needed.

This chapter is organized as follows. In a first part, in order to better understand the challenges of overlapping communication with computation, this chapter details the different types of communications available in MPI and their characteristics. We show there are two key factors needed to improve parallelism. First, background progress which guarantees communications are processed while computation is running. Second, the size of the window during which computation and communication can be run in parallel.

In a second part, we present our proposal, Commmama, which transforms blocking communications primitives into nonblocking ones to increase the potential for communication and computation overlap. Commmama guarantees both progress for these communications and the largest window during which communication and computation can be processed in parallel. Commmama combines both the simplicity of blocking communications and the efficiency of nonblocking ones to provide an efficient and hassle-free approach to MPI application design, thus avoiding the aforementioned trade-off between efficiency and simplicity of design.

Finally, we present an evaluation of Commmama and show that it increases parallelism by overlapping communication with computation. Our results show that when there is enough computation to overlap the communication phase, the whole communication is done in the background. In contrast, when there is not enough computation, we show that our approach still performs well, with little overhead compared to a standalone MPI runtime.

3.1 Background and motivation

Message Passing Interface (MPI) [37] is a specification designed to establish a standard for message passing libraries used in high performance applications. This specification is implemented by different open-source libraries such as OpenMPI [41] or MPICH [38]. Because of its efficiency and stability, it is used by many projects involving distributed applications or high performance computing.

The MPI specification consists in a set of primitives and semantics which must be respected by any given implementation. MPI focuses on message passing and thus is not designed for any particular data transfer medium. Among the different implementations, multiple transport layers are supported, such as shared memory, Ethernet or InfiniBand, making MPI a useful tool in many different environments.

This section is organized as follows. Section 3.1.1 presents the MPI specification, the different types of communications available and their semantics. Section 3.1.2 explains why some primitives can overlap communication with computation and some cannot. It then presents related work which aims to ensure background progress or maximize the overlapping ratio. Section 3.1.3 presents how our approach combines the simplicity of blocking primitives and the overlapping capability of nonblocking primitives.

3.1.1 MPI primitives and semantics

The MPI specification defines several primitives tailored for different use cases. MPI primitives can be divided in two main categories, blocking ones and nonblocking ones, based on their behavior. This section details these primitives and their semantics.

Blocking and nonblocking communications According to the MPI specification, an operation completes when the user can reuse resources specified as input parameters for this operation, this includes input communication buffers. When the communication completes, output parameters of the functions have been updated, including any output communication buffers. A buffer can be used by a primitive as both input and output.

A blocking communication does not return until the operation completes, therefore, until the message data are either stored by the MPI runtime or transmitted through the transport layer. In contrast, when using a nonblocking communication, the communication buffer must not be used before a waiting primitive, such as `MPI_Wait`, has been called. More precisely, the buffer cannot be written to, in case of ongoing send operations, or accessed in case of ongoing receive operations before the communication has completed.

In order for blocking communications to be able to return while the receiver is not ready to receive yet, the data contained into the buffer and the metadata of the operation must be stored somewhere. How data are stored is not specified in the standard and actually differs between implementations and transport layers. In most cases, the data is either copied in a system buffer or simply sent through the transport layer.

Synchronous and asynchronous communications While the blocking or nonblocking property of a communication primitive depends on whether the communication buffers can be reused or not when the primitive returns, the synchronous or asynchronous property of a primitive depends on whether senders and receivers must synchronize.

A synchronous communication completes only if a matching receive is posted and the receive operation has started receiving the data. In other words, a synchronous communication forces synchronization with the other nodes taking part in the communication. In contrast, an asynchronous communication primitive offers no additional information on the state of the other nodes and may or may not synchronize with them.

The synchronicity property of an MPI primitive is independent from its blocking or nonblocking behavior. MPI primitive can thus belong to any combination of blocking or nonblocking and synchronous or asynchronous.

For example, in a synchronous, nonblocking send operation, the sender returns immediately after starting the communication, the buffer cannot be reused before a call to a waiting primitive, which will synchronize with the receiver. The waiting primitive will complete only after the receiver has started to receive. In contrast, in an asynchronous, blocking send, the sender returns after the buffer has been copied in system buffers by the MPI runtime, the buffer can then be reused but the receiver node may not have executed the receive primitive yet.

Illustration: the `MPI_Send` function family The `MPI_Send` function family illustrates the different properties of blocking, synchronicity or lack thereof.

There are multiple variants of the "send" function: the standard `MPI_Send`, `MPI_Bsend`, `MPI_Ssend` and `MPI_Rsend` and their nonblocking counterparts. The distinction between these four functions depends on two concepts, the message buffering or lack thereof and the synchronous or asynchronous component of the primitive. The characteristics of the different blocking send primitives are summed up in Table 3.1, more details on each primitives follow.

Table 3.1: Blocking send modes

	Synchronous	Buffered
`MPI_Send`	No	Implementation dependent (for small message)
`MPI_Bsend`	No	Yes
`MPI_Ssend`	Yes	Irrelevant
`MPI_Rsend`	No	Implementation dependent (for small message)

As explained earlier, message buffering is the act of copying the data contained in the communication buffer to an alternate buffer in which the data will stay until the communication is completed. This allows the communication buffer to be reused by the application while the communication proceeds.

`MPI_Send` also called "standard mode" send is the basic send primitive, for performance reasons, most MPI libraries implements buffering for small messages when using the standard mode `MPI_Send` primitive. The primitive may thus choose either to synchronize with the receiver or to buffer the message depending on its size.

`MPI_Bsend` is similar to `MPI_Send` except that if the matching receive has not been posted `MPI_Bsend` must copy the message in MPI internal buffers and return, whereas `MPI_Send` can wait for the matching receive and avoid buffering.

`MPI_Ssend` is a synchronous communication primitive, it can return only after a matching receive has been posted, it never needs to copy the message content to MPI internal buffers as

it can directly initiate the sending operation after the matching receive is detected.

`MPI_Rsend` has the same semantics as a `MPI_Send` except that it informs the runtime that a matching receive has been posted already.

As nonblocking primitives only notify the MPI runtime that a communication starts, all these primitives return immediately, this is the case for example of `MPI_Isend`, and even `MPI_Issend`. However, the send-complete operation, ensured by a call to a waiting primitive, has the same semantics as its respective blocking version, presented in Table 3.1. For example, a `MPI_Wait` following a `MPI_Issend` is synchronous, and must wait for the matching receive to be posted. In all cases, MPI semantics specify that when the communication is completed, whether because the message has been buffered or not, the communication buffer is free to be modified by the application.

Point-to-point and collective communication primitives The MPI specification provides point-to-point primitives and collective primitives. A point-to-point primitive involves exactly two processes: a sender and a receiver. A collective primitive involves more than two processes.

These primitives represent different communication patterns, such as one sending to all (`MPI_Bcast`), all sending to all (`MPI_Alltoall`), or all sending to one (`MPI_Gather`) and their respective nonblocking versions, added in the third edition of the specification, `MPI_Ibcast`, `MPI_Ialltoall` and `MPI_Igather`.

In contrast to the point-to-point primitives above, collective communications have no synchronous versions, the only synchronous collective primitive is `MPI_Barrier`, other collective operations are not required to synchronize the processes inside of the communication group.

3.1.2 Overlapping communication with computation

While message buffering allows blocking communication primitives to overlap communication with computation to a certain degree, it also produces an overhead due to copying the communication data to an internal buffer. This becomes costlier when message size increases, hence the compromise used in multiple MPI implementations to buffer `MPI_Send` data only for small message sizes. The difference between blocking and nonblocking communications thus resides in the ability for nonblocking communications to be overlapped with computation.

To honor a send request, the runtime has two main options. First it can send the full data and let the receiving process buffer the data until a matching receive is posted. This first option is known as *eager*. Second, it can send only a protocol message indicating a send operation is posted and wait for the receiving side to respond when a matching receive is posted. In this second option, called *rendez-vous*, a handshake occurs before the data are sent, allowing the data to be received by the other process directly in the user provided communication buffer.

Figure 3.1 depicts such communications. The brown bar represents the location of the data (horizontal axis) through time (vertical axis). Thus a long brown bar means data stay a long time in a given memory zone, either user memory or internal MPI runtime memory.

For example, in the eager communication on the left, the data evolves as follow. At the beginning, data are stored in user memory, then during the call to `MPI_Send` they are copied to a buffer owned by the MPI runtime. The MPI runtime then proceeds to send the data through the network, which are received by the other process and buffered. The data are now stored in a buffer owned by the MPI runtime of the second process. When the second process calls the `MPI_Recv` primitive, the data are copied to the receive buffer of the user.

For *rendez-vous*, on the right, the data are stored in the user buffer until the handshake has occurred, when the matching receive has been posted. Data are then sent directly to the receiving user buffer during the end of the `MPI_Recv` primitive.

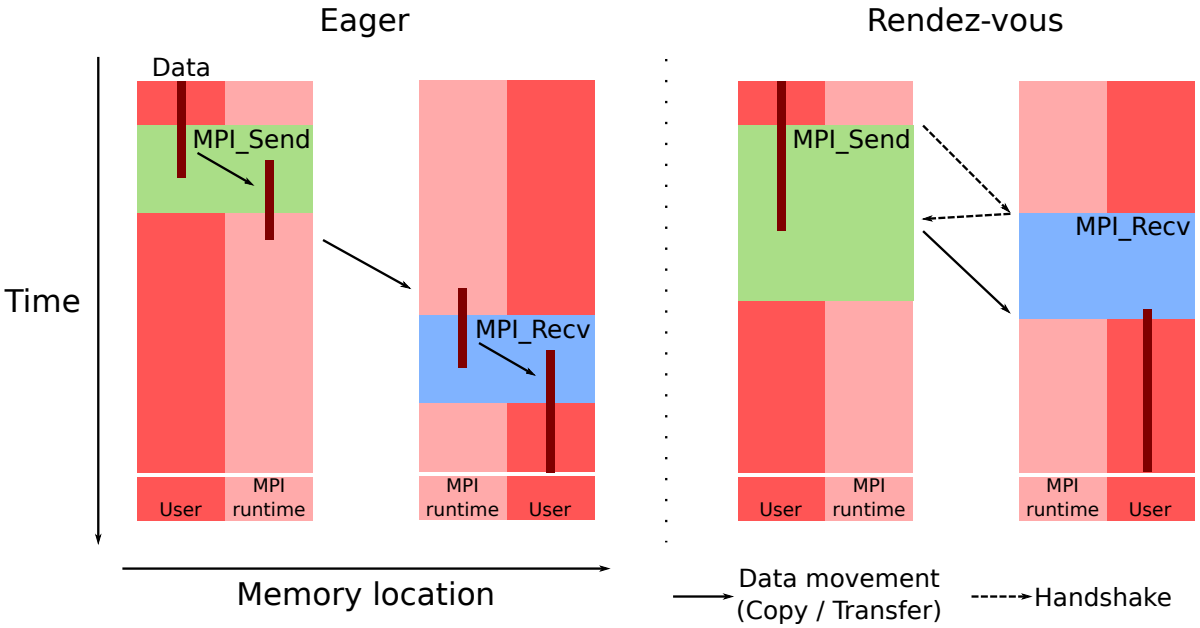


Figure 3.1: Comparison of *eager* and *rendez-vous* for blocking communications

With the *eager* protocol, the sender does not have to wait for the receiver to be ready to receive this particular message. In other terms, there is no need for synchronization between the sender and the receiver. While this can improve performances, it forces the MPI runtime on the receiver side to allocate sufficient buffer space to contain all the in-flight messages. Moreover, when the receiver calls the matching receive primitive, it still needs to copy the content of the already received message from the temporary buffer to the final application buffer. This protocol is thus not profitable for bigger message size.

In contrast, for the *rendez-vous* protocol, the sender only sends the message metadata (source and destination among others) to the receiver and waits for an acknowledgment before sending the data. This acknowledgment is sent back by the receiver when it enters the receiving primitive which in turns forces a synchronization between the two processes runtimes. This protocol directly writes the data into the application buffer given to the respective receiving primitive thus avoiding a copy and the need for internal buffer space.

For blocking communications using the *eager* protocol, the send primitive returns immediately after the data are copied. The application can thus continue its computation. Therefore, the communication is overlapped with computation except for the time taken to copy data to MPI internal buffers. In contrast, this is not possible due to the synchronizing nature of the *rendez-vous* protocol.

Nonblocking communication and progress In order to improve performances, nonblocking primitives were designed to help overlapping communication with computation.

Nonblocking primitives are composed of two parts, the "start" primitive (`MPI_Isend` for example) and the completion primitive (`MPI_Wait` for example). As the nonblocking version of the primitive only registers the intent to do a send or receive operation, it returns immediately, without waiting for the communication to complete. The theoretical overlapping window extends from the start of communication to the completion primitive call.

For *eager* protocol, data can be pushed into the distant runtime buffer between the call to the `MPI_Isend` primitive and the call to the waiting primitive. This data transfer does not require a corresponding receive to be posted. When the call to the receive-complete occurs, either because the call is blocking or because of a waiting primitive, the runtime buffer is copied to the local buffer. When supported by the communication controller, the sending or receiving procedure can thus take place concurrently and the program can continue its execution until the communication result is needed, at which point it calls a waiting primitive. This allows communication to be overlapped with computation.

For the *rendez-vous* protocol, this does not work in most of the cases because the MPI runtime shares the thread with the application. In details, for the *rendez-vous* protocol, the application needs to interact with the MPI runtime twice, a first time for the handshake and a second time for exchanging the data. This issue appears for nonblocking communications, even if the communication controller supports sending data in the background. If the `MPI_Isend` starts the handshake, when it receives the answer from the receiving process, the MPI runtime from the sender side will not be able to do any message matching while the sender runs computation, because the shared thread is busy. Thus, most of the time the acknowledgment from the sender is seen during the waiting primitive and all the data transfer occurs there. These two cases are represented in Figure 3.2

To avoid this issue, it is possible to force the application to enter the MPI runtime, giving the latter the opportunity to get the acknowledgment and start the data transfer. This can be

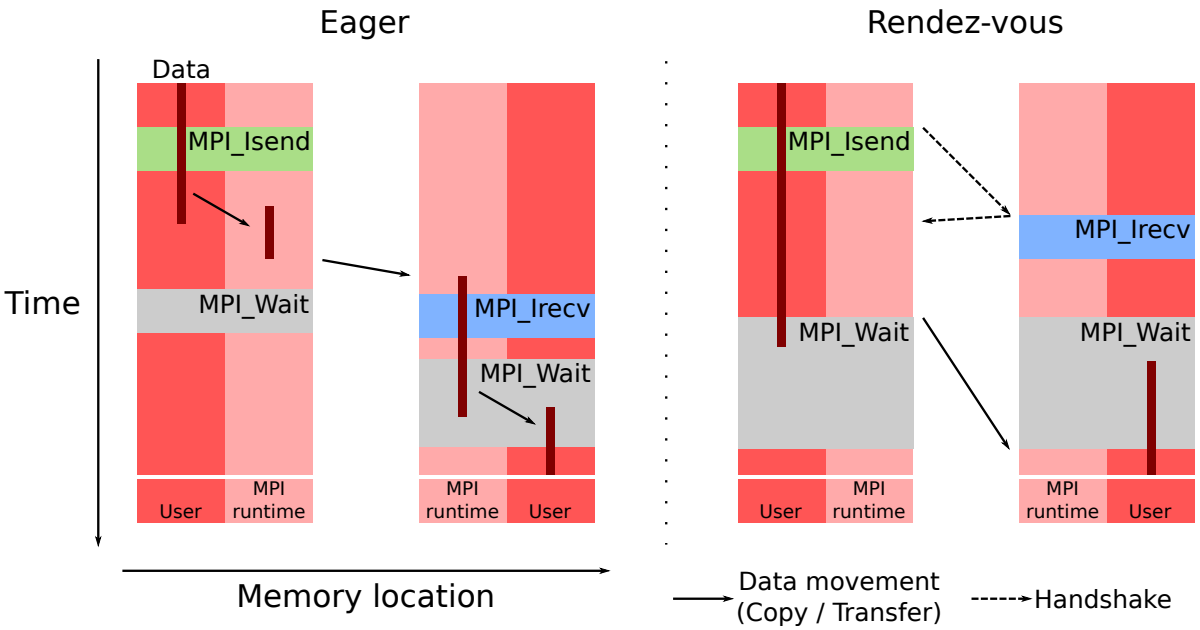


Figure 3.2: Comparison of *eager* and *rendez-vous* for nonblocking communications

achieved by calling the `MPI_Probe` primitive to help the MPI runtime get CPU time. While this basic solution can help overlapping communication with computation in some simple cases, it becomes overly complicated when the algorithm is not composed of simple loops. Furthermore, while this solves the latency issue caused by some process waiting for the handshake to terminate, depending on the type of transport layer, some of the copy operation will steal CPU time from application. In order to correctly process the *rendez-vous* acknowledgment and avoid stealing CPU time from the application, a background progress thread is necessary.

Ensuring background progress for communications In order to better overlap communication with computation, a number of techniques have been developed to ensure the progress of the ongoing communication. This is the case of Denis et al. [14] and Trahay et al. [54] which use threads to increase asynchronous progress by running communications in separate tasks. This avoids the issue mentioned above of stealing CPU time from the compute thread.

Vaidyanathan et al. [57], probably the closest approach to ours, delegate MPI nonblocking calls to a standalone thread. This approach reduces the CPU time consumed to perform the MPI calls in the application thread, and ensures the progression of offloaded communications. However, this does not increase the overlapping ratio, which directly depends on the waiting primitive placement in the application code.

Similarly, Min Si et al. [49] present an asynchronous progress system using processes in place of threads and is targeted at many-core architectures.

Didelot et al. [15] propose a different approach, with no added parallel execution unit, which uses idle cycles created by imbalance in computation time to improve progression.

Works presented here focus on making communication progress, most of the time using a separate execution context to process communications.

Maximizing the overlapping ratio While ensuring background progress of the communications is an important part in maximizing the efficiency of communication overlapping, ensuring an important overlapping ratio is also of the utmost importance.

Maximizing the ratio of communication overlapped with computation is the best method to leverage background progress and thus maximize performance. To this end, some approaches use compile time techniques to evaluate the best position for nonblocking primitives and specifically `MPI_Wait` calls to maximize the overlapping portion of communication and computation.

This is the case of Martin et al. [36] which use source to source translation to remodel the code. They use compiler pragmas to divide the code into three types of regions: send, receive and compute. This allows some communication to be overlapped with computation by running communication and compute regions in parallel on different cores. However, the compute region still needs to start with a waiting primitive (in most cases `MPI_Waitall`) which in turn seems too coarse grained to attain a satisfying overlap ratio in complex cases.

Fishgold et al. [16] propose a finer grained, automated technique which detects patterns presenting an opportunity for transformation by parsing the source code and then modifies it. This approach focuses on finding compute loops filling parts of communication buffers at each iteration and modifying them to send these parts when ready. While this technique covers a large portion of MPI applications, especially High Performance Computing related ones, our approach works for a broader field of application. Moreover, they focus on `MPI_Alltoall` collective while our work does not target a specific communication primitive.

A similar approach from Guo et al. [25] uses static code analysis to replace blocking communications by nonblocking communications in loop based computations thus overlapping communication step i with computation step $i + 1$. This approach uses compile time analysis, compute loops, and does the communication progression using `MPI_Test` primitives inserted in the loop body, which gives the MPI runtime some CPU time to make the loop communications progress. This approach is closely related to that of Murthy et al. [39].

Another attempt to increase the overlap of communication and computation is presented by Anthony Danalis et al. [12]. They describe an algorithm used to optimize the overlap window at compile time by replacing blocking primitives with nonblocking ones and placing the waiting primitive as far as possible to increase performances. However, unlike previous work they make the assumption that the compiler is aware of MPI semantics. While this allows for an even better increase in the overlap window, this makes the work of compiler

developers and MPI developers more complicated, which is not the case in our approach as it operates transparently. Nguyen et al. [40] propose a similar solution which uses clang to perform modifications.

The method presented by Saillard et al. [47] uses code instrumentation and offline analysis to determine code fragments where overlapping capabilities are not fully exploited. After the instrumented run has terminated, their tool uses the trace files to perform optimizations.

While the field of background progress for MPI primitives has been extensively studied with approaches using either compile-time or run-time techniques, all these proposals focus on the standard way of producing overlapped communications and computation: nonblocking primitives.

3.1.3 Commmama: blocking simplicity, nonblocking efficiency

As described in Section 3.1.2, the application can overlap some communication with computation using nonblocking primitives. Two key factors increase the efficiency of nonblocking communications. First, a progress engine transforms a potential overlapped communication into an effectively overlapped communication. Second, the size of the temporal window during which overlapping communication and computation occurs directly determines the amount of overlapped communication. However, using nonblocking primitive adds a degree of complexity in comparison to simple blocking primitives.

All nonblocking primitives must be completed using one of the waiting primitives, called on a request object which represents the ongoing communication. The code thus becomes more complex because of both needs to call the waiting primitive and to manage the different request objects. This is illustrated by comparing Listing 3.1 demonstrating a contrived example of a blocking `MPI_Send` usage and Listing 3.2, a nonblocking version of the same example.

```
1 void write_function(int* buffer, ...) {
2   int* result = do_long_computation();
3   memcpy(buffer, result, ...);
4 }
5
6 void example(int* buffer) {
7   MPI_Send(buffer, ...);
8   read_only_function(buffer, ...);
9   write_function(buffer, ...);
10 }
```

Listing 3.1: Simple example using blocking communication

Moreover, the placement of the `MPI_Wait` primitive in Listing 3.2, while allowing to overlap the send operation with the `read_only_function` execution, is nonetheless not optimal. If the `write_function` starts with some computation sequence not writing directly into the

```
1 void write_function(int* buffer, ...) {
2   int* result = do_long_computation();
3   memcpy(buffer, result, ...);
4 }
5
6 void example(int* buffer) {
7   MPI_Request handle;
8   MPI_Isend(buffer, ..., &handle);
9   read_only_function(buffer, ...);
10  MPI_Wait(&handle, ...);
11  write_function(buffer, ...);
12 }
```

Listing 3.2: Overlapping using nonblocking communication

communication buffer, the waiting primitive could be placed further in the execution sequence, after this computation, to provide a higher degree of overlapping and thus better overall performances.

While this contrived example easily demonstrates the misplacement of the waiting primitive, in a complex program containing deep nested functions or callbacks, placing the waiting primitive in its optimal place is close to impossible. Because statically placing the waiting primitive is difficult, and because blocking communications are easier to handle in general, we propose a method to transparently overlap blocking communication with computation.

In order to benefit from both the simplicity of blocking communications and the overlapping capabilities of nonblocking communications, we developed Commmama, an overlapping library and progress engine which automatically transforms blocking communications into nonblocking ones at runtime and manages the memory used for communication buffers to enforce the MPI specification semantics.

Commmama offloads communications to another thread, acting as an intermediary between the application and the MPI runtime. The offload thread is responsible for starting communications and ensuring background progress. This reduces tasks executed on the critical path for the computation thread to a minimum, maximizing the time dedicated to computation. In case the communication uses the *eager* protocol, the offload thread avoids the application thread having to wait for the data to be copied in the MPI runtime internal buffers. This is shown in Figure 3.3 on the left. When the MPI runtime switches to the *rendez-vous* protocol, the offload thread avoids the computation thread to be blocked while waiting for the handshake to complete (blocking case) or for the whole data transfer to occur in the waiting primitive (nonblocking case). This is illustrated on the right part of Figure 3.3.

Our proposal thus performs well on *eager* protocol but above all provides an important improvement on *rendez-vous* protocol by ensuring communications progress while the main thread continues to run computation until communication results are needed. Moreover, Com-

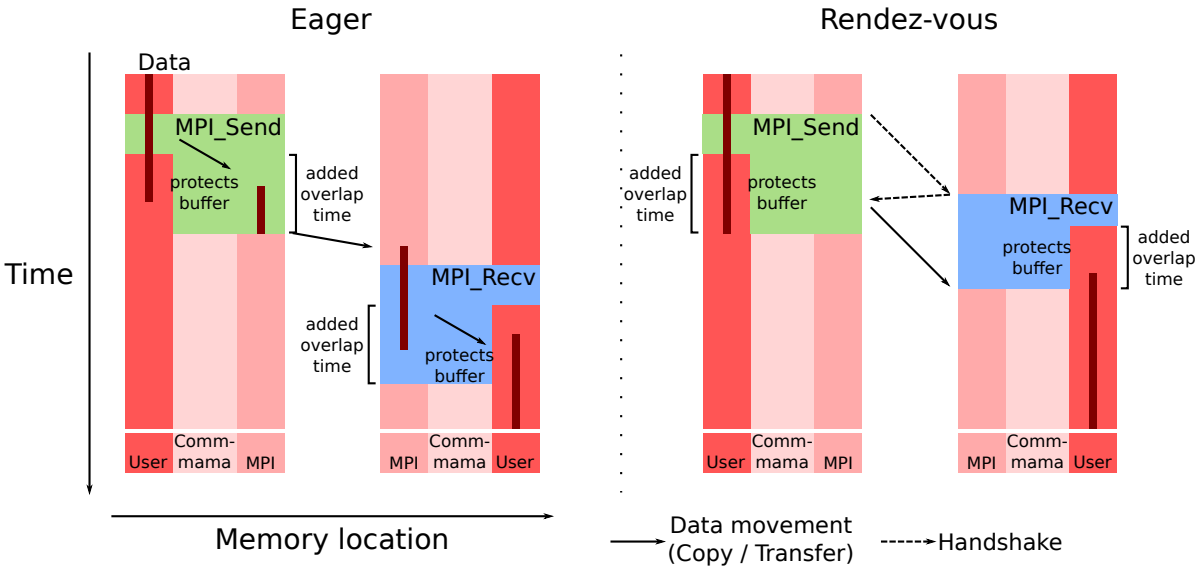


Figure 3.3: Comparison of *eager* and *rendez-vous* with Commmama

mmama does not need to introduce any modification in the application code such as adding calls to `MPI_Probe` because it directly replaces blocking calls with our custom versions. In order to increase efficiency while still enforcing the MPI specification, Commmama protects the communication buffer provided by the application. Commmama can thus detect any access which would otherwise ignore MPI semantics. When such an access occurs, Commmama forces the application thread to wait until the communication operation has completed according to MPI. The specifics of Commmama design and operation principles are described in the following section.

3.2 Our overlapping infrastructure: Commmama

In order to transform blocking communications into nonblocking ones, but still enforce the MPI specification, Commmama must guarantee the integrity of communication buffers. Moreover, to avoid the need to modify the user code, our proposal automatically replaces original MPI primitives with our custom ones. Finally, Commmama ensures ongoing communication progress to increase efficiency of communications overlapped with computation. These three features are the core of the project and thus direct its design.

Commmama is composed of three different layers. The interception layer, which intercepts and transforms standard blocking primitives into multiple calls to other layers of our library. The protection layer which manages memory for communication buffers and enforces MPI

semantics when an otherwise incorrect memory access would occur. The progress layer which is responsible for communications and offloads communications to a standalone thread.

These different layers and their interactions are represented on Figure 3.4 and detailed in the following sections.

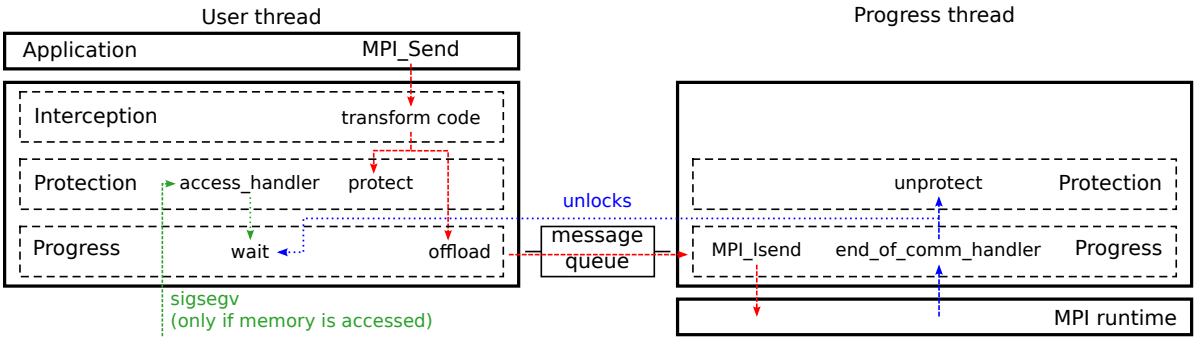


Figure 3.4: Commmama's architecture

3.2.1 The interception layer

The first step toward overlapping blocking communication with computation is to replace the base primitives of the MPI library by our custom versions. There are already multiple MPI libraries implementing the MPI specification, some of which are well-known and tested.

We chose to avoid modifying directly any MPI runtime to be compatible with any implementation and to avoid maintaining a forked version. Therefore, Commmama uses a shared library which replaces needed MPI functions by custom versions.

Our interception library first stores all the existing MPI library functions into dedicated function pointers and then declares functions with their original names. This library is then loaded before the real MPI library using *LD_PRELOAD*, effectively renaming MPI library symbols and replacing them with ours. To ease the use of this shared library, we provide an executable managing this configuration and using the interception library becomes as simple as `mpirun mpi_interceptor [args]`.

On top of being easy to use, this method allows a developer to use the library on any software without requiring any modification of the application code or recompilation. It is thus easy to benefit from the full efficiency of Commmama on existing software.

Moreover, using *LD_PRELOAD* gives our library complete control over all the MPI primitives, not only communication primitives. Among these, two are of specific interest: `MPI_Alloc_Mem` and `MPI_Free_Mem`. According to the MPI specification, these two primitives exist to allocate special memory which is dedicated to message-passing operations and could make such operations faster. In Commmama, we leverage these two functions to let

the user explicitly allocate specific memory for the use of our protection layer, as detailed in Section 3.2.3.

3.2.2 The offloading system

The offloading subsystem provides asynchronous progress for MPI communications. It consists of a standalone thread which is spawned when the MPI runtime is initialized (one thread per MPI process) and a lock-free FIFO queue in which are registered requests to be treated by the offloading thread.

The lock-free queue allows for multiple concurrent producers to enqueue new request which makes the offloading system designed to support multi-threaded calls, such as MPI+X workloads.

The offloading thread polls the queue for any new request to treat, if none are currently waiting, the thread performs calls to the MPI runtime to make pending communications progress. When a request completes according to the MPI runtime, the offloading thread invokes a callback function to which it passes the request metadata.

Metadata are stored in custom structures which are divided in two parts, a generic header part, common to all functions and a function specific part which contains the parameters needed for the function call. For efficiency reason, the header size has been reduced as much as possible, containing only an enum value denoting the type of the request, the underlying `MPI_Request`, the underlying `MPI_Status`, and a data pointer. Due to this division, extending the existing pool of available functions is easily done by adding a new structure.

Moreover, even if these structures are small in size, memory allocations on the critical path when calling communication primitives would decrease performances. To this end, the offloading system maintains a pool of preallocated metadata structures in ring buffers. This pool supports two different allocation policies. With the *strict policy*, asking the pool to provide a metadata structure when the ring buffer is empty results in an allocation error. With the *relaxed policy*, an allocation that cannot be fulfilled by the ring buffer will return a newly allocated structure using the system memory allocator. Metadata structures are freed by putting them back in the ring buffer.

Once again, the ring buffer structure supports multiple simultaneous enqueue and dequeue operations, thus allowing multi-threaded workloads.

The offloading library proposes functions with the same prototype than the MPI standard nonblocking functions for ease of use. These functions are proxy which fill the corresponding metadata structure and enqueue the structure in the message queue as represented in Figure 3.5.

After emptying the queue and thus starting the new communications, the offload thread must ensure ongoing communications are progressing. All real MPI runtime function used by the offloading thread are nonblocking, the progress can thus easily be done using a nonblocking test function for all ongoing communications at once, `MPI_Testsome`. Because the

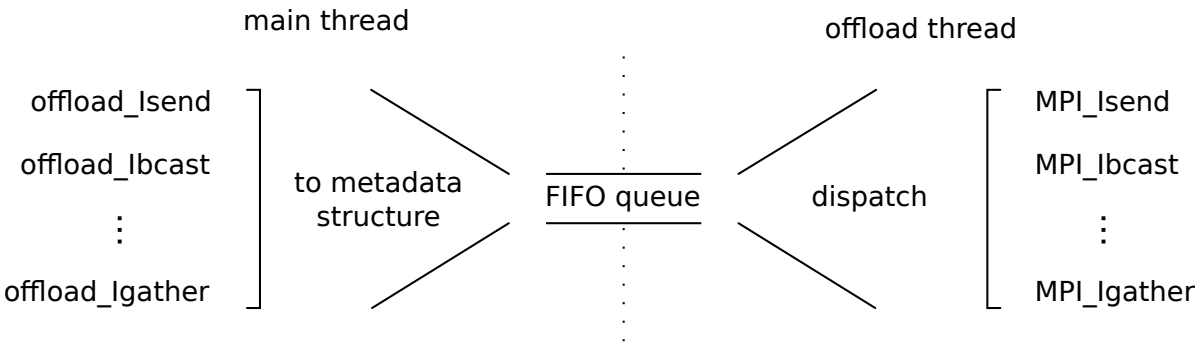


Figure 3.5: Queuing procedure, from local thread to offload thread

`MPI_Testsome` function tries to complete all the ongoing communications, faster or smaller communications can finish as soon as possible, independently of their starting order.

Finally, when a communication completes, the offload thread calls a callback to notify of the completion. While this callback allows for a completion to be treated as an event, offloaded requests can still be waited using the `offload_MPI_Wait` function, equivalent of the standard `MPI_Wait`.

As a standalone library, this offloading subsystem only takes care of communication progression and can thus be used on its own to provide asynchronous progress to nonblocking primitives. In addition to providing progress to nonblocking calls, the callback executed at the end of communications let blocking communications benefit from the progress engine as well, through the use of the protection mechanism.

3.2.3 The protection mechanism

The protection mechanism is responsible for enforcing the semantics of the MPI specification on communication buffer accesses. It uses the interface provided by the offloading subsystem to enable a memory protection mechanism on the communication buffer before offloading it and disable it when the communication buffer can be accessed freely. This corresponds, for classic nonblocking communications, to the end of the `MPI_Wait` function.

The protection framework provides two primitives respectively to enable and disable protection on a specific buffer. Both of these functions internally call the Posix `mprotect` function which allows a program to restrict access to its own address space as explained in Section 2.2.2. When an unwanted access occurs, the page fault occurring in the kernel is notified to the user space program as a segmentation fault signal (`SIGSEGV`).

According to the Posix standard, the behavior of the `mprotect` system call is undefined when called on a region of memory which was not obtained through a call to the `mmap` system

call. For this reason, we intercept and replace the calls to `MPI_Alloc_mem` and `MPI_Free_mem` by an allocating scheme using `mmap`.

While these two primitives take care of all the specifics, allowing the external interface to stay simple, internally, the protection mechanism must handle a variety of cases.

Simple sending primitive The simplest case is a simple sending primitive, considering that the program executing will not post simultaneously two send operations for which user provided communication buffers share some memory frames.

For send operations, the communication buffer cannot be modified until the communication completes. For blocking communication this is after the sending primitive returns, for non-blocking communications this is after a call to a waiting primitive on the suitable `MPI_Request`. Thus, when the original `MPI_Send` is called, it is replaced by a protection primitive which forbids write access by setting the permissions for the buffer as `PROT_READ` (read only).

Any write access following the change of permissions would result in a segmentation fault. Our custom handler for the segmentation fault signal first checks whether the memory address on which a bad access occurred is a zone managed by the protection mechanism. If the zone is protected by Commmama, it calls a modifiable callback function (called `access_handler` in the rest of the document). By default, this callback is set to automatically wait for the offloaded communication to complete.

Whether a forbidden access occurs or not, when the communication is completed, the offloading subsystem calls the callback function which in this case consists in changing the permissions back to the original setting (which should most often than not be `PROT_READ | PROT_WRITE`). The evolution of the memory state is represented in Figure 3.6. This figure presents the different states of the memory depending on the events occurring in the program execution. I is the initial state, F is the final state, P and W are intermediary states and stand for “Protected” and “Waiting” respectively.

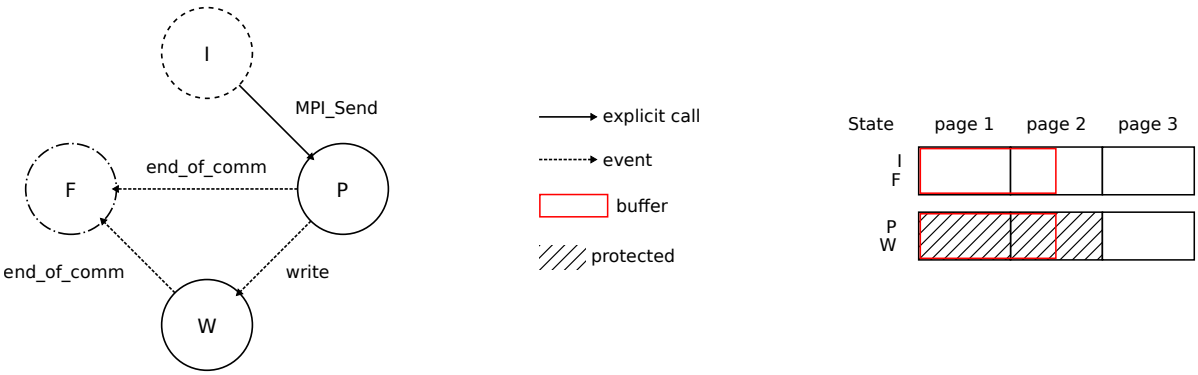


Figure 3.6: Evolution of the memory state through a simple send operation

The communication buffer being read-only, the underlying MPI primitive can still access the data and perform the send operation normally, while the specification required behavior is enforced.

Simple receiving primitive However, as mentioned in Section 2.2.2, even if the Posix API represents protection as an arbitrary combination of the three `PROT_READ`, `PROT_WRITE` and `PROT_EXEC` flags, the x86 page table system cannot support a write-only setting.

As a write-only mapping is impossible to obtain using the x86 page table, receiving primitives cannot be protected using the simple technique explained above. This concerns any primitive which actually receives some data, thus including both the functions of the `MPI_Recv` family and almost all collective primitives.

To solve this issue, the functions `MPI_Alloc_mem` and `MPI_Free_mem`, already used to allocate memory through a call to `mmap`, have been modified to produce a double mapping. One of these mappings is returned to the application which allocates the memory, the other is stored for further use. We call the buffer returned to the application the "public" buffer or simply the buffer and the buffer kept for internal use the "shadow" buffer.

When protecting communication buffers used for receive operations, the protection layer changes the permissions of the public mapping of the given buffer to `PROT_NONE` and starts the underlying communication using the shadow buffer whose permissions are unaltered. Setting the permissions of the public buffer to `PROT_NONE` satisfies the specification by catching all unwanted accesses during the communication. In the same manner, the shadow buffer allows the real MPI primitive to perform correctly. The evolution of the memory state is the same as in Figure 3.6 except for the access rights on the protected zone which are `PROT_READ` for the send primitive and `PROT_NONE` for the receive primitive. Accesses for the underlying communication are done through the shadow buffer.

Overlapping memory zones: serialization and merge While the previous mechanism is sufficient for simple send and receive operations, more issues arise when the communication buffer is shared between multiple communication primitives.

There are two different cases of communication buffer sharing. Firstly, because we transform blocking communications into nonblocking ones, a buffer may be used successively for different communications as illustrated by Listing 3.3.

In this example, the `ring` function receives a value from the previous peer of the communicator, prints it and sends it unmodified to the next peer. While printing the value forces the receive operation to be completed because it reads the value `val`, when the loop passes from one iteration to the next the sending operation is directly followed by the next receiving operation.

In general, this case we call serialization happens when the range of memory of the two communication buffers overlap at the byte level, as illustrated by the first case in Figure 3.7.

```
1 void ring(int* val) {
2   for (int i = 0; i < nb_iter; i++) {
3     MPI_Recv(val, ..., rank - 1, ...);
4     printf("rank: %d, token: %d\n", rank, *val);
5     MPI_Send(val, ..., rank + 1, ...);
6   }
7 }
```

Listing 3.3: Shared communication buffer due to unblocked communication

To avoid issues in this case, we track the precise memory location (starting address and length) which are involved in any ongoing communication. When a second operation tries to protect a memory range used in another communication, it triggers a modifiable callback (called `re_protect_handler` in the rest of the document). This function is currently set to wait for the end of the ongoing communication before starting the new one.

The second case of communication buffers sharing is less straightforward. It occurs because of the `mprotect` system call whose granularity cannot be smaller than a page frame. When a program allocates a range of memory and uses it in multiple communications for which address ranges are not overlapping, if these address ranges are not page aligned, a page frame may be shared between two different communications. This is illustrated by the first case in Figure 3.7 and is referred by Gaud et al. [19] as *page-level false sharing*.

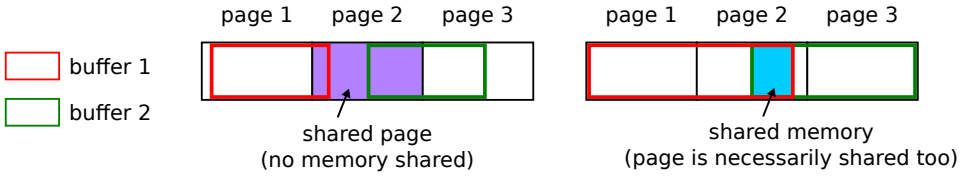


Figure 3.7: The two types of shared memory between buffers

In this case, the two communication buffers do not share any memory at the byte level but still share a page frame, making it impossible to choose two different sets of permissions for the two different communication buffers. In order to solve this conflict, the protection layer proceeds in three steps. First, it merges the two requests metadata structures by creating a dummy third one which is linked to the two conflicting ones. Second, it makes this dummy request the active one for the conflicting page frame. Finally, it changes the permissions for this page frame to the intersection of the permissions of the first and second requests. The evolution of the memory state for two requests involving a merge operation is illustrated in Figure 3.8. As for Figure 3.6, I, F, P, and W respectively stand for “Initial”, “Final”, “Protected” and “Waiting”. They are suffixed with the number of the request they refer to, P1 is a state in which the memory of first primitive is “Protected”. For any states a and b, $a1 + b2$ refers to

the combination of state a for the first primitive and state b for the second primitive.

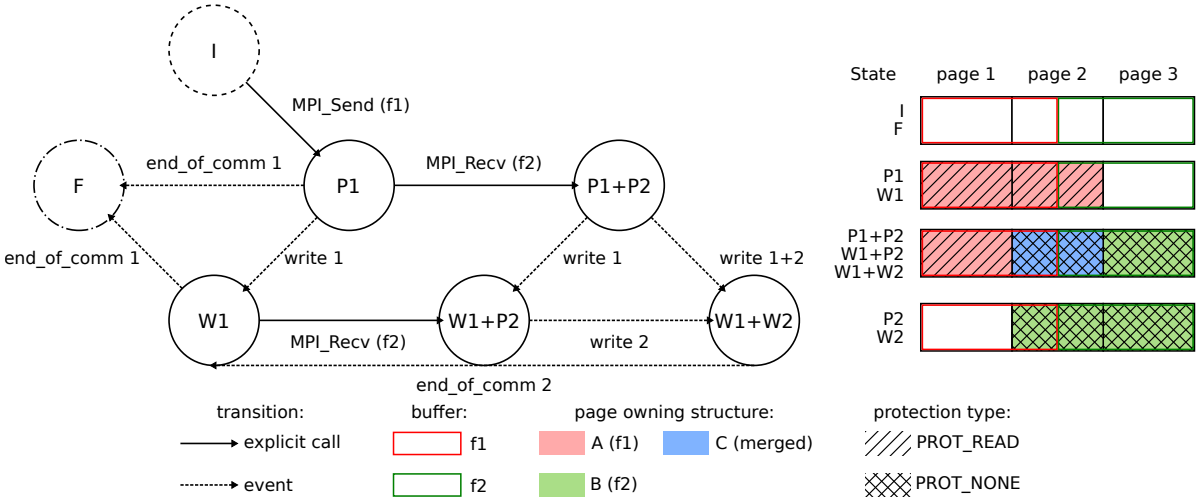


Figure 3.8: Evolution of the memory state during a merge procedure

When trying to access a memory zone in which a merged operation happened, the `access_handler` will wait for all the linked ongoing communications, thus enforcing the MPI specification. For all the zones belonging exclusively to one of the communications, the `access_handler` will only wait for the necessary primitive. Removing the protection for the given range will unlink the request metadata structure from the merged zone, essentially turning it back to a normal zone. This method has the benefit of avoiding any superfluous wait for non-shared page frames in case of an unauthorized access.

Collective communications While the different cases presented until now are focused on point-to-point communication or use point-to-point communication examples, the protection mechanism for collective communication primitives can be implemented using the same reasoning.

All collective communication primitives can be naively implemented using only point-to-point send and receive primitives. Extending the protection mechanism to include collective primitives can thus be done using a combination of the previous cases while still taking into account the specific semantics of each communication primitive. The Listing 3.4 is an example implementation of the `MPI_Gather` primitive using the protection mechanism.

For `MPI_Gather`, only the root of the operation will receive data, while all the other ranks will only send data. This explains the disjunction between normal senders which only call `protect` on the outgoing buffer and the root which protects both the outgoing buffer and the incoming buffer.

```
1 int MPI_Gather(const void* sendbuf, int sendcount, MPI_Datatype sendtype,
2                void* recvbuf, int recvcount, MPI_Datatype recvtype,
3                int root, MPI_Comm comm) {
4   const void* shadow_sendbuf = shadow_ptr(sendbuf);
5   void* shadow_recvbuf = shadow_ptr(recvbuf);
6
7   int stsize, rtsize;
8   MPI_Type_size(sendtype, &stsize);
9   MPI_Type_size(recvtype, &rtsize);
10
11  struct prot_req* send_prot =
12    protect(sendbuf, sendcount * stsize, PROT_READ);
13
14  /* mpii_infos contains info on the current call */
15  if (mpii_infos.rank == root) {
16    struct prot_req* recv_prot =
17      protect(recvbuf, recvcount * rtsize * mpii_infos.size, PROT_NONE)
18    link_pr(2, send_prot, recv_prot);
19  }
20
21  int ret = offload_MPI_Igather(shadow_sendbuf, sendcount, sendtype,
22                                shadow_recvbuf, recvcount, recvtype,
23                                root, comm, send_prot->req);
24  return ret;
25 }
```

Listing 3.4: `MPI_Gather` implementation using protection API

In the case of the root, the semantic implies that only read operations are allowed in the outgoing buffer while no accesses are allowed in the incoming buffer. This is correctly enforced by the permissions `PROT_READ` and `PROT_NONE` respectively. To ensure that any unwanted access to any buffer will trigger the `access_handler` and correctly wait until the communication is completed, the two protection metadata structures are linked (using `link_pr`). This is similar to the way merged memory regions are handled, but for the whole memory range of both communication buffers.

Finally, the offloaded version of `MPI_Igather` is called using the shadow buffers for both buffers, letting the underlying MPI primitive the freedom to read or modify buffers as needed.

The rest of the collective primitives are implemented using a similar method, by protecting the needed memory ranges and linking protection metadata structures when needed.

All the different cases presented above are handled by the protection mechanism internally. Using the `protect`, `unprotect` functions and occasionally linking multiple protection zones is sufficient to transform most blocking point-to-point and collective primitives into nonblocking ones.

The combination of the protection mechanism and the offloading library guarantees two important behaviors, every buffer on which memory protection is enabled during a call to a communication primitive will see its protection disabled eventually and any unwanted access to the memory will cause the application thread to wait as though a real call to `MPI_Wait` was issued. Therefore, the period during which communication is overlapped with computation is dynamically decided by the first forbidden access to the buffer, which in turns corresponds to the optimal limit we can afford according to the MPI specification.

3.3 Evaluation of the overlapping potential

This section presents the results of our evaluation of Commmama. All the experiments are done using the Grid'5000 [24] *grimoire* cluster. Nodes of this cluster consist of dual socket 8 core Intel Xeon E5-2630 v3 with 128 GB memory. These nodes are connected using 56 Gb/s Mellanox Technologies ConnectX-3 InfiniBand network adapters. We use GCC 8.3.0 and OpenMPI 3.1.3 [17]. All the results presented thereafter are produced using this test bed by running the mentioned benchmark 2000 times and taking a mean value.

This section is organized as follows. First we present an evaluation of the base overhead of our method. This is done by comparing our approach with a standard MPI runtime, OpenMPI, in the worst-case scenario (no computation, only communication). Then we present the results for different message sizes and different computation times. We show that our approach has a good overlap ratio, and even that when the computation time is sufficient, the communication is completely overlapped. We compare to the OpenMPI runtime, and with a good ratio of communication and computation Commmama reaches up to 73% speedup.

3.3.1 Base overhead analysis

In this section we compare our approach, Commmama, to a baseline version using only OpenMPI. This aims to study the effect of our system on base performances. This is done using a ping-pong microbenchmark with no computation to exhibit the overhead introduced by our method.

Figure 3.9 represents the round-trip latency of messages of varying sizes, both axes are represented using log scale. Depicted on this figure, the overhead incurred by Commmama is linear until important message sizes where it becomes negligible. This is mostly due to the cost of the protection subsystem and more specifically the linear cost of the `mprotect` system call.

This overhead is indeed due to the modification of the page table structure during the `mprotect` call used to enforce memory protection on the process address space. With the growth of message size, the range of memory protected by the system call grows which in turns increases in a linear fashion the number of page structures to modify. However, this overhead

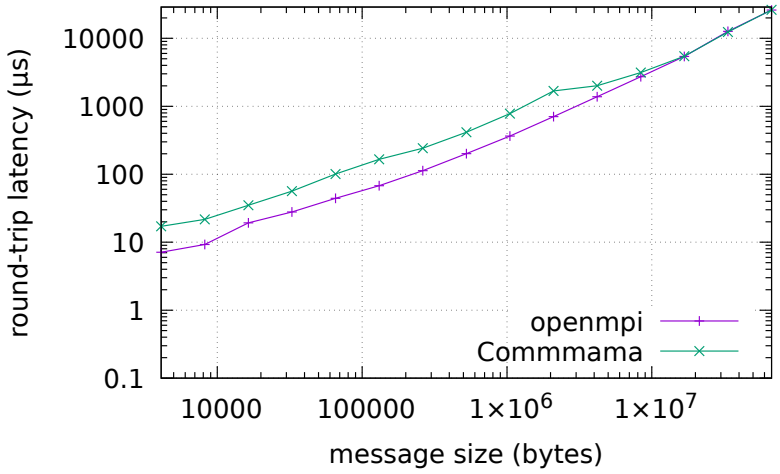


Figure 3.9: Round-trip latency with no compute

is not as important as to impact the capacity to overlap communication with computation as demonstrated in the next subsection.

3.3.2 Overlap evaluation

To evaluate the quantity of overlapped communication provided by our design, we use a modified version of the ping-pong microbenchmark with added computation time between communication phases.

This benchmark executes as follows. Process rank 0 calls `MPI_Send`, computes for a given time, writes one byte to the buffer then calls `MPI_Recv`, computes for a given time and reads one byte from the buffer. Process rank 1 does the same but starting by `MPI_Recv`, followed by the compute period and the read operation then `MPI_Send`, the compute period and the write operation.

In this second benchmark the memory operation plays an important role as it ensures the previous communication (either send or receive) is completed before continuing. This simulates the first access to buffer data in the computation period following a communication.

The expected execution timelines are presented in Figure 3.10.

The results presented thereafter use this benchmark with a varying amount of computation time, respectively 2×200, 2×2000 and 2×20000 microseconds. The *openmpi* curve represents the default OpenMPI behavior. The *Commmama* curve represents the behavior of our approach on top of the same OpenMPI runtime. The *optimal completion time* curve represents the theoretical optimum which is $\max(t_{compute}, t_{comm})$, where $t_{compute}$ is the total

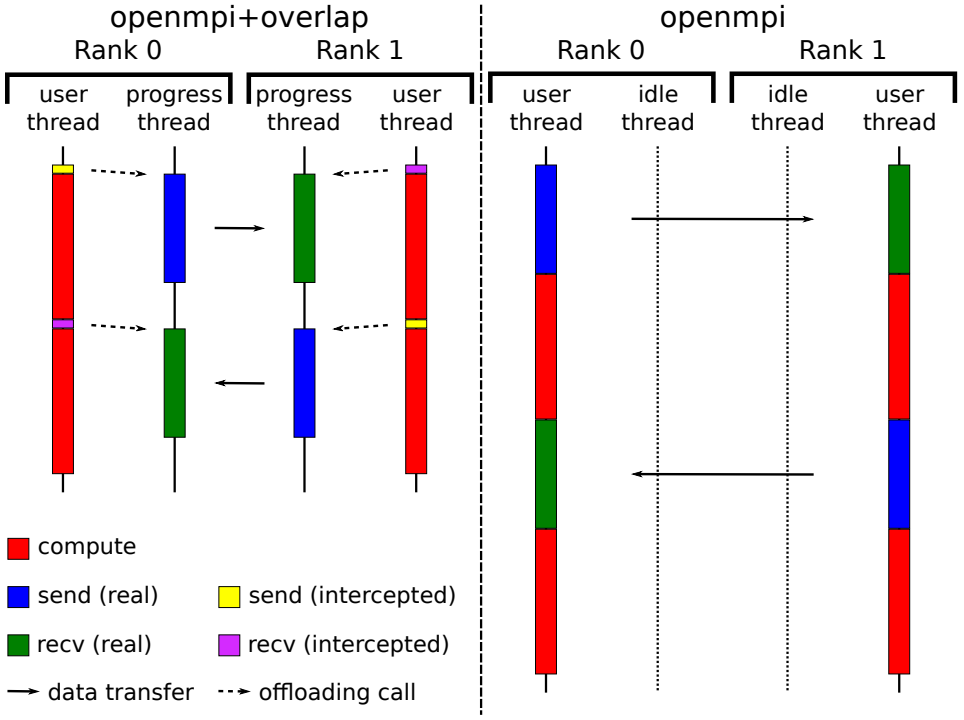


Figure 3.10: Expected execution timeline for microbenchmark

time of computation (i.e. the sum of both occurrences for a given process) and t_{comm} is the time of completion of the communication for the default OpenMPI implementation taken from Section 3.3.1.

Round-trip latencies in Figure 3.13, 3.11 and 3.12 all demonstrate two different behaviors. Between 4 KB and a certain value called $s_{threshold}$ below, *Commmama* remains very close of the optimal completion time, while *openmpi* is equal to $t_{compute} + t_{comm}$. Finally, between $s_{threshold}$ and 64 MB, both *openmpi* and *Commmama* increase gradually, with *Commmama* staying close to the optimal value.

The default OpenMPI implementation cannot overlap any communication with computation, resulting in an increased completion time. This is particularly visible when the communication time and computation time are close. For example, on Figure 3.11 for 16 MB message size, the raw measurement for *openmpi* gives a latency of 5419 μs which is close to $t_{compute} = 4000$ μs for this experiment. The measured time for Figure 3.11 at 16 MB is 9404 μs, almost exactly $t_{compute} + t_{comm}$.

On the contrary, Commmama performs well, showing a high overlapping capability. The raw measurement for *Commmama* is 5446 μs at 16 MB on Figure 3.11, only 30 μs more

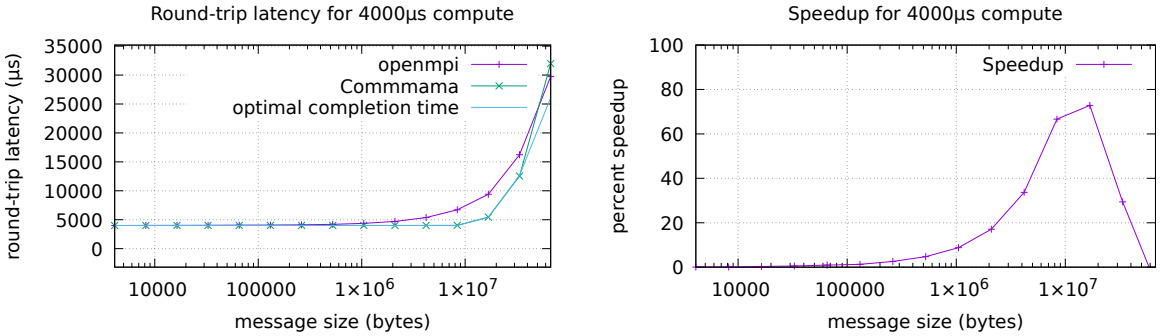


Figure 3.11: Round-trip latency and speedup for $t_{compute} = 4000$ μs

than native *openmpi*. With 4000 μs computation time, Commmama overlaps communication perfectly, with a measured time of 5440 μs. This second behavior persists until message size attains $s_{threshold}$, the message size until which $t_{comm} < t_{compute}$, which for this example is for message sizes between 8 MB and 16 MB.

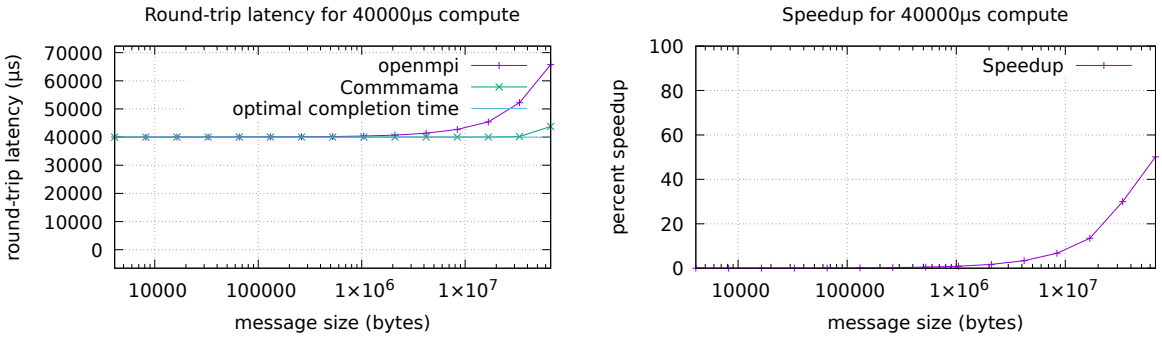


Figure 3.12: Round-trip latency and speedup for $t_{compute} = 40000$ μs

As for the second phase, when message size exceeds $s_{threshold}$, both *openmpi* and *Commmama* round-trip latencies start increasing, with *Commmama* still being close to the optimal completion time value which is now t_{comm} as exceeding $s_{threshold}$, t_{comm} has become the longest task required for completion. This behavior is better seen in Figure 3.12 for message sizes bigger than 2 MB. On this part of the curve, *openmpi* completion time gradually increases until reaching 65 ms while *Commmama* is close to $t_{compute}$ with only 40.185 ms for 32 MB and 43.800 ms for 64 MB.

While our proposal is less efficient for small computation times because the window for

overlapping communication and computation is smaller and thus closer to the potential overhead of our approach, Figure 3.13 still demonstrates good results, especially the speedup representation which reaches 10% for 128 KB and up to 42% for 512 KB.

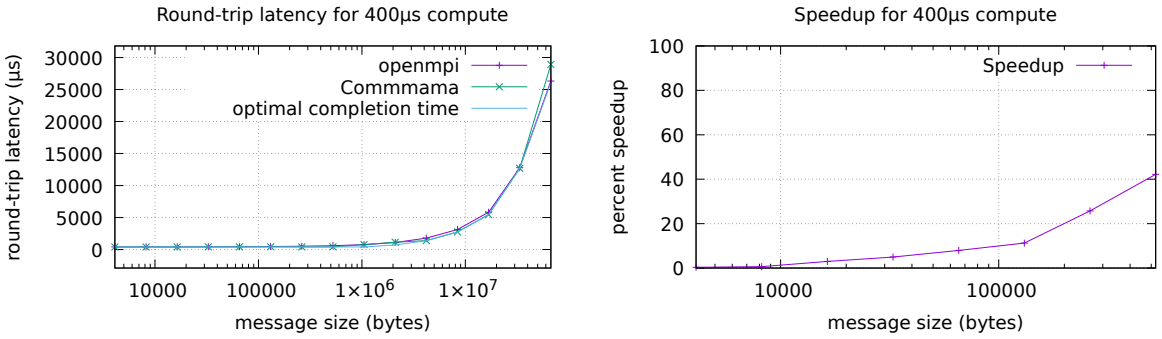


Figure 3.13: Round-trip latency and speedup for $t_{compute} = 400$ μs

Favoring communication and computation overlap, our approach can really shine when there is enough computation to overlap communication with. As shown by the speedup plot in Figure 3.11 in which speedup reaches 73% and is good in any case when $t_{compute}$ is slightly bigger than t_{comm}. For small communications, with a small compute time of 400 microseconds, speedup attains 42%, with 4 milliseconds of compute time, speedup attains its best value for medium-sized communications, 73% for 1 MB messages. After this point the speedup decreases due to the compute time becoming insufficient to correctly overlap communication for bigger messages. With 40 milliseconds of compute time, big communications are at their best with up to 50% speedup.

3.4 Conclusion

This chapter presents the challenges of overlapping network communication with CPU-bound communication in the scope of MPI. The MPI specification provides two types of primitives. Blocking primitives, which are simple to use but cannot overlap communication with computation, and nonblocking ones, which are more complex to use but allow some communication to be overlapped.

In this chapter we present the different types of primitives, their semantics, and explain which of these primitives can overlap communication with computation. We show that two key factors are important to reach efficient overlapping of communications, the window during which communication and computation are overlapped and the presence of a background progress mechanism. Works aiming to improve one of these factors are presented.

We then introduce Commmama which transforms blocking communications into nonblocking ones at runtime. Using this approach, Commmama combines the simplicity of blocking communications and the efficiency of nonblocking communications while enforcing the MPI specification. In order to enforce the MPI specification for communication primitives, Commmama protects the memory used by the application to store data sent or received with MPI. This allows to detect unwanted accesses and force the application to wait if needed. This is described in detail in the section related to the internal architecture of Commmama and its three layers: interception layer, offload layer, and protection layer. This architecture enables Commmama to be used easily by developers. As an intermediate layer, Commmama can be used without modifying legacy software and with most MPI runtimes.

Finally, this chapter presents an evaluation of Commmama's ability to overlap communication with computation. This evaluation shows that Commmama is able to reach good speedup (up to 73%) when computation time is sufficient to overlap communication.

To conclude, Commmama, increases network and CPU parallelism using memory protection as a layer between the MPI runtime and the application. In contrast to this user space solution, Scalevisor, the second part of our approach, manages resources at a lower level, below the operating system using virtualization techniques.

Chapter 4

A multicore resource management driver

As explained in section 2.1, computer architectures have evolved through the years and continue to do so. Because taking into account the topology is necessary to use a machine full potential, operating systems have to evolve and include complex rules each pertaining to different hardware configurations. Thus, each new topology creates its own specific needs in term of scheduling, memory management and I/O management.

Resource management in modern operating systems is often treated by having a piece of the system manage one specific resource. These subsystems are mostly isolated parts of the operating system, like the scheduler or memory manager, which do not communicate to take decisions about the optimal state of the system. More often that not, decisions are taken by considering the distribution of other resources as fixed or even to really make a specific resource static while having the other adjusted regularly.

While this method of resource management has been studied (some examples are presented in section 2.3) and can provide good results in simple cases, with the added complexity of new architectures and the multiplicity of applications, a new approach to resource management treating the system as a whole is needed.

Current operating systems were not designed to use this global approach, moreover, to accommodate the large number of applications, they include generic heuristics which are not tailored to provide a sufficient gain in performance. Because of the size of the code base and design of current operating systems, it seems impossible to implement new efficient policies and maintain them in multiple systems for multiple hardware configurations.

Given the advance of virtualization technology, especially in the field of hardware virtualization, we propose a new approach to resource management in complex topologies that uses virtualization to add a small layer dedicated to resource placement in the usual software stack. This intermediate layer, Scalevisor, placed between the hardware and the operating system,

manages resources placement and simulates a flat topology for the overlying operating system. This flat topology forces the overlying operating system to behave as if it were running on a simple topology and thus ignore the real topology when managing resources. Figure 4.1 represents a software stack with Scalevisor integrated in between the real topology and the operating system.

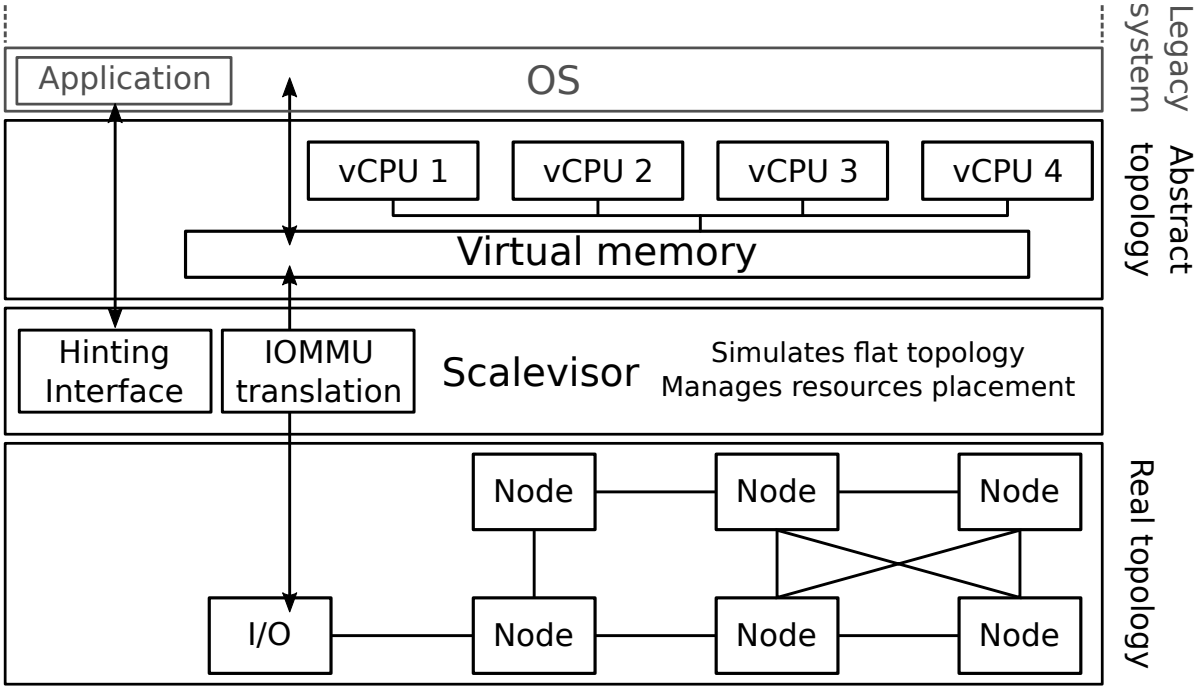


Figure 4.1: Software stack including Scalevisor

While Scalevisor primary purpose is to act as a black-box resource management system below the operating system, its design includes an hinting interface accessible to the guest and applications through paravirtualization. This interface can be used by applications to describe their operating patterns and thus make placement decisions more efficient.

The first step toward this multi-resources placement driver is to simulate an abstract memory and CPU topology using virtualization techniques. This chapter is organized as follows. Section 4.1 presents both software and hardware virtualization techniques, and describes overhead created by using such techniques. This section then presents related work on virtualization and more specifically about the issues of using virtualization on NUMA architectures. Section 4.2 presents Scalevisor, starting from its design and how to mask the underlying topology, then describes Scalevisor internal structure and components. Finally, this section presents the unrealized parts of the project due to a lack of time. While Scalevisor is able to boot a

Linux system, some parts are not mature enough to evaluate heuristics. We thus conclude with some reflections on the task of writing a new operating system given the complexity of current hardware.

4.1 Background and motivation

Virtualization consists in providing virtual resources to an operating system, most of the time in order to allow multiple systems to coexist on a single hardware unit. Virtual resources are created using different methods, currently for performance reasons, a lot of virtualized resources are created and managed using hardware-assisted techniques.

Real machine are mainly composed of compute units, memory and devices. In the same way, these constitute the main components of a virtualized machine, thus, the parts that need to be virtually duplicated in such a system are the CPUs, the page table responsible for memory mapping, and the device drivers.

This section first describes currently available virtualization techniques, then presents related work. It is organized as follows. Sections 4.1.1 and 4.1.2 present software techniques and hardware assisted techniques respectively. Section 4.1.3 details related work on virtualization, first presenting generic virtualization issues, then specific problems arising from the use of virtualization on NUMA architectures.

4.1.1 Software techniques

The concept of virtualization includes several notions but the property expected of a reliable Virtual Machine Manager (VMM) are threefold. First, the virtual machine should be functionally equivalent to the hardware machine as far as the guest operating system is concerned. Secondly, the guest should be isolated from the rest of the system, either from other virtual machines or the VMM itself. Finally, performances should be mostly equivalent to running directly on the hardware when inside the virtual machine.

According to Popek et al. [44], "For any conventional third generation computer, a virtual machine monitor may be constructed if the set of sensitive instructions for that computer is a subset of the set of privileged instructions". Sensitive instructions are divided in two categories, *behavior sensitive* instructions and *control sensitive* instructions. A *behavior sensitive* instruction is one that depends on the state of the processor (for example being in user or system mode, or the real memory location). A *control sensitive* instruction is one that modifies the processor state (changing from user mode to system mode).

Sensitive instructions need to be privileged to allow the VMM to catch any attempt from the guest software either to change the real state of the hardware to an unwanted state for *control sensitive* instructions or to react differently because of virtualization in the case of *behavior sensitive* instructions. This implies that to correctly virtualize the CPU behavior,

sensitive instructions must be intercepted and emulated by the VMM when executed by the guest operating system.

As detailed in Robin et al. [46], seventeen sensitive instructions in the x86 Intel Pentium CPU are not privileged, including `PUSHF`, `POPF`, some variations of `CALL`, `JMP` or `RET`. To overcome this issue, the guest code must not be executed directly on the hardware CPU but rather emulated, using binary translation to create an alternate version of the code and run this version on the CPU. This approach is the main software technique used to virtualize CPU as explained by Bugnion et al. [8], but unfortunately, introduces obvious performance issues when compared to real hardware execution.

While emulating the CPU behavior is necessary to virtualize a system, emulating the behavior of sensitive instructions is not enough to correctly isolate the guest system from the host, the guest must also execute in an independent memory space. Most modern operating systems use paging as their memory abstraction, this mechanism is explained in section 2.2.1. Hence, the guest operating system already uses its own page table to translate from guest virtual addresses to guest physical addresses as it would in a native environment. When guest code is executed and memory accesses occur the hardware MMU will translate using the page table currently loaded in the `cr3` register. For the page table to actually translate to the good machine address, the VMM must provide an alternate page table which contains mapping from the guest virtual space directly to machine addresses. This alternate page table is called a shadow page table as the VMM is responsible from transforming the mappings created by the guest in its page table to meaningful mappings when the code is executed.

The issue with such a technique is the recurring need for the VMM to modify the page table. To this end, the guest page table structures are marked read-only by the host which incurs a strong performance penalty forcing a context switch every time the guest page table is modified.

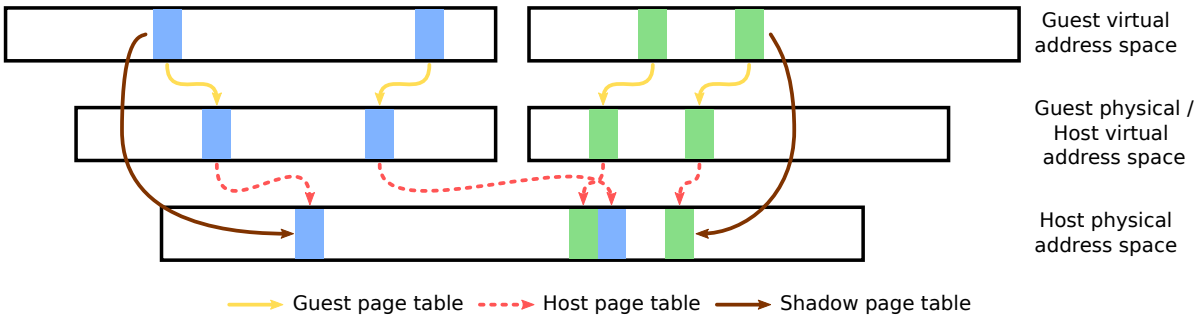


Figure 4.2: Shadow Page table

Finally, while the management of some devices can be left to the guest operating system, this is impossible when they are also needed by the host operating system and thus need to be

shared. Emulating devices is done differently depending on the type of device. For example, when the communication medium with the device are memory mapped I/O (MMIO), the VMM needs to create a read-only mapping in place of the device real I/O zone to generate a page fault when the device is used and take action depending of the guest input. This kind of emulation is expansive because it causes the VMM to regain control at each interaction with the device and cannot be avoided in general.

While different techniques either for CPU, memory or device emulation exists and allows a real software virtualization hypervisor to be achieved, each of them is quite costly in term of performance, and when combined can incur significant overhead.

4.1.2 Hardware techniques

As virtualization became ubiquitous, the need for faster, safer, virtualization techniques was tackled by introducing hardware features directly targeted at system virtualization.

The two big manufacturers of x86_64 CPUs, Intel and AMD both have added some hardware virtualization capabilities to their CPUs. This new hardware was originally presented by Intel [56] in 2005, and continues to evolve. Both Intel's and AMD's versions enhance the common instruction set with virtualization related instructions and a virtual machine control structure stored in memory. In Intel's case the control structure is called Virtual Machine Control data Structure (VMCS).

CPU virtualization: Intel VT-x and VMCS The VT-x extension adds the aforementioned VMCS control structure which occupies a four-kilobyte aligned memory page in physical memory. The pointer to this memory zone is used to reference a given VMCS for any subsequent operation on this structure.

There is a direct correspondence between the concept of virtual CPU (vCPU) and the VMCS structure which defines the behavior of a given vCPU. As such, it is possible and even required to have multiple VMCS to emulate multiple vCPUs but at any given time, only one VMCS can be the "current" VMCS for a given physical CPU (pCPU).

To interact with the virtualization capabilities and the VMCS, the Intel VT-x extension provides a number of new instructions. The first set of instructions is used to change which VMCS is the current one on the processor executing the instruction as well as make a VMCS active for a given processor. This first set of instructions contains:

- `VMCLEAR`: Reset the VMCS to an inactive, not current, not launched state
- `VMPTRLD`: Make a VMCS current and active for a given processor
- `VMLAUNCH`: Make the current VMCS launched
- `VMPTRST`: Return the pointer to the current VMCS

An "active" VMCS is one which is already attached to a given CPU, this means it can be loaded again with `VMPTRLD` on the same processor but not on another. A "launched" VMCS is one which is running guest code and cannot be launched again before being cleared. The different states and transitions are represented in figure 4.3.

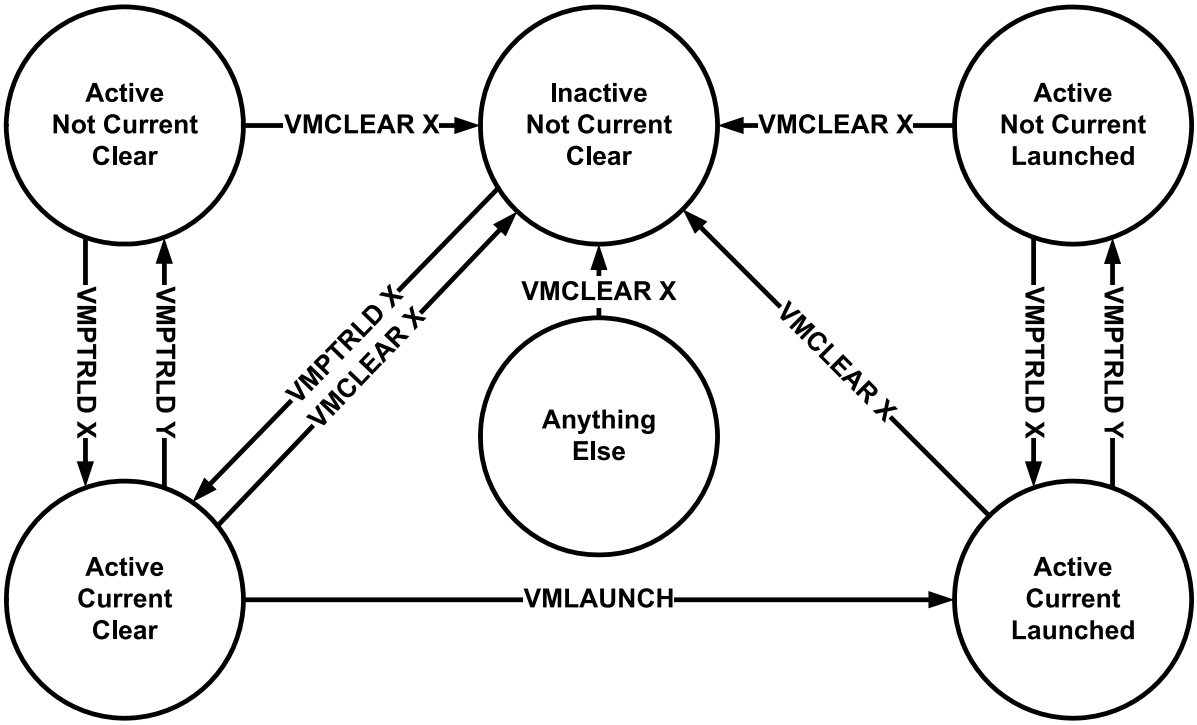


Figure 4.3: VMCS state machine (Intel documentation [28, chap. 24.1])

When a VMCS is successfully loaded in the processor using `VMPTRLD`, it can then be interacted with using the second set of instructions:

- `VMREAD`: Read a field from the VMCS
- `VMWRITE`: Write some data in a given VMCS field

While the VMCS is a structure stored in memory, it cannot be read or written through direct memory accesses for two different reasons. Firstly, the format of the VMCS is implementation specific and thus the real layout of the fields is not disclosed by the Intel documentation. Secondly, a loaded VMCS is partly cached inside its related pCPU which means a direct memory access would neither make sure that the content of the processor cache was flushed back

to memory in case of a read nor the current data in the processor cache was updated correctly after a write.

When a VMCS is launched using the `VMLAUNCH` instruction it enters guest mode, called VMX non-root mode for Intel in opposition to host mode which is called VMX root mode, the context in which is executed the hypervisor or Virtual Machine Manager (VMM) code. Entering VMX non-root mode is commonly called a VMEnter event while existing this mode and thus going back to VMX root mode is called a VMExit. The VMCS contains numerous configuration options, some of which can trigger a conditional VMExit, for example when a privileged register is accessed or a privileged instruction is used. Two instructions of the new VT-x extension interact with the VMCS when already launched:

- `VMCALL`: Trigger a VMExit (called by guest)
- `VMRESUME`: Return from a VMExit to VMX non-root mode (called by VMM)

The `VMCALL` instruction can be called when operating in VMX non-root mode to return control to the VMM directly, this is called mostly for paravirtualization to let the guest communicate with the hypervisor in the same manner as a process in a standard operating system would call a system call.

The VMCS and new instructions are the hardware mechanisms needed to virtualize the computation component of a system, the CPU. This solution avoids the need for binary translation mentioned in section 4.1.1 and reduces the overhead of virtualization as shown in section 4.1.3.

Second level page table: Intel EPT The shadow page table technique, while allowing a VMM to emulate a second level paging system introduces overhead. Mirroring the physical CPU page table to manage memory mappings, vCPUs obtained through the use of hardware assisted virtualization, introduce a hardware solution to second level paging.

As explained in section 2.2.1, for physical CPUs, the page table physical memory location is stored inside the specific `cr3` register. This register has its own equivalent in the VMCS called the Extended Page Table Pointer (EPTP). This pointer contains the address to a second level page table used to emulate the behavior of the normal page table when in VMX non-root (guest) mode.

This second level page table is called Extended Page Table on Intel processors and is used to translate guest physical addresses, equivalent to host virtual addresses, to host physical addresses which are real hardware addresses.

The EPT system acts in place of the shadow page table doing the translation and allows for guest physical addresses translation to be isolated from the VMM page table. On top of this virtualized paging system, the guest is free to use any memory management mechanism, either segmentation which is common for compatibility reasons when booting, or paging with its own

page table stored in the VMCS `cr3` register and acting as normal when in VMX non-root mode.

The EPT achieves an important performance gain by replacing the shadow page table in which every page fault caused by the guest would require a VMExit followed by the VMM actually doing the mapping, actualizing the shadow table and returning control to the guest. Evaluation of Intel EPT in the VMware platform has shown up to 48% gains for MMU-intensive benchmarks and up to 600% for MMU-intensive microbenchmarks [5].

However, this improvement is not free either, because of the nature of the page table system. A classic page table uses bits of the virtual address to navigate in the different levels of tables as explained in section 2.2.1. These tables are stored in physical memory and thus each level down the tree is accessed by getting its physical address from the upper level. With second level paging, the address stored in the guest page table is in fact a guest physical address which needs to be translated through the EPT.

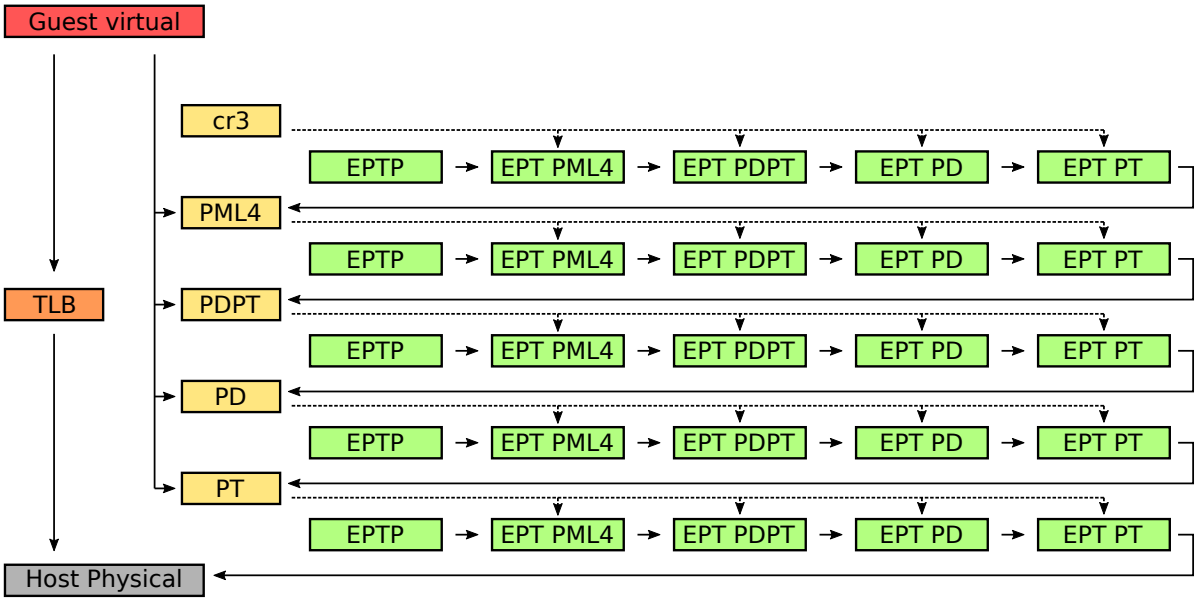


Figure 4.4: Second level translation overhead

As shown in figure 4.4, because the address stored in the guest page table needs to be translated using the EPT, translation from virtual (guest virtual) to real memory increases from following 4 pointers in the native page table to a combined 24 pointers in the EPT and guest page table.

Behavior of the TLB when using second level translation Fortunately, the second level translation system benefits from the TLB described in section 2.2.1 in the similar manner as a regular MMU. When EPT is enabled, the TLB caches two kinds of information, guest-physical mappings and combined mappings.

Guest-physical mappings are derived from the EPT paging structures, they include guest-physical translation, mapping guest-physical addresses to memory frames, and guest-physical paging-structure-cache entries, which link a part of the guest-physical address to an intermediate paging structures in the EPT. Combined mappings are derived from both the EPT paging structures and the guest page table located in the `cr3` register when EPT and paging are both enabled. These combined mappings contain cached translations, mapping guest linear addresses to physical addresses, and combined paging-structure-cache entries which link the upper portion of the linear address to the physical address of the corresponding guest page table paging structure.

When EPT and paging for the guest are enabled, combined mappings greatly increase the efficiency of the translation process by giving a direct guest linear address to host physical address translation. Moreover, the combined paging-structure-cache avoid going through the EPT table to retrieve the host physical address of the guest paging structure reducing the overhead described in figure 4.4, caused by using EPT and paging simultaneously.

As for non-virtualization behavior, when using virtualization extensions, the TLB must be invalidated when mappings become invalid. In section 2.2.1, the context transition between guest and host is mentioned as a reason of invalidation. While this is the case with the first version of VT-x included in older CPUs, it resulted in poor performances because even when using hardware virtualization, switching between VMX root and VMX non-root mode is quite frequent.

To avoid this issue, in modern CPUs, a VPID feature similar to PCID was introduced. When VPID support is activated, TLB entries are tagged using a 16-bit identifier stored in the VMCS and operations that would invalidate the TLB caches will only do so for the current VPID. VMX transitions no longer invalidate neither guest-physical mappings nor linear mappings. Two instructions, `INVVPID` and `INVEPT` are used to invalidate TLB cached entries at a fine grain, mirroring the behavior of `INVLPG` for the native page table.

The caching mechanism provided by the TLB is really important to offset the cost of repeated translations through the combination of the EPT and guest page table. Combined mappings, which directly map a guest virtual address to a host physical address allow a one hop translation, on par with TLB performances in a native environment.

Managing devices using the IOMMU: Intel VT-d While the Intel EPT provides second level translation for CPU memory accesses, the devices which are using memory to read or write data are not concerned by this behavior. In order to provide the same advantages to devices, a similar system named IOMMU is present in modern hardware.

An important number of devices use memory mapped input/output (MMIO) to communicate with the rest of the system, either to share data or for configuration. In the context of a native system, these devices simply access the physical memory, but in a virtualized environment, the guest physical address space is already built on top of a translation mechanism, the EPT. In this case, when the device tries to access memory, it needs to access the right guest physical address (not a real physical one) and for isolation reasons, its access needs to be limited to addresses accessible to the virtual machine.

The solution to this situation is to introduce a step of translation for memory accesses initiated by devices, called DMA (Direct Memory Access) remapping. In Intel VT-d system, this is done using a tree hierarchy to associate a translation structure similar to a page table for each address of the PCI bus on which devices are connected. The root table contains 255 entries, each one referencing a bus ID, and each entry contains the location to a context table with 255 entries corresponding to the 255 device addresses of a given bus. The IOMMU thus associates for each unique PCI device address (bus number and device number) a set of translation structures for the device. This tree hierarchy is depicted in figure 4.5

This answers the two previous needs, translation is done for a given device according to the translation structure associated with its ID and as a device is only given access to host physical addresses reachable through these paging structure, a device with non-present or limited translation structures will not be able to access unwanted zone of the hardware memory.

AMD-v, NPT, AMD-vi: different names same concepts As the second contender in the x86 CPU world, AMD CPUs also contain virtualization enhancing hardware. While not identical, these hardware features are very similar.

AMD-v, the VT-x equivalent, uses a similar structure in place of Intel's VMCS, the VMCB. In opposition to Intel's approach using dedicated instructions to interact with the content of the VMCS, the VMCB is a memory mapped structure accessible using classic memory accesses, other functionalities are similar.

AMD CPUs also include a second level translation system, named Nested Page Table (NPT) which is very similar to EPT, both being directly inspired by the standard x86 page table structure.

Finally, AMD-vi is the name of AMD's IOMMU, which while configured differently has an identical purpose of allowing devices to interact with memory while using second level translation.

Both software and hardware techniques presented previously are the cornerstone of virtualization as they are needed to simulate an abstract memory and CPU topology for the guest operating system. This abstract topology guarantees the isolation property and allows the hypervisor to manage resources independently of the overlying guest operating system.

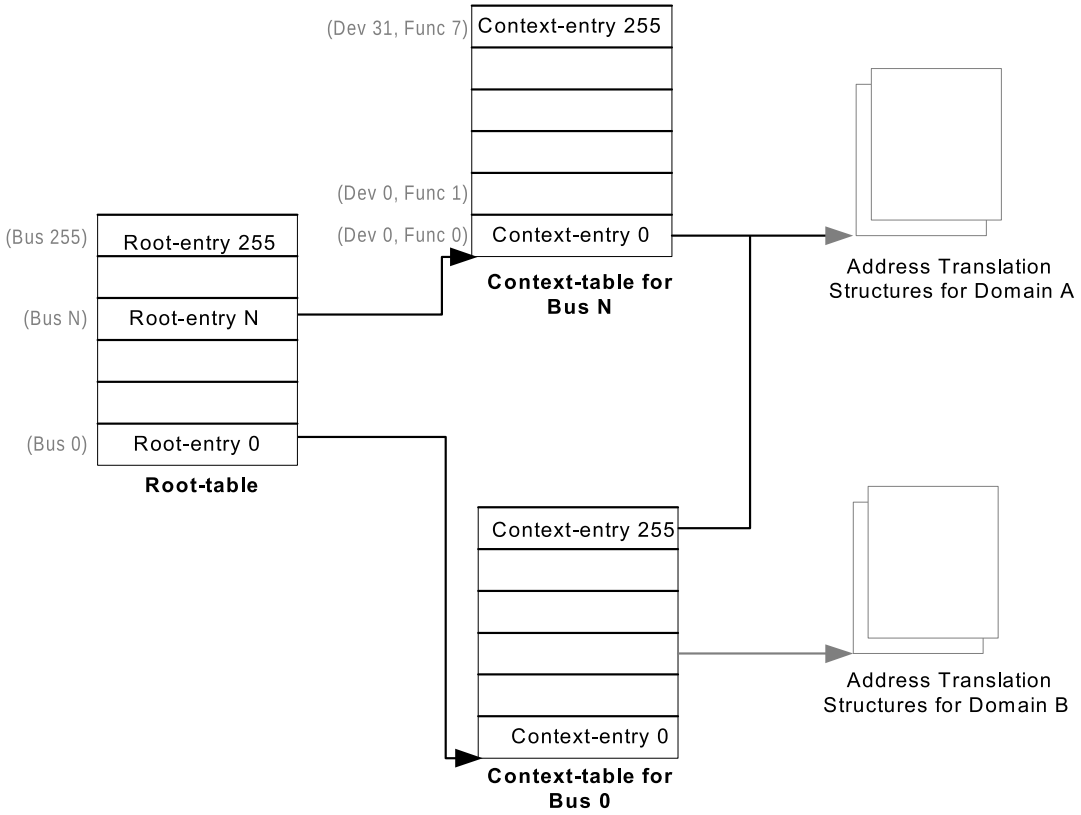


Figure 4.5: Intel VT-d tree hierarchy (Intel documentation [29, chap. 3.4.3])

4.1.3 State of the art: Virtualization

Virtualization techniques presented in previous sections are used by the VMM to manage resources. However, both software and hardware virtualization introduce some overhead which can lead to bad performances, especially when wrongly used.

This section presents work related to virtualization in general and more specifically on the interaction of virtualization and NUMA architectures.

Virtualization overhead and common issues

This section describes work related to common issues in virtualized environments, these include virtualization of interrupts and the Lock Holder Preemption (LHP) or Lock Waiter Preemption (LWP) issues among others. The solution described in these studies use techniques that can be classified in three categories. First, *hardware assisted* virtualization which leverages

existing hardware mechanisms. Second, *paravirtualization* when isolation is relaxed to allow interactions, in any direction, between the VMM and guest OS. Third, *transparency* when isolation is relaxed because the VMM shares information with the guest OS. Approaches are further detailed below, and summed up in table 4.1.

Table 4.1: Related work on virtualization

Proposal	Issues	Approach
Ganesan et al. [18]	–	–
DID	IPI	hardware assisted
Kilic et al. [30]	IPI	paravirtualization
vScale	LHP	paravirtualization
I-Spinlock	LHP + LWP	transparency

Empirical study of performance benefits of hardware assisted virtualization Applications or operating systems performances can suffer from virtualization as it introduces its own overhead. However, as explained in section 4.1.2, some hardware mechanisms were introduced in recent machines to decrease this overhead. Ganesan et al. [18] study the impact of hardware assisted virtualization on performances.

Because the VMM has to conserve the control on hardware resources, it needs to remove privileges from the guest operating system. Before hardware assisted virtualization was introduced, CPU virtualization used software techniques such as binary translation. Ganesan et al. evaluate the overhead of virtualization measuring three different metrics, CPU utilization, throughput and execution time in three different execution contexts, native (no virtualization), virtualization with no hardware assistance, and hardware assisted virtualization.

For CPU bound applications, they conclude that with only one vCPU and one instance of the benchmark the overhead is similar for both software virtualization and hardware virtualization with a 6% decrease in throughput. However, when scaling the number of vCPUs, they conclude that software virtualization speedup is 0.73 compared to the native linear speedup while virtualization speedup is 0.96, nearly linear.

For network bound applications, they measure the increase in CPU utilization and conclude that while both types of virtualization increase CPU utilization, hardware assisted virtualization is far superior with an increase of 18% compared to software virtualization at 110%. They also show that with hardware assisted network virtualization (SR-IOV) throughput is near native.

Finally, they measure the impact of the IOMMU on disk I/O performances, and show that while software virtualization presents significant I/O contention due to disk accesses, hardware assisted virtualization measured contention is near native, instead inducing an increase in CPU utilization.

While this work does not in itself address any issue, it shows that hardware assisted virtualization really reduces the overhead of virtualization for CPU and memory but not so much for I/Os. The overhead of I/O operations being partially due to the overhead of managing interrupts in a virtualized context.

A Comprehensive Implementation and Evaluation of Direct Interrupt Delivery – DID As demonstrated by Ganesan et al., the overhead introduced by virtualization has been largely mitigated by hardware assisted virtualization except for I/Os. Tu et al. [55] present a method, called *DID*, for reducing the cost of I/O virtualization by managing interrupt delivery to the guest vCPUs without involving the VMM. They identify two challenges, directly delivering interrupt to the guest OS and signal successful delivery to the controller hardware without involving the hypervisor in both cases.

To this end, Tu et al. leverage the interrupt remapping capabilities of the IOMMU. They send interrupts directly to the vCPU when its target VM is running else to the hypervisor which will queue incoming interrupts and deliver them as virtual interrupts when the vCPU is scheduled again. They also manage timer interrupts and migrate them from core to core to follow the migration of vCPUs. To avoid the overhead of Local APIC emulation, they use the hardware assisted virtualization feature known as APICv. By using both the IOMMU and the APICv, they avoid two causes of VMExit, those caused by devices sending interrupts or caused by CPU receiving interrupts targeted at their currently running vCPU.

Using this method, they reduce the latency of interrupt invocation from 14 μs to 2.9 μs, on their Memcached benchmark, VMExit rate is reduced from 97 k per second to less than 1 k per second.

Tu et al. tackle the issue of virtualizing inter-processor interrupts using hardware assisted features added in newer generation of CPUs, reducing the number of VMExit and thus the overhead of virtualization, especially I/O virtualization.

Overcoming Virtualization Overheads for Large-vCPU Virtual Machines As studied by Tu et al., interrupt virtualization is the cause of an important part of virtualization overhead. Kilic et al. [30] tackle this issue in the specific setting of virtual machines with more vCPUs than physically available cores.

Inter-processor Interrupts (IPIs) are a type of interrupt commonly used in operating systems to let a given CPU trigger the scheduling of some specific task on another CPU. For example, IPIs are used in Linux to flush the TLB caches of others CPU when changing mappings (TLB shootdowns). However, while processing IPIs in a non-virtualized context is fast, when running inside a VM, IPIs between vCPUs must be emulated which is slower. This overhead scales with the number of vCPUs of a VM.

Kilic et al. also treat the issue of double scheduling, which arises because the VMM scheduler and the guest OS scheduler are unaware of each other decisions and can thus lead to a vCPU

migration canceling the scheduling choice of the guest OS for a thread.

In order to solve these issues, Kilic et al. propose a paravirtualization approach in two steps. First, they dedicate a vCPU per thread, pinning the thread in the guest OS (using paravirtualization), this solves the double scheduling issue as well as reduces the number of IPIs used by the kernel to implement work stealing. Second, they replace IPIs used in TLB shootdowns by a hypercall which instructs the VMM to flush the TLB in place of the guest OS. Using these methods, they demonstrate that running a virtual machine with 255 vCPUs on a machine comprising 6 pCPUs adds little to no overhead compared to running one vCPU per pCPU.

In the specific case of virtual machines with an important ratio of vCPU per pCPU, the virtualization of interrupts takes another dimension, Kilic et al. propose a paravirtualization approach to reduce the number of such interrupts.

vScale: automatic and efficient processor scaling for SMP virtual machines To protect critical sections with shared data in the code, operating systems use synchronization primitives, such as spinlocks. A spinlock is a busy-waiting primitive, meaning the thread which does not acquire the lock (the waiter) waits on a loop checking the lock state.

In a native Linux system, both the lock holder and waiter cannot be preempted to avoid blocking the system. If the lock holder was preempted, it could not release the lock and thus would make every waiting thread waste CPU cycles for a long duration. However, as mentioned earlier, the VMM vCPU scheduler and the guest OS scheduler are unaware of each other. Thus, in a virtualized environment, vCPUs can be preempted by the VMM scheduler when holding the lock, this is known as the *Lock Holder Preemption* problem.

According to Cheng et al. [11], the Lock Holder Preemption issue is caused by the operating system not knowing the real share of compute power it possesses. Because all virtual machines on a server share the CPUs, and the number of vCPU of a given VM is fixed, this makes a VM's pCPU allocation dependent of the other VMs' CPU consumption.

Cheng et al. thus propose to make the guest operating system use a varying number of vCPUs which represent its real consumption. To the contrary of standard hypervisors changing the time share of vCPUs to increase or decrease available compute time for a VM, they keep constant time slices. They provide a kernel module to force the guest operating system to reschedule threads when changing the number of vCPUs for a given VM, thus avoiding a vCPU to be preempted while holding a lock.

The Lock Holder preemption issue is caused by the opacity introduced by virtualization. Cheng et al. use paravirtualization to coordinate changes in the amount of computing power available for a virtual machine with its guest scheduling.

The lock holder and the lock waiter preemption problems: nip them in the bud using informed spinlocks (I-Spinlock) Spinlocks can be implemented using a number

of methods. One of them, called ticket-lock consists of giving an always increasing value, the "ticket" to the thread trying to acquire the lock and give the lock to the one holding the smallest value. When the holder releases the lock, the current allowed value is bumped by one, allowing the next in line to hold the lock.

While this implementation has the advantage of being fair, it can lead to a slight variation of the Lock Holder Preemption problem discussed earlier. When a thread tries to acquire the lock, this thread obtains a ticket. If this ticket does not hold the smallest value but is preempted, all ticket holders with greater values will be forced to wait for this thread to acquire and release the lock before getting their turns. This create a *Lock Waiter Preemption* problem in which the previous lock holder has released the lock but the next in line blocks the whole lock by being preempted.

Taebe et al. [50] propose an implementation of the spinlock called *Informed Spinlock* (I-Spinlock) which only allows a thread to acquire a ticket if its time-to-preemption is sufficient to wait for its turn, enter, and leave the critical section. In order to implement this solution, they propose to share the remaining vCPU scheduling quantum with the guest operating system. When trying to acquire the lock, a thread first compares the remaining time slice of its vCPU with the time required to wait and execute the critical section.

This approach uses transparency to inform the guest operating system of the remaining quantum for a given vCPU, avoiding spinlocks inside the guest to suffer from both Lock Holder Preemption or Lock Waiter Preemption issues.

Studies presented before try to solve commonly seen issues in virtualized environment such as virtualization of interrupts, or the lock holder/waiter problems, which are specific cases of double scheduling. Presented solutions most of the time reduce the opacity between the hypervisor and the guest operating system, either by sharing some information or even introducing some form of communication through paravirtualization.

Virtualization on NUMA architectures

While virtualization inherently creates issues, combining a virtualized environment on top of a NUMA architecture introduces different issues, closer to those detailed in section 2.1.2. Most of the time, the studied problem arises either from contention on different resources, LLC, memory controller, interconnect, or bad locality of accesses. Because of the opacity induced by the virtualization layer, most approaches use a mix of performance counter monitoring (getting information without cooperating with the guest) or paravirtualization (cooperating with the guest). Related work detailed below are summed up in table 4.2. In table 4.2, with no further indication, contention means contention for LLC, interconnect and memory controller, PV stands for paravirtualization, "delegated" means that issues are delegated to the overlying operating system and applications.

Table 4.2: Related work on virtualization and NUMA

Proposal	Issue(s)	Managed Resource(s)	Approach
AQL_Sched	LHP + contention (LLC)	vCPU	monitoring
Merlin	contention	vCPU + memory	monitoring
BRM	contention	vCPU	monitoring + PV
Liu et al. [33]	contention + locality	memory	monitoring(PV)
vProbe	contention	vCPU	monitoring
Xen+	contention + locality	memory	monitoring + PV
XPV	delegated	vCPU + memory	PV

Application-specific quantum for multi-core platform scheduler – AQL_Sched In the same manner as the operating system scheduler, the VMM scheduler is responsible for managing compute resources by scheduling vCPUs. According to Taebe et al. [51], an important parameter of a scheduler is the quantum length, as a higher quantum length penalizes latency-critical applications but favors memory intensive applications by reducing cache contention. They thus claim that a fixed quantum length cannot benefit all applications and propose a scheduling policy for hypervisors using variable quanta, depending of the needs of the executed applications.

Their proposal, *AQL_Sched*, assigns an application type to each vCPU, which can evolve during its lifetime, and clusters vCPUs of the same type in a pool of pCPUs. They identify three types of applications with different patterns and related issues. First, CPU-bound applications which intensively use compute power and memory, these are susceptible to cache contention. Second, I/O intensive applications, which generate intense I/O traffic. These applications use interrupts to perform I/O requests and thus fail to use their CPU quantum while waiting. Third, concurrent applications which are composed of multiple threads and need to synchronize with one another, these are susceptible to the lock holder preemption issue mentioned before.

AQL_Sched identifies the type of application among the aforementioned three using four parameters, the number of I/O requests, the frequency at which they use spinlocks (detected by intercepting the `PAUSE` instruction), the total number of LLC requests, and the number of LLC misses among those requests.

Depending on the LLC miss metrics, AQL_Sched determines which vCPUs are running either cache friendly or cache trashing applications. The first step is to separate these two types of applications by clustering the first on a pool of pCPU and the other on another pool. Then, AQL_Sched assigns quantum lengths to the vCPUs. CPU/memory intensive types are given an important quantum length (90ms), vCPUs of the type "I/O intensive" or "concurrent", a small quantum length (1ms), finally, quantum agnostic trashing applications are given a medium quantum (30ms).

Taebe et al. tackle the issue of contention by classifying vCPUs behaviors using monitoring,

and segregating vCPUs into different pools of pCPUs. They also reduce occurrences of the Lock Holder Preemption problem by changing the quantum of vCPUs depending on their classification.

Merlin: Application- and Platform-aware Resource Allocation in Consolidated Server Systems As virtualization overhead decreases, the trend in datacenter is to collocate a maximum of services on a single machine using virtualization. In order to maximize machine utilization, efficient workload consolidation has become an important part of hypervisors. Because machine architectures are more and more complex, a change in one resource allocation can have an indirect effect on another type of resource. For example, changing the compute or memory share may lead to an increase in cache usage.

In order to limit these issues, Tembey et al. [53], propose the Merlin resource allocator which manages resources by taking all the different resource dimensions into account. Merlin seeks the best possible resource allocation that also avoids hurting other applications performances as a side effect.

Merlin groups set of applications in *virtual platforms* VPs and arbitrates VP resource shares by allocating resources while minimizing the differences between performance degradation of all applications. This leads to performance degradation being fairly shared between VPs.

They proceed in three steps, first, they evaluate the relative importance of each resource for each VP. Some applications may suffer greatly from cache contention or rather from memory contention. These metrics are evaluated from data provided by performance counters. Second, Merlin allocates the initial resources for all applications taking into account their specific needs. Third, using the performance counters, Merlin monitors the evolution of resource usage of each VP. Finally, Merlin adjusts the resources placement by choosing the cheapest possible reconfiguration method, among the following ones (from lowest resource consumption to highest consumption): changing CPU time shares, migrating vCPUs inside a NUMA node, migrating vCPUs from a NUMA node to another, migrating memory pages across nodes.

This proposal uses monitoring to evaluate pressure on different resource types and thus reduce contention through balancing of vCPUs and memory.

Optimizing virtual machine scheduling in NUMA multicore systems – BRM As described in section 2.1.2, resource management on NUMA machines must consider multiple architectural specificities, such as cache coherency or memory location to avoid the increased latency of remote accesses or congestion on the interconnect.

Because hypervisors have little information on the overlying operating system and applications, standard techniques described in section 2.3 are not applicable directly.

Instead Rao et al. [45] propose to characterize aforementioned problematic behaviors using a metric based on the penalty incurred by accessing the "uncore" memory. This metric, the *uncore penalty*, represents the number of stall cycles due to communicating on the network

outside the core. This number is defined as the number of L2 misses multiplied by the cost of an L2 miss, and represent the overhead due to a wrong memory placement forcing communication outside of the core.

Rao et al. exploit this metric by proposing a Bias Random vCPU Migration (BRM) algorithm to adjust the vCPU-to-core assignment. This algorithm is composed of three steps. First, evaluate the uncore penalty using hardware performance counters. Second, identify candidate vCPUs for migration. This is achieved by modifying the guest operating system to notify the hypervisor when a thread asks the kernel for a specific scheduling policy using the `sched_setaffinity` system call. Third, migrate a vCPU, randomly choosing the destination but with a bias toward the node which would minimize the uncore penalty.

The randomness in the migration decision statistically avoids simultaneously moving two competing or communicating vCPUs, which would reproduce the same situation after migration but on different nodes.

To sum up, Rao et al. introduce a metric based on monitoring data to reduce contention on the LLC, interconnect, and memory controller by balancing a subset of vCPUs. They use paravirtualization to detect which vCPUs are susceptible to need a special scheduling policy.

Optimizing virtual machine consolidation performance on NUMA server architecture for cloud workloads To enhance consolidation performances on modern machines, correctly managing these machines architectures is essential. For this reason, Liu et al. [33] study memory placement for NUMA architectures in virtualized cloud workloads. To this end, they distinguish four sources of overhead, LLC contention, memory controller congestion, interconnect congestion and remote memory access latency.

Their NUMA-aware memory management system estimates the memory access pattern using three metrics obtained through performance counters, the rate of instruction execution (IPC), the L3 cache hit rate and the L3 cache miss rate. These performance counters are updated by the guest through paravirtualization. Using these three metrics, they quantify the overhead relative to each of the four types. LLC contention is evaluated from the highest value of LLC hit rate among vCPUs of a VM. Memory controller congestion is determined using both cache hit rate and miss rate, as with a constant cache hit rate, cycles lost by cache misses are directly linked to memory controller congestion. Interconnect congestion is computed using the L3 cache miss rate of all the CPUs on a remote node. Finally, remote memory access latency is computed using the IPC ratio between local and remote accesses.

Using the estimated behavior of each memory zone, they manage memory with two different mechanisms, for allocations, the buddy allocator chooses nodes with the smallest overhead (preferring local node). During execution, they associate their overhead metric with each entry of the hypervisor second level page table. When a second level translation violation (equivalent of page fault) occurs, if the overhead for faulting memory zone exceeds a predefined threshold, it is exchanged with a page of another node with a lower estimated overhead.

Liu et al. use paravirtualization to collect metrics. They include this information in their hypervisor page table to reduce contention and locality when allocating pages for the guest system.

vProbe: Scheduling Virtual Machines on NUMA Systems The management of NUMA resources in virtualized environments is made more complex by the opacity introduced by the virtualization layer. As mentioned, some approaches use a black-box approach such as Rao et al. [45], others use paravirtualization such as Liu et al. [33].

However, according to Wu et al. [60], using only the *uncore penalty* metric, Rao et al. treat all performance-degrading factors equally which prevent the ensuing optimization to be precisely targeted at a specific factor. To solve this issue, and avoid breaking the isolation of the virtualization layer as Liu et al. do, they propose *vProbe* a NUMA-aware vCPU scheduler.

Similarly to other proposals, vProbe uses hardware counters to gather information on the running virtual machines, more specifically, about LLC accesses and the node containing the higher number of pager regularly accessed by a given vCPU.

vProbe classify vCPUs into three categories depending upon their LLC access pattern, akin to AQL_Sched [51]. LLC trashing vCPUs (LLC-T) cause many LLC misses, LLC fitting vCPUs (LLC-FI) intensively use the LLC but have a low LLC miss rate when there is no LLC contention, LLC friendly vCPUs (LLC-FR) are not impacted much by the LLC. Using this information, they introduce two different mechanisms to favor NUMA-friendly placement of vCPUs.

First, the *vCPU periodical partitioning* mechanism periodically reassigns all the memory intensive vCPUs (LLC-T and LLC-FI) to each node evenly, preferring their local nodes. This mechanism repeatedly selects the node with the least amount of memory intensive vCPUs assigned and adds an LLC-T vCPU, when no LLC-T remain, it does the same for LLC-FI vCPUs. This intends to reduce shared resource contention while preserving locality.

Second, to avoid the default scheduler of the hypervisor reverting changes introduced by the vCPU periodical partitioning algorithm, vProbe adds its own NUMA-aware load balancer. When a pCPU is idle, it steals work from other pCPUs by checking their run-queues, starting first by the pCPUs from the local node, preferably those with the highest workload. In order to avoid disrupting the balance of LLC contention, the stolen vCPU is chosen with the smallest LLC access footprint.

To conclude, vProbe reduces contention by favoring NUMA-friendly placement for vCPUs using different metrics. These metrics are obtained through monitoring using hardware performance counters.

An interface to implement NUMA policies in the Xen hypervisor – Xen+ Multiple studies focus on enhancing performances on NUMA architectures as detailed in section 2.3. In particular some describe new heuristics to manage NUMA resources at the OS level such

as Carrefour by Dashti et al. [13]. Moreover, previous research showed that, for virtualized environments executing on NUMA architecture, the main issue is the coordination between the hypervisor and the guest operating system.

In their work, Voron et al. [58] propose to reduce the overhead of virtualization on NUMA architectures by implementing resource management policies already existing in native environments inside Xen [2]. Their implementation contains four different policies. *First-touch*, which is the default memory allocation policy in Linux, allocates the backing memory of a virtual mapping the first time it is accessed on the node containing the accessing thread. *Round-4K* also part of Linux, allocates pages using a round-robin policy. *Round-1G* is the default policy of the Xen hypervisor, it allocates blocks of one gigabyte using a round-robin policy if possible, if the memory must be fragmented, it falls back to smaller sizes. *Carrefour* is the policy proposed by Dashti et al. and originally implemented in Linux. In Xen+, the policy can be chosen by the guest using a hypercall.

Policies are implemented by moving memory using the hypervisor's second level translation. When a page is allocated by the guest operating system, it signals the VMM using a dedicated hypercall, allowing the VMM to correctly place the page according to the chosen policy. Conversely, when a page is released by the guest operating system it signals the hypervisor which in turn changes its own mappings. This last case allows detecting page being reused after being released in particular for the first-touch policy.

Voron et al. implement different NUMA policies which can either reduce remote accesses or balance accesses to reduce congestion of memory controllers. The policy is chosen by the guest operating system through paravirtualization. The Carrefour policy, ported from the original Linux implementation uses monitoring to determine placement.

When eXtended Para - Virtualization (XPV) Meets NUMA When the hypervisor migrates vCPUs or memory, it changes the topology on which the virtual machine executes. However, in classic hypervisors, due to the opacity between the hypervisor and virtual machine, guest operating systems cannot take this virtual topology into account when managing resources.

Bui et al. [9] propose a new design to manage NUMA topologies in a virtualized context, which dynamically informs the guest operating system of the virtual topology offered to the virtual machine. This allows guest operating systems and NUMA-aware applications to reconfigure their behavior depending on the underlying changes advertised by the hypervisor.

Their method, "eXtended Paravirtualization" (XPV), is based on a split-driver. The hypervisor part of this driver is responsible for updating the NUMA topology presented to the guest when modifying vCPU or memory placement. The guest part of the driver retrieves the NUMA topology exposed by the other part and updates its internal structures and resource placement according to the new topology. However, this method forces to modify the guest kernel memory allocator and scheduler to take the information of the NUMA split-driver into

account. A strictly increasing counter representing the version of the NUMA topology allows to detect changes in the dynamic topology.

XPV uses paravirtualization to present the guest operating system and applications a dynamic map of the NUMA topology, guests operating systems and applications are responsible for using this information wisely.

Presented studies address the issues caused by virtualization, in general, and more specifically when running on NUMA machines. The opacity introduced by the virtualization layer is often the cause of these issues, explaining why these methods often rely on monitoring to circumvent the isolation. Moreover, the advent of efficient hardware assistance for virtualization, as described in section 4.1 reduces the cost of virtualization in general. While some propose to use transparency to help the operating system or applications better manage the specifics of the architecture, this often forces modification in the code of either the OS or both the OS and applications.

4.2 A driver for NUMA architectures: Scalevisor

Studies described in section 4.1.3 show that monitoring is a good solution to avoid opacity related issues when dealing with virtualization on NUMA architectures. However, our proposal is not a hypervisor in the common sense as it uses virtualization techniques to extract the role of managing resources from the operating system but does not pretend to guarantee any other classic properties of hypervisors.

First of all, this limits the features which need to be implemented and thus makes Scalevisor lightweight, in term of both the size of the code base and the execution behavior.

Second, as Scalevisor does not aim to support multiple guests, our only focus is on isolating our only guest operating system from the hardware NUMA topology. We can give full access to devices not used by our system. The opacity in our approach is reduced to presenting the guest with a simpler topology.

This section first presents how this abstract topology is built, how it is used to manipulate compute and memory resources underneath the guest operating system, and finally, describes Scalevisor internal structures. We conclude with some teachings about our approach.

4.2.1 Presenting an abstract memory/CPU topology

By presenting an abstract topology to the overlying operating system, Scalevisor allows the operating system resource management to stay the same regardless of the real hardware topology.

Virtualization techniques are used to provide a simple, linear view of the memory to the operating system and a pool of computation units, vCPUs. To this end, we first need to

discover the topology of the real hardware. Multiple ACPI tables include information about the architecture of the machine, the number of CPUs, the size of RAM and the distribution of both of these in the different NUMA nodes.

Scalevisor uses these tables to determine the topology at boot time and reserve a part of the resources necessary for its operational needs. The rest of the resources is used to fabricate the abstract memory and CPU view given to the overlying operating system. As shown in figure 4.6, this mechanism transforms the distributed topology of NUMA architecture into a linear SMP-like architecture in which the guest physical memory and CPU pool are composed of the multiple units of each NUMA nodes not used by Scalevisor.

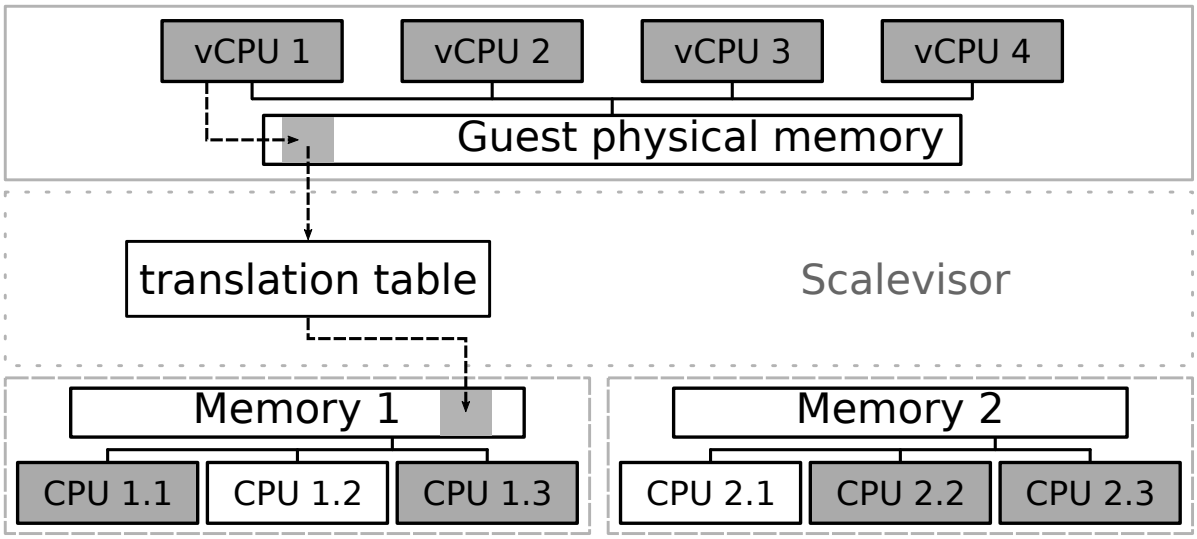


Figure 4.6: Abstract topology

While generating this abstract topology is necessary for the overlying operating system needs, its not sufficient to dissimulate the actual distribution of resources. Most operating systems contain a topology discovery subsystem not unlike the one contained in Scalevisor, and would thus either not function correctly if not given the needed ACPI tables or try to use the real topology if access to the real tables was granted. This is solved by generating ACPI tables which correspond to the abstract topology exposed to the guest system and removing from the guest memory space any table which would hint at a NUMA architecture.

The actual creation of the guest physical space is done using the EPT mechanism (second level address translation for Intel) available with modern CPU hardware virtualization. As explained in section 4.1, this table allows guest physical addresses to be translated to host physical addresses. The exposed CPUs are vCPUs controlled by the VMCS data structure presented in the same section.

4.2.2 Managing memory migration

The abstract topology is used by the operating system to allocate memory and schedule tasks. As explained before the ideal abstract topology is one were the physical topology does not play a role at all. To the opposite, our system needs to correctly manage memory placement in order to reach better performances.

Scalevisor already uses the EPT to create this abstract topology and thus uses it to manage memory migration.

In order to migrate the memory, the targeted page is marked read-only in the corresponding EPT structure to forbid any write access, thus making it safe to copy it to the new location. If a write access would occur during the migration process, it will cause a VMExit for the vCPU intending to write, and the vCPU will be preempted until the migration process is over. When the page is copied to the new memory location, the mapping in the EPT structure is changed to match the new page location as shown in figure 4.7.

If an EPT violation occurs due to the page being marked read-only, the corresponding TLB entry is invalidated decreasing performances, but as the mapping is changed, the TLB entry will need to be invalided anyway. As explained later in section 4.2.3, if the hardware supports VPIDs, others TLB caches are not modified by the VMExit do to an EPT violation.

These procedures are correct when migrating memory which is only used by CPUs. When some memory is used to interact with the I/O subsystem, and more specifically when it is the source or target of a DMA operation, migrating the page would make the current I/O operation fail or the migrated data incomplete.

In current operating systems, avoiding this issue is done by keeping a list of ongoing I/O operations, but when using the IOMMU to give control over some device to the guest operating system, it becomes impossible to monitor I/O operations, thus making it probably impossible to implement correct memory migration efficiently. In our prototype, we do not handle this problem: if Scalevisor migrates a page while it is written by a device, the write performed by the device is lost.

4.2.3 Managing CPU migration

Managing CPU migration is akin to scheduling thread except for the VMCS structure which is located in memory and thus also benefits from memory locality. While the vCPU structure and thus the execution context will move, from the operating system point of view, the OS thread will still execute on the same CPU, allowing an execution context to be moved without the overlying OS noticing. As explained in section 4.1.3, for classic virtualization purposes, this opacity tends to diminish performances of virtual machines because of double scheduling. However, by exposing a flat virtual topology, the operating system will have less reasons to move threads around and thus double scheduling occurrences will remain uncommon.

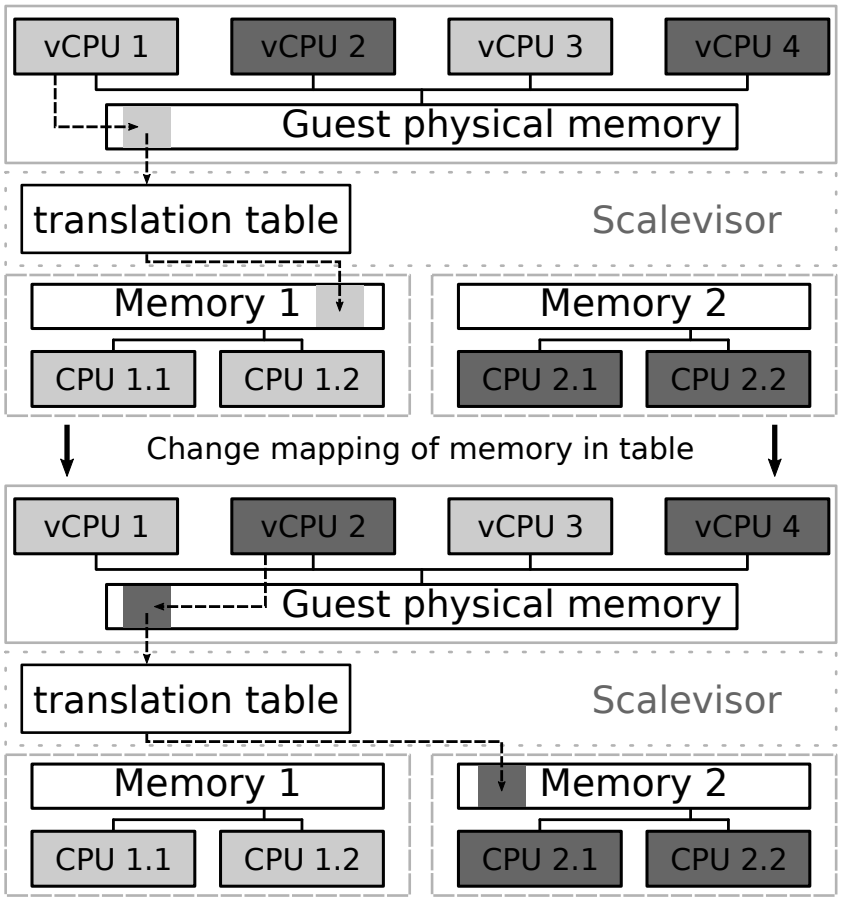


Figure 4.7: Migration using EPT mappings

Several operations are constitutive of migrating a vCPU. Firstly, the vCPU needs to be stopped as a result of a VMExit. It is thus possible to use any cause of VMExit to reschedule a vCPU on another physical CPU, but in order to have a reliable frequent source of VMExit, a specific VMCS option named "VMX preemption timer" can be used. This option allows the VMM to set a counter at a given value which will be decreased at a rate proportional to that of the time stamp counter (TSC, counts the number of CPU cycles). When the counter reaches zero, this will automatically cause a VMExit, thus allowing the VMM to perform needed management operations regularly.

When the physical CPU on which the vCPU is currently running exits VMX non-root mode, the VMM takes control and can either call `VMRESUME` if no changes are needed or choose to migrate the vCPU by calling `VMCLEAR` on the current physical CPU and `VMPTRLD` on the

destination CPU. The vCPU is then ready to be launched again using `VMLAUNCH` as represented in figure 4.3.

While migrating vCPUs is needed to make better use of a given machine depending of its architecture, it has performance implications.

Firstly, while VMX transitions do not force a TLB flush neither for guest-physical nor linear mappings when VPID is enabled (see section 4.1), the TLB caches are per CPU caches and thus not shared across different physical CPUs. This means whether VPIDs are supported and enabled in VMCS or not, migrating a vCPU on a new physical CPU will make it run without the new TLB caches containing the previously used mappings.

Secondly, the VMCS memory region is used to store data from the vCPU state, and even though some of its content is cached directly in the physical CPU on which the VMCS operates, the interaction between this memory region and the corresponding physical CPU must be efficient. Thus, migrating the vCPU to another NUMA node can suffer from any of the memory issues introduced in section 2.1.2, such as bad locality or causing congestion on the interconnect. Moreover, migrating vCPUs frequently can cause these same issues and thus greatly reduce performances for the guest operating system.

Finally, same as physical CPUs using interrupts to communicate with other hardware, vCPUs need to be able to send and receive interrupts. When vCPUs are pinned on their attributed physical CPU and never migrated, most interrupts can just be directly received by the CPU without leaving VMX non-root mode and without VMM interaction. If vCPUs are to be migrated, interrupts must be correctly directed to the target vCPU. This can be achieved by making the VMM reprogram each interrupt attached to the previous physical CPU which targets the migrated vCPU to be assigned to the destination CPU. However, changing targets for multiple interrupts is neither atomic nor efficient, thus using interrupt remapping capabilities of the IOMMU to perform dynamic interrupt migration is better as described by Tu et al.

4.2.4 Scalevisor internals

Scalevisor can be divided in two main parts. First, an operating system part, as the project needs to be able to boot on the target hardware. This includes the kernel which manages the boot process, memory allocation and driver communication, as well as drivers needed to interact with the minimum necessary hardware (timers, ACPI, IOMMU). The second part is composed of the “resources driver” capabilities and requires a working set of virtualization features and the vCPU scheduling and memory migration features. Figure 4.8 represents these different parts of Scalevisor grouped into the four logical groups: kernel, drivers, virtualization and scheduler.

Scalevisor’s kernel Although called minimal, because it contains only the minimum set of functionalities to boot and run, the kernel is quite complex, as any operating system, even

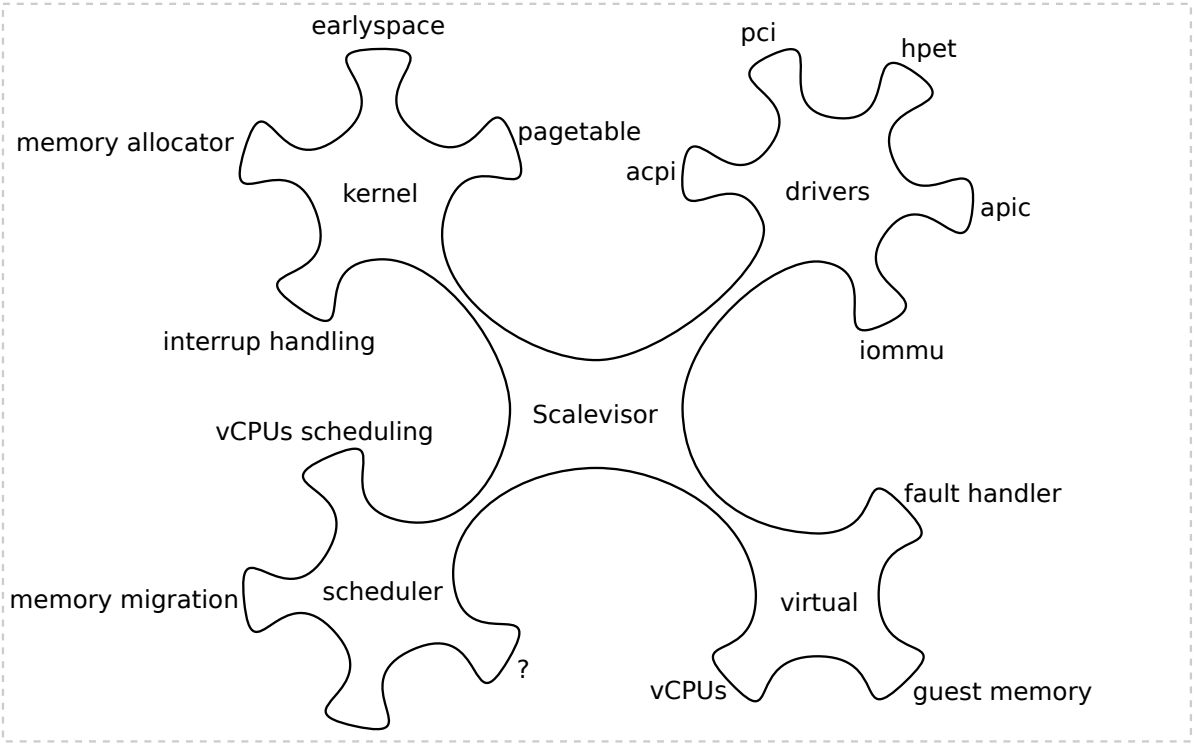


Figure 4.8: Software architecture of Scalevisor

minimalist, must deal with numerous hardware parts or behaviors.

Scalevisor is different from a classic hypervisor because it has been designed to manage resources in NUMA architectures. In Scalevisor, the architecture is represented by a Domain structure which encompasses all the information about a NUMA domain. This structure contains the CPUs assigned to a domain, the memory regions which belong to the given domain, and a number of methods which interact with the domain. Each domain has its own memory pool composed of the free memory regions and its own local memory allocator.

The aforementioned CPU structure contains all the logic used to control the CPU, electing a thread or executing a remote task using an Inter-Processor Interrupt as a trigger. It also abstracts vendor specific functionalities.

Domains and CPUs are globally managed through a super structure called Machine which contains most other non-NUMA related hardware, access to the console, and more importantly drivers.

For retro-compatibility needs, booting on a modern bare-metal machine means booting in real mode (16-bit, segmentation memory management scheme) and switching to protected

mode then long mode. Page table and paging memory need to be set up during these steps, more specifically when transitioning from protected mode to long mode as shown in figure 4.9.

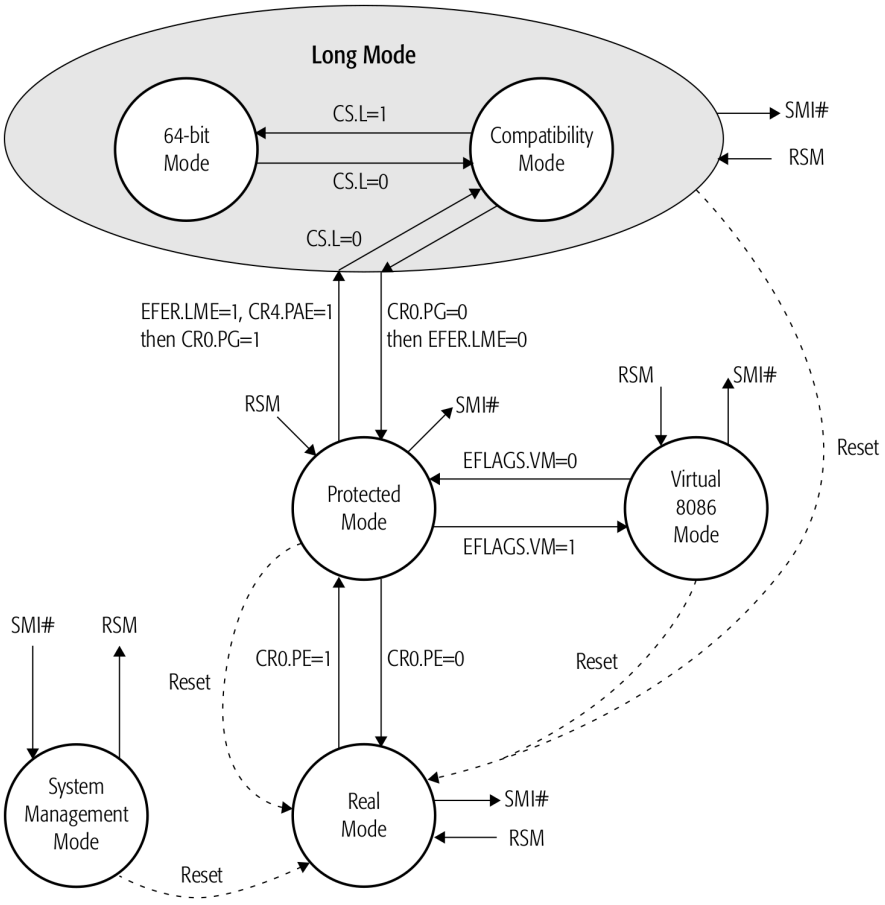


Figure 4.9: Operating modes of x86_64 architecture (AMD documentation [1, chap. 1.3])

During the process of booting, the Machine super structure is progressively filled using information from various sources while corresponding hardware is initialized.

At first, a fake domain is built using memory ranges given by the bootloader, this domain and the Bootstrap Processor (BSP) are used by our Machine structure to discover the rest of the topology. The number of CPUs and real memory ranges are then discovered using ACPI tables, Multiple APIC Description Table (MADT) and System Resource Affinity Table (SRAT) respectively, leading to a correct creation of Domain structures. This step concludes the bootstrapping of our kernel which is now able to correctly allocate and map memory but still uses the BSP as its sole processor. In order to wake up the other processor, called Auxiliary

Processors (APs), we need to use Inter-Processor Interrupts (IPIs) which are managed through the local APIC.

Device drivers The local APIC is the part of a core which is responsible for sending and receiving interrupts for this given core. This piece of hardware must be configured before it can be used for a variety of purposes, such as having a basic method of counting elapsed time, or communicating with other processors. The booting process effectively starts on one processor, the BSP, while the other processors, APs are woken up when receiving a specific interrupt sent from the BSP local APIC.

The second part of Scalevisor, device drivers, includes this local APIC, but also the High Precision Event Timer (HPET) which gives a more reliable and precise timer or the IOMMU which is used latter by the virtualization subsystem. Currently when multiple IOMMUs are present in a physical machine, the virtual machine cannot benefit from hardware assisted interruption remapping. When using complex topologies in state of the art hardware, multiple IOMMUs are often present and manage different devices. Once again, our IOMMU structure abstracts the differences between Intel and AMD IOMMUs.

The initialization of the IOMMU device is done by reading a specific ACPI table, the DMA Remapping Reporting table (DMAR) which indicates which hardware unit is located at which physical address, and when multiple IOMMUs are present which devices is a given IOMMU responsible for.

Virtualization For virtualization management, the architecture follows a design similar to the kernel part. The Guest structure contains the memory ranges available for a given guest and the related EPT mappings and IOMMU mappings. This structure is linked to one or more vCPUs represented by a same-named structure.

The Vcpu structure contains all the hardware CPU virtualization related logic and abstracts vendor specific configurations. For Intel vCPUs, this structure populates and controls the VMCS by providing a set of higher level methods to access fields. It also enforces hardware constraints by checking the global state of the VMCS is coherent with the vendor specification, which currently represents 246 different constraints.

The Vcpu is responsible for managing the different VMExit conditions using a set of cascading handlers, thus abstracting virtualization instructions such as `VMRESUME`, `VMCLEAR` or `VMLAUNCH`.

This layer is responsible of creating the abstract topology presented in section 4.2.1. The first step toward presenting a linear memory abstraction is to intercept the classic detection features used by the guest operating system to determine the topology.

The main interface to detect features supported by the CPU is through a call to the `CPUID` instruction, this instruction is thus intercepted and emulated using a Vcpu handler. Most of the values are returned without being modified, but some settings need to be. For example to

disable support for some instructions which cannot be directly executed because they modify privileged state but cannot be emulated either. This is the case of the `XSAVE` instruction which is shown as not available when queried through the `CPUID` interface.

When booting, the guest operating system tries to discover the memory topology by interacting with the BIOS. These functions are accessed by setting some values inside the CPU general registers and raising a software interrupt using the `INT` instruction. Multiple BIOS functions exist [7] to discover the memory map, linked to interrupt number 15 and identified by different values of the `eax` register, such as 0xe801, 0x88 or 0xe820.

The BIOS function bound to interrupt number 15 with `eax` = 0xe820, is the most common and precise method of discovering memory. It returns a list of memory zones, in which zones are described by their starting address, length and memory type (usable RAM, not usable, ACPI memory).

In order to present the guest operating system with an abstract memory topology, Scalevisor intercepts the interrupt number 15 and emulates the behavior of the BIOS for the e820 interrupt. If done correctly, other methods are not used by Linux to further confirm the results obtained with the e820 method. Therefore, we only have to intercept the e820 method which simplifies the implementation of our system.

Virtual scheduler The virtual scheduler is the part of Scalevisor which should control how resources are allocated and managed for the guest operating system. Managing resources can be seen as a three step process.

First, the scheduler accumulates data from the running guest operating system through different sources such as Intel's Precise Event Base Sampling (PEBS) for which Scalevisor has an interface. This interface produces raw data about which memory address is actively used by which CPU, or which memory address causes cache misses. The scheduler must then decide whether a given resource should be migrated and where. Finally, it actually migrates the given resource.

The memory migration system first allocates memory from the wanted domain corresponding allocator and uses the Guest structure described earlier to replace EPT mappings for a given memory range. As explained in section 4.2.2, the EPT mapping is changed to read-only during the copy and in case a write operation occurs, the vCPU will be stopped until the end of the procedure.

As for the vCPU migration, Scalevisor implements kernel-level threading, which allows a vCPU to be started in a thread context. When a window occurs during which the vCPU is stopped, the corresponding thread is then elected through an Inter Processor Interrupt (IPI) on the target CPU and the vCPU structure naturally resumes executing the previous context through a call to `VMLAUNCH`.

While the data collection of the first step and the mechanisms needed to migrate resources in step three are implemented, the decision mechanism, mainly based on heuristics has not

been implemented due to a lack of time.

4.2.5 Assessments

While heuristics haven't been implemented due to a lack of time, this approach of a driver for resource management is rich in findings. Scalevisor contains 20 k lines of code. It implements a large working set of functionalities: system level memory management, system interrupts, vCPU and VMCS structure, guest level memory management and masked topology. Scalevisor is able to boot a Linux guest entirely, but some parts are not mature enough to run evaluations.

This is especially the case for devices such as the IOMMU, whose complicated specification brought to light the impossibility for an individual or a small team to sufficiently master modern hardware. Nowadays, modern computer hardware includes a lot more devices with each its own quirks making development of a full operating system very complicated, even more in the case of a hypervisor which must leverage additional functionalities and specific hardware to reach satisfying performances. The complexity of hardware does not only affect devices, even CPUs became quite complicated, Intel's Software Developer's Manual [28] contains approximately five thousand pages of documentation and does not even include devices related information such as the IOMMU documentation or ACPI specification.

In general, it revealed that implementing a complex system such as a hypervisor in the constrained time of a PhD thesis is close to impossible, especially without a full team dedicated to the project.

Future work may build upon the already working set of functionalities, using the existing implementation to produce a viable test bed for experiments on heuristics.

Chapter 5

Conclusion and future work

Their important computation power has lead multicore architectures and NUMA in particular to become ubiquitous in context where this power is required. Nowadays, High Performance Computing applications, data analysis or artificial intelligence are all benefiting from these new architectures which are frequently encountered in datacenters or experimental setups. However, the NUMA architecture increases the computational power by making the internal machine topology more complex, with multiple CPUs and memory controllers.

This thesis studies how to manage resources in order to increase parallelism in NUMA architectures. This work focuses on two different approaches at two different levels in the software stack.

First, increasing CPU and network parallelism through automatically overlapping communication and computation in user space. This approach, Commmama, shows that using memory protection can enforce the constraints of the MPI specification while still increasing the parallelism by ensuring background progress for communications. Evaluation shows that, without having to modify neither the application code nor the MPI runtime, Commmama greatly increases the overlapping ratio between communication and computation. When the application time is balanced between communication and computation phases, the speedup obtained through this method reaches up to 73%.

Second, introducing an intermediate layer between the hardware and the operating system, Scalevisor aims to manage the complex topology of the machine. To this end, Scalevisor uses virtualization techniques to present an abstract topology to the overlying operating system. This additional software layer aims to ease the creation of new resource management policies by extracting architecture specific policies outside of the operating system. While the current implementation of Scalevisor is incomplete, it includes most of the needed functionalities of a standard operating system and implements enough of the virtualization features needed to run a guest Linux system. Because it is not mature enough, this work lacks evaluation to present the impact of Scalevisor on performances. However, working on Scalevisor revealed

that implementing such a system using modern virtualization hardware is close to impossible, in the time constraint of a PhD thesis. This reinforces our views that the complexity of current and future hardware needs to be abstracted using an approach such as Scalevisor.

Future work

Both Commmama and Scalevisor need further study in order to provide their full potential. This section presents three possible improvements for Commmama, one related to evaluating the effect on other primitives, and two which focus on decreasing the overhead of the memory protection layer. This section then follows with three possible improvements for Scalevisor, finishing implementation of missing features, resource management heuristics, and an hinting interface to better manage applications with unusual behaviors.

Commmama As far as Commmama is concerned, while the maximum overlapping ratio obtained depends on the ratio of communication and computation in the application, we showed that for balanced application, Commmama was able to almost completely overlap communication with computation. Thus, most of the refinement targets identifying and reducing the causes of overhead in Commmama.

First, while our evaluation of point-to-point primitives with Commmama shows this approach is promising, Commmama supports collective primitives, which have not been evaluated. Moreover, Commmama has not been evaluated yet with large scale applications. As different applications produce different memory access patterns and communication patterns, identifying these could provide clues for improving Commmama.

Second, Commmama current implementation uses the standard `mprotect` system call to change memory access rights on user provided communication buffers. While the results using this approach are satisfying, most of the overhead currently introduced by Commmama can be attributed to this memory protection system call. As mentioned in Section 2.2.2, when supported by the machine and kernel, the protection keys feature is another mechanism available to protect memory ranges. This mechanism could decrease the base overhead of Commmama's method. However, the number of protection keys and thus the number of possible different permission sets is limited to 15. Modifying Commmama to dynamically use protection keys or standard paging permissions could be a good trade-off between low overhead and broad use-case support.

Third, Commmama uses `MPI_Alloc_Mem` and `MPI_Free_Mem` to provide buffers with double mapping to the user application using the `mmap` system call. However, memory ranges obtained through this mechanism are allocated with each call to `MPI_Alloc_Mem` and freed with each call to `MPI_Free_Mem`. While this is not an issue if the user application reuses the reserved memory for multiple communication, this could increase the overhead of our approach otherwise. Introducing a better memory management scheme by reserving pages in advance would prevent this issue for applications not carefully designed.

Scalevisor While Scalevisor was not mature enough to provide a reliable testing environment for evaluations, we think our experience with the complexity of hardware shows that this approach is an interesting one to study.

First, completing the implementation of Scalevisor in its current state should be feasible. Most of the needed features are already implemented, and the missing features, while still complex to add, are few. Moreover, the code base of the project is still manageable for individuals with approximately 20 kSLOC, whereas the code of modern operating systems such as Linux, with more than 20 MSLOC, cannot be fully mastered by an individual. Missing functionalities include some critical drivers (hard drive support for example) and some less important features such as hardware assisted virtualization for some tasks (virtual interrupt management for example).

Second, developing and studying resource management heuristics is at the heart of the Scalevisor project. Some leads can be found in studies presented in both Section 2.3, focused on NUMA architectures and Section 4.1.3, presenting issues linked to the use of virtualization. As Scalevisor is neither a full-fledged hypervisor nor a standard operating system, it can be expected to present some issues pertaining to hypervisors but not all.

Third, Scalevisor primary purpose is to act as a black-box resource management system between the standard operating system and the hardware. However, we anticipate issues with applications featuring very unusual access patterns. As presented by related work on virtualization, paravirtualization is often a good solution to break the isolation of the hypervisor and solve issues related to this isolation. Therefore, introducing an hinting interface using paravirtualization would allow applications with unusual patterns to inform Scalevisor. Such an interface is shown in Figure 4.1.

Bibliography

[1] AMD. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*, April 2016. `https://www.amd.com/system/files/TechDocs/24593.pdf`.

[2] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In M. L. Scott and L. L. Peterson, editors, *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003*, pages 164–177. ACM, 2003.

[3] A. Baumann, P. Barham, P. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new OS architecture for scalable multicore systems. In J. N. Matthews and T. E. Anderson, editors, *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, pages 29–44. ACM, 2009.

[4] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In L. Rudolph and A. Gupta, editors, *ASPLOS-IX Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems, Cambridge, MA, USA, November 12-15, 2000*, pages 117–128. ACM Press, 2000.

[5] N. Bhatia. Performance evaluation of intel ept hardware assist. Technical report, VMware, 2009. `https://www.vmware.com/pdf/Perf_ESX_Intel-EPT-eval.pdf`.

[6] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, M. F. Kaashoek, R. T. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: An operating system for many cores. In R. Draves and R. van Renesse, editors, *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 43–57. USENIX Association, 2008.

[7] R. Brown. Ralf brown's interrupt list. `https://www.cs.cmu.edu/~ralf/files.html`.

[8] E. Bugnion, S. Devine, M. Rosenblum, J. Sugerman, and E. Y. Wang. Bringing virtualization to the x86 architecture with the original vmware workstation. *ACM Trans. Comput. Syst.*, 30(4):12:1–12:51, 2012.

[9] V. Q. B. Bui, D. Mvondo, B. Teabe, K. Jiokeng, P. L. Wapet, A. Tchana, G. Thomas, D. Hagimont, G. Muller, and N. D. Palma. When extended para - virtualization (XPV) meets NUMA. In G. Candea, R. van Renesse, and C. Fetzer, editors, *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019*, pages 7:1–7:15. ACM, 2019.

[10] G. Chatzopoulos, R. Guerraoui, T. Harris, and V. Trigonakis. Abstracting multi-core topologies with MCTOP. In G. Alonso, R. Bianchini, and M. Vukolic, editors, *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*, pages 544–559. ACM, 2017.

[11] L. Cheng, J. Rao, and F. C. M. Lau. vscale: automatic and efficient processor scaling for SMP virtual machines. In C. Cadar, P. R. Pietzuch, K. Keeton, and R. Rodrigues, editors, *Proceedings of the Eleventh European Conference on Computer Systems, EuroSys 2016, London, United Kingdom, April 18-21, 2016*, pages 2:1–2:14. ACM, 2016.

[12] A. Danalis, L. L. Pollock, D. M. Swany, and J. Cavazos. Mpi-aware compiler optimizations for improving communication-computation overlap. In M. Gschwind, A. Nicolau, V. Salapura, and J. E. Moreira, editors, *Proceedings of the 23rd international conference on Supercomputing, 2009, Yorktown Heights, NY, USA, June 8-12, 2009*, pages 316–325. ACM, 2009.

[13] M. Dashti, A. Fedorova, J. R. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quéma, and M. Roth. Traffic management: a holistic approach to memory placement on NUMA systems. In V. Sarkar and R. Bodík, editors, *Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, TX, USA - March 16 - 20, 2013*, pages 381–394. ACM, 2013.

[14] A. Denis. pioman: A pthread-based multithreaded communication engine. In M. Daneshtalab, M. Aldinucci, V. Leppänen, J. Lilius, and M. Brorsson, editors, *23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2015, Turku, Finland, March 4-6, 2015*, pages 155–162. IEEE Computer Society, 2015.

[15] S. Didelot, P. Carribault, M. Pérache, and W. Jalby. Improving MPI communication overlap with collaborative polling. In J. L. Träff, S. Benkner, and J. J. Dongarra, editors, *Recent Advances in the Message Passing Interface - 19th European MPI Users' Group Meeting, EuroMPI 2012, Vienna, Austria, September 23-26, 2012. Proceedings*, volume 7490 of *Lecture Notes in Computer Science*, pages 37–46. Springer, 2012.

[16] L. Fishgold, A. Danalis, L. L. Pollock, and D. M. Swany. An automated approach to improve communication-computation overlap in clusters. In G. R. Joubert, W. E. Nagel, F. J. Peters, O. G. Plata, P. Tirado, and E. L. Zapata, editors, *Parallel Computing: Current & Future Issues of High-End Computing, Proceedings of the International Conference ParCo 2005, 13-16 September 2005, Department of Computer Architecture, University of Malaga, Spain*, volume 33 of *John von Neumann Institute for Computing Series*, pages 481–488. Central Institute for Applied Mathematics, Jülich, Germany, 2005.

[17] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. Open MPI: goals, concept, and design of a next generation MPI implementation. In D. Kranzlmüller, P. Kacsuk, and J. J. Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary, September 19-22, 2004, Proceedings*, volume 3241 of *Lecture Notes in Computer Science*, pages 97–104. Springer, 2004.

[18] R. Ganesan, Y. Murarka, S. Sarkar, and K. Frey. Empirical study of performance benefits of hardware assisted virtualization. In R. K. Shyamasundar, L. Shastri, D. Janakiram, and S. Padmanabhuni, editors, *Proceedings of the 6th ACM India Computing Convention, COMPUTE 2013, Vellore, Tamil Nadu, India, August 22 - 24, 2013*, pages 1:1–1:8. ACM, 2013.

[19] F. Gaud, B. Lepers, J. Decouchant, J. R. Funston, A. Fedorova, and V. Quéma. Large pages may be harmful on NUMA systems. In G. Gibson and N. Zeldovich, editors, *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014*, pages 231–242. USENIX Association, 2014.

[20] L. Gidra, G. Thomas, J. Sopena, and M. Shapiro. A study of the scalability of stop-the-world garbage collectors on multicores. In V. Sarkar and R. Bodík, editors, *Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, TX, USA - March 16 - 20, 2013*, pages 229–240. ACM, 2013.

[21] L. Gidra, G. Thomas, J. Sopena, M. Shapiro, and N. Nguyen. Numagic: a garbage collector for big data on big NUMA machines. In Ö. Özturk, K. Ebcioglu, and S. Dwarkadas, editors, *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015*, pages 661–673. ACM, 2015.

[22] J. R. Goodman and H. H. J. Hum. Mesif: A two-hop cache coherency protocol for point-to-point interconnects. Technical report, University of Auckland, 2004. `http://hdl.handle.net/2292/11593`.

[23] J. R. Goodman and H. H. J. Hum. Mesif: A two-hop cache coherency protocol for point-to-point interconnects. Technical report, University of Auckland, 2009. http://hdl.handle.net/2292/11594.

[24] Grid'5000: A large-scale and flexible testbed for experiment-driven research. https://www.grid5000.fr.

[25] J. Guo, Q. Yi, J. Meng, J. Zhang, and P. Balaji. Compiler-assisted overlapping of communication and computation in MPI applications. In *2016 IEEE International Conference on Cluster Computing, CLUSTER 2016, Taipei, Taiwan, September 12-16, 2016*, pages 60–69. IEEE Computer Society, 2016.

[26] Intel® xeon® processor scalable family technical overview. https://software.intel.com/content/www/us/en/develop/articles/intel-xeon-processor-scalable-family-technical-overview.html.

[27] Intel. *Intel® Xeon® Processor E5 v2 and E7 v2 Product Families Uncore Performance Monitoring Reference Manual*, February 2014. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/xeon-e5-2600-v2-uncore-manual.pdf.

[28] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual*, October 2017. https://software.intel.com/content/dam/develop/external/us/en/documents-tps/325462-sdm-vol-1-2abcd-3abcd.pdf.

[29] Intel. *Intel® Virtualization Technology for Directed I/O*, November 2017. https://software.intel.com/content/dam/develop/external/us/en/documents/vt-directed-io-spec.pdf.

[30] O. Kilic, S. Doddamani, A. Bhat, H. Bagdi, and K. Gopalan. Overcoming virtualization overheads for large-vcpu virtual machines. In *26th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, MASCOTS 2018, Milwaukee, WI, USA, September 25-28, 2018*, pages 369–380. IEEE Computer Society, 2018.

[31] R. Lachaize, B. Lepers, and V. Quéma. Memprof: A memory profiler for NUMA multicore systems. In G. Heiser and W. C. Hsieh, editors, *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*, pages 53–64. USENIX Association, 2012.

[32] B. Lepers, V. Quéma, and A. Fedorova. Thread and memory placement on NUMA systems: Asymmetry matters. In S. Lu and E. Riedel, editors, *2015 USENIX Annual Technical Conference, USENIX ATC '15, July 8-10, Santa Clara, CA, USA*, pages 277–289. USENIX Association, 2015.

[33] M. Liu and T. Li. Optimizing virtual machine consolidation performance on NUMA server architecture for cloud workloads. In *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*, pages 325–336. IEEE Computer Society, 2014.

[34] R. Marotta, M. Ianni, A. Pellegrini, A. Scarselli, and F. Quaglia. A non-blocking buddy system for scalable memory allocation on multi-core machines. *CoRR*, abs/1804.03436, 2018.

[35] M. M. K. Martin, M. D. Hill, and D. J. Sorin. Why on-chip cache coherence is here to stay. *Commun. ACM*, 55(7):78–89, 2012.

[36] S. M. Martin, M. J. Berger, and S. B. Baden. Toucan - A translator for communication tolerant MPI applications. In *2017 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2017, Orlando, FL, USA, May 29 - June 2, 2017*, pages 998–1007. IEEE Computer Society, 2017.

[37] MPI-3.1 standard reference document. https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf, 2015.

[38] MPICH: A high performance and widely portable implementation of the mpi standard. `https://www.mpich.org`.

[39] K. Murthy and J. Mellor-Crummey. A compiler transformation to overlap communication with dependent computation. In *Proceedings of the 2015 9th International Conference on Partitioned Global Address Space Programming Models*, page 90–92. IEEE Computer Society, 2015.

[40] V. M. Nguyen, E. Saillard, J. Jaeger, D. Barthou, and P. Carribault. Automatic code motion to extend MPI nonblocking overlap window. In H. Jagode, H. Anzt, G. Juckeland, and H. Ltaief, editors, *High Performance Computing - ISC High Performance 2020 International Workshops, Frankfurt, Germany, June 21-25, 2020, Revised Selected Papers*, volume 12321 of *Lecture Notes in Computer Science*, pages 43–54. Springer, 2020.

[41] OpenMPI: A high performance message passing library. `https://www.open-mpi.org`.

[42] Y. Paek, A. G. Navarro, E. L. Zapata, J. Hoeflinger, and D. A. Padua. An advanced compiler framework for non-cache-coherent multiprocessors. *IEEE Trans. Parallel Distributed Syst.*, 13(3):241–259, 2002.

[43] M. S. Papamarcos and J. H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In D. P. Agrawal, editor, *Proceedings of the 11th Annual Symposium on Computer Architecture, Ann Arbor, USA, June 1984*, pages 348–354. ACM, 1984.

[44] G. J. Popek and R. P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, 1974.

[45] J. Rao, K. Wang, X. Zhou, and C. Xu. Optimizing virtual machine scheduling in NUMA multicore systems. In *19th IEEE International Symposium on High Performance Computer Architecture, HPCA 2013, Shenzhen, China, February 23-27, 2013*, pages 306–317. IEEE Computer Society, 2013.

[46] J. S. Robin and C. E. Irvine. Analysis of the intel pentium's ability to support a secure virtual machine monitor. In S. M. Bellovin and G. Rose, editors, *9th USENIX Security Symposium, Denver, Colorado, USA, August 14-17, 2000*. USENIX Association, 2000.

[47] E. Saillard, K. Sen, W. Lavrijsen, and C. Iancu. Maximizing communication overlap with dynamic program analysis. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region, HPC Asia 2018, Chiyoda, Tokyo, Japan, January 28-31, 2018*, pages 1–11. ACM, 2018.

[48] S. Schneider, C. D. Antonopoulos, and D. S. Nikolopoulos. Scalable locality-conscious multithreaded memory allocation. In E. Petrank and J. E. B. Moss, editors, *Proceedings of the 5th International Symposium on Memory Management, ISMM 2006, Ottawa, Ontario, Canada, June 10-11, 2006*, pages 84–94. ACM, 2006.

[49] M. Si, A. J. Peña, J. R. Hammond, P. Balaji, M. Takagi, and Y. Ishikawa. Casper: An asynchronous progress model for MPI RMA on many-core architectures. In *2015 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2015, Hyderabad, India, May 25-29, 2015*, pages 665–676. IEEE Computer Society, 2015.

[50] B. Teabe, V. Nitu, A. Tchana, and D. Hagimont. The lock holder and the lock waiter pre-emption problems: nip them in the bud using informed spinlocks (i-spinlock). In G. Alonso, R. Bianchini, and M. Vukolic, editors, *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*, pages 286–297. ACM, 2017.

[51] B. Teabe, A. Tchana, and D. Hagimont. Application-specific quantum for multi-core platform scheduler. In C. Cadar, P. R. Pietzuch, K. Keeton, and R. Rodrigues, editors, *Proceedings of the Eleventh European Conference on Computer Systems, EuroSys 2016, London, United Kingdom, April 18-21, 2016*, pages 3:1–3:14. ACM, 2016.

[52] B. Teabe, P. Yuhala, A. Tchana, F. Hermenier, D. Hagimont, and G. Muller. Memory virtualization in virtualized systems: segmentation is better than paging. *CoRR*, abs/2006.00380, 2020.

[53] P. Tembey, A. Gavrilovska, and K. Schwan. Merlin: Application- and platform-aware resource allocation in consolidated server systems. In E. Lazowska, D. Terry, R. H. Arpaci-Dusseau, and J. Gehrke, editors, *Proceedings of the ACM Symposium on Cloud Computing, Seattle, WA, USA, November 3-5, 2014*, pages 14:1–14:14. ACM, 2014.

[54] F. Trahay, A. Denis, O. Aumage, and R. Namyst. Improving reactivity and communication overlap in MPI using a generic I/O manager. In F. Cappello, T. Hérault, and J. J. Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 14th European PVM/MPI User's Group Meeting, Paris, France, September 30 - October 3, 2007, Proceedings*, volume 4757 of *Lecture Notes in Computer Science*, pages 170–177. Springer, 2007.

[55] C. Tu, M. Ferdman, C. Lee, and T. Chiueh. A comprehensive implementation and evaluation of direct interrupt delivery. In A. Gavrilovska, A. D. Brown, and B. Steensgaard, editors, *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, Istanbul, Turkey, March 14-15, 2015*, pages 1–15. ACM, 2015.

[56] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. M. Martins, A. V. Anderson, S. M. Bennett, A. Kägi, F. H. Leung, and L. Smith. Intel virtualization technology. *Computer*, 38(5):48–56, 2005.

[57] K. Vaidyanathan, D. D. Kalamkar, K. Pamnany, J. R. Hammond, P. Balaji, D. Das, J. Park, and B. Joó. Improving concurrency and asynchrony in multithreaded MPI applications using software offloading. In J. Kern and J. S. Vetter, editors, *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2015, Austin, TX, USA, November 15-20, 2015*, pages 30:1–30:12. ACM, 2015.

[58] G. Voron, G. Thomas, V. Quéma, and P. Sens. An interface to implement NUMA policies in the xen hypervisor. In G. Alonso, R. Bianchini, and M. Vukolic, editors, *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*, pages 453–467. ACM, 2017.

[59] D. Wentzlaff and A. Agarwal. Factored operating systems (fos): the case for a scalable operating system for multicores. *ACM SIGOPS Oper. Syst. Rev.*, 43(2):76–85, 2009.

[60] S. Wu, H. Sun, L. Zhou, Q. Gan, and H. Jin. vprobe: Scheduling virtual machines on NUMA systems. In *2016 IEEE International Conference on Cluster Computing, CLUSTER 2016, Taipei, Taiwan, September 12-16, 2016*, pages 70–79. IEEE Computer Society, 2016.

Résumé en Français

Avec l'émergence et l'évolution rapide de domaines scientifiques tels que l'analyse de données ou l'intelligence artificielle, les besoins en puissance de calcul ont fortement augmenté. Depuis des années, en raison de contraintes physiques, l'augmentation de la puissance des processeurs se fait au travers d'une augmentation du nombres de cœurs et non plus d'une augmentation de la fréquence.

Cette augmentation de la puissance de calcul et du débit réseau a été rendue possible grâce à l'utilisation de machines dotées d'architectures matérielles complexes. Les améliorations de la fréquence du processeur ont créé une différence importante entre la vitesse d'exécution des instructions et celle à laquelle la mémoire peut répondre aux requêtes de données. Pour réduire cette différence, les processeurs modernes incorporent de petites zones mémoires très rapides appellées des caches. Ces caches permettent de réduire la latence des accès à la mémoire. De plus, le nombre de requêtes en mémoire augmentant avec le nombre de processeurs, la pression occasionnée sur le contrôleur mémoire est devenue trop importante pour des machines dotées d'un grand nombre de cœurs. Afin de pallier le manque de bande passante, plusieurs contrôleurs permettant de gérer la mémoire ont été ajoutés, distribués à l'intérieur d'une seule machine. Cette nature distribuée entraîne une différence de latence selon les positions relatives du processeur qui accède à la mémoire et de la zone de mémoire acccédée par celui-ci. Ces architectures sont appellées Mémoire à Accès Non-Uniforme (NUMA).

Ce nouveau paradigme nécessite une évolution du logiciel afin de pouvoir développer toute la puissance de ces machines, faisant ainsi du parallélisme une pierre angulaire de la pile logicielle. Ce document propose une approche en deux parties.

Commmama Le premier obstacle à une augmentation du parallélisme est de recouvrir la communication réseau avec le calcul. Pour cette partie, nous nous sommes concentrés sur le niveau utilisateur dans le cadre de l'environnement d'exécution MPI (Message Passing Interface). MPI propose deux types de communications, les communications bloquantes et non-bloquantes. Les communications bloquantes sont simples d'utilisation pour le développeur mais ne peuvent pas recouvrir les communications avec du calcul, à l'inverse, les communications non-bloquantes sont plus complexes mais permettent de recouvrir une certaine quantité de communication.

Dans le but de combiner la simplicité et l'efficacité, nous proposons *Commmama* qui transforme automatiquement les communications MPI bloquantes en communications non-bloquantes au cours de l'exécution. Nous montrons ainsi que deux principaux facteurs contribuent à l'augmentation du recouvrement. Premièrement le progrès en arrière plan, qui garantie que les communications ont lieu pendant que s'effectue le calcul. Deuxièmement, la taille de la fenêtre temporelle pendant laquelle la communication et le calcul peuvent avoir lieu en parallèle.

Pour s'assurer de la conformité à la spécification MPI lorsque les communications bloquantes sont transformées en communications non-bloquantes, Commmama doit garantir que les buffers de communications ne sont pas accédés de façon incorrecte pendant la durée de la communication. Commmama est composé de trois couches différentes. La couche d'interception remplace les primitives bloquantes par des primitives non-bloquantes, évitant ainsi au développeur de modifier le code. La couche de protection protège la mémoire des buffers de communications à l'aide de la primitive système `mprotect` au début de la communication et retire cette protection lors de la fin de la communication. Cette couche permet d'intercepter les accès incorrects lorsque la communication est en cours. Enfin, la couche de progrès effectue la communication dans un fil d'exécution séparé, permettant ainsi au calcul de s'effectuer en parallèle.

Nous avons évalué Commmama en comparant deux exécutions, l'une utilisant uniquement OpenMPI, l'autre utilisant Commmama au-dessus de OpenMPI. Les résultats confirment qu'OpenMPI ne parvient pas à recouvrir correctement la communication avec du calcul, avec un temps de calcul supérieur au temps de communication, Commmama au-dessus d'OpenMPI effectue quasiment la totalité de la communication en parallèle avec le calcul, résultant en une accélération de l'exécution du programme de 73%.

Scalevisor Bien que le recouvrement de la communication avec du calcul permet d'augmenter le parallélisme, la contention sur des ressources matérielles partagées diminue également la capacité du logiciel à exploiter les performances de la machine.

Comme expliqué précédemment, la complexité des machines multicoœurs a grandement augmenté pour répondre au besoin de puissance de calcul. Cependant, cette complexité des architectures NUMA crée d'autres problèmes dont peuvent souffrir les machines si le logiciel gère de façon incorrecte les ressources. La présence de plusieurs contrôleurs mémoire peut causer un encombrement du réseau interne reliant les différentes parties de la machine. Cet encombrement est aggravé par les caches qui utilisent eux-mêmes ce réseau interne pour que les données stockées dans ces caches soient cohérentes.

Pour masquer la complexité du matériel et gérer la mémoire efficacement, la deuxième partie de notre approche utilise des techniques de virtualisation pour gérer les ressources de façon transparente. *Scalevisor* est une couche légère placée entre le matériel et le système d'exploitation. Cette couche permet de présenter une topologie matérielle simple au système

d'exploitation tout en gérant les ressources en tenant compte de la véritable topologie.

Scalevisor utilise les techniques de virtualisation matérielle récentes pour gérer les ressources. Le deuxième niveau de pagination, EPT, fourni par les processeurs Intel récents permet de migrer la mémoire simplement et de façon transparente d'un emplacement mémoire à un autre. De même, la virtualisation matérielle des processeurs permet de migrer les vCPUs. Pour faciliter l'intégration de nouvelles politiques de placement, Scalevisor est composé de quatre modules. Premièrement, une partie noyau, responsable du démarrage et du fonctionnement propre de Scalevisor. Deuxièmement, une partie virtualisation contenant les primitives nécessaires à l'utilisation de la virtualisation matérielle et permettant la gestion du système invité. Troisièmement, une partie pilotes, contenant les pilotes essentiels, en particulier la gestion de la communication inter-processeurs. Finalement, une partie ordonnanceur, contenant les politiques de gestion des ressources.

ECOLE DOCTORALE

Titre : Gestion mémoire pour les systèmes d'exploitation et environnements d'exécution

Mots clés : Gestion mémoire, Virtualisation, informatique en nuage

Résumé : Avec l'émergence et l'évolution rapide de domaines scientifiques tels que l'analyse de données ou l'intelligence artificielle, les besoins en puissance de calcul ont fortement augmenté. Depuis des années, en raison de contraintes physiques, l'augmentation de la puissance des processeurs se fait au travers d'une augmentation du nombres de cœurs et non plus d'une augmentation de la fréquence.
Ce nouveau paradigme nécessite une évolution du logiciel afin de pouvoir développer toute la puissance de ces machines, faisant ainsi du parallélisme une pierre angulaire de la pile logicielle.
Les systèmes d'exploitation, directement concernés, doivent inclure différentes règles permettant la bonne gestion de différents types de machines. Cependant, la gestion de ressources est souvent divisée en différentes unités responsables chacune d'une ressource spécifique, qui prennent des décisions sans vision globale du système. De plus, en raison de la complexité et de l'évolution rapide du matériel, les systèmes d'exploitation ont de plus en plus de difficultés à tenir compte des variations subtiles entre deux machines. L'important développement de la technologie de virtualisation nous permet de proposer une nouvelle approche pour la gestion de ressources qui utilise la virtualisation pour ajouter une couche de gestion des ressources dédiée entre la machine et le système d'exploitation habituel.
Au même titre que les systèmes d'exploitation, les applications doivent exécuter une partie de leur code en parallèle pour obtenir des performances élevées. C'est le cas en particulier pour les environnements d'exécution tels que MPI qui ont pour but d'aider à la parallélisation d'applications. Avec les architectures matérielles modernes dotées de réseaux rapides, le recouvrement de la communication réseau avec du calcul est devenu partie intégrante du parallélisme applicatif. Une certaine quantité de recouvrement peut être obtenue manuellement mais cela reste une procédure complexe. Notre approche propose de transformer automatiquement les communications bloquantes en communications non bloquantes, augmentant ainsi le potentiel de recouvrement. Pour cela, nous utilisons un thread séparé pour les communications et contrôlons les accès à la mémoire des communications. Nous garantissons ainsi la progression des communications et une meilleure parallélisation de celles-ci et des calculs.

Title : Memory managment for operating systems and runtimes

Keywords : Memory managment, Virtualization, Cloud computing

Abstract : During the last decade, the need for computational power has increased due to the emergence and fast evolution of fields such as data analysis or artificial intelligence. This tendency is also reinforced by the growing number of services and end-user devices. Due to physical constraints, the trend for new hardware has shifted from an increase in processor frequency to an increase in the number of cores per machine.
This new paradigm requires software to adapt, making the ability to manage such a parallelism the cornerstone of many parts of the software stack.
Directly concerned by this change, operating systems have evolved to include complex rules each pertaining to different hardware configurations. However, more often than not, resources management units are responsible for one specific resource and make a decision in isolation. Moreover, because of the complexity and fast evolution rate of hardware, operating systems, not designed to use a generic approach have trouble keeping up. Given the advance of virtualization technology, we propose a new approach to resource management in complex topologies using virtualization to add a small software layer dedicated to resources placement in between the hardware and a standard operating system.
Similarly, in user space applications, parallelism is an important lever to attain high performances, which is why high performance computing runtimes, such as MPI, are built to increase parallelism in applications. The recent changes in modern architectures combined with fast networks have made overlapping CPU-bound computation and network communication a key part of parallel applications. While some degree of overlap might be attained manually, this is often a complex and error prone procedure. Our proposal automatically transforms blocking communications into non-blocking ones to increase the overlapping potential. To this end, we use a separate communication thread responsible for handling communications and a memory protection mechanism to track memory accesses in communication buffers. This guarantees both progress for these communications and the largest window during which communication and computation can be processed in parallel.

Institut Polytechnique de Paris
91120 Palaiseau, France