Scalevisor: using hypervision to build a memory driver for NUMA machines

M2 Internship Report

Nicolas Derumigny
ENS de Lyon - UVSQ

Gaël Thomas
Télécom SudParis

September 6, 2018

People who are really serious about software should make their own hardware.

Alan Kay

Acknowledgements

Remerciements

This report is the emerged part of a long work starting from the end of March until the end of September at Télécom SudParis in the HP2 team. This internship is also part of my studies at the Université de Versailles Saint-Quentin-en-Yveline for my Master Degree in Computer Science, following my cursus at the ENS of Lyon.

I would like to particularly thank Miss Nahid EHMAD for the direction of the master, as well as Mr Gaël THOMAS for this internship's supervision. I thank also Miss Anne BENOIT and Miss Laure GONNORD for their pieces of advice on my orientation, Mr Alexis LESCOUET and Mr Remi DULONG for their presence and help during these seven months of research; and finally my brother Alexis DERUMIGNY and more generally my family for all their support.

Contents - Sommaire

1 Introduction

Introduction

Modern microprocessors have become more and more complex on both the compute and the memory side. Indeed, nowadays scientific clusters are massively multicore's and thus expose a complex NUMA topology to the operating system, which is not always able to cope efficiently with the scheduling problem the latter raises. Moreover, from a software engineering point of view, the specificity of each machine has to be disseminated throughout the kernel, when this should be avoided for maintainability and scalability reasons.

To address this issue, Scalevisor is an hypervisor that uses virtualization extensions to run transparently a non-modified version of Linux version to which a flat topology is exposed. Below the guest system, Scalevisor will re-schedule virtual CPUs and memory chunks on the physical server in order to minimise as much as possible the NUMA effect, playing the role of a CPU and memory driver. Thanks to the flat topology Linux detects, we can safely assume the Linux Completely Fair Scheduler not to move neither memory nor threads to fit to the underlying hardware and such not ruining Scalevisor's work.

The scope of this project goes beyond a mere seven-months internship. Even though Scalevisor had already one year and a half of development when I stared to work, not all features were available; and the path to go to a bare metal[1] booting system was not established. That is the reason why my work is split into two contributions. On the one hand, a few modules of different importance for the project itself were implemented: some tools to port the booting procedure of the Applications Processors, a wrapper surrounding the `CPUID` instruction in order to query and activates x86 extension sets, and a standalone library bringing the PEBS API to a more high-level interface for the programmer. On the other hand, I programmed a small benchmarking tool that measures several memory timing such as local and distant RAM load latencies and cache coherency switching latencies. This tool was used to test the target system of the Scalevisor project, based on four Intel Skylake-SP CPUs, as well as an other server of the team, based on a dual-socket AMD's EPYC processor architecture.

[1]A bare metal working system is reached when the hypervisor directly boots on the hardware and is stable enough not to be emulated by an other software as we do during its development.

2 About Télécom SudParis
A propos de Télécom SudParis

Télécom SudParis is an engineering school located in Evry, part of Institut Mines-Telecom (IMT). My internship took place in the Samovar laboratory as part of the HP2 (High Performance and Parallelism) team. The latter team is specialised in HPC (High Performance Computing), as a subdivision of the Computer Science Department.

2.1 Generalities
Généralités

IMT is a public institution of international fame oriented in research and teaching in the numeric and engineering field under the direction of the ministry of Economy and Industry. Every year, more than 13 000 engineers, managers and PhD. are graduated from the school members (IMT Atlantique, IMT Lille Douai, IMT Mines Albi, IMT Mines Alès, Institut Mines-Télécom Business School[2], Mines Saint-Etienne, Télécom ParisTech and Télécom SudParis). Thirteen schools have partnerships with IMT all around France and in Tunis, in order to increase its influence on distant territories. IMT also has close relationships with industrial actors to sponsor projects and to cultivate a fructifying link with the engineers in the making.

2.2 About IMT
A propos de IMT

The goal of the HP2 team is to improve the usage of computing resources, both on the performance side (less consumption and less execution time) and on the software side (simplifying the API, making the accelerators more easy-to-use). One of the major objective of the team is to bring parallelism to a simpler interface, in order to soften the usage of heterogeneous platforms so that development costs are reduced while keeping the same level of performance for HPC applications. In this background, the HP2 team mainly focus on experimental scheduling heuristics at the system level, as well as performance analysis and micro-tuning of hardware parameters.

[2]That is *strictly forbidden* to shorten in ”IMT Business Scool”

2.3 About the SAMOVAR laboratory
A propos du laboratoire SAMOVAR

The SAMOVAR laboratory (Distributed Services, Architectures, Modelling, Validation and Network Administration) is a mixed research unit established in Télécom SudParis, focused on services, network and telecommunication. It received its accreditation from the CNRS in 2003, 2007, 2010 and 2014. The HP2 team is part of the ACMES (Algorithms, Components, ModEls and Services for distributed computing) research team, one of the five teams included in the SAMOVAR group: ACMES also includes MARGE (Algorithms, Components, ModEls and Services for distributed computing) and SIMBAD (Semantic Interoperability for Mobile, collaBorative and ADaptive applications) along with the HP2 team I was part of. The SAMOVAR laboratory counts more than a hundred doctorates, 79 permanent employees, 17 of which are part of ACMES; and only 4 of them form the HP2 team.

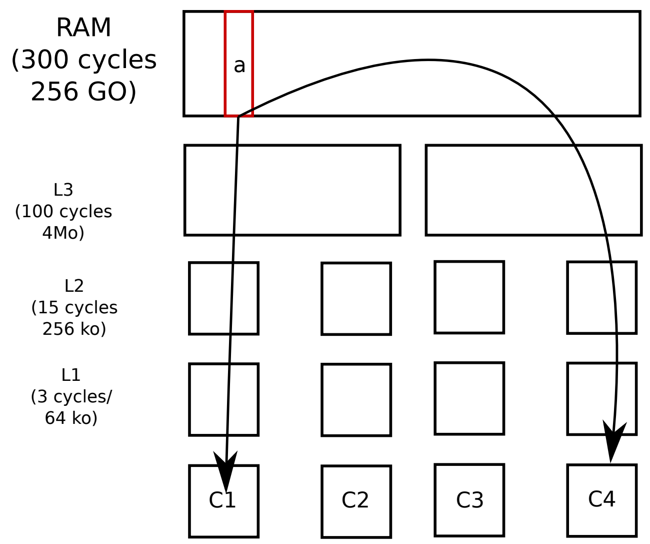


Figure 1: Block diagram of the cache hierarchy for a 4-core system (C1...C4) with shared L3 between two cores.

3 State of the art Etat de l'art

3.1 Goals of the internship Buts du stage

Technical background

Multiprocessing and cache hierarchy My work follows the need of a simpler interface between the hardware and the operating system. As the race to reach higher frequencies stopped about 15 years ago, stabilising at around 3 GHz, more and more cores have been introduced in servers, as Moore's empirical law were still valid[3]; this results in a complex memory hierarchy.

To decrease the time required to load a value from the RAM, caches were already developed in the eighties. They consist of small amount of on-die memory, quicker to access for the system than the main memory; but they have to keep a specific coherency with RAM modules. The algorithm used in this latter goal is thus named *Cache Coherency Protocol*. When more compute processors were added, this protocol became more complex as data can now be shared and even accessed by two processors at the same time. Synchronisation instructions were implemented in order to keep memory coherency among the cores, lead-

[3]"The density of integration of transistors in microchips doubles every two years."

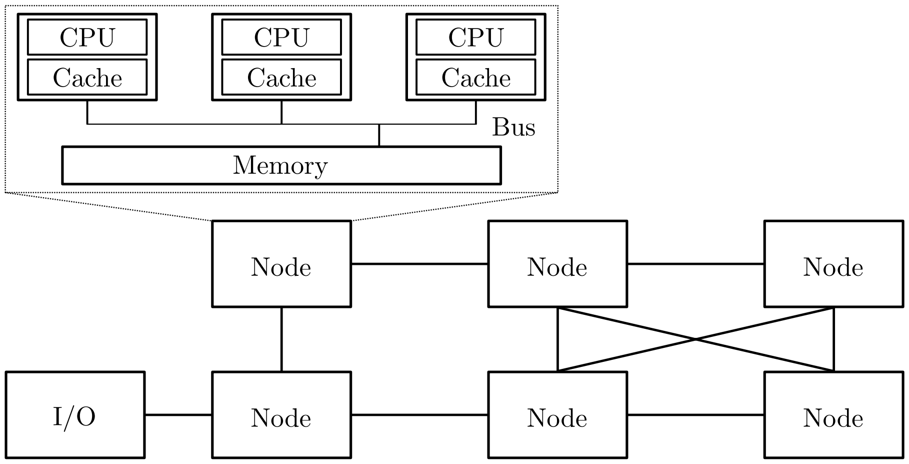


Figure 2: Logical view of a NUMA machine

ing to nowadays processors. Today, standard X86 processors have 3 level of cache named L1/L2/L3 (ordered by increasing size and latency of access). Usually, the L3 cache is *shared*, meaning that a set of cores can equally access the data gathered inside; whereas the L1 and L2 cache are *private*: only the bound core can access to its own data, illustrated in figure 1.

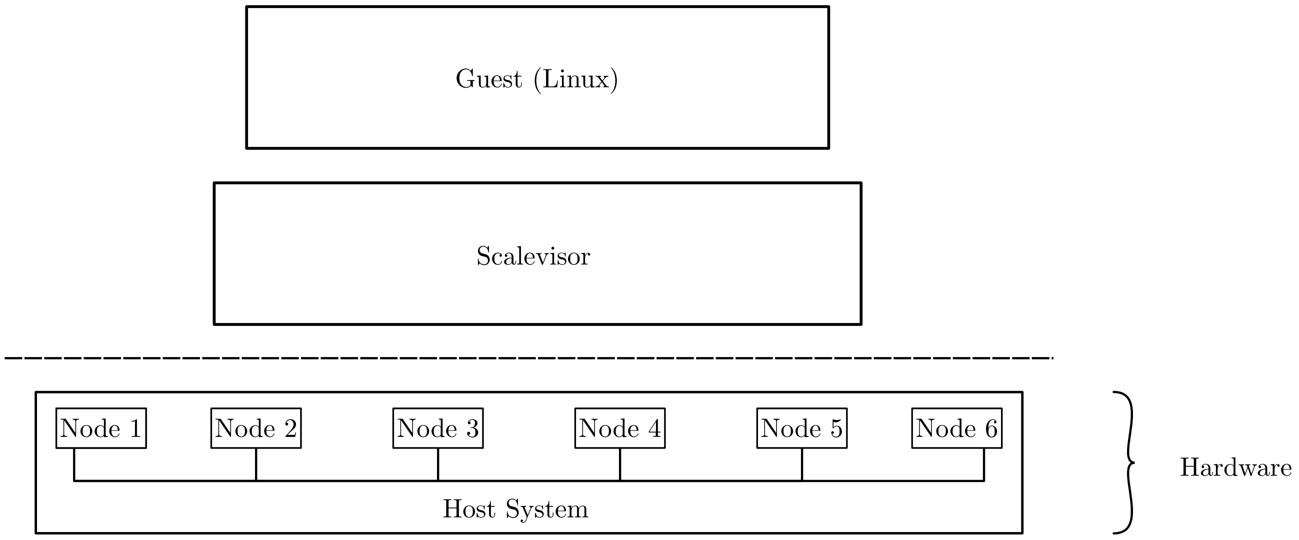


Figure 3: Block diagram of an hypervisor

NUMA effect and hypervisor When the industry started to offer multi-socket server, another issue rose. The physical location of the data was directly linked with the time needed to access it. For example, if a variable was stored in a memory cell belonging to the RAM controlled by the socket 1, a few additional hundred cycles were needed when loading the value in the register of a core physically present in the socket 2. This was called *NUMA* effect, for *Non-Uniform Memory Access*, illustrated in figure 2.

Until now, NUMA-aware policies were directly implemented - under Linux - in the scheduler. If that first lead to increasing performances with minimal code restructuring, it is now a sprawling part of the sources that can hardy be fully understood. Moreover, we suspect the default scheduler to be inherently inefficient due to its sub-optimal design: that lead to the idea of Scalevisor.

Usually, an hypervisor is simply a layer between virtual machines, named *guests*, and the *host* server, see figure 3. They are mainly used in cloud-based services for sharing part of a server between several clients in order to maximise the usage of a many-core machine. With Scalevisor, we aim at using an hypervisor to hide the underlying topology to the guest system, and re-schedule threads *and* displace memory according to CPU-specific policies: that forms a CPU/memory driver. One major feature of Scalevisor relies in the fact that it is designed to be totally transparent to the user, in the sens that a vanilla Linux image[4] can be booted without any additional configuration.

Context of the internship Six month before my internship, a first draft of Scalevisor was up and running on an older AMD machine based on a Opteron CPU. Unfortunately, this processor lacked an essential virtualization feature: the vAPIC, that allows selective redirection of the interruptions. Without vAPIC, Scalevisor would need to emulate its behaviour, which would be way slower than the native performances. As our goal is to overcome these native performance, changing the server was the only option. The new machine turns out to be an Intel one, so the previous work had to be adapted for Intel's own way of configuring a CPU. This required a major part of technical digging and was not fully completed before my internship.

My role in the team As the software development tasks that have to be done in order to boot the Linux kernel were not parallelisable (it consisted mostly of a suite of bug solving and tricks to handle either the processor undocumented features or the way Linux deals with its own booting sequence[5]), my role in the project was to develop some parts of Scalevisor that were not necessary at the moment but would be useful later. Even tough we were during my internship only two main contributors to the project, I had to follow coding standards for readability and reuse purpose; this was mandatory, given the fact that the implementations

[4]During the development, we use the latest ArchLinux Iso

[5]For example, even if Linux has detected via the `cpuid` instruction that it runs on an AMD CPU, it will still try to access some Intel-specific registers, that will trigger a fault. Once caught, Linux finally starts the standard AMD booting sequence.

I did are to be used later by scientists I would have never met. Moreover, the project follows a scrupulous git branching model in order to keep a decent history of the implementation and follow the regressions that some features can add to the project.

Scalevisor is entirely coded in C++ with some early boot stage in assembly, but as it is an hypervisor, some features of the language cannot be used until a specific boot sequence has been executed[6]. Memory leaks can become a lot more serious compared to any other project as the hypervisor should never leak information to the host system, even though they shared theoretically the same memory space. Moreover, overflows can occur during the early boot stage, leading to random crashes depending on the data overwritten. For example, a page table overflow was causing the system to regularly hangs depending on the log level chosen - these type of issue being very hard to detect and solve.

After having developed the modules listed in section 4.1; we realised that we missed a tool describing the effective micro-architecture of our CPU. With four sockets composed of one die on which lies 16 cores each, we have to benchmark ourselves the latencies if we want - when we will reach the scheduler implementation part - to fine tune Scalevisor as a memory driver tailor fit for our server. That is the reason why I was charged of the implementation of a micro-benchmarking tool, with two main goals: on one hand being easy to compile and run so that the software would be quickly rebuilt if the machine were to be reset, and on the other hand being as exact and configurable as possible, with no restrictions on the administrative rights it requires - given the constraint that it cannot be executed in kernel (ring-0) mode. This benchmark turns out to be very attractive for an other member of the team that has also received an (AMD) server for his work on the possible NUMA effects mitigations. This opportunity allows me to compare two radical different micro-architectural implementations of the x86 instruction set: the Intel one based on monolithic single-die sockets using a mesh, and the AMD one relying on muti-core-module sockets linked by a proprietary interconnect.

Once the measurements have been made, I had to analyse the results and put them into perspective with the official documentation of the vendor, that can turn out to be misleading or even false. Such characteristics also explain the behaviour described in the literature concerning NUMA optimisation. If the main idea has long been to bring closer to the core the memory it accesses, later results tend to prove that a better strategy consists in avoiding saturation of the memory controller by balancing the accesses amongst the available nodes[5].

[6]Global variables especially are never used, and static members of classes will be uninitialised during early boot stage

3.2 Related works
Travaux connexes

3.2.1 About NUMA
A propos de l'effet NUMA

Mitigating the NUMA effect is one of the major issues of the two last decades. If scientific programs can heavily benefit from manycore systems, the NUMA effect is interfering and can slow the execution, although recovering of the extra latency can be realised. It was believed for a long time that the interconnect latency was to blame for the NUMA-related performance degradation[5], but it has been experimentally verified not to be the case on latest processors, but rather saturation of this interconnect. That is why the default Linux kernel policies, first-touch or round-robin are particularly inefficient on certain cases[2], where customised policies leads to improvements reaching a diminution of the execution time up to 3.6 times.

On more specific programming, NumaGiC[4] aims at minimising the NUMA effect on Java's garbage collection mechanism by increasing memory locality and work sharing through garbage collector threads, with overall performances increased up to 5.4 times; the worst case occurring when the heap size is especially small with a maximum degradation measured as 8%.

3.2.2 About hypervisor
A propos des hyperviseurs

The first roots of an hypervisor on NUMA machines comes from the Disco[1] project at Stanford, that allows to run multiple instances of the IRIX operating system[7] at the same time on a single machine. Its idea was to hide the NUMA-ness of the memory architecture by migrating or duplicating memory and move threads dynamically to balance the load, achieving an improvement of 37% compared to a non-NUMA-optimised OS.

NUMA-friendly policies have also been recently implemented in Xen[7] (an open-source hypervisor mainly used for server sharing through vitalisation extensions), resulting in a performance gain of more than 2 times on 9 applications out of a selection of 29, realised on an AMD 48-core machine. This was mainly due to the centralisation of I/O tasks to one virtual machine, bottlenecking the whole execution for useless requests.

[7]A UNIX-based proprietary OS, discontinued in 2013

3.2.3 About performance measurement on NUMA machines
A propos des mesures de performances sur les machines NUMA

Performance monitoring has been the subject of many open-source projects. The standard tool for performance profiling on Linux is `Perf`, a command using performance counters and kernel pass-through to monitor an application with a highly configurable interface, tough slightly complicated.

For NUMA systems, MemProf[5] uses thread instrumentation and Instruction Based Sampling (IBS) to trace the creation an destruction of objects and help the developer to optimise his code, given the fact that the program is not cache-efficient. The output analysis often leads to simple modification that improve the execution time from 6.5 to 161% in less that 10 lines edited.

An other profiling application is MAQAO[6], developed at the university of Versailles, which also gathers information during the execution of an application; but these data are related to the course of the execution - in order to optimise later the structure of the application - and not measurements of static characteristics of the CPU. But nowadays, CPU complexity is increasing and specifications given by the founder are becoming scarcer and more confusing: from this state of affairs comes the need of a customised benchmark.

For specific micro-benchmarking of CPUs, Agner Fog's experiments[3] are well-known for their accuracy on X86 port occupations, but do not concern memory latency and cache for our architectures.

4 My contributions
Mes contributions

4.1 Modules for Scalevisor
Module pour Scalevisor

4.1.1 Coherent boot of APs
Boot cohérent des APs

When the system initially boots, only one processor is executing the entry code. This processor is named BSP (BootStrap Processor), contrarily to the AP (Application Processors) that have to be woken up following a specific timed sequence of signals. Note that no assumption can be made on the ID[8] of the BSP processor, so hard-coding a 0 value is not a durable solution. Once the AP are launched, they executes their own portion of code. I found that the issue was a wrong member of a C++ class read in the assembly code in the early boot stage of the AP, due to an encapsulation of the former class. It indeed was originally loaded from its the address of a standalone class, but this class was derived from an abstract class following a code re-factoring due to the switch from AMD's server to Intel's. This caused a pointer to the vtable to spawn, adding and offset of 8 bytes to the effective address of the member. We also found that one of the ACPI tables - the SRAT, that gives the processor and memory topology - exposed by Scalevisor was wrong because it was overwritten by the paging table due to insufficient memory reservation at the early boot stage.

Once these issues fixed, I had to write in assembly a simple Hello World multicore test program (see A.1) with a locking mechanism to verify that every CPU was effectively up and running when booting the guest system.

Exception handling
Gestion des exceptions During this implementation, I found that the fault handling mechanism was not working anymore. When a (regular) hardware fault occured, a C++ exception was raised due to the internal architecture of Scalevisor. But this exception was itself raising another (unexpected) error; that lead to an unreadable error report. I found that this was caused by an unaligned data used with a SSE2 instruction called during the internal C++ exception subsystem, which lead me to the conclusion that the stack was misaligned: it should be on a multiple of 16 bytes but was shifted by 8 bytes. This was caused by the lack of push of the very first function pointer, as the `main()` called during the

[8] APIC ID

early boot stage was never added to the stack. Once solved, Scalevisor exception handling subsystem worked as expected.

4.1.2 CPUInfo module
Module CPUInfo

With an intensive use of the virtualization extensions, but also of SSE2/AVX ones, we found that a great improvement of the project could be to centralise the detection and the activation of such extensions in a single class, accessible in every part of the project. The CPUInfo module was therefore born. It consists of a single static class with only `constexpr` members that can easily be updated if a new x86 feature is created. This class is a simple wrapper for the `cpuid` instruction, that feeds it with the necessary value in the EAX and ECX register and parse the given output to deduce whether the functionality is available or not. If needed, a custom activator and de-activator can be specified so that a vendor-agnostic routine can be called when the feature is needed in other part of the code. Finally, the programmer can also dynamically log the available features aside with a quick description of their role at boot time, as Linux does with its own module.

4.1.3 PEBS module
Module PEBS

Instruction Based Sampling (IBS) is a convenient way to gather data from a running program without modifying the underlying code. It consists in taking regular snapshots of the system state[9] every fixed occurrence of a given event. For example, we can log one memory load each thousand and thus gather an estimator of the actual data being treated by the program. Applied to Scalevisor, the idea is - at term - to gather miss locations (level of cache and/or physical location in RAM) in order to efficiently move threads at scheduling time.

Intel IBS implementation is called *Precise Event Based Sampling* (PEBS). It is particularly fast as it is directly present in the hardware, at the microcode layer: no interrupt is raised when a collection event occurs; which is a major advantage compared to manual stops or analysis of the code. The downside of a hardware implementation resides in the fact that it is controlled by a set of ill-documented MSRs that varies with the CPU generation and often breaks backward compatibility, that has necessarily consequences on the readability of PEBS-wrapping code.

Intel PEBS is in fact implemented as a layer over the standard *Performance Monitoring Counters* (PMC). Nowadays processors are equipped with 8 PMC per physical core, that are divided into 4 for each logical core when HyperThreading is enabled. Each PMC only

[9]The snapshot consist on register values and a few information of the instruction that triggered the log

counts the number of occurrence of a given event. This event is set in an associated register, that also states whether an interrupt should be raised in case of overflows. Only a subset of the PMC's events are compatible with PEBS; but the configuration is rather simple. The dumped information are written each time the counter overflow; then the counter is reset to a fixed value configured in the PEBS configuration MSR. Note that the initial value present in the PMC register has to manually be set.

My module is a simple wrapper of the PEBS interface. It hides the number of PMCs to the programmer for simplicity reason; enabling PEBS just needs to state the event to log and the frequency of sampling. Of course, a trade-off has to be made, as a higher sampling rate gives more precise information about the running program but is also slower, both because of the logging (that is not instantaneous) but also because of the analysis that will be made at schedule time.

My module also includes a C++ iterator with its corresponding C++ classes wrapping the PEBS logging space in order to treat and reset the snapshots in a readable way. As up to 4 counters can be used simultaneously but all of them share the same logging space, the traversal was optimised for a faster execution to the detriment of memory complexity.

4.2 Microbench: a memory latencies measurement tool for NUMA systems
Microbench : un outils de mesure de latence mémoire et cache pour systèmes NUMA

4.2.1 General Presentation
Présentation Générale

Microbench is a small tool that aims at measuring cache and memory latency on NUMA machines. It is freely available on Gitlab[10] and support both Linux and OSX (Windows can be also targeted through the Windows Subsystem for Linux, using the bash terminal), written entirely in C with dedicated tools to output a R graph from the measurements.

4.2.2 Quick overview of the targeted architectures
Présentation rapide des architectures cibles

Intel Our Intel server is a quad-socket Skylake Scalable Platform machine, see figure 4. Each processor is a Xeon Gold 6130 featuring 16 actives hyperthreaded cores, connected in a complete graph by 3 Ultra Path Interconnect[11], for a total of 128 threads. The die used

[10] `https://gitlab.com/NicolasDerumigny/microbench`

[11] UPI is a bi-directional low-latency coherent interconnect, with an operating speed of 10.4 GT/s.

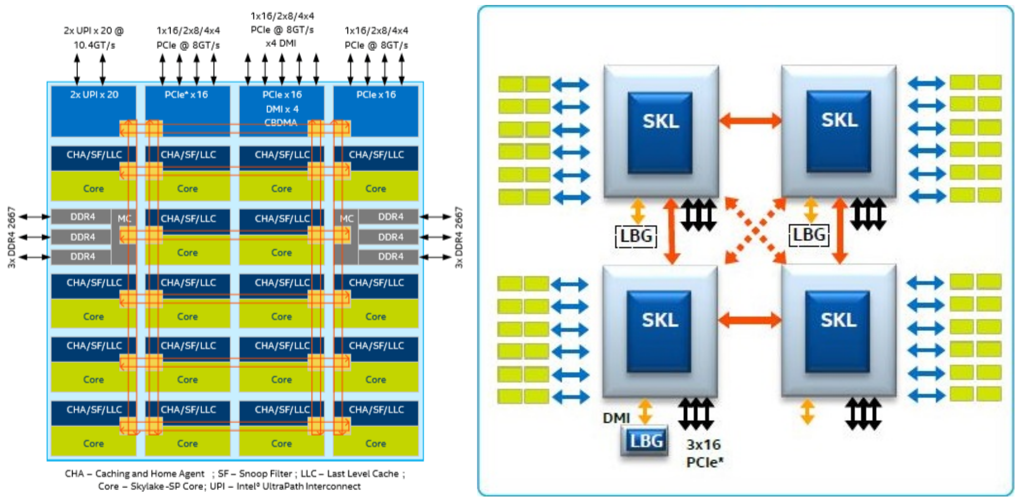

2x UPI x 20 @ 10.4GT/s
1x16/2x8/4x4 PCIe @ 8GT/s
1x16/2x8/4x4 PCIe @ 8GT/s x4 DMI
1x16/2x8/4x4 PCIe @ 8GT/s
2x UPI x 20
PCIe* x 16
PCIe x 16 DMI x 4 CBDMA
PCIe x 16
CHA/SF/LLC
Core
DDR4
MC
3x DDR4 2667
CHA – Caching and Home Agent ; SF – Snoop Filter ; LLC – Last Level Cache ; Core – Skylake-SP Core; UPI – Intel® UltraPath Interconnect
SKL
LBG
DMI
3x16 PCIe*

Figure 4: Logical view of the Intel Scalable Platform die (High Core Count). On our CPU, 2 cores per die are disabled to reach 16 cores per die. On the right, schematic view of the 4-die topology of our machine.

is the Skylake-SP's Mid Core Count with two cores disabled out of the 18 initially present. The Skylake micro-architectures introduces a new *mesh* in-die topology, and redesigns the cache distribution with 64 KB of private L1 (data cache)[12], 256 KB of private on-chip L2, 768 KB of off-chip private L2 (see part 4.2.6 for more details about this partitioning) and 22 MB of shared L3 (1.375 MB per core).

AMD Our AMD server is a dual-socket EPYC machine (see figure 5); but each socket is composed of four distinct dies. These dies named *Zepplin* are themselves composed of two Compute Clusters (CCX). Finally, a CCX contains 4 (multi-threaded) cores and 4 MB of shared L3 cache, but the cache is also shared inside the Zepplin die. Each core features also 512 KB of private L2 cache and 32 KB of L1 (data cache)[13]. The link used between cores and sockets is AMD's *Infinity Fabric*; a low-latency coherent point-to-point interconnect designed with for scaling purposes. The throughput of this fabric reaches 42 GB/s with bidirectional link. Our machine uses two EPYC 7451 CPU with 24 cores supporting SMP, for a total of 96 threads, one core being disabled per CCX.

[12]And an additional 64 KB of instruction cache
[13]With also 64 KB of instruction cache

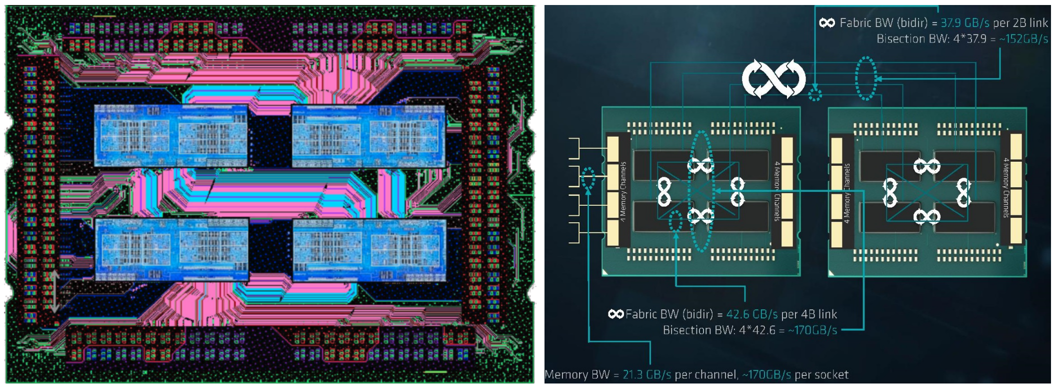

Fabric BW (bidir) = 37.9 GB/s per 2B link
Bisection BW: 4*37.9 = ~152GB/s
4 Memory Channels
4 Memory Channels
4 Memory Channels
4 Memory Channels
Fabric BW (bidir) = 42.6 GB/s per 4B link
Bisection BW: 4*42.6 = ~170GB/s
Memory BW = 21.3 GB/s per channel, ~170GB/s per socket

Figure 5: Die shot and commercial presentation of the AMD EPYC micro-architecture. Note the Infinity Fabric (in pink on the die shot), the four Zepplin dies on one socket and the two CCX per die.

4.2.3 Structure
Structure

The basic idea is rather simple. We just aim at measuring the time taken to complete a single load. The idea is therefore to store in an array the next memory cell to load, so that we can reuse the content of this cell directly in the next instruction, avoiding extra instructions. We also use the `RDTSCP` instruction that empties the pipeline once issued in order to avoid extra latency due to other instructions. The assembly code used is available figure 6.

But nowadays processors are equipped with prefetching mechanisms, that are able to detect and predict linear accesses of the main memory. Therefore, we need to load a new random address after each load, so we generate this random address before the timed section. One other issue was that a random address can lead to a cycle, for example loading first `a[0]`, then `a[32]`, then `a[26]` that point again to `a[0]`, so that only three different addresses are loaded, fitting therefore in the L1 cache even if we aimed a measuring the L3 or RAM accesses. In that goal, we generate a one-cycle permutation of the array that we load using the algorithm detailed in 1.

4.2.4 Measuring cache transitions and NUMA effects
Mesure des transitions de cahces et de l'effet NUMA

NUMA effect The NUMA effect is relatively simply to measure. The only issue relies in being sure that the data we load comes effectively from a chip that is linked with a distant memory controller. For this fact to be true, we first use the algorithm 1 to generate a one-cycle array, then we pin our process to the desired core using the `sched_setaffinity()` function. Then, we copy it using the `memcpy()` function to a fresh memory location allocated

```
Input: $n$ the dimension of the array
Output: $T$ an array of size $n$ with a one-cycle permutation
1  $T \leftarrow [-1, -1, ..., -1]$ ;                                  // $n$ elements
2  $first \leftarrow rand(0, n)$ ;                  // the starting point of our generation
3  $former \leftarrow first$ ;                                // the next element to fill
4  for $i \leftarrow 0$ to $n-1$ do
       // Generation of the address to put in $former$
5      $j \leftarrow rand(0, n)$;
       // It should be random, nor $first$ nor $former$ nor anything already
          filled
6      while $n[j] \neq -1 \vee j \overset{?}{=} 0 \vee j \overset{?}{=} former$ do
7          $j = j + 1 \mod n$;
8      end
9      $T[former] \leftarrow j$;
10     $former \leftarrow j$;
11 end
   /* Conversion from indexes to addresses                                      */
12 for $i \leftarrow 0$ to $n-1$ do
13     $T[i] \leftarrow (T[i] * sizeof(uintptr\_t)) + T$;
14 end
```
Algorithm 1: Generating a one-cycle permutation array

```
Input: R9, a register containing the address of an element of a one-cycle array
Output: EAX, a register containing the number of cycle taken to execute the loads
1 RDTSCP;
2 MOVL %EAX, %R10;
3 MOVL %EAX, %R11 ;    // saving timestamp as the beginning of the loads
4 MOV (%R9), R9;
5 MOV (%R9), R9;
6 MOV (%R9), R9;
  // 8192 hardcoded MOV for the L1/L2/L2b, 32768 for the L3/RAM
7 RDTSCP;
8 SUBL %R10, %EDX;
9 SUBBL %R11, %EAX ;  // substracting timesamps to obtain the numbner of
   cycles elapsed
```

Figure 6: Assembly measurements of the cycles taken for memory loads

with the `mmap` function and the MAP_PRIVATE | MAP_ANON arguments so that memory is allocated following the first-touch policy on the accessing node, which happens during the `memcpy`. Finally, a simple double-for loop allows us to iterate amongst all allocater/loader couple.

Cache coherency protocol Multicore processing has lead to new challenges, including cache coherency protocol. Depending on their replication and their meanings, the cache lines are labelled with a state. The basic protocol MSI consists in three states: Modified, Shared, Invalid, but has been improved over time to the MESIF one that we time. To time some of these transitions, we set the cache in a specific state, the load data we know fit in the cache from another processor and time these load. In order to have accurate measure, we do not want to load two times the same data as they will not return back to their original location - which is the case when we just aim at loading from the processor's own cache. Moreover, the processor does not load a single cache line but a more general chunk of data of unknown size when a miss occurs, that is why reducing the number of accesses was crucial in cache coherency runs.

4.2.5 Issues and mitigations
Problèmes et solutions

Noise due to the system

Issue Running a benchmark always comes up with the issue of the accuracy of the measure. The host system has indeed always background tasks such as a timer that regularly send interrupts to the cores in order to verify their (online) state. As such an interrupt cost a few hundred cycle and is repetitive, we aims at removing as much as possible those perturbations.

Mitigation When running the benchmark, we switch the scheduler to a FIFO policy and set the benchmark to the highest priority, so that we are sure never to be interrupted. Additionally, all measurements are repeated 32 times, alternating the first core to be tested amongst the nodes, and only the minimum number of cycle is kept[14]. Indeed, there is no physical reason for the hardware to be faster for random loading, but software interrupt can lengthen the measures.

[14]For the RAM benchmark, we chose to keep the median of the measurements due to efficient prefetching techniques that successfully fits in the L3 cache some of the access

Fitting data in the cache

Issue As cache are sometimes inclusive, we are not sure that the data we load comes effectively from the targeted cache. Due to either cache eviction protocols or prefetching, data may be located in nearer or farther cache than expected.

Mitigation To avoid eviction, we always take an array smaller than the size of the cache we want to measure, for example 128 kB for a cache of size 256 kB. Moreover, we always run ten warm-up turns in order to be sure that data is indeed already present in the cache before running the measurements, except for the RAM timing, because warm-up would fill the cache and have the opposite effect on the measurements.

DVFS

Issue The trickier issue comes from the Dynamic Voltage-Frequency Scaling (DVFS). First, the Turbo Boost: it allows the processor to increase its frequency when running a higher load, so that the latter finishes faster. This is fairly simple to deactivate, as one MSR field is dedicated to this usage. However, the processor also enters an idle state when not used, lowering its frequency to 800 MHz in order to limit power consumption and heat. When launching a task, the processor takes a few milliseconds to switch to the base frequency, thus influencing the results.

Mitigation The RDTCP instruction count cycles elapsed by the main core clock regardless of the effective core frequency, so it was mandatory to fix the CPU to a constant speed. This is usually done by switching governor to performance, but with high-end CPUs, this behaviour is not observed[15] - probably due to their high core count per socket. To reduce the effects of frequency variation, we repeat the measurement 32 times and take either the mean or the minimum of the result - depending of the cache level we aims at measuring. Moreover, we experimentally remarked that only the first measure for each socket was slowed by the DVFS switch, so we alternate the first core to measure inside the socket to get rid of this effect.

4.2.6 Results
Résultats

[15]On the Intel platform, both the Intel-Pstate and the legacy ACPI driver failed to manually set the frequency to a fixed state, whichever tool were used. One the AMD's one, the default controller also not responds to standard governor commands, probably because of its recent release.

[16]Including the 128 kB of on-core L2

[17]Combined with the use of the other caches.

Cache level	Size of the cache (in kB)	Array size used for measurement (in kB)	Latency (local/distant) (in CPU cycle)
L1	32	2	4
L2	256	128	10
Off-core L2	1024[16]	358	13
Local L3	1408 per core	4096	18[17]

Figure 7: Access Latency for the Intel Server

Intel The mesh architecture seems really efficient on the Intel server, as we never observe any die-level NUMA effect. We suspect that the 10 warm-up runs runs before issuing the real benchmarks are enough for the CPU to fit the accessed data in the L2 without being evicted later. The ideal solution would be to increase the number of loads, but due to time constraints, this was not tested yet. The latency for the other caches are quite coherent - see figure 7, with an emphasis on the ”off-core L2”, that we suspect of simply being reused silicon from the shrinking of the L3. Indeed, in the former Broadwell generation, the L3 offered reached 2.5 MB per core, whereas Skylake is only equipped with 1.375 MB!

The RAM latency, represented in figure 8 is much more interesting. We materialise the well-known NUMA effect, with a latency for local accesses of 220 cycles, versus 360 for remote accesses. The mesh structure of a big monolithic die take here all its interest, as access time to the memory controller can be centralised, contrary to the observation for the AMD machine (cf next paragraph). We nevertheless observe verticals patterns for the load latency, corresponding to the time needed for the value to pass from the controller to the on-chip core. We can therefore conclude that the disabled core is not always the same along the dies, but this is not surprising are the disabled core is often a defective one from the silicon etching, thus not controlled.

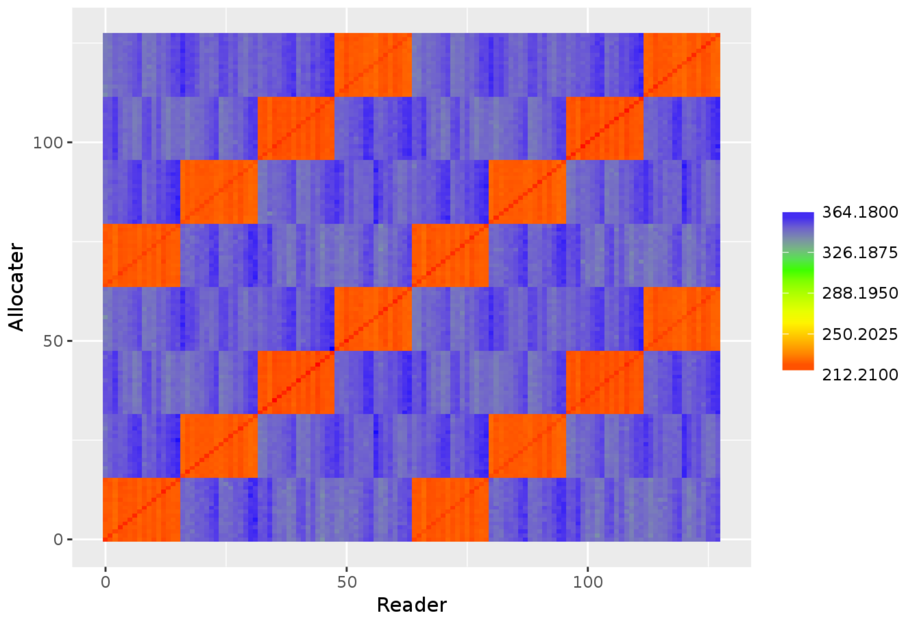
AMD Despite some widely spread belief, the AMD architecture was more efficient on cache access that the Intel's one, as represented in figure 9[19]. Once more, no NUMA effect were detected on random access over the L3 cache, which is probably due to the same causes that on Intel's chip.

The RAM latency is quite surprising. We knew that a die-local load would be the fastest (185 cycles), followed by a socket-local load (300 cycles), and finally a socket-distant load (630 cycles). But two effects were not expected:

- Even on one die, the ”remote” load from an other CCX is around 20 cycle slower. This may be the time needed to consult the L3 of the other CCX, and then requesting from

[18]Combined with the use of the other caches.

[19]But one should not forget that the effective latency is also linked to the frequency!



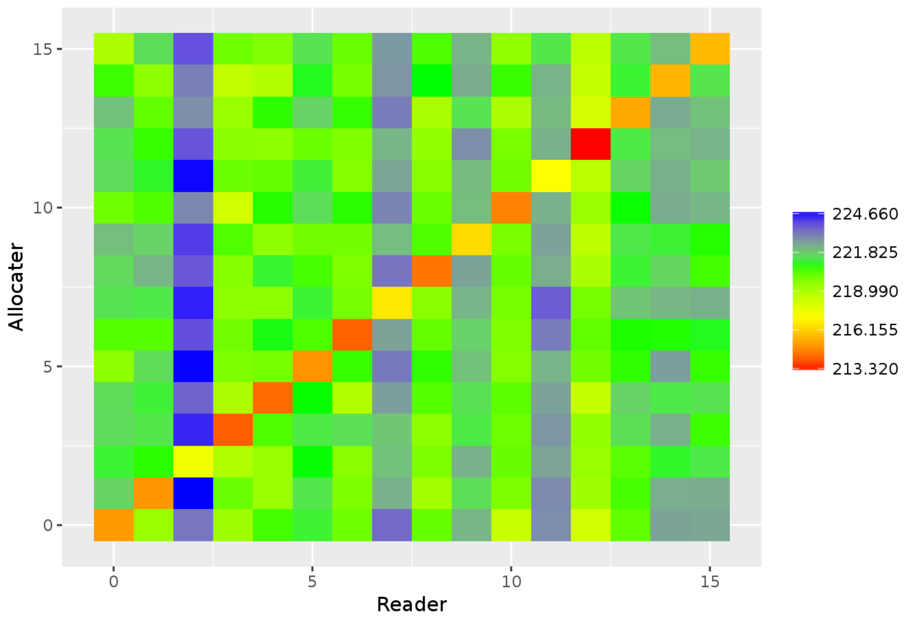


Figure 8: Latency (in CPU cycle) of RAM loads depending on the allocator and the loader, for every core including SMT ones on the top, and only for one die on the bottom. The physical cores expands from 0 to 64, grouped by socket (cores 0 to 15 for the first socket, 16 to 31 for the second, etc).

Cache level	Size of the cache (in kB)	Array size used for measurement (in kB)	Latency (local/distant) (in CPU cycle)
L1	32	2	3
L2	512	128	7
Local L3	4096 MB per CCX	4096	28[18]

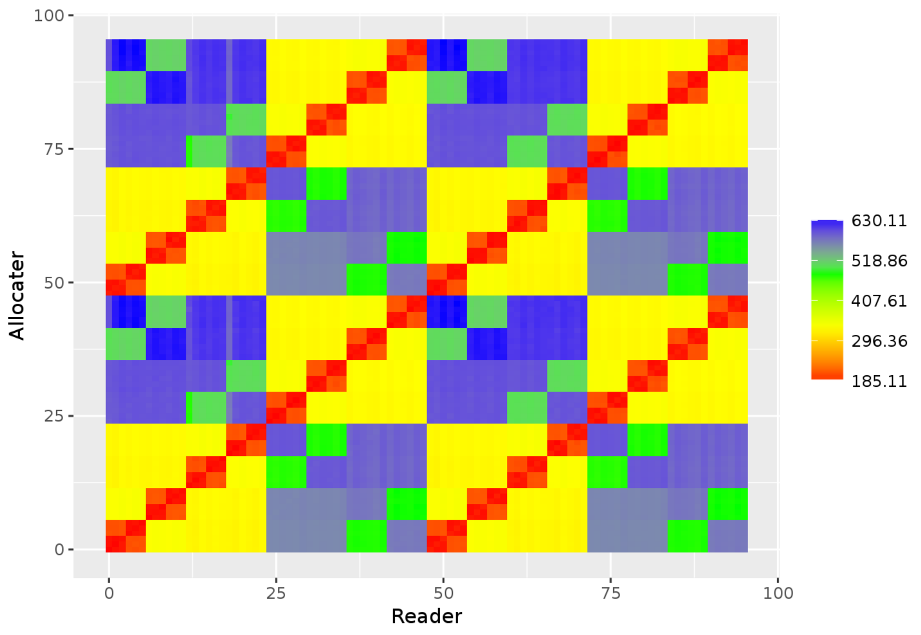
Figure 9: Access Latency for the AMD Server

the memory.

- The naive idea of the dual-socket EPYC architecture is a tree : two sockets of 4 dies of 4 CCX of 3 cores (excluding the SMT dual threads). This measure reveals that it is false: the connection between sockets are die-to-die ones and not centralised. That means that each die has a preferred accessed correspondent on the other die, translating in a green rectangle on the figure 10. When loading from a DIMM linked with the latter's controller, the latency reaches 480 cycles, whereas a distant load from the other DIMMs take up to 630 cycles. When tweaking the scheduler and the allocation policy, this piece of information will reveal very useful, as it means that a CPU on a socket A has also more affinity with some memory location controlled by socket B.

Sequential accesses and cache coherent protocol For both processors, a prefetcher is implemented in order to improve linear accesses within an array. The latter seems very efficient, as in the two cases, sequential loading takes the same latency than L1.

The cache coherent protocol measurements are still under development. It is highly plausible that the relatively high number of accesses causes collisions, i.e. that the underlying cache loading subsystem has already changed the state of a cache line when loading it. Indeed, modern processors are known to load a chunk of several contiguous bytes in the cache when loading a variable, as this technique often outputs very good empirical results. Due to the paradox of the anniversaries, the CPU may have already loaded the line we aims at measuring when timing the latency, hence a wrong result. We also have a similar effect when trying to load a variable from an other NUMA node: no cache coherency protocol should be used apart invalidating the owner and loading directly from the RAM the content. But while loading, the chunk of memory brought on-th-fly by elapsed load in the cache also leads to cached loads instead of distant access to DIMMs, that is the reason why these results are not significant for off-NUMA-node values. An example of the type of graph obtained in such case is represented figure 11.



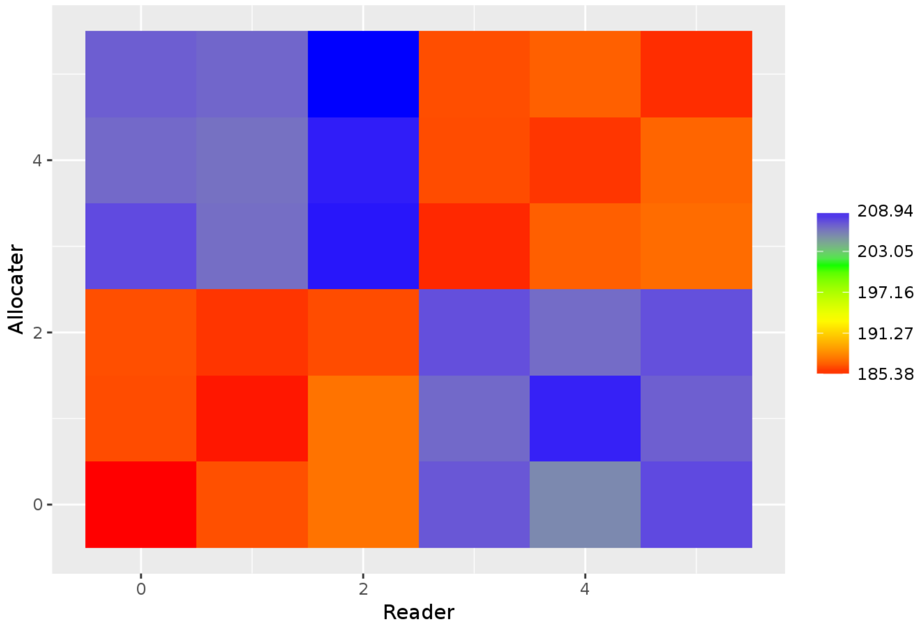


Figure 10: Latency (in CPU cycle) of RAM loads depending on the allocator and the loader, for every core including SMT ones on the top, and only for one die on the bottom. The physical cores expands from 0 to 48, grouped by CCX (cores 0 to 3, 4 to 6, ...); by die (core 0 to 7, 8 to 15, ...) and by socket (core 0 to 23, 24 to 47, ...).

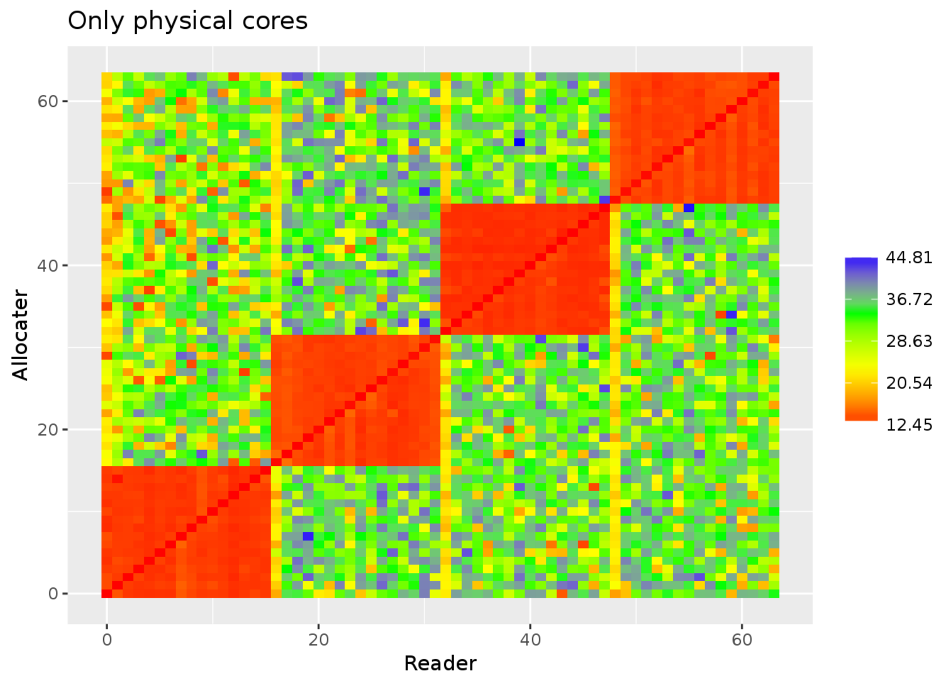


Figure 11: Typical example of cache coherency latency: switching from Shared to Forward state on the L2b cache on the Intel platform, showing only physical cores, distant loads being not significant.

5 Conclusion

Conclusion

Even though this internship falls only in the preliminary stages of a far more ambitious project, a few teachings boil down from it.

First and foremost, the hardware on which every application is run is inherently complex; not only by its structure but also by its implementation and compatibility constraint. My CPUInfo module is designed to ease this complexity when trying to enable a feature, whereas the PEBS module grants the programmer with a more usual API to use Instruction-Based Sampling on Intel's CPU. More generally, Scalevisor's goal is all about getting part of the complexity of the CPU out of the kernel to a dedicated module.

My other contribution, Microbench, has been built in the goal of presenting part of the hardware complexity (the memory one) to the programmer, so that optimising for a specific microarchitecture can be realised based on quantitative values and not experimental measurements gathered at runtime. The results outputs coherent behaviour, but more features are still in development, especially concerning cache-coherency latency measurements.

A Annex
Annexe

A.1 Assembly mutex on bootsector
Mutex assembleur sur le secteur de démarrage

```
        .globl _start
        .code16
_start:
        jmp entry

mutex:
.word 0x1 ;initialisation of the mutex to 1



entry:
        xor %cx, %cx
        xchg (mutex), %cx ;exchange the mutex to 0
        test %cx, %cx ;loop over the mutex
        jnz print
        jmp entry



print:
        mov $msg, %bx ;load the message
        add %cx, %bx
        ;load the character corresponding to the value
        ;of the mutex
        sub $1, %bx
        mov (%bx), %al
        mov $0x0e, %ah
        mov $0x00, %bh
        mov $0x07, %bl
        int $0x10 ;BIOS interrupt to print and configuration
        add     $1,%cx  ;increase the mutex
        xchg (mutex), %cx ;re-set the mutex to an non-zero value
        jmp end

end:
        jmp end

msg:
        .ascii "Hello World and very long message in case of\0"
```

References

[1] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Trans. Comput. Syst.*, 15(4):412–447, 1997.

[2] Mohammad Dashti, Alexandra Fedorova, Justin R. Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quéma, and Mark Roth. Traffic management: a holistic approach to memory placement on NUMA systems. In Vivek Sarkar and Rastislav Bodík, editors, *Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, TX, USA - March 16 - 20, 2013*, pages 381–394. ACM, 2013.

[3] Agner Fog. Software optimization resourcesd Blog. `https://www.agner.org/optimize/blog/`.

[4] Lokesh Gidra, Gaël Thomas, Julien Sopena, Marc Shapiro, and Nhan Nguyen. Numagic: a garbage collector for big data on big NUMA machines. In Özcan Özturk, Kemal Ebcioglu, and Sandhya 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.

[5] Renaud Lachaize, Baptiste Lepers, and Vivien Quéma. Memprof: A memory profiler for NUMA multicore systems. In Gernot Heiser and Wilson C. Hsieh, editors, *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*, pages 53–64. USENIX Association, 2012.

[6] P. Sadayappan, Manish Parashar, Ramamurthy Badrinath, and Viktor K. Prasanna, editors. *High Performance Computing - HiPC 2008, 15th International Conference, Bangalore, India, December 17-20, 2008. Proceedings*, volume 5374 of *Lecture Notes in Computer Science*. Springer, 2008.

[7] Gauthier Voron, Gaël Thomas, Vivien Quéma, and Pierre Sens. An interface to implement NUMA policies in the xen hypervisor. In Gustavo Alonso, Ricardo Bianchini, and Marko Vukolic, editors, *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*, pages 453–467. ACM, 2017.