Diploma Thesis

Memory and Thread Management on NUMA Systems

Daniel Müller

Monday 9th December, 2013

Technische Universität Dresden
Faculty of Computer Science
Institute of Systems Architecture
Chair of Operating Systems

Supervising professor: Prof. Dr. rer. nat. Hermann Härtig
Supervisors: Dr.-Ing. Michael Roitzsch
Dipl.-Inf. Carsten Weinhold

TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik Institut für Systemarchitektur, Professur für Betriebssysteme

Aufgabenstellung für die Diplomarbeit Dresden, 4. Juni 2013

Name des Studenten:	Müller, Daniel
Studiengang:	Informatik
Immatrikulationsnummer:	3487204
Thema:	Speicher- und Thread-Verwaltung auf NUMA-Systemen

In Mehrprozessor- und Mehrsockelsystemen sind die Entfernungen zwischen CPUs und Speicherbänken nicht mehr uniform, stattdessen sind einige Teile des Arbeitsspeichers über mehr Indirektionen zu erreichen als andere. Dies führt zu unterschiedlichen Zugriffslatenzen. Für optimale Performance muss die Speicherverwaltung des Betriebssystems die Hardware-Topologie und Zugriffsmuster von Anwendungen berücksichtigen.

Im Rahmen der Diplomarbeit sollen Speicherverwaltung und/oder Scheduler auf Fiasco.OC und L4Re so angepasst werden, dass eine möglichst gute Gesamtleistung auf einem NUMA-System erreicht wird. Denkbar sind dabei sowohl die Migration von Threads zwischen CPUs, als auch die Migration von Speicherinhalten zwischen den Speicherbänken verschiedener Sockel. Als möglicher Ansatz für die Realisierung bietet sich die Analyse von Performance-Monitoring-Units und gemessenen Ausführungszeiten an.

Für die Bereitstellung notwendiger Topologie-Informationen können Ad-Hoc-Lösungen eingesetzt werden. Das Hauptaugenmerk der Arbeit liegt auf Strategien zur automatischen Analyse und Platzierung von Anwendungslasten und deren Daten. Dafür ist eine geeignete Anwendung auszuwählen anhand derer verschiedene Platzierungsstrategien untersucht werden. Mindestens eine Strategie soll implementiert und deren Effektivität evaluiert werden.

verantwortlicher Hochschullehrer:	Prof. Dr. Hermann Härtig
Betreuer:	Dipl.-Inf. Michael Roitzsch
	Dipl.-Inf. Carsten Weinhold
Institut:	Systemarchitektur
Beginn:	10. Juni 2013
Einzureichen:	9. Dezember 2013

Unterschrift des betreuenden Hochschullehrers

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich diese Arbeit selbstständig erstellt und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Dresden, den 9. Dezember 2013

Daniel Müller

Acknowledgements

Foremost, I would like to thank Prof. Hermann Härtig for giving me the opportunity to write this thesis. His lectures and seminars provoked my interest in operating systems and low-level system software in the first place. The possibility of working as a student assistant at his chair gave me many interesting insights in computer science related topics but also taught me lessons in thinking outside the box.
My supervisors, Dr. Michael Roitzsch and Carsten Weinhold, provided me with help and input whenever I needed it along my journey of writing this thesis, for which I am very grateful. In addition, I would like to thank Björn Döbel for his guidance before and during my internship as well as Adam Lackorzyński and Alexander Warg for ideas regarding implementation matters and the insights they gave me into kernel and user land of our L4 system.
Last but not least, my thanks go to Gesine Wächter for criticism and comments on my version of English—her helpful and patient nature helped me in various situations.

Contents

List of Figures

List of Code Listings

List of Tables

List of Algorithms

1 Introduction

1.1 Motivation

In the past, we saw a steady increase in transistor count over time due to decreasing feature sizes and more sophisticated development processes: the number of transistors on an integrated circuit doubled approximately every two years. This phenomenon, later dubbed *Moore's Law* after an engineer at Intel Corporation, was found to be a reliable way to forecast development of chips as well as systems based on them [Moo00, Cor05]. At first, these additional transistors were mainly used for increasing single core performance by raising clock frequencies, building larger caches for masking the inability of memory to keep up with CPU speed, and implementing prefetching units as well as advanced out-of-order and execution-prediction logic. For the application programmer, things did not change in significant ways: without programmer intervention programs executed faster.

However, due to physical constraints like the speed of light and maximum heat dissipation per area, a continued increase in clock frequency and ever more complexity could not be incorporated into a single core. Research and industry found ways to *scale out*: the newly available transistors were used for placing more processing cores onto a single chip. In such SMP (Symmetric Multiprocessing) systems, all cores are considered equal: they have similar processing speeds and access to memory as well as peripheral hardware is relayed over a *shared bus* without preference of any component.

Unfortunately, adding more execution units in such a uniform fashion is impossible without sacrificing performance: every new core causes more contention to shared resources and infrastructure, especially the shared memory bus and the—in direct comparison—slow memory subsystem. As a result, contended resources got duplicated as well, resulting in multi-processor systems with more than a single main-processor or multi-core architectures with several memory controllers. In both cases the result is comparable: components located physically closest to a certain processor can be accessed fastest for they are *local* to it. *Remote* accesses on the other hand, i.e., requests to a processor further away, experience a slow-down due to additional communication and longer travel times of the electric signal. Non-Uniform Memory Access (NUMA) architectures were born.

NUMA systems—unlike SMP or Uniform Memory Access (UMA) systems—pose more challenges to programmers: in order to achieve maximum application performance, the peculiarities of the architecture have to be understood and incorporated into resource allocation decisions.

In this thesis, I examine the influence which this type of non-uniformity has on the memory subsystem as well as the impact of shared resource contention. In a first step, I investigate possible effects on software performance under different workload scenarios. Based on these insights, I design and implement an infrastructure for making the Fiasco.OC microkernel and L4 Runtime Environment (L4Re) NUMA-aware by incorporating the system's topology and architecture into memory allocation decisions. As systems are dynamic, I furthermore analyze the impact of thread scheduling and memory migrations to cope with imbalances when they arise.

1.2 Document Structure

This work is structured as follows: chapter 2 introduces the fundamentals that are required to understand this thesis. In this chapter, I represent NUMA systems and the hardware involved and take a closer look at important memory related concepts. On the software side, I outline the mechanisms used in Linux to gather and then incorporate knowledge about the underlying NUMA architecture in decisions regarding the assignment of resources to processes. I also introduce the L4 microkernel and an important building block for user space applications: the L4Re. Last but not least, this chapter also gives an overview about other work done in the field of or otherwise related to NUMA architectures.
Chapter 3 depicts my approach for making the system NUMA-aware. First, I quantify the effects of non-uniform memory access times and shared resource contention on software performance. Thereafter, I introduce the NUMA manager—a component that intercepts thread placement and memory allocation requests and enforces certain placement policies itself. Finally, I integrate the NUMA manager into the L4 system. I explain necessary changes to kernel and user land allow this component to provide its service.
I evaluate my NUMA manager service in chapter 4. The first problem I investigate is the overhead induced by the approach of object virtualization that I followed. The degradation of performance caused by thread and memory migrations is also part of the evaluation. Last but not least, I examine what benefits my component provides in different workload scenarios.
Chapter 5 concludes my thesis by providing an outlook for future work to be done on this and related topics and summarizing the achieved goals.

2 State of the Art

This chapter explains the basics necessary to understand this thesis. First, I give an overview about NUMA architectures and the involved hardware. I discuss their advantages and problems that engineers have to cope with. Section 2.2 dives into memory architectures and memory management in today's systems, differentiating between virtual and physical memory. The following two sections, 2.3 and 2.4, describe what the two operating systems Linux and L4 offer in terms of NUMA awareness and support. Since my work is based on L4, I also provide a deeper architectural overview of it. The final part, section 2.5, sums up research targeting NUMA architectures. Therein, I represent work that investigates problems such architectures may cause, how to spot these problems, as well as different approaches on how to solve them.

2.1 NUMA

This thesis is concerned with *NUMA* systems. In order to understand later design and implementation decisions as well as evaluation results, this section explains the required details. Furthermore, it provides an overview about the typical hardware components that are of interest in the context of this thesis and establishes a common terminology.

2.1.1 NUMA Architectures

Today's server systems are typically made up of multiple CPUs linked to each other. Each of these processors is connected to a portion of physical memory that is said to be *local* to it. On the other hand, there is *remote* memory—memory that is, from one CPU's viewpoint, connected to a different CPU. Remote accesses, i.e., reads from or writes to memory that is local to another processor, experience penalties in the form of lower throughput and higher latencies compared to local accesses. This decreased performance stems from the additional communication between CPUs that is required to fulfill remote requests. Architectures that exhibit this difference in memory access times are referred to as *NUMA* architectures. Figure 2.1 on the following page illustrates a NUMA system with four CPUs, each featuring four cores and the corresponding memory local to it.

In the following, I refer to a processor and the memory local to it as a *NUMA node* or simply *node* for short. This terminology is consistent with that of other authors [LBKH07, KSB09, BZDF11, MG11a].[1]

[1] Sometimes but less often, the term *socket* is found for this conglomerate of CPU and memory in the literature—both terms are used interchangeably.

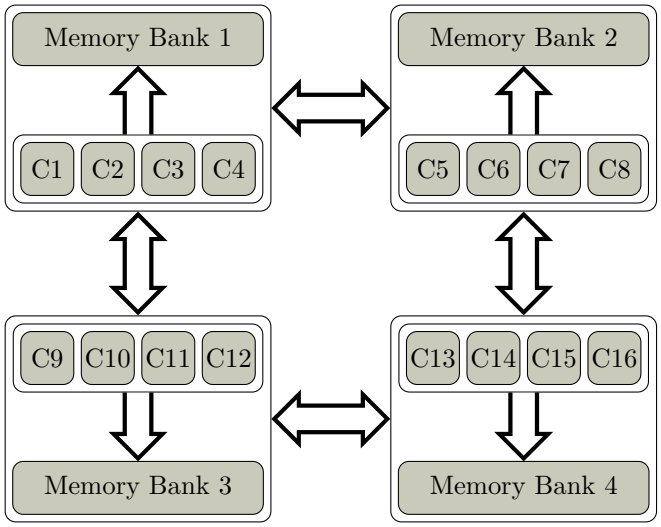


Figure 2.1: NUMA System Featuring Four CPUs

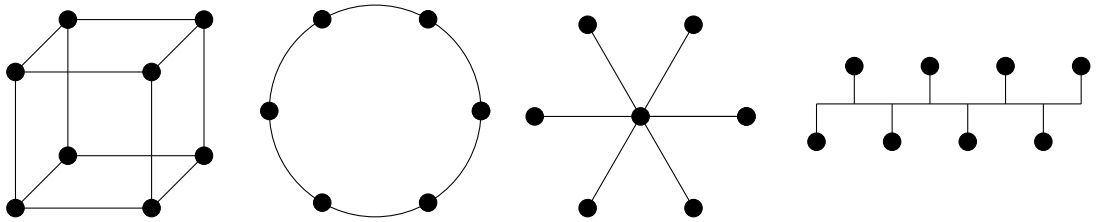
2.1.2 Interconnects

As explained above, the notion of remote memory arises from the fact that memory which is connected to a certain processor cannot only be accessed from that CPU but also from different CPUs within the system. This access has to be possible because established and most commonly used programming languages and environments such as C, C++, Java, and Python assume or are designed for a *shared address space*, i.e., one where all executing entities can access all parts of the address space in a uniform way via normal memory operations.
For performance reasons, the means of granting accesses between all memory nodes has to be provided on the side of the hardware. Thus, hardware vendors connect processors among each other in a point-to-point way. This connection is established using *cross-processor interconnects* [OHSJ10].[2]
These *interconnects* allow for remote requests to be fulfilled by asking the remote CPU to read or write a data item from or to its local memory on the requesting processor's behalf. The most prominent CPU vendors, Intel and AMD (Advanced Micro Devices), rely on different mechanisms for this purpose: Intel developed their *QuickPath Interconnect (QPI)* [Coo09, Cooa] and AMD licensed *HyperTransport (HT)* [Con04, AMD13] for their products. For HPC (High Performance Computing) systems, the no longer existing company Sun Microsystems offered the *Fireplane* interconnect [Cha01].

Topologies

On small systems, each CPU is typically connected directly to every other CPU in the system in a point-to-point fashion—they are fully meshed. As connection networks on HPC clusters and other large scale networks have shown, such an approach does not scale for larger networks as the number of required interconnects is quadratic in the

[2] Occasionally, the term interlink is found in the literature [Lam06a].



(a) Hypercube **(b)** Ring **(c)** Star **(d)** Bus

Figure 2.2: Selection of Possible Topologies

number of nodes in the system. To cope with these scalability issues, nodes can be arranged in certain *topologies* [Dre07b]. Common topologies are, for instance, full mesh, tree, crossbar, ring, bus, star, and hypercube.
Figure 2.2 depicts four topologies. Numerous research projects have investigated the influence of topologies on network properties such as latency, scalability, reliability, and evolvability [RW70, YS80, Gav92, CY04].

Today, commodity systems exist that are not fully meshed but where some of the remote CPUs are only accessible through more than a single *hop*, i.e., using one or more intermediate steps over other processors. With larger topologies that are not fully meshed, inequalities in access times increase as more hops are required to reach a CPU. In the ring topology as illustrated in figure 2.2b, for instance, three hops are necessary for one processor to communicate with the opposite one. An example for such a system using a ring topology is the Celestica A8448 with four AMD Opteron CPUs [AJR06].

NUMA Factors

To quantify the overhead imposed for communication between two CPUs, I employ the notion of a *NUMA factor*. This term is also commonly found in the literature [Dre07b, PRC+11, MG11b, SLN+11]. The NUMA factor approximates the relative slow-down an access from one node to another one experiences. A factor of 1.0 describes local accesses and higher NUMA factors imply higher overhead, i.e., remote accesses.
As an aside, another term that is sometimes used in a similar context is the *NUMA distance* or simply *distance* [ANS+10, MA12]. As for the NUMA factor, a higher distance entails a higher access penalty but the exact meaning is not as clearly defined.[3]
Typical NUMA factors on today's systems range between 1.0 and 2.0 [MG11a]. For an AMD Opteron system for example, a NUMA factor of 1.4 was reported by Kupferschmied et al. [KSB09]—meaning that off-node accesses experience a performance degradation of 40 %.

NUMA factors might increase significantly with increased load on the system, specifically on the interconnect. AMD has investigated this problem in more detail for an Opteron

[3] ACPI also uses the term distance in the context of NUMA systems [CCC+06], however, it should not be seen as the actual penalty factor for remote accesses but more as an information that memory is closer to one processor than to another—most of the time it is simply too imprecise or even wrong [Dre07a, Dre07b].

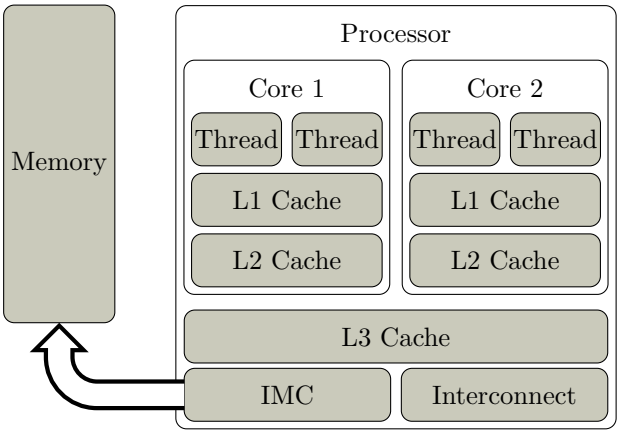


Figure 2.3: Overview of a Typical Processor

system with four dual-core processors [AMD06]. The term NUMA factor as I use it, however, is always seen in the context of a system that is only busy with this particular read or write operation required for estimating the access penalty. I examine contention on shared resources separately.

2.1.3 Processors

The previous sections assumed that a single processor or CPU (I use both terms interchangeably here) is the unit of execution. Nowadays, however, a processor is much more complex than that. Figure 2.3 illustrates a typical processor. Its main parts are explained subsequently.

Cores

Being unable to make a single processor ever faster due to physical constraints, multiple cores were placed on a single chip. Each core can be regarded as a mostly self contained unit being able to perform calculations, to access memory, and to execute input and output operations on external devices. However, cores share parts of the infrastructure that exists on the processor, for instance, the physical connection to the rest of the system, i.e., the socket.
Such sharing makes sense in order to reduce costs when adding another core to the processor. If an entire processor was added, the whole infrastructure embedding it onto the motherboard would need to be duplicated, too. Also, as cores are integrated into a processor, distances between cores are lower than those between entire processors (which are naturally located on separate chips). This lower distance, again, allows for higher transfer speeds and lower access latencies among them.

Hardware Threads

A different level where sharing of resources is possible is for the complex infrastructure used for data prefetching, access prediction, and speculative execution of parts of the code, as well as the whole instruction pipeline for performing the execution.
For that matter, the concept of *Simultaneous Multithreading (SMT)* was introduced—a technique for running multiple *hardware threads*. When one hardware thread is waiting for a slow memory or I/O operation to finish, for example, others can issue more instructions in the meantime. This allows the core to continue performing work although one of its threads is currently stalled. The costs for adding a new hardware thread are lower than those for adding a core: mainly the register set has to be duplicated. The gain is a better utilization of the components shared between hardware threads.

The operating system treats hardware threads as the unit of execution: they are the means of executing application (software) threads and the granularity at which scheduling happens. When referring to such a unit, I subsequently use the term *logical CPU*. If a core has no support for SMT it contains a single hardware thread and represents one logical CPU.

Memory Controllers

Yet another resource that is shared is the Integrated Memory Controller (IMC) on today's CPUs.
One of the reasons for NUMA architectures to emerge are the scalability problems of previous *UMA* architectures. On these architectures, all a system's processors share the same memory controller and with it the memory bus connecting them to the machine's entire memory [Coo09]. Everytime one of the CPUs performs a memory access, the bus is locked for that CPU, causing all others trying to access memory concurrently to be stalled until the former request is served. With an increasing number of CPUs or cores, however, this approach does not scale well because bus contention becomes too high.

To work around this issue, processor vendors integrated memory controller and bus into the CPU [Coo09, AMD13] where it is connected to a portion of the system's memory with full speed—the local memory introduced earlier. This IMC is typically shared between all of the processor's cores.

Caches

Even with an IMC and only small distances to main memory, memory still is a bottleneck in today's systems because processors have reached processing speeds memory is unable to keep pace with [WM95, Sut09]. One way to cope with this difference in processing and accessing speeds is the introduction of fast intermediate memories that store data the CPU works on temporarily: caches. A cache is a fast lookup memory that masks the latency for accessing slow main memory.
Hardware-managed caches, as they are used in processors of the x86 architecture [Coo13a], cache data automatically when it is accessed by the CPU: if the datum does not exist in the cache, it is fetched from main memory and stored in the cache until

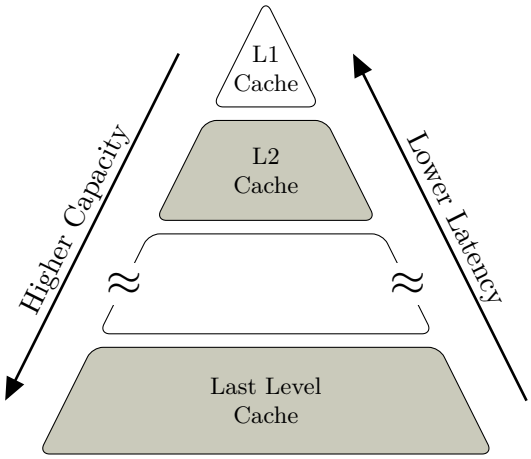


Figure 2.4: Illustration of the Cache Hierarchy

other data evicts it. This process of failing to find a particular datum in the cache is called a *cache miss*. Contrary to a *cache hit*, where the requested datum is found, a cache miss is slower because the main memory has to be accessed. This memory access takes orders of magnitude longer than a cache access. For software-managed caches, data has to be loaded into the cache by software explicitly [EOO+05, BGM+08, GVM+08].
Caches store data in the form of *cache lines*: a contiguous block of bytes, typically a power of two in size. Even if only one byte of memory is accessed, a whole cache line is fetched and stored. Typical cache line sizes are 32 bytes, for instance, on certain processors of the ARM architecture [Lim07], or 64 bytes on x86 processors, e.g., of Intel's Netburst architecture [Coo13b],

In a typical configuration, a CPU comprises multiple caches, forming a *cache hierarchy* [SZ08, OHSJ10]. Caches can be another shared resource on processors. The *First-Level Cache* is local to each core but shared among hardware threads. The same is true for the *Second-Level Cache*. The *Third-Level Cache*, however, is shared between all the processor's cores [SZ08].
The levels of the hierarchy differ in the times they need to search for and access a requested datum as well as their capacity, i.e., the amount of data they can cache. Figure 2.4 shows such a hierarchy. It illustrates that the access times increase the farther away a cache is from a CPU, with the *LLC (Last-Level Cache)* having the highest delay. The capacity, however, increases as well, providing room for more data to store.

2.1.4 Cache-Coherency

Strictly speaking, the term NUMA as introduced before is only one special form of a NUMA architecture, typically referred to as *Cache-Coherent NUMA (ccNUMA)* [LL97, Kle04]. As the name implies, architectures of this type ensure cache coherency across all of a system's CPUs and cache levels using hardware mechanisms. This method contrasts to systems where the programmer or a software component is responsible for providing

coherence between data kept on different nodes and in different caches [OA89, AAHV91, ADC11]. Doing this work in hardware allows for an easier programming model, as the application or systems programmer is relieved from additional complexity.

Since I do not consider NUMA machines that do not provide cache coherency in hardware, the term NUMA is used in the remaining part of this thesis although strictly speaking only ccNUMA is meant.

2.2 Memory

My work investigates the difference in access times between physical memory connected to different CPUs in a multi-processor system. In order to fully understand all memory related parts, this section provides an overview of memory management on today's systems. Since this thesis specifically targets x86_64, my remarks concentrate on this architecture—although the general concepts of this section apply to nearly all of today's systems with no or only a few deviations.

When talking about memory, there are two views that are particularly interesting in the context of my work: physical and virtual memory.

2.2.1 Physical Memory

Physical memory can be regarded as the memory as it exists in the form of DRAM (Dynamic Random-Access Memory) banks in the machine and as it is visible at boot up.[4] The *physical address space* is the entity which provides addresses uniquely referencing each cell in physical memory.
Typically, the total physical memory is not represented as a contiguous block within the address space but rather fragmented: there are certain regions that reference no memory at all and ones that are reserved for special purposes. Examples include video memory, BIOS (Basic Input/Output System) reserved memory, and ACPI (Advanced Configuration and Power Interface) tables. Other regions are free for use by software. System software—typically the kernel of an operating system—is the component that manages this memory by granting usage rights to applications.

2.2.2 Virtual Memory

Virtual memory on the other hand is *set up* by the operating system. By using virtual memory, each process can have its own *virtual address space* that can be used solely for its purposes and is isolated from other virtual address spaces.[5]
In this address space, regions of virtual memory *map* to regions of physical memory. This mapping is usually established by software but enforced by the hardware: the *Memory Management Unit (MMU)*, a component in the CPU, is dedicated to this purpose. To the software, the translation from virtual to physical addresses happens transparently and processes only work directly with virtual addresses.

2.2.3 Page Management

Creating such a mapping for each byte in an address space would not be a viable option because the bookkeeping and management overhead would be too high. Due to this overhead, the notion of a *page* was introduced. A page is a fixed size block of virtual

[4] It is not entirely correct to directly correlate the memory regions visible at start-up with the physical memory banks: the memory controller can usually be configured in different ways, creating different mappings from physical memory to the actual memory banks as they are reported later.
[5] Typically, a region of this address space is reserved for the kernel and neither accessible nor directly usable by the application.

memory—on x86 typically 4 KiB or 4 MiB—that references a block of the same size in physical memory.
With this coarser granularity, the virtual-to-physical mapping can be managed using *page tables*: special objects created and managed by software whose structure is also known to hardware. Each *Page Table Entry (PTE)*, i.e., an element within such a table, maps to one page in physical memory. The hardware, in turn, is then able to traverse these tables in order to find out the desired physical memory destination to a virtual address. I do not explain details of page table management because those are not required to understand my work—I can rely on existing APIs that handle page table management for me.

2.2.4 Memory Management

Putting it all together, figure 2.5 gives an overview about the virtual memory principle. On the left and on the right the virtual address spaces of two processes are shown. In the middle, the physical memory is located. Both virtual address spaces contain references to regions in the physical memory address space. These physical memory regions are accessed when memory on the corresponding virtual memory address is read or written. Because a page is the smallest mappable unit in this scheme, each region is a multiple of a page in size. The virtual-to-physical mapping is established by system software on process creation or later when the process allocates additional memory.

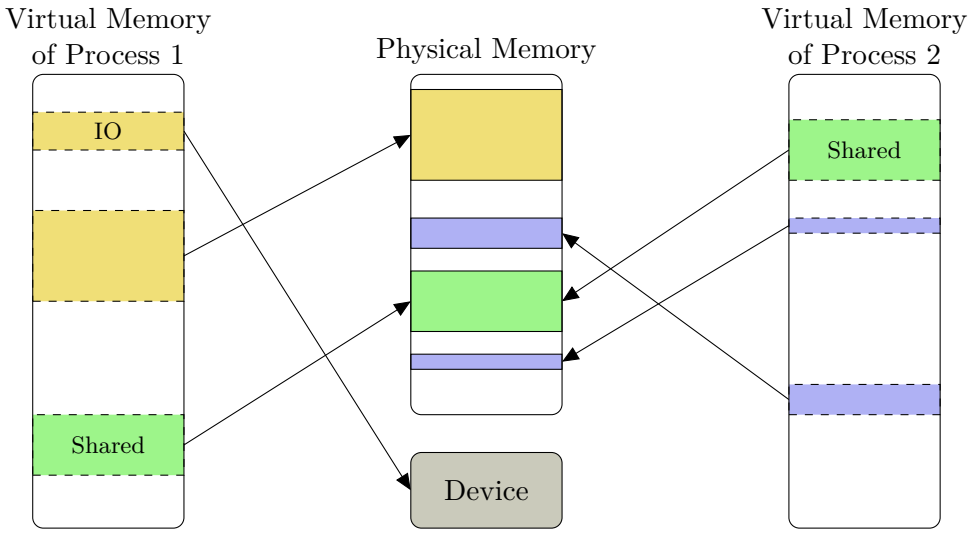


Figure 2.5: Virtual & Physical Memory Interaction

In addition to the already mentioned isolation between processes, virtual memory provides four advantages that are not immediately obvious:

- *Memory regions can be shared.*

  It is possible for two or more regions of memory in distinct virtual address spaces to reference the same physical pages. In this case, the content of the regions is

said to be *shared* among the processes owning these address spaces. Figure 2.5 on the preceding page illustrates this aspect for the virtual memory blocks labeled "shared".

- *Pages can be annotated with access rights.*

  In a system where sharing is possible, it is usually also important to provide a way to restrict access to shared regions. For instance, it can be useful to only allow read accesses to happen on certain shared regions—modifications might be prohibited. On x86 systems, read, write, and execute bits are associated with each page and evaluated by the hardware, allowing for this type of rights management to be enforced.

- *Mapping of arbitrary data into the virtual address space can be simulated.*

  It is possible to map a file located on a device to a location of the virtual address space, for example. In this case, the operating system reads the file's contents once into memory and all subsequent references to this memory region could indirectly work on the file's data. Of course, for writes to appear in the file on the device, the operating system has to write it back once the work is finished. This form of access provides better performance than an interaction with the slow device for every file operation. The virtual memory region dubbed "IO" in figure 2.5 on the previous page depicts this scheme.

- *Mapping does not have to happen eagerly.*

  When memory is requested by a process, the operating system might decide to note that it handed out a certain amount of memory to a program without actually backing this region with physical memory. This way, allocation requests can be fulfilled faster. When the application tries to work on this memory later, it triggers a page fault because the hardware cannot find a virtual-to-physical mapping. The operating system is invoked to handle this fault and remembers that the application allocated this region. To that end, it just allocates physical memory *lazily* (i.e., after the fault), establishes the mapping for the page where the access happened, and lets the program resume execution afterwards. A similar process can be used for memory mapped files as explained earlier where the relevant data of a file is read into memory upon first access. For the application, this process happens transparently.

Subsequently, I explicitly differentiate between virtual and physical memory unless this information can be safely inferred from the context.

2.3 Linux

Having illustrated the important aspects of NUMA hardware architectures, it is also important to see what can be done on the software side to deal with the peculiarities of such systems. The aspects I have a look at are the operating system as well as applications and libraries close to it.
This section investigates NUMA support on the *Linux* operating system.[6] Linux is both a major player in the server, desktop, and mobile markets as well as a system used extensively for developing research prototypes.

2.3.1 NUMA Awareness

Linux is NUMA-aware: it uses information about the system's NUMA topology in user space as well as in the kernel. Basic support for NUMA architectures dates back to kernel 2.4 [Fou13]. Around 2004, with Linux 2.6, the foundation for the policy based architecture as it is used today was laid [AJR06, Fou13]. To gather topological information, the kernel queries ACPI tables, foremost the SLIT (System Locality Information Table) [Lam06a, CCC+06], upon system boot.

From user space, access to the acquired information is granted through the *sys* and *proc* pseudo file systems. Using these file systems, node distances between cores can be retrieved or data about a process' virtual memory regions and the backing NUMA nodes acquired, for example. In the kernel, NUMA knowledge is also used for improving performance with regard to at least two other areas:

- *Thread Scheduling:*

  In order to balance load equally among all cores of the system, the scheduler might decide to migrate a thread from one core to another [Sar99]. This migration may even happen across node boundaries. The scheduler, however, has the information that migration to a core on another NUMA node is more expensive. This cost is twofold: first, there is a direct cost because migration, i.e., data transfer, takes longer if the destination core is remote; second, indirect costs are induced when caches on the destination CPU do not contain any data the newly scheduled thread will access because more cache misses will occur. Based on this information, the scheduler can decide if a thread migration to a remote node is still advantageous.

- *Memory Allocation:*

  Linux provides different APIs for memory allocation within the kernel and in user land. In the kernel, a counterpart exists for each major allocation function where the node from which to allocate memory can be specified explicitly [Lam06a]. Examples include the `kmalloc` and `kmalloc_node` functions. If such an explicit node-aware function is not used, memory from the node the requesting thread is running on is used for allocation.

[6] Strictly speaking Linux is only the name of the kernel. Distributions provide the necessary libraries and programs on top required to run typical end-user programs on top of the kernel. For this thesis, however, this distinction is not of interest and for simplicity, I refer to Linux as the whole software product and not only the kernel.

2.3.2 First Touch

By default and if no special NUMA memory allocation API is used, current versions of Linux use a policy called *first touch* [AMD06, DFF+13, Fou13]. When memory is first requested from the operating system, it is typically not mapped into the requesting process' address space and also not yet backed by physical memory. Only when the first reference to one of those requested pages is made, i.e., a read or write access happens, a physical memory frame is allocated and the actual virtual-to-physical mapping is established. This *demand paging* is common practice in operating systems [Tan09].

With a first-touch strategy, when this very first access happens, the physical memory that gets used for backing the mapping is allocated on the node where the accessing thread is running on.[7] In certain scenarios this first-touch strategy works well, for example, when memory is allocated and used only by one and the same thread and when migration of threads to other NUMA nodes happens rarely or not at all [AMD06]. Unfortunately, neither precondition can be guaranteed in practice:

- *Applications might be programmed in a way that "manager" or "factory" threads allocate, initialize, and hand out memory to other threads.*

  In this case, the first access is performed by the allocating thread which might run on a different node than the thread working on the memory. This scenario can be regarded as a worst case for this type of strategy, but in complex systems and with software written in a platform-independent fashion, it constitutes a possibility.

- *It is hard for the operating system or a dedicated component to determine beforehand how applications might behave over time.*

  Linux schedulers usually ensure that upon start-up of an application the load is well balanced across all execution units by placing threads on an underutilized core [Lam06a]. However, applications typically run in phases: after a certain time CPU utilization, memory access frequency, and/or device input/output might change significantly [SPH+03, IM03, LPH+05, VE11]. This change can lead to imbalances.
  The most prominent way to cope with this uneven load is the migration of threads to different cores that are less heavily loaded [Lam06a, Lam06b]. These cores may possibly reside on different NUMA nodes.

2.3.3 NUMA API

Kleen devised an API for Linux that provides NUMA related functionality [Kle04, Kle05]. Current versions of mainline Linux integrate it in an extended form [Kle07]. This API consists of user space components as well as changes to the kernel itself. It focuses on memory allocations but also offers functionality that is concerned with thread affinity in the context of systems that have a NUMA architecture.

[7] Obviously, it is possible that all physical memory local to the node of interest is already allocated to other processes. In this case, it depends on the implementation what is done—typically, memory is allocated from the closest other node still having memory available.

Policies

Kleen's proposal is built around policies governing memory allocation. Four policies are available [Kle07, Fou13]:

Local	The *local* policy specifies that allocation happens on the node the process is currently running on, i.e., the local node. Local allocation is the default policy. The local policy is the same as the first touch strategy I explained in section 2.3.2 on the facing page.[8]
Preferred	With the *preferred* policy, allocation is first tried on a particular set of nodes. In case of a failure, it falls back to the next closest node.
Bind	The *bind* policy strictly allocates from a specific set of nodes. Different from the preferred policy, no fallback is tried in case an error occurs but the allocation fails instead.
Interleaved	Allocations can also span multiple nodes. Each page of an allocated region can be allocated in a round-robin fashion on a set of NUMA nodes. The *interleaved* policy provides this type of allocation.

These policies can be specified for entire processes as well as for a particular region of virtual memory. A policy specified for a process also affects the allocation of kernel internal data structures that are allocated on the process' behalf. Children inherit the policy of their parent process on a `fork` operation.

Kernel

The kernel is the component where the policies are implemented and enforced. In his original proposal, Kleen suggests the introduction of three system calls [Kle04, Kle05]:

`set_mempolicy`	With `set_mempolicy`, it is possible to set the memory allocation policy for the calling process.
`get_mempolicy`	The current process' allocation policy can be queried using the `get_mempolicy` system call.
`mbind`	The `mbind` system call is used to set a policy for a specifiable region of virtual memory.

In order to scale to large systems with varying NUMA factors, the notion of a *nodemask* is provided. A nodemask represents a freely configurable set of nodes. The three presented system calls can be provided with such a mask instead of working only on a particular node or all nodes of the system, respectively.
This way, it is possible to use the `mbind` system call to interleave allocations on the three nodes that have the lowest latency to the allocating thread's processor, for example.
The Linux kernel itself does not perform automatic memory migration of pages that are not allocated optimally. As motivated before, threads might be migrated by the scheduler due to load imbalances. Such imbalances can occur over time as new programs are run and others terminate or long running programs change phases (see section 2.3.2 on the preceding page). In this case, data which was allocated locally is then remote.

[8] I use the term local here as this is also how it is referred to in the documents provided specifically for Linux—the man pages, for instance.

The main reason for the missing automatic memory migration is that until now, no heuristic was found which performs equally well or better than the no migration case among a large set of workloads [Lam06b]. User space applications knowing their workloads better than the kernel does, might improve their performance by applying memory migrations that incorporate knowledge about the system's NUMA topology.
To support this behavior on the kernel side, two system calls were introduced to facilitate memory migrations between NUMA nodes [Kle07, GF09]:

`migrate_pages`	This system call can be used to migrate *all* pages that a certain process allocated on a specifiable set of nodes to a different node.
`move_pages`	Contrary to the `migrate_pages` system call, `move_pages` moves the given pages—no matter what NUMA node they are allocated on—to a specific node.

These five system calls make up the interface between kernel and user space.

User Space

The user space part of the NUMA API is made up of several components. The most important one is *libnuma*—a shared library that provides a more convenient interface compared to directly interacting with the kernel using the previously described system calls [Kle07]. When linked to applications, these can access functionality such as NUMA-aware memory allocation, reallocation, and deallocation incorporating one of the mentioned policies; querying the relationship of nodes to CPUs and the number of nodes on the machine; working comfortably with nodemasks; and migrating memory pages between nodes.
Another component is the *numactl* command line application. This program can be used to modify or retrieve NUMA settings of running processes or to start programs with certain defaults overwritten. Compared to libnuma, no modification of a program is required and its source code does not have to be available.
In addition to the two main components just described, there are also other tools available that are closely related to numactl and provide more specific functionality. Two of these applications are *numastat* for gathering statistics about NUMA allocations [Gra12] and *numademo* which illustrates the influences the different policies have on the system.

2.4 L4

As this work is based on L4, the section at hand covers the relevant details of the microkernel as well as concepts and abstractions employed in user space. Furthermore, it introduces a common terminology.

L4 is the name of an ABI specification for microkernels [Lie96a].[9] At Technische Universität Dresden, the *Fiasco.OC* microkernel is developed that implements a derivate of this ABI. Microkernels strive to be slim [Lie96b]: they usually only implement security and performance-critical parts directly in the kernel and allow them to run in privileged mode. Everything else runs in user land without direct access to hardware or privileged instructions.

2.4.1 Mechanisms & Policies

In this spirit, a separation between mechanisms and policies is employed [LCC+75, GHF89, Lie96a, Tan09]. Typically, the kernel only provides mechanisms, i.e., the means of doing something. Policies, on the other hand, that are used to specify how something is done, are provided by user space services and libraries.
An example for this separation is the process of handling a page fault [HHL+97, SU07, Tan09]. The actual handler where the page fault is received first is located in the kernel. It is a mechanism in the sense that it gets invoked on a page fault, it extracts the information provided by the hardware, e.g., the faulting address and the type of access, and supplies a user level pager with this knowledge. The pager should know how to resolve the page fault, for instance, by loading data from a file into memory, in case of a memory mapped file. It might also use application or workload specific knowledge to decide how many pages to map in advance, for example.

The L4 microkernel offers three basic mechanisms [Lie96c]:

Threads	Threads are the means of serial code execution [Lie95]. A thread is also the granularity at which scheduling happens on logical CPUs [LW09].
Address Spaces	Address spaces provide the necessary isolation between user level applications. L4 uses the virtual memory principle as explained in section 2.2.2 on page 10 to achieve this isolation.
IPC	One way to intentionally break the isolation between address spaces is *Inter-Process Communication (IPC)*. IPC allows for data to be sent between two threads in a synchronous way [Lie95]. Synchronous means that both the sender and the receiver have to rendezvous for the communication to take place. If one party is not yet ready, the other is blocked until the former becomes ready or a configurable timeout elapses. In addition to sending data, IPC can also be used for mapping virtual memory regions [Lie96b] or capabilities to another task [LW09].

[9] Some parts in this section are taken more or less identically from my previous undergraduate thesis [M12].

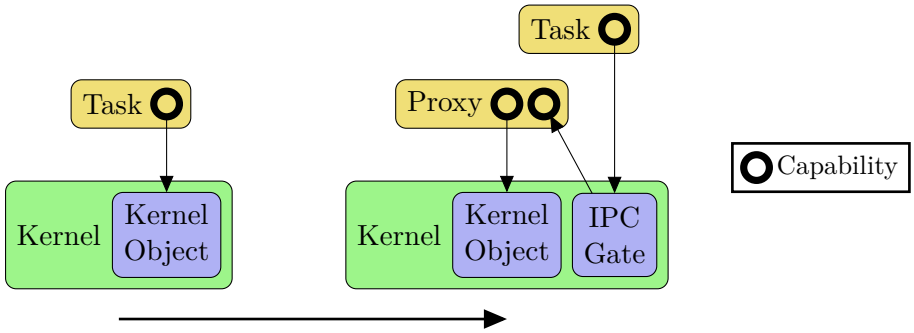


Figure 2.6: Capabilities and Virtualization of Objects

A word on names: an address space and the thread or threads belonging to it form a *task*. Other systems such as Linux often use the term *process* for similar means. I use both terms interchangeably.

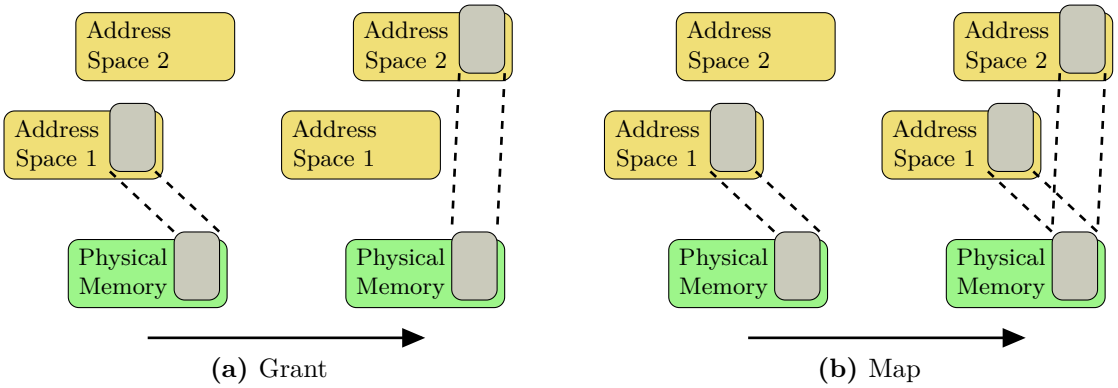
2.4.2 Capabilities

Another important building block of L4/Fiasco.OC are capabilities [LW09]. There are different flavors of capability systems. In the case of L4, the so called object capability system is used [MYS03].[10] Capabilities are tamper-resistant references to objects managed by the kernel. They are the means of protecting as well as naming and accessing resources within the system and across address spaces.

For this work, the naming properties of capabilities are of paramount interest: in order to communicate with another task or thread an *IPC gate* is used. It represents one endpoint of an IPC message transfer. IPC gates are kernel objects and in user space they are represented by a capability. A communication can be started by passing the capability of the destination IPC gate along with potential payload to the IPC system call. However, the kernel not only knows IPC gates: threads, tasks, IRQs, and more are also kernel objects and represented by capabilities in user space. Like memory regions, capabilities can be mapped between tasks as to transfer the right for invoking the underlying object.

Capabilities have the property that the invoker does not have to know where the destination object resides—it can be located in another address space or in the local one, even in the kernel itself. It is also possible to virtualize the object represented by a capability by intercepting requests and forwarding them to the original destination. Figure 2.6 shows an example of interposing direct access to “Kernel Object” from a client task by redirecting control transfer through the “Proxy” task.

[10] The *OC* in Fiasco.OC is short for “object capabilities”.



(a) Grant **(b)** Map

Figure 2.7: Grant and Map Operations

2.4.3 Memory Management Primitives

Basic management of memory (excluding kernel memory) is performed at user level. The kernel provides the means of mapping memory regions at the granularity of a page between address spaces in the form of three mechanisms [HHL+97]:

`grant` The `grant` operation maps a range of pages to a destination address space and removes the mapping from the source address space. Figure 2.7a illustrates the granting of a memory region from one address space to another.

`map` Mapping memory into another task's address space is similar to granting the memory, except that the mapping in the source address space is left untouched (see figure 2.7b). The mapped pages are shared subsequently between the mapper and the mappee. Mapping a memory region can only happen with equal or less rights.

`unmap` In order to revoke a mapping, the `unmap` operation can be used. The mapping is removed from all address spaces that directly or indirectly received it, but left untouched in the task requesting the `unmap`. Unmapping is only possible for pages that are *owned* by the address space requesting the operation.

All three mechanisms just mentioned can be used at user space but they rely on kernel functionality to provide their service. The `map` and `grant` operations directly use the IPC functionality provided by the kernel [Lie96a]. Therefore, the sender and the receiver of a mapping have to agree on when to exchange a mapping.
The `unmap` operation is implemented as a distinct system call and does not require the consent of the task the mapping is revoked from—the receiver of a mapping implicitly accepts that the latter can potentially be revoked on the mapper's discretion. The kernel keeps track of which regions are mapped to which address spaces in a *mapping database*. Based on this database, memory can be unmapped from all address spaces.

2.4.4 L4Re

On top of Fiasco.OC, the *L4 Runtime Environment (L4Re)* is developed. The L4Re provides basic abstractions for interfacing with the kernel from user space as well as infrastructure to ease application development [Gro12]. Examples include services and libraries such as `l4re_kernel` that helps to start and bootstrap new L4Re tasks, `l4re_vfs` providing virtual file system support, `libkproxy` wrapping kernel functionality in a convenient way for user level applications, as well as adapted versions of `libstdc++` and `µclibc`—implementations of the standard libraries for C++ and C, respectively.
Four abstractions provided by L4Re are of main interest and are now explained in more detail: Dataspaces, Memory Allocators, Region Maps, Pagers.

Dataspace A dataspace is a user space abstraction for any memory accessible object [LW09, Gro13a]. It can be used for representing normal chunks of memory (for anonymous memory, for example), memory mapped files (and parts of a file such as the text or data sections), or memory mapped I/O regions of a device.
As dataspaces are implemented on top of virtual memory, their memory can be shared between tasks. Moreover, a page is their minimum size. Dataspaces might also provide different properties or guarantees to clients: there might be dataspaces that use a contiguous region of physical memory or ones that are pinned and may not be swapped out.

Memory Allocator Memory allocators are used for memory management at the granularity of a page [Gro13e]. They provide clients with an interface for requesting new and returning no longer required memory in the form of dataspaces.
Memory allocators may also enforce further policies including quota support to prevent excessive memory usage by a single client.

Region Map A region map is the entity that manages a task's virtual address space layout: it remembers which regions are free and which are occupied [Gro13h]. A region map works at the granularity of a dataspace.[11]
For interaction with clients, region maps provide an interface that consists of methods for reserving a virtual address region, attaching and detaching a dataspace, and searching for a dataspace associated with a certain region. Since region maps know the virtual address space layout, they know which regions might be reserved for special purposes (and should not be used by clients) and are involved in page fault handling as performed by pagers.

[11] To be precise: region maps work with regions. A region can, in theory, be backed by multiple dataspaces but this functionality is not provided in the default implementation and not relevant for this work.

Pager L4Re uses the concept of user level pagers [Lie96c, HHL+97]. Conceptually, a pager is a component that is responsible for resolving page faults using the `grant` and `map` memory management operations presented in section 2.4.3 on page 19. Such a component is always a thread that understands and speaks a page fault IPC protocol. A separate pager can be associated with each "ordinary" application thread but typically one pager handles page faults for all threads of an application.

Hierarchical Memory

Pagers and Memory Allocators (and the address spaces they belong to) can form a hierarchy in L4 (sometimes the term *recursive address space* is used [Lie96a]): there is a root component that initially owns all physical memory in the form of 1-to-1 mapped virtual memory regions [SU07]. Other pagers and allocators can be created that rely on this root component for allocating memory which they, in turn, hand out to their clients.

This approach has the advantage that pagers can implement different policies and provide other guarantees. For example, it is possible that one pager supports swapping, i.e., moving pages of memory to secondary storage (a hard disk drive or comparable means) in case of memory pressure—a job which requires access to a service driving the hard disk. Since applications might have varying requirements on the paging strategy, it is not possible for the initial pager to suite all of them equally well.

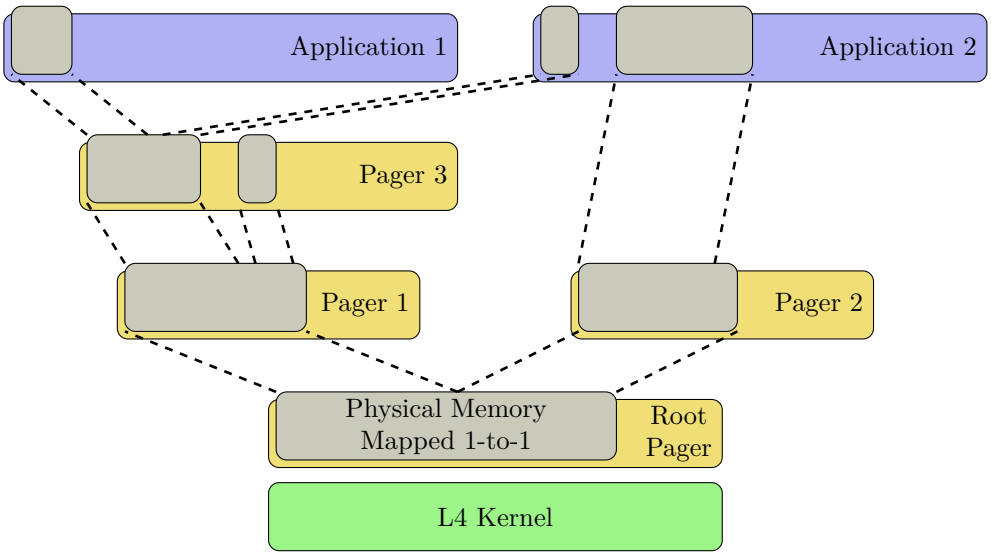


Figure 2.8: Hierarchical Paging in L4

Figure 2.8 illustrates such a pager or memory allocator hierarchy. Memory can be mapped or granted from one task to another. This latter task, in turn, might be a pager for a child task that gets mapped memory upon a page fault (shown by "Pager 2") or may be a normal task that uses this memory itself (e.g., "Application 1"). Tasks might even interact with multiple memory allocators or pagers as illustrated by "Application 2".

Overview

Putting it all together, figure 2.9 provides an overview about memory allocation and page fault handling on L4.

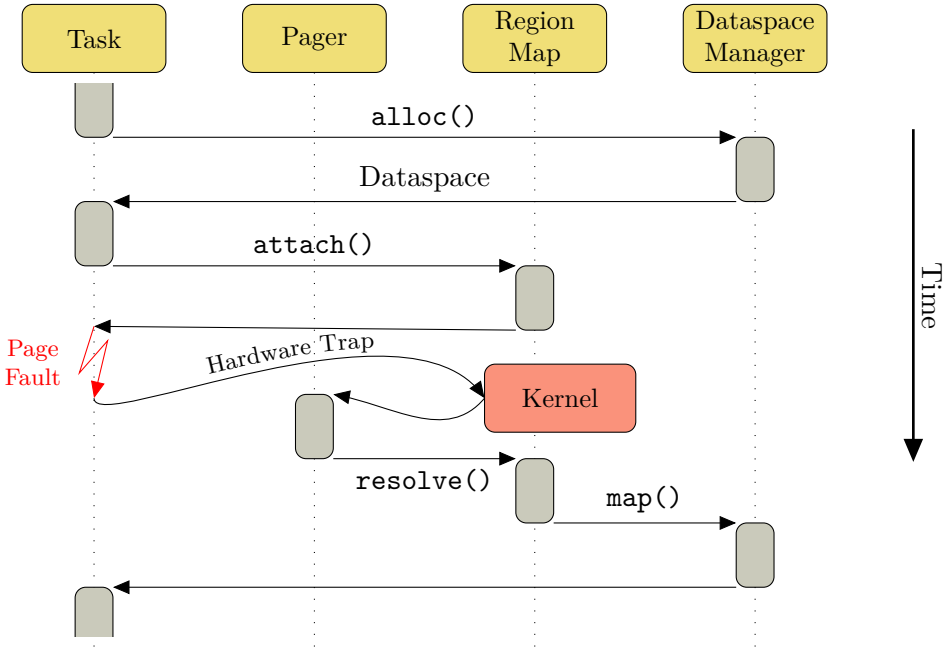


Figure 2.9: L4Re Component Interaction

When a client needs memory, for instance, for an anonymous memory region as the heap, it requests a dataspace with the required size from the dataspace manager. In order to make this region accessible, it has to be embedded into the address space. An "attach" request from the client to the region map along with the previously obtained dataspace arranges for the registration within the address space. The region map can be advised where to attach the dataspace or have it search for a suitably sized region and return the starting address.
Typically, the dataspace is only mapped but not yet backed by physical memory. Upon first access by a thread, the hardware triggers a page fault. Execution continues in the kernel's page fault handler. The kernel synthesizes an IPC operation to the faulting thread's pager in user space. In this operation, the kernel passes the faulting address. The pager now has to resolve the fault. It forwards this request to the region map that knows which dataspace represents the region where the fault happened. The dataspace then has to map its data by requesting physical memory to back the region where the fault happened or by copying data from a file in case of a memory mapped file, for example.

Note that unimportant details are abstracted from in this illustration for simplicity. Specifically, communication between the main components, task, pager, region map, and dataspace manager involves the kernel because it is based on IPC (even if two

components reside in the same address space). Furthermore, all communication happens in the form of a *call*, i.e., the caller waits for a reply from the callee until it resumes execution.

2.4.5 System & Memory Overview

Having discussed the mechanisms the microkernel provides as well as the abstractions the L4Re system offers in user space, this section's final part outlines the "flow" of memory when booting the system and names involved components.

Boot Loader Independent of the operating system being booted, the boot sequence starts with a boot loader that is the first software to run. It performs basic initializations and loads and starts further software components.[12] Before the boot loader executes the next program involved in the boot process, it queries a map of physical memory. A software interrupt on which the BIOS reacts by providing its view on the memory—which regions are used or reserved and which are free—is one example of how to retrieve this map. This information is stored in a free region of memory. After this step, the next program is executed.

Bootstrap On our L4 system, the next application is *Bootstrap*. Bootstrap receives the information where the memory map is stored in memory. It parses this map to provide a more convenient way of working with it. Using this knowledge, Bootstrap lays out the modules, i.e., program and library binaries, in free memory regions to make them ready for execution. From this data, along with the initially queried memory map, the *Kernel Interface Page (KIP)* is created. This page of memory contains information on how the system is set up and which kernel and system call version are used. From then on, the KIP is used for passing information to the next component that is executed.

Until now, all memory references referred to physical memory directly. Bootstrap is the component that sets up virtual memory by setting the corresponding bit in one of the processor's special purpose registers and by creating the required page table infrastructure (for more details on the difference between virtual and physical memory please refer to section 2.2 on page 10) [Kuz04]. Afterwards, it starts the next entity in the booting process: the kernel.[13]

Fiasco.OC When the microkernel, Fiasco.OC, is started, it performs necessary initializations. For instance, the number of CPUs in the system is queried, all of them are initialized, and a timing reference is configured to allow for scheduling of user programs. It also allocates the memory it needs for its own, kernel-internal data structures. The kernel queries this information from the KIP. Therein, it also marks which regions are no longer free, such that other programs do not try to

[12] My remarks are based on systems that rely on the BIOS for basic initializations—newer systems typically rely on *EFI (Extensible Firmware Interface)* or its successor, *UEFI (Unified Extensible Firmware Interface)*. The first steps differ there.

[13] Strictly speaking, there is one component that is run before the actual kernel: the kernel loader. For brevity and because this separation is of no interest here, this distinction is not made and the kernel loader and the kernel are treated as a single unit.

use that memory as well.
The kernel always operates in privileged processor mode, with unrestricted access to the hardware, all memory, as well as privileged instructions. Normal applications, however, run in user mode, where certain instructions that modify system state cannot be executed—a property enforced by the hardware—and only access to the task's own address space is allowed directly. Creating two of these user tasks is the next step the kernel performs: the root pager, *Sigma0*, and the root task, *Moe*.

Sigma0 Sigma0, the root pager, is the component that the kernel starts first. The kernel supplies it with the KIP and, based on the information therein, Sigma0 claims all user-available memory. Since the kernel already set up virtual memory but the information in the KIP is about regions in physical memory, all the virtual memory that Sigma0 manages is mapped 1-to-1 to physical memory. Sigma0 also manages platform specific resources like I/O ports.
Being the root pager, Sigma0's main responsibility is to resolve page faults of the root task that is started afterwards and then running concurrently [Gro13b]. In order to do so, Sigma0 speaks the same IPC page fault protocol as the kernel. However, dataspaces and region management are neither available at this early stage nor provided by Sigma0 itself. Hence, the paging functionality only comprises memory allocation tasks from main memory in its simplest form. Sigma0 itself has no associated pager, it is loaded entirely into memory before start-up such that no page faults can occur.

Moe Moe is the first regular L4Re task and forms the root of the task hierarchy: all other processes are either directly started by Moe or by a program started by it [Gro13f]. The kernel supplies it with several capabilities to kernel objects—among others, a factory for creating new objects that it may hand out to derived applications.
Moe is also the task that provides the abstractions presented in section 2.4.4 on page 20: it offers default implementations for multiple dataspace types with different properties. Examples include dataspaces for anonymous memory with copy-on-write semantics or ones that are physically contiguous. Furthermore, it implements the factory interface for creating new kernel objects and acts as the memory allocator, region map, and pager for subsequently started tasks [Gro13f].

Ned The default init process on L4Re is *Ned* [Gro13g]. As such, it is responsible for starting both further services needed by the system as well as tasks the user wants to execute. Ned can be configured and scripted using the *Lua* scripting language [ICdF13].
Using Ned, it is also possible to overwrite the set of default capabilities passed to each program. Examples include a memory allocator capability used for requesting memory in the form of dataspaces, as well as region mapper, and scheduler capabilities.

More services for different purposes are available: *IO* for management of input/output peripheral devices, *Mag* as a simple GUI multiplexer, and *Rtc* as the means of accessing

the system's hardware realtime clock from multiple clients [Gro13b]. They are left out here for they are neither mandatory nor of interest on the context of this work.

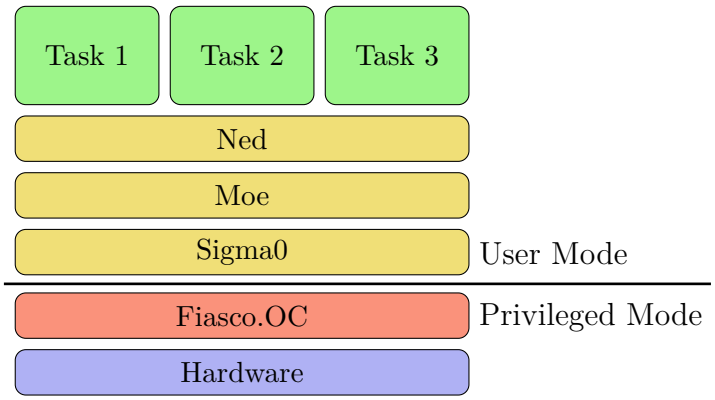


Figure 2.10: Overview of an L4Re-Based System

Figure 2.10 provides an overview of the components just explained. It does neither show the bootloader nor Bootstrap because they are only used when starting the system—afterwards both terminate. From this picture, the distinction into privileged (kernel) and unprivileged (user) mode is also visible.

NUMA Awareness

In its current implementation, there is no NUMA awareness in the kernel or the user land of the L4 system. The kernel neither queries nor grants access to information about the NUMA topology like Linux does and has no information about which CPUs and physical memory regions belong to or are local to which NUMA node.
Also, none of the mentioned L4Re services nor any of its abstractions like dataspaces rely on NUMA knowledge: they all work under the assumption that all memory accesses are equally fast.

Running the system under these circumstances on a NUMA machine can lead to suboptimal performance in certain scenarios. Therefore, the next part of my thesis, beginning at chapter 3 on page 33, investigates possible problems and discusses solutions to them.

2.5 Related Work

NUMA architectures provide better scalability with more processing units and so future systems are expected to follow this approach of non-uniformity in terms of access and transfer times between tightly integrated components. For that matter, research has focused on various parts of the software and hardware stack.
This section presents other work done in the context of multicore and multiprocessor architectures—approaching the scalability and contention problems arising from them. It covers scheduling and memory migration mechanisms that address penalties on NUMA systems and gives an overview of shared resource contention as well as new cache architectures.

2.5.1 NUMA Effects

AMD was one of the first microprocessor vendors that provided a NUMA architecture: their Opteron processor replaced the shared FSB (Front-Side Bus) with an on-chip memory controller [KMAC03]. For providing performance advice to developers, they also evaluated the effects a four processor Opteron system exhibits [AMD06].
They show that NUMA memory architectures can have a profound effect on performance: for remote accesses requiring two hops to read data, a performance degradation of 30 % is reported. For write operations with two hops the slow-down can reach 49 %. They argue that in order to achieve good performance, locality is an important factor.

Mccurdy and Vetter investigated how to best track down NUMA related problems [MV10]. They make a strong argument towards the usage of performance counters. Basic performance counters allow no correlation between an event, such as a remote access, and the source of the event, i.e., the instruction. With newer means such as event-based or instruction-based sampling this correlation can be established.
Based on this knowledge, they designed and implemented *Memphis*, a toolset that uses instruction-based sampling to find out which variable in a program causes problems in the context of a NUMA architecture. The effectiveness of their approach is demonstrated by improving profiling benchmarks and scientific programs and fixing their hot-spots. The result is a better balanced access ratio between local and remote nodes than before and a performance increase of up to 24 % for certain applications.

Bergstrom concentrates on the evaluation of systems of the two major suppliers, Intel and AMD [Ber11]. He uses the *STREAM* benchmark to compare two high-end systems: a 48 core AMD Opteron system and a 32 core Intel Xeon. The benchmark is adapted to feature local, remote, and interleaved allocations, as well as thread affinity settings to avoid automatic thread migrations.
The findings suggest that significant differences between the architectures of the two systems and—as a consequence—different scalability characteristics exist: the AMD system provides a higher peak bandwidth than the Intel system. At the same time, it shows a large performance gap between interleaved and non-interleaved accesses and scales less predictably to a higher number of threads. The author concludes that NUMA penalties for the Intel architecture are less severe, primarily due to higher interconnect bandwidth.

Tang et al. conducted an evaluation based on a less synthetic but more realistic set of applications [TMV+11]. They investigate the influence of NUMA effects on common datacenter applications and workloads. Their focus lies on data sharing—within an application and between applications. The authors identify three important properties characterizing an application: the required memory bandwidth, its cache footprint, and the amount of data shared within the program itself. Based on these characteristics, a heuristical and an adaptive (i.e., self-learning) approach for mapping threads to cores is presented.
The paper reports performance gains of up to 25 % for websearch and 40 % for other applications—bigtable, a content analyzer, and a protocol buffer, for instance—by only varying the thread-to-core mapping. They also point out that contention on shared resources such as memory controllers and caches can have a profound effect on application performance.

2.5.2 Shared Resource Contention

Previous remarks (see section 2.1.3 on page 6) explained why certain hardware resources are shared among execution units: sharing of resources can lead to lower production costs and better utilization. In certain scenarios, such as the usage of shared caches, it can even improve application performance. However, it can also lead to contention.

Majo and Gross investigate contention on NUMA systems, especially for interconnects and memory controllers [MG11b]. They aim at achieving high memory bandwidth. Like Bergstrom [Ber11], they use the STREAM memory benchmark for their evaluation. However, they rely solely on the *Triad* workload that is part of it. For simplicity, the authors run this benchmark in a multiprogrammed fashion with up to eight instances.
Using different workload combinations on their two processor Intel Nehalem machine, Majo and Gross show that relying only on local access (i.e., enforcing locality under all circumstances) is not sufficient for achieving the highest possible memory throughput. One of the factors that has to be taken into account is the system's *Global Queue (GQ)*. The GQ is responsible for arbitration between accesses of the different processors in the system and it directly influences the ratio between local and remote accesses when the system is loaded.
Based on this knowledge, they devise a model for prediction of achievable bandwidth in a loaded system with local and remote memory requests. In a last step, the researchers investigate the same effects on a successor system based on Intel's Westmere microarchitecture. They conclude that qualitatively the same effects are visible although specific numbers and ratios differ.

Dashti et al. take a different approach of dealing with the peculiarities of NUMA systems [DFF+13]. They use traffic management as an inspiration for their proposal: their goal is the minimization of traffic hot-spots while preventing congestion on memory controllers and interconnects. The authors argue that remote access latency is not the main factor for slow software and that locality in general is not always preferable. They believe that contention on shared resources can slow down software more significantly.

The group devises the *Carrefour* system, an algorithm for load distribution on Linux. Carrefour uses four mechanisms to achieve its goals: page co-location (i.e., page migration towards the accessing thread), page interleaving, page replication, and thread clustering. Based on metrics such as memory controller imbalance, local access ratio, memory read ratio, and CPU utilization, the system decides which of the mechanisms to enable—while taking their respective overheads into account.
The evaluation is based on single and multi-application workloads and compares the Carrefour system to default Linux, AutoNUMA (according to the article, a patch considered to provide the best available thread and memory management to Linux), and manual page interleaving. It shows that Carrefour outperforms the other systems in most workloads. For the single-application case, performance is improved by up to 184 % over default Linux. The multi-application workload experiences—depending on the evaluation machine—a speed-up of up to 90 %. Furthermore, Carrefour never hurts performance by more than 4 % over default Linux.

2.5.3 Scheduling

Employing scheduling techniques is one way of coping with contention on shared resources and the peculiarities of NUMA systems.
A pure scheduling approach for handling shared resource contention is taken by Zhuravlev et al. [ZBF10]. They state that a scheduling algorithm consists of two parts: 1) a classification scheme that gathers information about which processes can run in conjunction and which combinations cause too much contention on shared resources and 2) a scheduling policy that assigns threads to logical CPUs.
In a first step, the authors investigate the quality of five classification schemes they collected from previous research on contention-aware scheduling algorithms. They compare these schemes to the optimal classification which is calculated offline by examining different scheduling combinations of two programs in an otherwise unloaded system. Over a solo run of each program, performance degradation due to shared resource contention is visible. The results indicate that two schemes perform best and within 2 % of the theoretical optimum: *Pain* and *Miss Rate*. Pain uses stack-distance information that contains information about LLC usage and can either be created online with sophisticated sampling techniques or in a previous profile run. Miss Rate only uses the LLC miss-rate as input which can be gathered online easily. For simplicity, the authors chose to further examine the Miss Rate heuristic.
The second step comprises the refinement of the Miss Rate classification scheme and the creation of a scheduling algorithm based on it: *DI*. Their scheduler is implemented entirely in user space on Linux and relies on processor-affinity related system calls for changing the thread to CPU assignments.
For their evaluation, the researchers created several workloads based on the *SPEC CPU 2006* benchmark suite. Although they use two eight core NUMA machines for their evaluation, they do not incorporate any topology information into scheduling decisions. Zhuravlev et al. show that their DI algorithm performs better than the default Linux scheduler in nearly all cases. They attribute this improvement to reduced contention on LLC, memory controllers, memory buses, and prefetching units. Moreover, they

state that their approach provides better *Quality of Service (QoS)*: run-time variance of the workloads decreases over the default case, causing more stable execution times and better worst-case performance.

2.5.4 Memory Allocations & Placement

Section 2.3.2 on page 14 introduced the first-touch strategy used in Linux and other operating systems. First touch has deficiencies making programs perform suboptimally under certain circumstances. Goglin and Furmento implemented a *next-touch* strategy for Linux that copes with some of the issues of the default first-touch policy [GF09].
The authors argue that thread migration due to load balancing by the operating system scheduler can happen very frequently. If such migrations cross node boundaries, previously locally allocated memory is no longer local to the executing processor and, thus, induces constant access penalties. Although it is possible to perform memory migrations from within an application to the new node, doing so manually involves knowledge about the application's behavior which is hard to gather. Also, the performance of the `move_pages` system call (see section 2.3.3 on page 15) as the standard way of doing this kind of work from user space is bad compared to a simple `memcpy` operation.
For that matter, the researchers enhanced the Linux kernel with a next-touch policy that automatically performs memory migrations towards the threads that access the memory. The implementation is based on the existing copy-on-write handler. It removes read and write access bits for the corresponding PTE, so as to cause a page fault upon access, and performs the page migration transparently to the user program. The authors evaluate the approach by measuring achievable throughput when migrating large amounts of pages using their policy and estimating the overhead of parts of the implementation. Furthermore, they differentiate their kernel-level approach against a user space implementation, stating the pros and cons for both sides. Unfortunately, the evaluation is not performed for a wider set of workloads in different scenarios.

Another point to investigate in more detail in the context of NUMA architectures is the allocation of variable-sized memory chunks ready for application usage. In this context, Kaminski wrote a NUMA-aware heap manager [Kam09].
Based on the publicly available *TCMalloc* that is designed for UMA systems, the author describes his adaptations to make the allocator perform well in NUMA contexts. He designs two prototypes: the first one aims at high portability and does not rely on any operating system specific NUMA API. Instead, the only knowledge about which NUMA node memory is located on is inferred by assuming a first touch policy—which means when memory is allocated from the operating system and initially accessed, the accessing thread's node is remembered. Kaminski claims that although this approach is feasible, it makes assumptions about accessing behavior that are not always valid and can lead to problems. Due to these and other problems, a second prototype is devised. In his second approach, Kaminski incorporates NUMA APIs, specifically libnuma on Linux. Using these APIs, memory can be allocated on a specific node or, if in doubt about the physical location of already allocated memory, the corresponding NUMA node can be queried. The overall allocator's structure comprises three components: a cache local to each thread that is used for small allocations, a central cache that manages

memory in free lists annotated with the node the memory therein belongs to, and a central page heap for both serving large memory allocations and carving out smaller pieces for the other levels of the allocator hierarchy.
The allocator's performance is analyzed with a custom synthetic benchmark with multiple worker threads allocating memory, performing some work, and releasing it again. The author notes a performance improvement of up to 40 % over the unmodified TCMalloc. He attributes this result to better locality and reduced interconnect usage. The adapted TCMalloc also outperforms the *glibc* allocator in terms of usable memory bandwidth. Concluding remarks include ideas on how to further improve the allocator as well as design recommendations for an operating system's NUMA API.

Awasthi et al. use memory migration techniques to best distribute allocations among multiple memory controllers [ANS+10]. They argue that in the future, the number of cores on a processor will increase and so will the number of on-chip memory controllers in order to provide more memory bandwidth—a trend that can already be seen in recent architectures. With multiple IMCs, the disparity of memory access times increases even more and keeping data local to the memory controller becomes more important.
The authors investigate latencies on a lower level compared to the work presented earlier: they state that in addition to communication latencies to access remote memory, memory controller queuing-delays and DRAM row-buffer conflicts also have a large impact on overall memory performance. To cope with the identified problems, they use two complementary approaches that influence memory placement: *Adaptive First-Touch (AFT)* and *Dynamic Page Migration (DPM)*. AFT works comparable to the first-touch policy but incorporates a more sophisticated cost function into the decision where to initially allocate memory. DPM uses this cost function as well but applies it later in order to rebalance the memory controller load using page migrations. Whether to migrate pages is decided after each epoch—a configurable but constant time interval—and the induced overhead is taken into account.
Awasthi et al. assume a 16 core system with four IMCs which they do not have as a physical machine but simulate in software. The simulator illustrates the need for multiple memory controllers in the first place by showing that throughput increases up to a certain point and request-queue lengths decrease. In their evaluation, the authors demonstrate that AFT can achieve an average speed-up of 17 % and that with DPM nearly 35 % improvement are possible. Furthermore, the authors demonstrate that their approach reduces the row-buffer misses of the DRAM modules by up to 23 % and show that queuing delays are also decreased for their workloads.

2.5.5 Other Non-Uniformity

Non-uniform access times are not only present between memory local to different processors. Rather, the same problem can emerge within one processor, where a single cache might exhibit different access times depending where the access originates from—so called *Non-Uniform Cache Access (NUCA)* architectures.
Hardavellas et al. investigate such NUCA architectures and propose a new approach to data organization within caches [HFFA09]. The authors claim that with uniform cache access times, scaling cache sizes to more cores and still keeping latencies low is not

possible. Also, a reasonable trade-off for the ratio between shared and private caches is hard to find, as it depends on the workload at hand.
Research has shown that distributed or sliced caches can provide a solution: a larger cache is split into slices that are physically located closer to a set of cores than others. Slices in close proximity to a core can be accessed with lower latency than others farther away. Still, the slices form one logical cache and are interconnected to allow for data to be shared to avoid wasting capacity on duplicated blocks. In such a scenario, data or block placement within the caches becomes an issue because requests served by a slice on the opposite side of the die can be several times slower than ones serviced by the local slice.
The authors propose an approach based on access type classification of data. In a first step, they examine access patterns of typical server, scientific, and multi-programmed workloads. They find that data can be grouped into three classes based on the accesses it experiences: private data, shared data, and instructions. Different placement strategies within the cache are suggested: private data is to be located close to its requester's slice, shared data is distributed evenly among all a sharer's slices, and instructions are replicated into slices close to accessors at a certain granularity (not every neighboring slice gets a copy as that would cause capacity pressure).
In a second step, the researchers design R-NUCA, a placement algorithm for such sliced caches. They present two methods for indexing data: address interleaving and their newly devised rotational interleaving. The classification algorithm that works on a page granularity is explained and the implementation of the placement discussed briefly.
The evaluation is performed in a cycle accurate simulator and shows promising results: the classification is wrong in only 0.75 % of all accesses and the Cycles Per Instruction (CPI) value is lowered by approximately 20 % due to less cache misses. These improvements translate to average performance gains of 14 % while staying within 5 % of the performance of an ideal cache design.

Memory and cache access times do not have to be the only variable in a system. Emerging systems also exhibit different performance characteristics for the executing entities. Such *Asymmetric Multiprocessing (AMP)* systems contain processors that provide different processing speeds or various levels of complexity. The result is one processor being more suitable for running workloads that have a higher priority or higher QoS requirements while others being better at executing specialized workloads such as a memory-intensive application much faster, for example.

Li et al. examine the effects of such AMP architectures [LBKH07]. Their research is motivated by heterogeneous architectures being expected to become more prevalent in the future. Heterogeneity in their sense refers to processors or cores that use the same instruction set but exhibit different performance characteristics. Causes for these variable characteristics can be higher or lower clock speeds, more or less sophisticated prediction, and out-of-order execution, for example. The authors argue that for achieving optimal performance, the operating system's scheduler should map threads to cores depending on workload properties such as throughput-orientation or single thread performance.

The researchers explain why existing schedulers do not perform well on architectures with asymmetric performance properties and outline *AMPS*: an *asymmetric multiprocessor scheduler* for Linux. AMPS is designed to incorporate *SMP* as well as NUMA-style performance-asymmetries into scheduling decisions. It contains three components: 1) an asymmetry-aware load-balancing unit ensures that faster cores receive more load than slower ones; 2) the faster-core-first scheduling policy guarantees that if faster cores are underutilized, they receive new work first or existing work is migrated towards these cores; 3) a NUMA-aware thread-migration unit estimates migration overheads and performs migrations as appropriate. The first two components rely on the concepts of *scaled computing power* and *scaled load* that quantify processor speed and utilization based on the system's slowest core. Thread-migration overhead is estimated based on the NUMA distances of the source and destination cores as well as the resident-set size.
Their evaluation is based on two AMP systems constructed from processors with different complexities connected to each other: an SMP system with 8 cores and a NUMA system with 32 cores distributed among 8 nodes. The authors state that AMPS proves advantageous over the default Linux scheduler in three aspects: performance, fairness, and variance. For the SMP system, an average speed-up of 16 % is reported. The NUMA system outperforms stock Linux in every workload with speed-ups between 2 % and 161 %.

This chapter explained basic principles required to understand the main part of my work. I represented the basics of NUMA architectures and the concept of virtual memory. On the software side, I covered NUMA awareness in the context of Linux and the L4 microkernel environment for which I develop my work. Moreover, I presented several research projects related to NUMA architectures that illustrate that application performance can benefit from incorporation of system information into resource-allocation decisions.
The next chapter explains design and implementation of a component that makes the system NUMA-aware by intercepting thread-placement and memory-allocation decisions.

3 Design & Implementation

This chapter elaborates on the process of making our L4 microkernel environment NUMA-aware. First, I give a more precise summary of the goals and requirements this work has to meet. In the next part, section 3.2, I describe three experiments I have conducted. Among others, these experiments illustrate possible performance penalties that remote memory accesses and shared resource contention can have on application performance on my machine. Section 3.3 depicts design and implementation of a component that provides automatic scheduling, memory allocation, and memory migration—mechanisms which are necessary to mitigate the previously perceived NUMA effects. This chapter's final section, section 3.4, presents adaptations to the L4 system required for the integration of the NUMA management component described beforehand.

3.1 Challenges

As motivated in the previous chapter, incorporation of knowledge about a system's NUMA topology can have a profound influence on application performance. Making our microkernel environment NUMA-aware is a general problem and can be tackled in a variety of ways. In this thesis, however, I cannot integrate all ideas and incorporate solutions to all possible problems arising in this context. This section points out the main goals and challenges of this thesis in more detail. Moreover, it defines a couple of non-goals: points that I intentionally ignore in this work.

3.1.1 Goals & Requirements

My work aims at providing NUMA awareness in the context of our microkernel environment. I have identified two goals to be of paramount interest:

Performance The main reason—from a bird's eye perspective—for NUMA architectures to emerge in the first place is the need for applications to perform better than with previous UMA architectures.
In this spirit, my approach aims at enabling more efficient usage of these architectures on the L4 system. Thus, the most important goal is to improve overall application performance throughout the system.

Ease of Use Any component or system is only successful if it has users. One of the points discouraging users to employ a library or software program is complexity handed over to the user: if installation and configuration take up too much time or the expected outcome does not outweigh the effort to get it up and running, a piece of software will not be used.
I aim for a user friendly approach that does not require modification of applications

that should behave in a NUMA-aware fashion. This means that the source code does not have to be provided and no recompilation has to happen. Furthermore, required configurations are to be minimized to allow for a short setup phase.

Moreover, there are four requirements guiding my design and implementation decisions:

R1) *Use thread and memory migrations for alleviating problems.*

Thread affinity and memory allocation decisions can be used to manage problems caused by NUMA effects: if a thread accesses remote memory, the thread can change its CPU affinity to execute locally to the memory or the memory can migrate towards the accessing thread.
This work aims at investigating the advantages and disadvantages of both approaches as well as their limitations.

R2) *Perform automatic rebalancing during run time.*

A simple approach to cope with NUMA penalties is to perform thread placement when a thread is created and allocate memory close to it. This *static* approach works if there is not much load on the system and if applications do not exhibit different phases with varying characteristics.
Although providing a good starting point for this thesis, I also want to investigate the advantages of using a *dynamic* approach for handling imbalances and emerging problems during application run-time.

R3) *Incorporate contention management into placement decisions.*

As recent work has shown, penalties induced by remote memory accesses are only one source of performance degradation caused by improper placement decisions [BZDF11, ZBF10, MG11a, MG11b, TMV+11]. Contention on shared resources such as interconnects, memory controllers, and caches can also become a bottleneck for application performance.
Such shared resource contention is another aspect my approach has to take care of.

R4) *Minimize changes to existing components.*

A lot of infrastructure exists on our L4 microkernel environment. Most of it is tailored for a special purpose in the spirit of the "do one thing and do it well" mentality [Sal94].
This document describes the efforts required for the creation of a component offering services specific to NUMA systems. Since not all systems are based on such an architecture, providing a dedicated component for that purpose—that can be used if required—appears to be a better approach than integrating functionality into commonly used components where it is used only in rare cases.

3.1.2 Non-Goal Criteria

A lot of aspects can be considered when designing NUMA and contention-aware software. The following points are not of interest in this thesis:

- *Energy Consumption*

  Energy consumption is of major interest these days with ever smaller devices and an increasingly "green" IT infrastructure. Due to timing constraints and complexity considerations, I do not consider energy consumption in this work.

- *Devices connected to a certain node*

  Drepper [Dre07a] and Lameter [Lam06a] state that the entire southbridge or certain I/O devices such as network interface cards or storage devices might also be connected locally to a NUMA node. Since I do not possess such a system, I concentrate on local and remote main memory and leave devices aside.

- *Other architectures than x86_64*

  The only test system I have is based on the x86_64 architecture. Due to a lack of other systems as well as my main expertise being based on that platform, I ignore other architectures in this work.

- *Querying topological information at run time*

  Usually, to retrieve more information about the system and its current configuration, the operating system queries certain sources at boot or run time. One of these sources is the ACPI interface with a variety of tables. Such tables also contain knowledge about the systems NUMA topology.
  It is no trivial task to parse these tables and because I only have one system available for evaluation, I hardcode required topological information for this thesis.

- *CPU and memory hot-plugging*

  Typically, modern hard- and software—especially on server systems—supports some form of hot-plugging where a CPU or memory bank can be added and/or removed without rebooting the system. This functionality introduces additional complexity which distracts from the problem I focus on. For this work, I ignore hot-plugging entirely.

3.1.3 Mechanisms

Previous work [Lam06a, Dre07b, BZDF11, KHMHb10, DFF+13] identified four mechanisms for coping with contention as well as other penalties imposed by NUMA systems:

- *Thread migration*

  Thread migration refers to moving a thread's execution from one logical CPU to another one. It can help to eliminate remote access penalties and to alleviate shared resource contention.
  Thread migration is a simple to implement yet efficient and effective way to cope with the aforementioned problems.

- *Page Migration*

  Page migration means moving a memory page physically to a different NUMA node. Page migration can be regarded as the counter operation to migrating a thread to the data it accesses: the difference being that the thread is fixed and the memory is moved. Resulting from this property, it can be used for solving mainly the same problems.
  Memory migration as I require it is enabled by virtual memory: due to the additional level of indirection it introduces, the physical memory backing a virtual memory region can be exchanged without the client application being neither notified nor actively involved.

- *Page Replication*

  It is possible to replicate pages for each accessing entity. When accesses to data happen from threads scheduled on two distinct NUMA nodes, for example, each thread could be provided with a replica of the page. Using this approach, each of the two threads could access the contents of this page locally.
  Unless memory is only read, a form of software coherency protocol is required so as to ensure synchronization between the replicas [DFF+13]. As a consequence of the additional complexity introduced by this approach, however, I ignore it for this work.

- *Page Interleaving*

  Memory can be interleaved across a subset of the available memory nodes. Usually, this interleaving happens at the granularity of a page. When accesses span multiple pages, penalties are decreased on average because a share of the accesses occurs locally while another occurs remotely. Interleaving can also help when multiple threads running on different NUMA nodes access shared data.
  Furthermore, when data is distributed among different nodes, contention can be reduced for memory-intensive tasks because utilization is spread across multiple memory controllers and interconnects [DFF+13].

3.2 NUMA Effects

Previous work estimated the effects which NUMA topologies can have on system performance (see section 2.5.1 on page 26). This section presents experiments performed to get a better understanding of the factors that influence application performance: remote access penalty and contention on shared caches, such as memory controllers, and interconnects. I also have a look at interleaved allocations and their influence on the NUMA factor.

3.2.1 Dell Precision T7500

For testing and evaluating my work I can use a Dell Precision T7500 workstation. This system features two processors—each on its own socket—making it a NUMA architecture.
Both processors are Intel Xeon® X5650 CPUs with a core frequency of 2.66 GHz [Coo12b]. The X5650 is of Intel's Westmere microarchitecture which in turn is a Nehalem with smaller feature size (32 nm instead of 45 nm). Each of the CPUs provides six cores. The cores support Intel's HyperThreading technology [Coo12b]—a form of SMT providing two hardware threads (see section 2.1.3 on page 7 for more information on SMT). In total, this system comprises 24 logical CPUs.
Each core has private L1 and L2 caches for storing 32 KiB and 256 KiB of data, respectively. The LLC (Last-Level Cache) is shared between all the cores on one physical CPU and is 12 MiB in size. Moreover, each processor can access 12 GiB of local DRAM, summing up to a total of 24 GiB primary memory for the whole system.
Intel equips its processors with TurboBoost which increases a core's clock frequency dynamically depending on the load [Coob]. However, I disabled this feature for it can bias measurements in unforeseeable ways.
Figure 3.1 depicts the Dell system schematically, showing the different cores with their hardware threads as well as the cache hierarchy and the main memory.

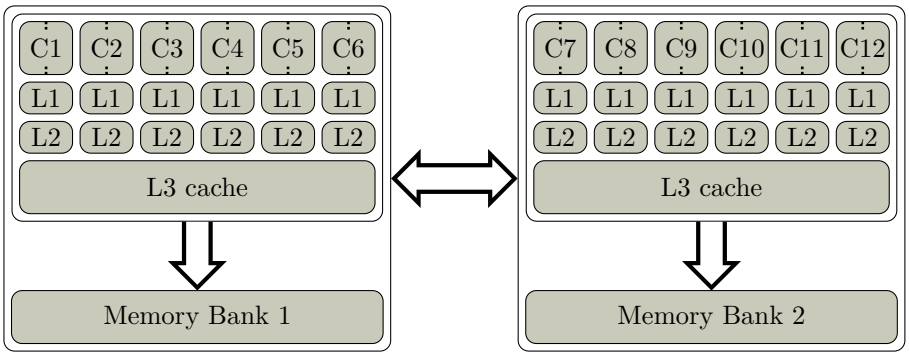


Figure 3.1: Dell Precision T7500 System Overview

3.2.2 NUMA Factor

My first experiment is to measure the NUMA factor, i.e., the factor that an access to remote memory is slower than a local access (see section 2.1.2 on page 5 for a more detailed explanation of this term). To quantify this penalty, I modified the benchmark provided by Kiefer et al. [KSL13] to run on L4.
The adapted benchmark allocates a large chunk of memory—512 MiB in my case—either on the local or the remote NUMA node and performs a fixed number of either reads or writes to random addresses in this region. For evaluation, the benchmark's run time is measured.
Two facts are important for obtaining reliable results not biased by other hardware mechanisms than the DRAM:

- *The array's size has to exceed the size of the LLC.*

  In order to measure memory and not cache accesses speeds, one needs to ensure that most of the memory accesses are not served by the cache but rather reference main memory. The important factor here is the LLC: it is the largest cache in the system and, thus, can hold the most amount of data. However, if the data to access is larger than the cache, cache lines are evicted and replaced for each accessed item. To that end, the size of the working array should exceed the system's LLC size by a large factor. For the STREAM benchmark, which aims for the comparable goal of estimating memory bandwidth, the recommended working data size is four times the size of the LLC [McC].
  With an array size of 512 MiB and an LLC size of 12 MiB for the Dell system, I satisfy this property.

- *Accesses should not happen sequentially.*

  Modern CPUs prefetch data that they suspect to be accessed next from main memory into caches. This process happens transparently as a background activity. When the data is used later on, the request can be served by the cache. The prediction of next accesses uses a simple heuristic: locality (in conjunction with possible access patterns). If an access occurs at a specific address in memory, chances are that a close, subsequent memory cell is referenced next. To circumvent the prefetching logic, accesses should happen at unpredictable addresses.
  Therefore, I use a random number function for generating the next memory address to read or write. In conjunction with the large dataset I use, the chance of finding the requested data in the cache (where it was loaded by the prefetcher) is negligible.

Compared to Kiefer's original version, the random number generation function is slightly modified: I employ an entirely deterministic random number generator—it relies solely on bit shift and mask operations to produce new values and the initial seed is hard coded—which is why it always creates the same sequence of values.
All subsequent measurements are performed using Fiasco.OC and L4Re revision 42648 from the subversion repository of the operating systems group. The benchmark uses the API I wrote for node-aware allocations as presented in section 3.4.1 on page 58.

Measurements

Comparable to other authors [Bre93, MG11a], I measure the run time of the benchmark for my evaluation: the slower the memory access is, the longer the program runs. I use eager mapping of memory to avoid the overhead for resolving page faults.

Operation	Access	Time	Std Dev	NUMA Factor
read	local	37.420 s	$6.60 \cdot 10^{-2}$ s	1.000
read	remote	53.223 s	$3.53 \cdot 10^{-2}$ s	1.422
write	local	23.555 s	$7.17 \cdot 10^{-3}$ s	1.000
write	remote	23.976 s	$1.48 \cdot 10^{-3}$ s	1.018

Table 3.1: Benchmark Execution-Times for Local and Remote Accesses

Table 3.1 lists the run time of the benchmark for read and write accesses being performed locally and remotely, respectively, as well as the inferred NUMA factors. Shown is the mean value of five runs and the derived standard deviation from this mean value.

Evaluation

The results indicate that read and write accesses show a different performance degradation when performed on a remote node: for reads, a degradation of 42.2 % can be reported. Remote writes are only marginally slowed down with a decrease of 1.8 % over the local case.
I double-checked this information by performing essentially the same benchmark on a Linux installation booted on the same system—using a marginally modified version of the original benchmark with the same allocation size of 512 MiB.[1] The setup runs a 64 bit Debian 7 with kernel 3.2.41-2. The measured numbers are identical with a standard deviation of less than 0.1 s compared to the previous numbers acquired on L4.

My explanation for this major difference in penalties between reads and writes is the *store buffer* of modern x86 CPUs. According to Intel, the Xeon® CPUs I am using contain a store buffer with 32 entries [Coo13a]. This component, located between processing unit and first level cache, buffers write operations so that they can be acknowledged immediately [McK10]. A buffering of writes allows the CPU to continue processing further instructions. It does not have to be stalled until the write eventually finished. In my case, this buffer masks the increased latency of writing data to a remote NUMA node.

These results contrast to ones Drepper acquired for an Opteron system [Dre07a]. He notes that writes are slower than reads by an average of 15 %. I believe this disparity stems from different systems possibly exhibiting different NUMA penalties. Particularly, as the comparison is between systems of two CPU vendors and of two generations, it

[1] I disabled the RANDOM_MASK feature which is enabled by default and made the random number function behave as previously explained.

is hard to reason about all the differences influencing the results. That significant differences can exist even between same-generation systems of two vendors is discussed in a publication by Bergstrom [Ber11] which I summarized in section 2.5.1 on page 26.

3.2.3 Interleaved Mode

To further corroborate my initial results, I ran the same benchmarking program in a slightly modified setup: the *interleaved* mode that is configurable in the BIOS of my system is turned on ("Memory Node Interleaving"). This mode, typically used in conjunction with operating systems that are not NUMA-aware, configures the memory controller in a way as to interleave memory regions on the NUMA nodes that make up the system's physical memory.

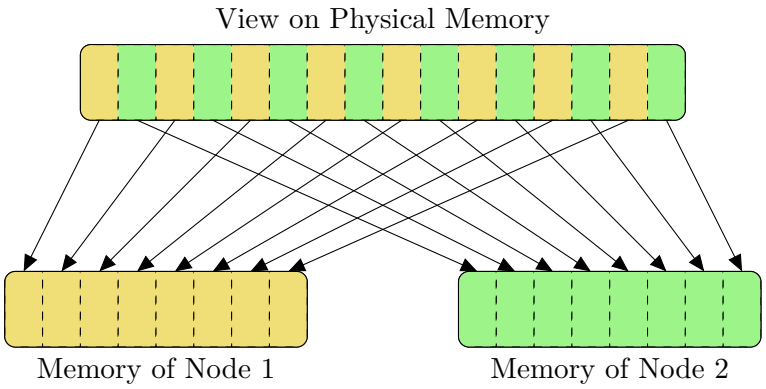


Figure 3.2: Physical Memory Interleaving

Figure 3.2 illustrates the concept of interleaved memory. As shown there, the physical memory is divided into fixed size regions, each referring to memory from the first and the second node in an alternating fashion. Unfortunately, the size of interleaving, i.e., the granularity, is not mentioned in the system's manual—it could be as small as an L1 cache line, that is, 64 bytes, or as large as a typical page (4 KiB). Kleen reports the same feature for an Opteron system [Kle04]. For his system, he states an interleaving granularity of the size of a page.

For accesses spanning multiple interleaved regions, I expect average memory access times to approximate the arithmetic mean of those times for accessing memory on the same node and those for accessing memory on all other nodes:

$$t_{\text{average}} = \frac{\sum_{i=1}^{\text{node count}} t_i}{\text{node count}} \tag{3.1}$$

Since my system consists of only two nodes, the ratio between local and remote interleavings is 1-to-1. For a not NUMA-aware operating system—for which this mode is designed—the performance can be expected to be more balanced compared to the case

where some allocations refer only to (faster) local and some only to (slower) remote memory.
I want to emphasize, however, that this memory interleaving is invisible to the operating system. It is established by the BIOS (by instructing the memory controller to set it up) before any other software is run. Still, the results give an estimation of the performance to expect in the case of implementing or using an interleaved allocation scheme for a NUMA API on L4.

Measurements

For measuring the speed with interleaved memory enabled, I use the same benchmark as before.

Operation	Access	Time	Std Dev	NUMA Factor
read	local	37.420 s	$6.60 \cdot 10^{-2}$ s	1.000
read	interleaved	48.405 s	$3.20 \cdot 10^{-2}$ s	1.294
read	remote	53.223 s	$3.53 \cdot 10^{-2}$ s	1.422
write	local	23.555 s	$7.17 \cdot 10^{-3}$ s	1.000
write	interleaved	23.976 s	$1.18 \cdot 10^{-3}$ s	1.018
write	remote	23.976 s	$1.48 \cdot 10^{-3}$ s	1.018

Table 3.2: Benchmark Execution-Times with Interleaved Mode

Table 3.2 shows the execution times with interleaved mode and the newly inferred NUMA factors.

Evaluation

As before, writes to interleaved memory are not influenced at all and show almost no difference to remote or local ones. Read accesses, on the other hand, do not perform as well as local accesses but they are also not as slow as remote accesses. On average, the performance degradation is 29 %.
This result is slightly worse than the approximately 21 % degradation that equation 3.1 on the preceding page suggests. Still, performance is better than for accessing only remote memory. The findings suggest that for data shared among execution units located on different NUMA nodes or for load distribution among multiple memory controllers, interleaved allocations can be useful.

3.2.4 Contention

As motivated earlier, contention can become a bottleneck: when multiple threads use a resource or component shared among them, each thread typically experiences a slowdown. Previous work has discovered three main factors of contention in a NUMA system: the LLC, the memory controller, and the interconnect [BZDF11, ZBF10, MG11a]. Other components of interest in this context include the memory bus and prefetching units.

Unfortunately, the limited time frame that is available for my work does not allow for an in-depth analysis of possible contention for each shared resource. I have decided to concentrate on the memory subsystem in more detail and to leave components closer related to processors—e.g., prefetching units or the global queue—aside.

In order to utilize my test machine in a scalable way, I run multiple instances of a modified version of the STREAM benchmark which I ported to L4 in a parallel way [McC]. Like Majo and Gross, I use solely its *Triad* workload [MG11b]. In test runs I executed beforehand, I witnessed the highest memory pressure out of the four available workloads. This workload uses three arrays (`a`, `b`, and `c`; in my setup, each 64 MiB in size) and performs a floating point multiply-add operation for each element: `a[j] = b[j] + scalar * c[j]`. These operations, performed in rapid succession, cause a significant amount of memory to be accessed and, thus, can illustrate possible bottlenecks or emerging contention patterns.
My setup starts several instances of this benchmark in a multi-programmed fashion, each executing on a logical CPU allocated solely to it. Each instance performs the multiply-add operation on the allocated data in a loop. It measures the time each loop takes to execute. I am interested in the maximum execution time for a single loop in order to factor out start-up delays between the different STREAM Triad instances. I assume that once all instances are running, interference is the highest and the time it takes for one instance to finish is the longest. I compute the data rate available to the instance from the acquired data by incorporating information about the arrays' sizes.

Measurements

Figure 3.3 on the facing page shows the calculated data rates for an increasing number of Triad instances started and running in parallel.
The diagram differentiates between two cases: when memory allocation is local (▬) and when it is remote (▬). The instances' threads are distributed on the two NUMA nodes of my machine: The first twelve instances are started on the second socket (logical CPUs 12 to 23) and the remaining instances on the first socket of my machine (logical CPUs 1 to 11).
I start the first twelve instances on the second socket because the first logical CPU in the system, CPU 0 (on the first socket), is used by other services in the system, such as Moe and Ned. In order to avoid interference with the running services, I do not use this CPU for a STREAM instance. Döbel et al. describe this problem for one of their setups [BD13].
Depending on the type of allocation, the memory is allocated on node 1 for local access for the first twelve instances and on node 0 for local access of the remaining instances or the other way around in case of remote allocation (on node 1 and node 0, respectively).
Furthermore, the plot illustrates the total bandwidth accumulated over all instances (●) as well as the average data rate for a single instance (▲). The data shown is the mean out of five performed runs. The standard deviation among these runs is less than 2.49 % of the mean and is too small to show up in the plot.

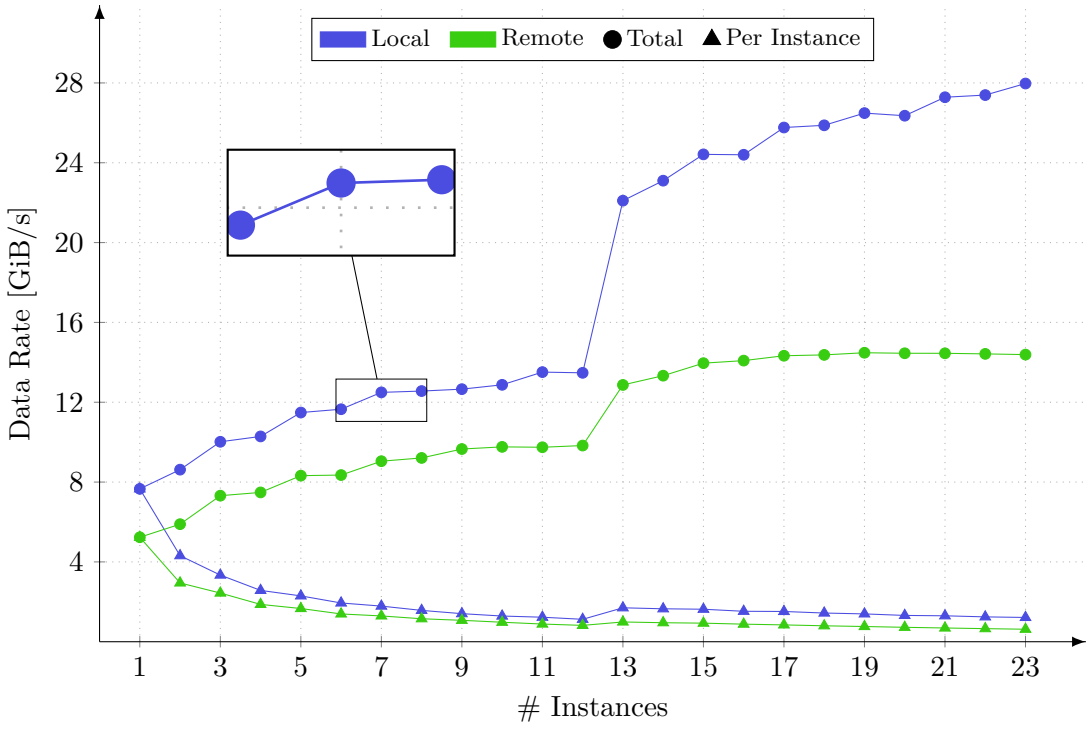


Figure 3.3: Data Rates for Parallel STREAM Triad Instances

Evaluation

The graph shows an intuitive behavior: with more instances started, each instance can utilize less bandwidth for its own needs. Starting with approximately 7.8 GiB/s and 5.4 GiB/s, for local (—▲—) and remote allocation (—▲—), respectively, for a single instance that causes no contention at all, bandwidth lowers to roughly 1.2 GiB/s and 0.8 GiB/s, respectively, for 23 instances.
The data illustrates contention on the Last-Level Cache: different STREAM Triad instances evict each other's cache lines causing subsequent accesses to produce cache misses and requiring a slow memory access. This behavior leads to only a share of the totally available bandwidth being used by each instance.

Still, the accumulated total bandwidth to memory usable by all instances increases. For the remote case (—●—), a stagnation is visible starting with nine Triad instances at a data rate of almost 10 GiB/s. I attribute this stagnation to contention on the interconnect—it is the limiting factor when memory is allocated remotely and requests have to travel between processors.
The remote case also shows a steep increase in the accumulated data rate with 13 instances. This increase is caused by the new instance executing on the next NUMA node where no STREAM task is running yet. Being the first instance, there is no contention on the LLC and access can happen at full speed. Although the interconnect was saturated before, there still is an increase in accumulated bandwidth for the 13th

instance (and subsequent ones). I attribute this increase to the QPI used for connecting the processors in my machine. It forms a unidirectional connection, meaning that there are two distinct interconnects for data traveling from NUMA node 0 to node 1 and the other way around.
The same behavior of a large data rate increase is also clearly visible with local allocation (—●—) where the difference between 12 and 13 instances is almost 8 GiB/s—nearly as high as the data rate for a single running instance. In this case, the memory controller is the main limiting factor and no cross-processor interconnect is involved.
Although the accumulated data rate increases less with each added instance, no stagnation is visible, yet. I interpret this observation as the memory controller having a higher maximum bandwidth than the interconnect and not being saturated by twelve STREAM Triad instances.

The usage of Hyperthreads, i.e., hardware threads running on a single core, causes another interesting fact. Each new STREAM Triad instance is started on a different logical CPU in increasing order, i.e., the workload with six instances uses logical CPUs 12 to 17. On a machine with two hardware threads per core (as has my test machine), every even logical CPU represents the first Hyperthread on a core and every uneven one the second Hyperthread.
On a closer look, the plot shows almost no difference in the total data rate between 7 and 8 or 9 and 10 instances, for example. When only a single Hyperthread is used on one core (for every uneven instance count), this thread can use almost the same bandwidth as two Hyperthreads running on the same core.
Although I cannot pinpoint this behavior to a single hardware component, it is clear that either contention on a component inside the core or the work distribution algorithm is causing this behavior.

Overall, the results motivate that remote access penalties and shared resource contention can become a bottleneck. The next section describes the development of a component that can serve to alleviate these problems using the previously explained mechanisms: thread migrations, page migration, and page interleaving. Due to time constraints, I leave page replication to future work.

3.3 NUMA API

Having conducted experiments showing that there does exist a difference between local and remote accesses and that contention can become a bottleneck and decrease application performance significantly, the next step is to implement the means of coping with these possible problems or even preventing their emergence in the first place.

My library or service needs to provide at least a mechanism for changing thread affinity, i.e., the logical CPU a thread is running on, and for NUMA-aware allocation of memory. As presented before, the L4Re is a major component in our L4 microkernel environment and used by numerous applications. For that matter, my implementation is based on L4Re mechanisms and abstractions.
This decision implies that I use threads and dataspaces as the main parts in my implementation. Further requirements were already stated (see section 3.1.1 on page 33): the resulting component has to support thread and memory migrations (requirement R1) and it has to provide the possibility of dynamic rebalancing (requirement R2).

In order to especially fulfill the latter requirement, a simple library approach is not enough: I need an active component that is involved in scheduling and allocation decisions and that checks for imbalances periodically.

3.3.1 NUMA Manager

The first question to answer relates to the location of the implementation. I see two possibilities:

- *Enhancing Moe.*

  Moe, as the default factory for creating kernel objects like threads and tasks, provides an implementation of a scheduling component and hands out memory to applications in the form of dataspaces. Having control over these objects, the required NUMA functionality could be provided directly in Moe.

- *Developing my own NUMA component.*

  It is possible to create a component that contains all NUMA-related functionality. This component would need to wrap and intercept scheduling requests and memory allocations as normally served by Moe and invoke appropriate rescheduling or memory migrations in case contention or remote memory accesses are detected.

I can think of two facts in support of an approach that modifies Moe directly. First, an adaptation of Moe instead of a creation a new component requires less new code and can rely on better tested functionality because default implementations of certain L4Re features can be reused and adapted accordingly. Second, compared to an approach with a separate NUMA management task, implementing the functionality in Moe causes less indirections: a new component would acquire certain objects from Moe and hand out wrapped or proxied objects to clients. This approach can cause performance degradation because additional communication is required from a client to the NUMA component and from the latter to Moe.

Although valid, these two points are outweighed by the disadvantages: in my opinion, Moe's codebase is barely in a state to work with. There exist almost no comments, no separation between interfaces and implementations is visible, expressively named constants are missing but instead "magic numbers" are spread throughout the code, and the used variable names are vacuous. Also, one of the main requirements for my work is to touch as few existing components as possible (requirement R4) which is clearly violated by adapting Moe in such a profound way.
Considering the previous two arguments, a decision in favor of the creation of my own component is inevitable. Later, I investigate whether or not performance degradation is visible due to these additional indirections.

The next parts examine how to create a component with the required scheduling and allocation functionality. Since my new component is responsible for handling NUMA related duties, I refer to it as *NUMA manager*. Also, when talking about a task my NUMA manager is to manage, I often use the term *NUMA task*.
Note that this section of the thesis only describes the mechanisms available to enforce my goals. The policies which can be build on top of these mechanisms are discussed in the next chapter, Evaluation.

3.3.2 Thread Handling

First, I start with an implementation of a scheduling component that can be used for influencing thread-to-CPU affinity and thread priorities from user space. It is important for three reasons:

1) As I am working on a capability-based system, interacting with a thread in any way requires a capability to it.

2) I need to know where threads are currently running in order to decide where to place other threads. Also, knowledge about which thread is local or remote to which dataspace is important. By default, no component provides this type of information.

3) For avoiding performance degradation due to NUMA penalties, I need to detect remote accesses and contention first. Per thread performance metrics are one way of estimating performance penalties. For querying them, I need knowledge of all managed threads.

These points illustrate the need for as well as basic requirements of my scheduling component.

Scheduler Interface

L4Re provides the `Scheduler` interface for scheduling threads at user level [Gro13d]. Unlike scheduling within the kernel, that is concerned with which thread to execute next on a particular logical CPU, scheduling in user space can change a thread's affinity, i.e., the logical CPU where it is supposed to run. The `Scheduler` interface comprises four methods:

`info`	The `info` method is used to retrieve information about the scheduler object and the system. The maximum number of processors supported by the system or the logical CPUs this `Scheduler` object manages can be obtained, for example.
`idle_time`	For querying a logical CPU's idle time, i.e., the time this CPU performed no actual work, the `idle_time` method can be used.
`is_online`	The online status of a logical CPU can be retrieved using `is_online`. This method is mainly important in the context of hot-pluggable CPUs.
`run_thread`	Everytime the scheduler is supposed to change a thread's parameters, such as CPU affinity or priority, the `run_thread` method is invoked.

The `run_thread` method represents the most important part for my work. It accepts both a capability of a thread to schedule and parameters such as the desired priority and CPU affinity. I ignore the given affinity and decide where to run the thread myself. The priority value is used directly. Operation is then redirected to the kernel in the form of a system call to schedule the thread.
The NUMA manager keeps all thread capabilities for later use, i.e., it maps them locally and stores them in a suitable data structure. This way, I have a set of all threads managed by this scheduler that can be used for gathering per-thread statistics, for instance.

Scheduler Registration

Within a task, the `Scheduler` object can be referenced through a capability registered in the environment. The environment is created at start-up of a task. The start-up is also the moment when all initially available capabilities are registered.
In order to make use of the created `Scheduler` object, I need to modify the initial environment of any NUMA task and replace the `Scheduler` capability with one referencing my NUMA manager. When the started NUMA task then invokes this capability, the NUMA manager can see the request and can react on it.
Fortunately, this modification of a task's environment is precisely one of the functions Ned offers. Ned—a loader for starting new L4Re tasks—also sets up their initial environment. Using functionality accessible from the Lua start scripts, it is possible to directly pass in a different `Scheduler` capability to use and register. All I have to do is start my NUMA manager via Ned and use the Lua functionality to create a channel for communication with it. Under the hood, such a channel is represented as an IPC. A capability to this IPC gate is then used to overwrite the new task's default `Scheduler` capability. When the NUMA task invokes the scheduler registered in the environment, the request is sent to the NUMA manager which can then react on it.

Thread-to-Task Relationship

One information that can guide thread-placement decisions is the task (i.e., address space) a thread belongs to. For example, if multiple threads run within the same task, it is likely that scheduling them on the same NUMA node (or pairs of them on the same

core, in case some form of SMT support exists) yields better performance than placing them farther away from each other.
The rationale behind this thought are locality and data sharing: threads within the same address space are likely to access the same data during execution. Explicitly shared data several threads work on and parts of the executable code form two examples of such sharing. Scheduling these threads on the same processor or NUMA node can lead to better cache usage because data needs to exist only once within the shared cache. If a thread was to execute on a different processor, its cache would contain its own private copy of the data. Furthermore, false sharing can occur and cause a degradation of performance [BS93]. For that matter, I need information about which thread belongs to which address space.

Identifying which task a thread belongs to is not trivial: a thread capability contains no user-accessible information about the address space it is bound to. I found two possible ways of establishing this thread-to-task relationship:

- *Explicitly communicate a thread to my NUMA manager upon creation.*

  This approach is very direct: whenever a new thread is created, its capability, along with the task it runs in, is sent to the NUMA manager. The implementation can happen at various levels within the software stack with different advantages and drawbacks. For instance, every application to run could be modified to explicitly perform such a send operation whenever it wants a thread to be managed by the NUMA manager. It is also possible to wrap or adjust all functions usable for thread creation accordingly, e.g., by replacing a thread creation function in a library.
  Either strategy allows for finding the task to which a created thread belongs by means of the environment because threads execute in the task's context and the environment is accessible from there.

- *Virtualize threads.*

  The second approach virtualizes threads. Comparable to the `Scheduler` object used for scheduling threads, a `Factory` object exists which is responsible for creating them in the first place. This object can be replaced. All thread creation requests are then directed to the new `Factory` object.
  For establishing the relationship from threads to tasks, a trick could be used: using the `ex_regs` system call (short for *exchange registers*), a thread's register state can be modified. The registers comprise, among others, the instruction pointer used for finding the next instruction to execute. By setting this value, a thread can be redirected to continue execution at arbitrary positions in code.
  One could inject a library into all started NUMA tasks that contains code for communicating the task's capability to my NUMA manager. Once the manager created a thread, it could use the `ex_regs` functionality to make the thread execute this piece of code and then continue normal execution. Doing so, interaction with my NUMA manager would be completely transparent to the application.

In my opinion, the second approach is superior to the first one: it is generic in the sense that it does not matter which libraries an application may use for thread creation—at the lowest level they all rely on a `Factory` object on the L4 system. Moreover, no modification of application or library code is required—an advantage when running workloads with a lot of different applications or benchmarks that are only meant to be run in an unmodified way. The approach, however, is not trivial to implement due to the code injection required for all NUMA tasks.

After pondering furthermore over the problem at hand, I found a solution that combines the directness of the first approach with the genericity of the second one.

- *Create a new Factory object for every NUMA task.*

  A `Factory` object serves, among others, requests for thread creation. By creating a separate `Factory` object for each task, the thread-to-task relationship can be established easily: every thread created by `Factory` X belongs to task X, while all threads created by `Factory` Y belong to task Y.
  The implementation requires only a simple per-task setup before creating a NUMA task: a new factory object has to be created and registered. However, using Ned and a Lua start-up script, the implementation is straightforward and can mostly be done in Lua code (the curious reader is referred to section 3.3.4 on page 53, where I present the required Lua functionality briefly).

Having a per-task `Factory` object not only helps to establish a relationship between threads and tasks. Because dataspaces are created using a `Factory` as well, this approach can also be used to determine which task a dataspace belongs to.[2]

3.3.3 Memory Management

The second important part of my NUMA manager is management of memory. As explained earlier, I target an approach based on L4Re abstractions. This decision implies that the abstraction I have to work with in terms of memory is the `Dataspace` [Gro13a] (see section 2.4.4 on page 20 for an overview of L4Re abstractions in general and dataspaces in particular).
As discussed as requirement R1, migration of memory has to be supported. This leaves me with the following two possible approaches:

- *Virtualize Dataspaces.*

  The `Dataspace` API is based on IPC and hence can be interposed. As a result, I can replace a `Dataspace` object with a proxy object. This process is comparable to what I did with the `Scheduler` and `Factory` objects earlier. Virtualizing a dataspace in that way would allow for deciding whether to use my own implementation for providing a functionality or to forward a request to an existing implementation as provided by Moe, for instance.

[2] A dataspace can be shared among different tasks—only the initial ownership can be queried using this approach.

- *Integrate my changes into Moe.*

  A different approach can use the circumstance that Moe already provides a sophisticated `Dataspace` implementation (or better: multiple implementations because there are more than three different types of dataspaces available). Using Moe as a starting point gives me full control over all physical memory allocation aspects, e.g., if a dataspace's memory is allocated only after a page fault occurred or eagerly during dataspace creation.

Both approaches have their advantages: with my own dataspace implementation, I do not have to touch Moe's codebase and the code can be contained solely in my NUMA manager. Using Moe would allow for reusing existing and tested code while not introducing another level of indirection.
However, I chose to virtualize dataspaces and to provide my own dataspace implementation. This approach provides most flexibility in terms of implementation choices. This way, I have greater freedom for the implementation of memory migrations, for example. Adapting Moe's `Dataspace` implementations restricts me to a design that fits best into the existing code. Moreover, this decision minimizes changes to existing components (as stated by requirement R4).

Dataspace Implementation

The `Dataspace` interface comprises several methods. The most important ones in the context of this work are the following three:

`map`	The `map` method maps a memory region of a `Dataspace` to another task using the IPC map mechanism. This is the method invoked by region maps in order to resolve a page fault.
`allocate`	A dataspace's memory is typically allocated on demand, i.e., when an access happens and a page fault is triggered because a part of it is not yet mapped. Using `allocate` can speed up page-fault processing by preallocating the memory backing a mapping.
`copy_in`	A memory region of a `Dataspace` can be copied from a source `Dataspace` using the `copy_in` functionality.

Further methods exist for mapping larger regions, clearing parts of a dataspace, querying certain parameters such as the size or the physical address of a dataspace's memory area, and taking and releasing references to such an object.

My dataspace implementation is straightforward: it is built around a `Dataspace` object allocated from Moe. Most of the functionality described earlier can just be directly forwarded to this dataspace. The `map` method is implemented from scratch. A simple redirection to the wrapped object does not work, as this would map the region of interest only into the NUMA manager's address space.

Dataspaces are usually created using objects implementing the `Mem_alloc` interface [Gro13c]. Like `Scheduler` and `Factory`, a capability to an object implementing this interface exists in every L4Re task. By default, this capability refers to Moe but

my NUMA manager provides an implementation of this interface as well. I use the same mechanism as for the `Scheduler` object to overwrite this capability for any newly started NUMA task in order to redirect the allocation request to my NUMA manager.

Region Management

Unfortunately, virtualizing dataspaces works from a conceptual point of view but the implementation is not working correctly out of the box. The problem is the region map implementation provided by Moe: as depicted in section 2.4.4 on page 20, the region map is the component that manages the virtual address space. As such, it is responsible for deciding at which address to attach a dataspace and, the other way around, querying which dataspace belongs to a certain address so as to resolve a page fault, for example. The second case is causing problems with virtualized dataspaces: Moe's region map implementation does not support dataspaces that were not created locally, i.e., within Moe itself.
Normally, when a page fault occurs, the region map receives a dataspace in the form of a capability. Two cases are distinguished here: when the dataspace implementation resides in the same task the region map code is located in, a reference to the local object (a memory address) is received. When a dataspace references a remote object, i.e., one implemented in another address space, a capability to this object is mapped locally for the time it takes to handle the page fault.
This latter case is precisely what happens for my virtualized dataspace objects for they are located in my NUMA manager and not in Moe. When a NUMA task triggers a page fault, Moe's region-map object is invoked. It receives a capability to a dataspace object residing in my manager task. For such a remote object, Moe is unable to handle the page fault and returns an error code.

In order to solve this problem, I implement my own region-map class. Most of the code for attaching and detaching dataspaces and resolving page faults already exists in L4Re and can be reused. However, a lot of scaffolding code has to be provided as well to make use of this functionality.

Memory Migrations

The ability to perform memory migrations is one of the goals of my work. The design space for a possible migration implementation is large.
First, there is the question of the type of migration:

Eager Using an eager approach, when the migration request is received, the whole dataspace is migrated subsequently. Typically, this migration happens in the background (i.e., in another thread). The request is acknowledged in the meantime and execution can continue.

Lazy Migration can also happen lazily. In that case, only accessed pages are migrated from one node to another when a page fault occurs. All pages stay where they are for the time being.

Both schemes have their advantages and disadvantages: eager migration can put a lot of pressure on the memory subsystem while the entire dataspace is copied. This can influence other applications running in parallel. Furthermore, if migrations happen frequently, data can be copied to a different NUMA node without even being accessed between two migrations. This is particularly a problem for large memory regions that are accessed only selectively. Other research suggests circumventing this problem by copying only the working set [LS12].
The working set comprises a certain number of pages that got accessed last in the past. Determining the working set is a viable option but requires knowledge about which pages belong to this set in the first place. Acquiring this knowledge using Fiasco.OC is not trivial because the API usable for retrieving page access information is limited: it only provides an accumulated access count for a range of pages with each system-call operation. For detailed information about which page got accessed, this system call has to be performed for every page—a significant overhead for large memory areas.
Lazy migrations do not suffer from the two aforementioned problems. However, as data is only copied upon access, i.e., when a page fault occurs, handling these page faults takes longer than usual caused by an additional copy operation being performed. Also, because data is not mapped eagerly, more page faults occur in the first place.

Due to the additional overhead in terms of potentially unnecessary memory traffic and the complexity of estimating the working set, I made a decision in favor of lazy migrations.
The first step for making memory migrations possible to clients is to adapt the `Dataspace` interface. I added a single method:

`migrate` Migrate a `Dataspace` object from one NUMA node to another. It is up to the implementation to decide how to treat requests on a dataspace that is not migratable.

With this change, clients have the ability to request migration of the entire dataspace to another node. Like for all other methods, a new opcode had to be defined that identifies the IPC operation which requests the migration. On the server side, i.e., within the dataspace implementation, a handler for this opcode needs to start the migration.
When it receives a migration request, the handler unmaps the entire dataspace from the task it belongs to as well as from all the NUMA manager's clients that received the dataspace. This unmapping leads to the receipt of page faults for subsequent accesses to the dataspace's memory. The dataspace then allocates a new dataspace on the destination node. When a page fault is received and redirected to the `Dataspace` object's `map` method, it looks up the page in which the page fault occurred and determines on which node it is currently allocated. If this node does not match the migration's destination node, a migration of this page is performed.
For keeping track which node a dataspace's pages are allocated on, I use one byte of additional meta data per allocated page. This meta data is also used to determine when the migration is completed and the internal source dataspace can be released: if no pages are resident anymore on the source node.

Interleaved Dataspaces

One possible mitigation strategy for remote access penalties and shared resource contention represents interleaved allocation of memory. To support this type of allocations, I created a new dataspace type within the NUMA manager: `DataspaceInterleaved`. A `DataspaceInterleaved` object comprises regular dataspaces allocated on the different NUMA nodes of the system. For my Dell workstation, two dataspaces are used.
The implementation is straightforward: each dataspace is attached in the NUMA manager's address space. Upon a `map` request for the `DataspaceInterleaved` object, pages are mapped out from each regular dataspace in a round-robin fashion.
Objects of this new type do not support memory migrations using the newly introduced `migrate` method because the memory is already distributed among multiple nodes. To that end, a memory migration request using the previously developed `migrate` method is always acknowledged directly with an indication of success.
For requesting a dataspace with interleaved memory, I enhanced the `Mem_alloc` interface with an additional flag that can be passed to the `alloc` method: `Interleaved`. A `Mem_alloc` object that receives an allocation request with this flag and that supports memory interleaving can decide to hand out a dataspace with this property. My NUMA manager—as an object implementing the `Mem_alloc` interface—can return an object of the just described `DataspaceInterleaved` type.

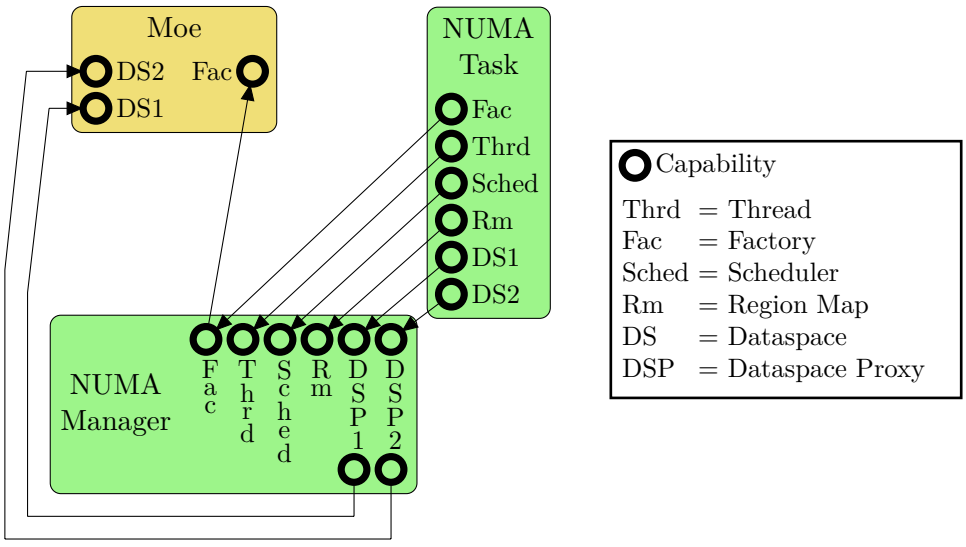


Figure 3.4: NUMA Manager and Virtualized Objects

3.3.4 Overview

The previous sections elaborated on the basic design of my NUMA manager and the virtualization techniques for L4Re objects it uses. Figure 3.4 summarizes the design by giving a schematic overview of the NUMA manager and a managed NUMA task. It also illustrates the interaction with Moe. Moe (or better: its `Factory` object) is used for creating new L4Re objects such as tasks, threads, and IPC gates.

Following is a summary of the intercepted objects and the rationale for the virtualization.

`Factory` The NUMA manager needs a way to intercept creation of L4Re objects, such as tasks and threads. I virtualized the `Factory` object as to redirect creation requests through my manager component. Most requests get forwarded to Moe directly but the created objects stay mapped within the NUMA manager to work with them later on. The `Rm` object forms a special case in this regard for it is not created by Moe (see below).
For each managed task, a separate `Factory` object exists in order to establish a thread-to-task relationship.

`Scheduler` The `Scheduler` object is used for changing the CPU affinity of managed threads which were previously created through a `Factory` object. All of a client's scheduling requests pass through the NUMA manager and the desired CPU affinity is overwritten to enforce a certain scheduling policy.

`Mem_alloc` The `Mem_alloc` interface allows to intercept the creation of dataspaces.[3] For the NUMA manager, I extended this interface with a new flag value for requesting a dataspace with interleaved memory as well as with an additional parameter to specify on which node to allocate memory (see section 3.4.1 on page 58).

`Dataspace` Once an allocation request reaches the `Mem_alloc` object, it decides what type of `Dataspace` object to create. My manager provides dataspaces with interleaved memory, dataspaces that use memory from a certain node, and dataspaces copied from unmanaged ones (not described in more detail). All dataspaces wrap other ones created by Moe. Visible to clients are only the NUMA manager created dataspaces—they act as proxy objects.
Additionally, my `Dataspace` implementation is enhanced with memory migration support. For that matter, I extended the `Dataspace` interface with a `migrate` method.

`Rm` Because of limitations in Moe's `Rm` class, the NUMA manager has to provide its own `Rm` implementation to clients. Objects of this type can handle all types of managed dataspaces (see page 51 for more details).

As explained earlier, virtualization of L4Re objects is possible through Ned. Its scripting capabilities allow for an easy replacement of objects in tasks which are to be created. To give the reader an impression of this process, a sample configuration file is shown in listing 3.1 on the next page. The listing depicts the `start_taskv` function from the `numa.lua` Lua package that I created. The function accepts a channel to the NUMA manager as one of its parameters.

[3] No `Mem_alloc` object is shown in figure 3.4 because internally, the corresponding protocol is handled by the `Factory` object as well.

The important parts are as follows. Line 8, where a per-task factory object is created. Lines 13 to 20, where four out of the seven default object capabilities are replaced. The `Scheduler` capability references the NUMA manager directly. The `Factory` ("`factory`"), `Mem_alloc` (here dubbed "`mem`"), and `Rm` ("`rm_fab`") capabilities refer to the per-task factory object. With regard to threads, knowledge of this per-task factory helps to establish the thread-to-task relationship.
Line 29 embeds the NUMA manager capability into the client task's environment. By using this capability, the client can communicate with the NUMA manager and, for example, query on which CPU one of its threads is running.
Finally, line 30 invokes the `startv` function in order to start the new task. The function call is not different from that used for regular application start-up in L4Re using Ned.

```
1  function start_taskv(numa_mgr, env, cmd, posix_env, ...)
2    -- create a new factory that is to be used for 'factory',
3    -- 'mem', and 'rm_fab' objects for every newly started task;
4    -- the 'factory' serves general requests like task, thread,
5    -- and ipc gate creation, whereas the 'mem' factory
6    -- implements the mem_alloc interface for creating
7    -- dataspaces
8    local numa_factory = numa_mgr:create(L4.Proto.Factory);
9
10   -- overwrite the 'factory', 'scheduler', 'mem', and 'rm_fab'
11   -- capabilities and refer them to the numa_manager instance
12   -- and the per-task factory, respectively
13   local numald = L4.Loader.new({
14       scheduler = numa_mgr,
15       factory = numa_factory,
16       mem = numa_factory,
17       log_fab = L4.default_loader.log_fab,
18       ns_fab = L4.default_loader.ns_fab,
19       rm_fab = numa_factory,
20       sched_fab = L4.default_loader.sched_fab,
21     });
22
23   if not env["caps"] then
24     env["caps"] = {};
25   end
26
27   -- insert a capability into the environment under which
28   -- the numa manager is reachable in client tasks
29   env["caps"]["numa_mgr"] = numa_mgr;
30   return numald:startv(env, cmd, posix_env, ...);
31 end
```

Listing 3.1: NUMA Task Object-Virtualization

3.3.5 Clean-Up

With the NUMA manager in place, a new problem arises: the clean-up of applications. Normally, when an application exits in any way, clean-up of resources is performed. Memory is then reclaimed by the system as to allow for future use by other tasks, for instance. Also, kernel-level data structures for the task of concern are released at that time.
The latter point is also important in my context: if too many tasks are left in a zombie state, that is, they no longer actively execute code but still exist in the form of an address space, the kernel eventually runs out of kernel memory. The problem exists even on x86_64, i.e., a 64 bit system, with large amounts of virtual memory available.
This phenomenon manifests itself when a task requests new virtual memory mapping but not enough space for page-table creation is available, for instance. As a result, the kernel quits the request with an out-of-memory error.

Reference Counted Kernel Objects

A task can vanish entirely when all references to it are dropped. In the narrow sense of the word, a task is just the task capability: a kernel object representing an address space. All kernel objects are reference counted and tasks are only one kind of kernel object (see section 2.4.2 on page 18 for a brief presentation of other types of kernel objects). To enforce this behavior, the kernel needs to know the number of tasks that hold references, i.e., capabilities, to such an object. This knowledge is available because mapping or granting a capability to a different task can only be performed with support by the kernel. Only when the last capability to an object is unmapped from a task, the kernel can remove the actual object and release internal data structures.

The NUMA manager acts as a proxy for certain L4Re objects. In this function, it keeps a list of capabilities to all tasks and threads it created. All those capabilities are mapped in the NUMA manager which means that it holds a reference to the corresponding kernel object. In order to release this reference, the manager has to unmap the capability from its own task. For doing so, it needs the information that a managed task has finished execution.
In the normal flow of operation, a parent is notified via an IPC message once a task terminates. Unfortunately, the parent is Ned for it is the default loader I use. Due to this aspect, there is no notification of the NUMA manager.

Notification of NUMA Manager

In order to notify my NUMA manager that a task is no longer running and does not have to be managed anymore, I have two choices:

- *Virtualize the Parent protocol.*

  The notification sent to the parent whenever a child task terminates uses IPC functionality and IPC gates. Being IPC based, it is eligible for virtualization as I performed earlier for the `Scheduler` object, for example. Virtualizing this `Parent`

protocol would allow my NUMA manager to intercept the notification, react to it, and forward it further to the actual parent that started the task.

- *Adjust the loader to inform the NUMA manager.*

  The direct modification of the loader to inform my NUMA manager forms constitutes another approach. A loader is a component that starts a new task by loading the executable into memory, requesting creation of the address space as well as the initial thread, and pointing the thread to the first instruction to execute. Because the loader starts a program, it usually registers itself as the parent in order to free allocated resources after termination.
  The loader could be modified in such a way as to inform the NUMA manager about the termination of a client once it received that notification itself.

The first approach is the more generic one: it works with all loaders. In the ideal case, it would also only require additional code on the NUMA manager's side and a small bit of glue code that replaces the default parent capability when a NUMA task is started. Unfortunately, the loader I use—Ned—does not support virtualization of the `Parent` protocol from within the Lua scripting language. As adding such functionality in a well-conceived way is not trivial, I opted for the second approach.

Implementation

My approach uses an implementation detail to accomplish the goal of notifying the NUMA manager. If the manager is in use, I have to virtualize the `Factory` object, i.e., replace the default `Factory` capability with one pointing to my NUMA manager (see section 3.3.2). The moment Ned starts a new task, it uses this `Factory` capability to request, among others, creation of the actual task capability and of a region map object. I store the `Factory` capability within Ned's data structures dedicated to that task.
When Ned now receives the notification that the application terminated, I perform an IPC send operation based on the `Parent` protocol to the object behind the stored capability. In my case, this object is the NUMA manager. It behaves as if it was the task's parent itself and can unmap the client's capabilities. After Ned dropped its references to the task's objects, the reference counts decrease to zero and the kernel can release all memory allocated to that task.
The case that Ned is used without my NUMA manager is also covered by this approach: the send operation does not block forever but times out after one second. If no one receives the it or the `Parent` protocol is not understood by the receiving object, Ned continues execution as usual after the time-out elapsed.
The disadvantage of this approach is that it only works for Ned and requires additional modifications in case other loaders are used in parallel. However, this circumstance does not pose much of a limitation in the context of my work—I rely on Ned's functionality for virtualizing the other objects anyway. In any case, exchanging the loader requires additional work.

3.4 NUMA Manager Integration

At the lowest level, the NUMA manager relies on several mechanisms offered by the underlying system. This section describes three of these mechanisms that I had to adapt for the manager to provide its functionality: allocation of memory, naming of CPUs with respect to the system's topology, and support of performance counters.

3.4.1 Node-Aware Allocations

One requirement of the NUMA manager implementation is the support of node-aware allocations: when requesting memory from the system, it should be possible to specify from which NUMA node it is to be allocated.
Theoretically, the first point of decision regarding the implementation is whether to use a user level or kernel land approach. However, in a microkernel environment with all user-visible memory managed at user level, the kernel is of no concern for the implementation of this part. At user space, three possible approaches come to mind:

- *Enhancing Sigma0.*

  Sigma0 is the first component that gets mapped all user-space available memory in the form of 1-on-1 physical-to-virtual mappings.

- *Adapting Moe.*

  Moe is the first L4Re service and also acts as the root memory allocator. It requests a 1-on-1 mapping of all memory from Sigma0. Thus, Moe also has knowledge about physical addresses and can relate memory regions to NUMA nodes.

Choosing Sigma0 as a starting point would allow for the lowest possible implementation level: all memory is initially mapped to Sigma0 and memory-related abstractions are built on top of other components that rely on it. However, the latter point is also the greatest disadvantage. All "ordinary" L4 programs rely on L4Re. They use abstractions such as dataspaces and memory allocators for obtaining memory for their use. Making Sigma0 NUMA-aware would also require adjusting Moe, the root L4Re task, to make use of this knowledge.
As it already provides memory allocations within L4Re, adjusting Moe is the more appropriate solution.

NUMA Node Information

As described in section 3.2.1 on page 37, I use a Dell workstation for testing my work. Since this machine is the only one I can effectively evaluate my work on and because acquiring topological knowledge dynamically at boot or run time is out of scope of this thesis, I choose to hardcode the assignment of physical memory to NUMA nodes into Moe.
When talking about memory, the physical memory address provides the piece of information on which NUMA node the memory resides. In order to get this information for

all memory ranges of interest, I booted the system with a Linux installation (I described this setup briefly in section 3.2.2 on page 39).
As explained before in the State of the Art chapter (see section 2.3 on page 13), Linux queries a variety of architectural data for the machine. At boot time, it prints out the memory ranges that belong to each of the system's NUMA nodes. Using the `dmesg` command, I am able to extract this information.

Adapting Moe

Moe uses a list-based allocator for serving allocations at the granularity of a page. At its start-up, Moe frees all available memory regions to this allocator—based on information provided in the KIP. Later, when memory is requested by clients in the form of a dataspace, it carves out a chunk, creates a dataspace, and hands out this dataspace.

My modifications to Moe duplicate this allocator: I use one allocator per NUMA node. When memory is reserved initially, I decide, based on the address of the memory region, which allocator to grant the memory to. In case a region to reserve spans two nodes, I perform a splitting operation. Freeing memory back to the allocator works analogously: based on the provided address I determine which allocator to free the memory to.

This approach, as-is, suites my needs for memory allocation perfectly. However, for systems with a large number of small memory nodes, problems can arise for large allocations of physically contiguous memory. If more memory is requested than is available from a single allocator, memory from other allocators has to be used for the request as well. As such systems are not of concern for the work at hand and only few use-cases require physically contiguous memory at all, I ignore these problems.

Adapting the Allocation API

Having adjusted Moe still leaves the outside with no option for specifying the NUMA node to allocate data from. In order to provide clients with such an option, the memory allocation API has to be adjusted.
This memory allocation API, `Mem_alloc` in L4Re jargon, consists of two functions that can be invoked on the memory allocator object (which is represented by Moe, in my case) [Gro13c]:

`alloc` A `Dataspace` object can be allocated using the `alloc` method. It accepts parameters such as the desired size of the dataspace and a set of allocation flags for requesting additional properties, for instance, physically contiguous allocation, usage of *Superpages* (Superpages are the L4Re term for memory pages of a larger size: 4 MiB instead of the default 4 KiB per page for the x86 architecture) or pinned memory (i.e., memory that cannot be swapped out).

`free` The `free` method can be used to free a dataspace back to the allocator. It is deprecated in favor of a more generic approach where a dataspace frees itself when its reference count drops to zero. If the dataspace is not shared among multiple clients that still hold references to it, the memory can be reused for other dataspaces.

My change to this API comprises a new parameter for the `alloc` method that specifies the NUMA node from which to allocate the dataspace's memory. Moe directly uses this parameter to allocate memory from the corresponding list-based allocator.

3.4.2 CPU IDs

When conducting experiments that refer to different CPUs, for instance, for placing application threads only on cores of a certain NUMA node or for identifying on which NUMA node a thread that triggered a page fault is running, CPU IDs are used. These IDs should help to establish a relationship between a core and its location in the system (the NUMA node, for example).
CPU IDs are continuous numbers starting at zero which the kernel assigns to each logical CPU, i.e., each executing unit a thread might be scheduled on. For example, on a machine with four NUMA nodes each containing four cores as previously shown in figure 2.1 on page 4, IDs 0 to 3 might reference all cores on the first node, IDs 4 to 7 the cores on the second node, and so on.

Non-Deterministic IDs

On Fiasco, however, CPU IDs are assigned in the order the CPUs are initialized during kernel boot-up. This behavior makes them non-deterministic between reboots: there is no relationship between the CPU's ID and the system's topology because CPUs might start up in any sequence.
In order to make experiments reproducible, these IDs need to be fixed with regard to the topology—it must be known to which NUMA node each CPU ID belongs.
There are two possibilities of how to approach this problem:

- *Implement a CPU ID mapping in user space to translate kernel assigned IDs.*

  This approach, implemented entirely in user land, builds a map of logical IDs that are always fixed with regard to the topology. A user level implementation is feasible because instructions such as `cpuid` for x86_64 are unprivileged. This instruction allows for querying an identifier for the core the instruction is issued on [Coo13c]. This identifier, the *APIC ID*, is unique among all of the system's cores and stays the same between reboots.
  Based on this information, a mapping from kernel IDs to these topologically fixed IDs can be created. Everytime the kernel passes a logical CPU ID to user space, it has to be translated by means of this map and can subsequently be used in contexts where a fixed relationship between logical CPUs and physical cores is required.

- *Adjust the kernel to work with fixed IDs.*

  Since the kernel assigns these IDs to logical CPUs, it is possible to adjust this component directly to fix the problem of non-determinism. As in the previous approach, the `cpuid` instruction could be used to identify each core. A logical CPU ID can then be derived from the reported identifier.

Both options are equally viable but the user-level approach presumably bears more problems.
When creating a translation map in user space, as suggested by the first approach, it has to be used for every interaction with the kernel where logical CPU IDs play a role. As in general there are multiple user land tasks running, a centralized service or a shared library that handles this translation needs to exist. Providing this infrastructure is a cumbersome task and requires modification of existing services and adaptation of all new programs to rely on this translation.
The kernel, on the other hand, is already centralized by design, as only one instance is running on the system at a given time.[4] If it was to work with these fixed logical CPU IDs in the first place, any translation would be superfluous. By adapting the kernel, it is not possible to mistakenly forget about places where a translation as provided by the first approach might be required.
Due to these advantages, I pursue the second approach of enhancing the kernel.

APIC IDs

For approaching the CPU ID problem, I started by implementing a function that, when executed on a logical CPU, returns a unique identifier for that CPU. As explained briefly before, the `cpuid` instruction can be used for this purpose [Coo13c].
Intel provides documentation and sample code about this instruction and information on how to retrieve this unique identifier, the APIC ID, on a multicore system [Coo12a]. I ported the corresponding retrieval function, here called `get_apicid`, to the Fiasco.OC microkernel.

Continuous IDs

It is not possible to directly use the APIC ID as the logical CPU ID because the kernel relies on all IDs being consecutive and starting at zero. However, `cpuid` does not guarantee this property: a sorted list of APIC IDs can contain holes. This characteristic exists because the APIC ID includes information about the position of the logical CPU in the system: a SMT ID describing logical processors sharing the same core, a core ID identifying the core within the CPU package, and a package ID referencing the CPU package [Coo12a].
Because the Fiasco.OC microkernel requires logical CPU IDs to be consecutive, I provide a lookup table that translates possible APIC IDs reported for my system into logical CPU IDs. I inferred the topological information required for this translation from the APIC ID itself.
The resulting IDs, generated by `get_cpuid`, can subsequently be used to infer knowledge about the system's topology: IDs 0 to 11 correspond to cores on the first NUMA node of the system, while IDs 12 to 23 belong to cores in the second node. With Intel's HyperThreading enabled, an even logical CPU ID (0, 2, …) represents the first hardware thread and the subsequent uneven ID (1, 3, …, respectively) refers to a core's second one.

[4] Virtualization and its ramifications are intentionally not considered here.

Adaptation

In the final step, the kernel has to be adapted to use the `get_cpuid` function. For Fiasco.OC, the place to hook in is during the boot process when all the system's CPU are brought up. Beforehand, only one CPU, the *boot CPU*, is used for executing kernel code. It starts the initialization process of all others by notifying them via IPIs (Inter-Processor Interrupts). Upon receipt of such an IPI, the CPU executes initialization code for, among other things, initialization of CPU local data.
This is precisely the moment when the logical CPU ID is set, for it is CPU local data as well. Using this knowledge, my patch replaces the creation of the logical CPU ID from an incrementing counter shared among all CPUs with a call to the `get_cpuid` function.

Since the kernel's logical CPU IDs are not changed later on and are used throughout the whole kernel, this modification is sufficient to have topologically fixed CPU IDs in the kernel and in user space.

3.4.3 Performance Counters

The NUMA manager uses performance counters to detect potential problems arising at run time. These counters are provided by the hardware and monitor hardware events. Examples of these events include, among others, LLC misses, retired instructions, TLB (Translation Lookaside Buffer) misses, and accesses to remote memory. Usually, the monitorable events vary between hardware vendors and generations.
The counters store their values in registers to which access is granted through the `rdmsr` and `wrmsr` instructions. As these instructions are privileged, kernel support is required in order to read or write the register values.
Basic infrastructure for configuring the available counters and reading their values via a system call is available in the form of a patch for Fiasco. Using this infrastructure, the performance counters are programmed for the executing logical CPU, i.e., all threads running on this logical CPU use the same counter.[5]

Access Interference

Unfortunately, for a deeper analysis at the level of threads as is preferable for my work, the setup as just described is not suitable.
Figure 3.5 on the next page illustrates the performance counter problem. It shows two threads scheduled on the same logical CPU and competing for the same performance counter register.
First, the scheduler decides to give "Thread 1" time to run. While executing, it triggers two events causing the performance counter register to increment twice (resulting in a value of two). After a certain time, the thread gets preempted and "Thread 2" is eligible to run. In a *Preemptive Multitasking System* such as L4, preemption of user tasks can happen at any moment and is completely transparent to threads from a user space perspective. The second thread also triggers a counter event and then reads the counter's value and resets it to zero.

[5] There are also performance counters that are located in the *Uncore*, i.e., in an off-core but on-die area where other shared hardware resources like the LLC reside.

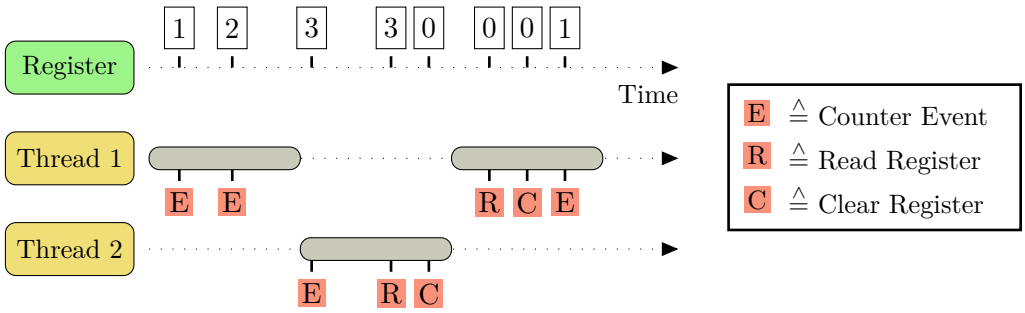


Figure 3.5: Threads Using a Counter Register

When "Thread 1" is scheduled again, it reads the register and retrieves a wrong value (zero in this case), not representing the actual number of events this thread triggered (which would be two).

Multiplexing Thread Accesses

To gain insight into which thread causes which effect on a certain performance counter, access to each counter register has to be multiplexed among threads. Multiplexing has to take place within the kernel because that is the location where preemption happens and scheduling decisions regarding the next thread to run are made. Note that this type of scheduling is different from the one happening at user level, for instance, in the previously presented `Scheduler` object (see section 3.3.2 on page 46).

To achieve this multiplexing, I hook into the `Context` class in the kernel's scheduling code. Everytime a new thread is scheduled to run, the `switch_cpu` method is invoked. Its parameters are, among others, the source context (i.e., the thread that was running before) and the destination context (the thread that is to be scheduled).
I extend this class to provide additional storage for the counter registers and the corresponding configuration registers whose values determine which event to monitor. Using this thread-local storage, I can now load and store the current register state for every thread upon an invocation of `switch_cpu`: store the current hardware register values into the source context's register storage and load their new values from the destination context's shadowed registers.
This way, interference with other threads cannot happen anymore: at every point in time, only one thread is able to execute on a given logical CPU. This thread alone is allowed to access the core's performance counter registers. Every other thread potentially running has a shadow copy of the registers. Only once scheduled for execution, a thread overwrites the hardware counter value with its shadow copy. The moment the thread gets preempted again, it stores the content back into its local copy. To that end, the performance counter register values appear to be isolated from each other.

Figure 3.6 on the next page illustrates my solution using the example shown earlier. Initially, "Thread 1" contains the local register value 0. When it is scheduled, it loads this value into the performance counter register. Subsequent events increase it to 2.

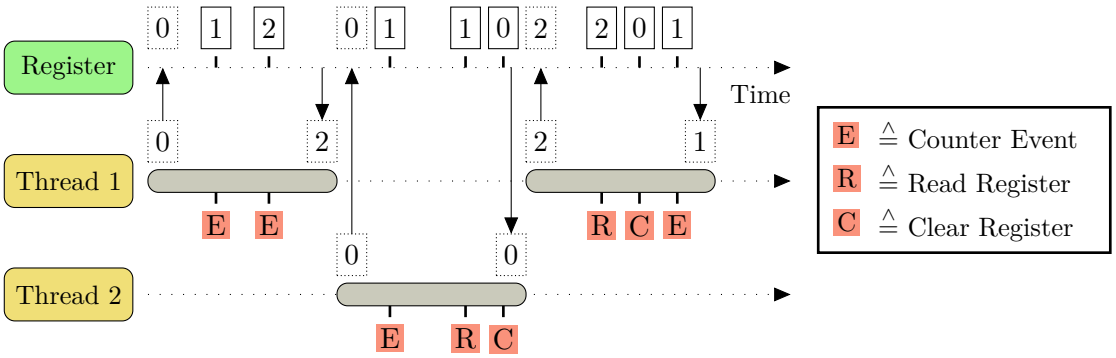


Figure 3.6: Threads Multiplexed on a Counter Register

After the thread got preempted, it writes this last value back into its shadow copy. "Thread 2" then loads its shadow-copy values, triggers an event, reads the new hardware counter value, clears the register, and stores the state locally. When "Thread 1" is scheduled again, it can restore its saved value, thereby ignoring any previous events that changed the register value while it was not executing.

This chapter summarized the design of my NUMA manager and explained the adaptations that were required to make it work on the system. I chose an implementation based on virtualization of L4Re objects which enables unmodified client applications to benefit from NUMA-aware thread and memory management.
The next part addresses the evaluation of the program so as to determine what overheads it causes and what benefits it can provide.

4 Evaluation

In this chapter, I present experiments that evaluate my work. In section 4.1, I analyze the overhead the NUMA manager causes over the case where it is not used. The subsequent part, section 4.2 on page 70, investigates performance of the two mechanisms the manager uses frequently: migration of threads and memory as well as reading and configuring performance counters. Section 4.3 elaborates on the design of thread-placement and memory-allocation policies for the NUMA manager and investigates what performance benefits this component can offer to applications. This chapter's final section, section 4.4, provides insights into the complexity of the code required for the manager service.

4.1 NUMA Manager Overhead

The NUMA manager causes additional indirections: it intercepts interaction with most of the L4Re objects. Later on, it receives requests for the virtualized objects and forwards them to the wrapped ones. Such a behavior leads to additional communication which can influence performance negatively. The section at hand examines the overhead this concept introduces.

4.1.1 Application Start-Up

While developing and testing features for my NUMA manager, I witnessed a delay in application start-up over the case where no NUMA manager is used. In order to quantify this delay, I compare start-up times for a minimal application written in C when started from Ned directly and when managed by the NUMA manager. The application has an empty `main` function and does not perform any additional work.
For brevity, I use the terms "unmanaged" and "managed" to refer to an application started in the default setup, i.e., without any NUMA manager involvement, and started using the NUMA manager, respectively.

Measurements

I executed the aforementioned application 100 times in a row in a managed and unmanaged fashion and measured the time it takes until it terminates. I then calculated the average time for a single application invocation. Table 4.1 on the next page shows the mean of those average times for five runs as well as the standard deviation from this mean.
Note that the times comprise both the start-up as well as the tear-down time of the application. The tear-down time is interesting because of the additional clean-up that my NUMA manager has to perform (see section 3.3.5 on page 56).

Type	Time	Std Dev
unmanaged	20.36 ms	$1.03 \cdot 10^{-1}$ ms
managed	50.24 ms	$2.02 \cdot 10^{-1}$ ms

Table 4.1: Application Start-Up Times for Managed and Unmanaged Tasks

Evaluation

The data illustrates that the NUMA manager is not for free: compared to the approximately 20 ms it takes for an unmanaged task to start and terminate, a managed one requires roughly 50 ms. This 30 ms delay can be attributed to the indirections introduced by my NUMA manager. These indirections mainly manifest in the form of additional inter-process communication. For instance, when a task creates a new thread, it sends the respective request to the NUMA manager. The latter then forwards this request to its default factory, i.e., Moe. Moe creates the thread and returns the capability. The NUMA manager can then acknowledge the initial request back to the client task and map the thread capability.

Although 30 ms are not a negligible amount of time, the time required for an application to finish its work in normal scenarios exceeds the start-up and tear-down times significantly. To that end, that overall performance due to this additional work remains almost unchanged in most cases.
For this reason, I do not investigate start-up and tear-down slow-down in more detail but concentrate on other aspects that have an impact on application performance at run time.

4.1.2 Page Fault Handling

As explained in the context of the NUMA manager's design, I provide my own `Dataspace` objects and hand them out to NUMA tasks. Internally, these objects wrap dataspaces created by Moe. This wrapping can have implications on page-fault handling performance due to the additional communication required between a NUMA task and the manager.
In order to assess page-fault handling times and to compare them for the managed and unmanaged case, I created an application that allocates and attaches a dataspace and sequentially touches each of the pages. This operation triggers a page fault which has to be resolved.

Measurements

I executed this application with different dataspace sizes (being a power of two) for the managed and unmanaged case. In each experiment, the sequence of allocating, touching, and releasing the dataspace is performed ten times in a row. Then, the average run time is calculated.

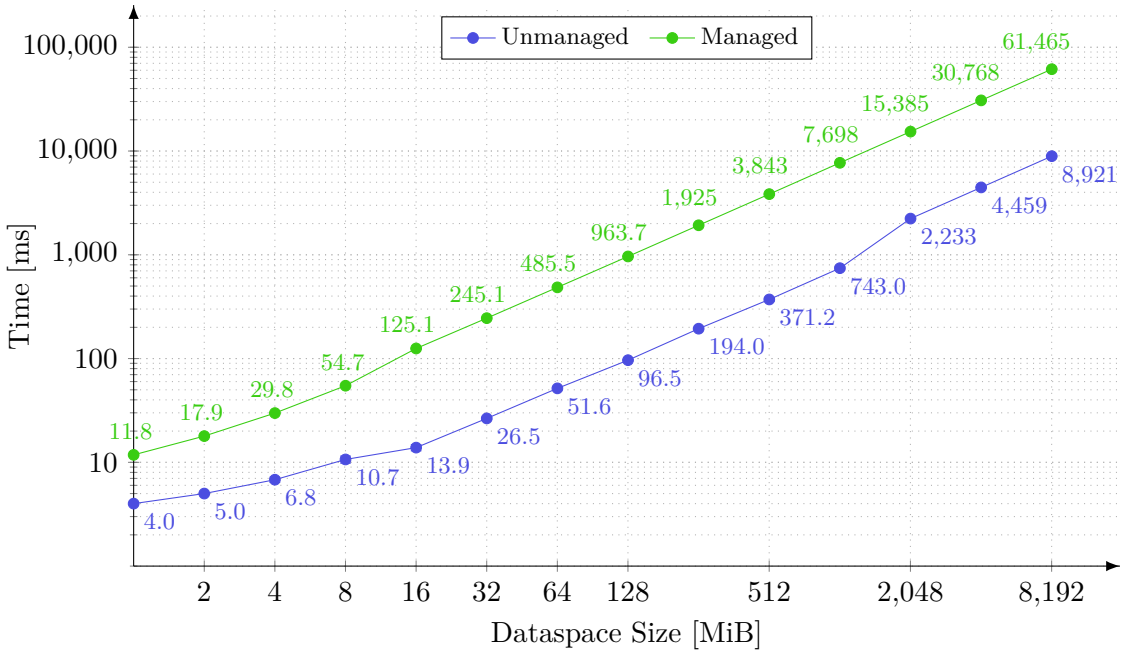


Figure 4.1: Dataspace Mapping-Times

Figure 4.1 depicts the time it takes (y-axis) to completely map a dataspace of the size specified on the x-axis for the managed (—●—) and unmanaged case (—●—). Note that both axes have a logarithmic scale.
Shown is the arithmetic mean of five performed runs. The standard deviation is less than 7.86 % of the mean and too small to show up in the plot.
The times include allocation and release of the `Dataspace` object itself as well as allocation for each of the memory pages before they get mapped. However, as both the managed and unmanaged case have this overhead, the values can be compared.

Evaluation

The data indicates that mapping a dataspace through consecutive accesses in a managed application takes significantly longer than in an unmanaged one. The difference in the time it takes to resolve the page faults between both cases can be as high as a ten-fold increase for the managed case (with 965 ms vs. 97 ms in the case of a 128 MiB dataspace, for instance). Overall, however, the managed version's mapping times scale linearly with the size of the dataspace. The run-time behavior is within the same complexity class.

The large difference in map times between the managed and unmanaged versions lead me to investigate the problem further. As it turns out, Moe, the component providing the dataspace implementation in the unmanaged case, uses a more sophisticated algorithm for mapping pages in case of a page fault: depending on the dataspace's size and the offset where the fault occurred, it maps a larger chunk of memory to the client. This behavior leads to less overall page faults because more memory is already mapped.

On Fiasco.OC, the element describing a memory mapping is a *flexpage*. Such a flexpage imposes two constraints on the area that is to be mapped in order to require only a single machine word of storage. First, a mappable region's size always has to be a power of two (and has to be greater or equal to the hardware page size). And second, the region's start address has to be aligned to its size [Lie96a].

Both constraints are trivially satisfied for a single page. This is precisely what my `Dataspace` implementation does in case of a page fault: it maps one page (the page where the page fault occurred) to the client.

A Changed Implementation

To allow for better comparability of the results and to increase overall performance, I adjust the mapping algorithm my `Dataspace` uses: I employ a similar approach to Moe where a flexpage describing a region that is larger than a single page is generated. Instead of the hand-crafted algorithm Moe uses, my implementation utilizes the `l4_fpage_max_order` function that creates the largest possible flexpage for the given dataspace region which includes the address where the fault occurred. The change is very small, only five lines have to be modified.

Figure 4.2 evaluates the changed implementation using the same experiment as before. In addition to the new managed version (—●—) and the unmanaged one (—●—), the curve of the previous implementation is plotted as well (- - -).

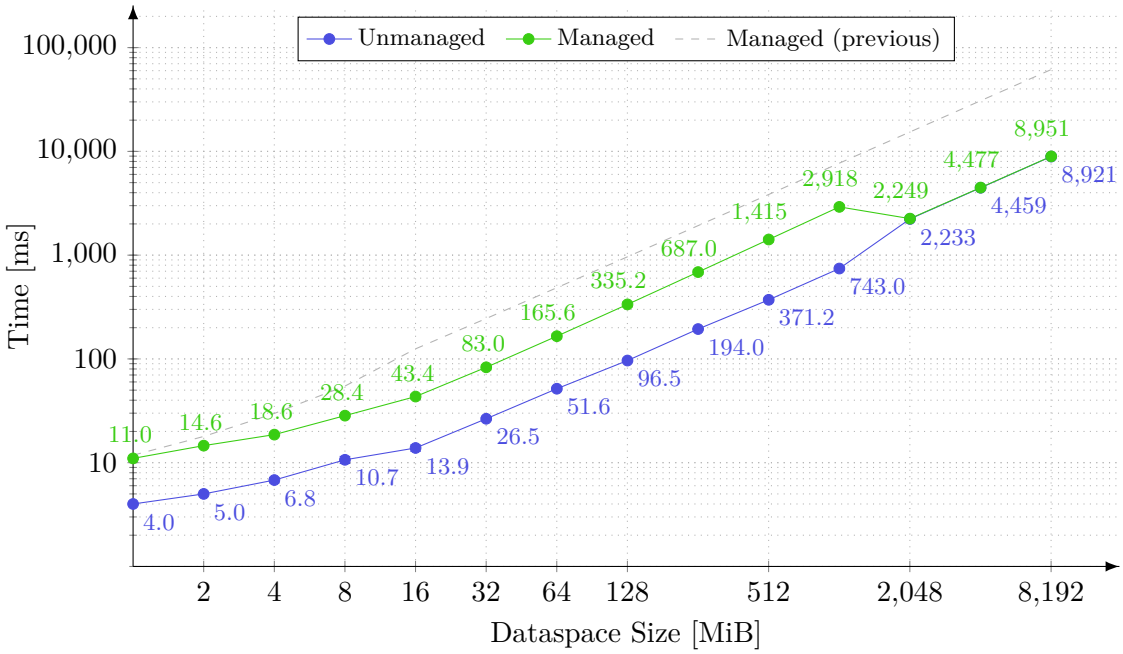


Figure 4.2: Dataspace Mapping-Times With New Map Algorithm

The new version provides a better performance than the previous one. The degradation over the unmanaged case went down to less than a factor of four in the worst case.

This factor can be explained with the additional communication and mapping overhead. Comparable to the application start-up experiment, communication in the case of a page fault happens between three parties: the client application that caused the page fault, the NUMA manager, and the component providing memory to the NUMA manager (Moe in my case). Moreover, all memory is mapped from Moe to the NUMA manager and to the client. Establishing this extra mapping induces overhead as well due to the page-table management required by the kernel.
With the new algorithm, a different phenomenon emerges: *less* time is required to completely map a 2,048 MiB dataspace compared to a 1,024 MiB memory region. This acceleration can be explained with the way the implementation works now: for a 2 GiB dataspace, a 1 GiB flexpage can be created and mapped in a single request. This reduces the number of follow-up page faults. In the 1,024 MiB case, such a large flexpage cannot be used because of the alignment constraint explained earlier—the region is only page size aligned. Such being the case, a smaller region of memory gets mapped and more page faults occur.

Overall, the diagram shows that the overhead for the managed version—although decreased significantly over the first approach—is still high. A factor of almost four is not negligible. However, I am confident that this degradation is not caused by a potentially bad implementation within the NUMA manager. The actual mapping code executed in the `Dataspace` implementation upon a page fault (i.e., the code lying in the hot path) consists of only a few lines plus parameter marshalling and unmarshalling as well as checking for error conditions.

My conclusion is that using the approach of virtualizing `Dataspace` objects might not be the best solution in terms of performance—implementing the desired dataspace functionality directly in Moe may be advantageous as this would remove the virtualization costs entirely (see section 3.3.1 on page 45 where I explain why I did not chose an implementation in Moe). The current implementation imposes a certain inherent overhead to the resulting system. Higher-level benchmarks on fully-fledged applications have to show whether or not dataspace mapping performance has significant influence on overall application performance.

4.2 Inherent Costs

The NUMA manager relies on migrations of threads and memory to alleviate architecture specific problems. For the assessment of such bottlenecks, it furthermore needs to configure and read performance counters.
This section examines the costs of these operations so as to understand the performance penalties they introduce and to estimate their value for dealing with problems such as remote access latencies.

4.2.1 Migration

My NUMA component employs two means to alleviate penalties on NUMA architectures: thread migrations change a thread's CPU affinity to make it execute on a different core—the goal being to move execution closer to the data being worked with. Migration of memory, on the other hand, moves the data towards the accessing entity.

In order to understand the downsides each type of migration has on program run-time, I conducted experiments that examine the different performance characteristics. The program I wrote performs a constant amount of work (sequential read and write accesses on a dataspace of 10 MiB; 10,000 iterations). Like before, I measured the application's run time. In the first experiment, the accessing thread is migrated periodically between CPUs on my system's two NUMA nodes. The second experiment migrates the dataspace's memory in the same fashion.

Measurements

Figure 4.3 on the facing page shows the program run-time as a function of the migration frequency for a thread (—●—) and a dataspace (—●—), respectively. Displayed is the mean of ten runs. The standard deviation is less than 6.29 % of the mean. Note that the migration frequency's scale is logarithmic.

Evaluation

The figure illustrates the main fact I want to point out: thread migrations are less expensive than memory migrations. Only at much higher migration frequencies—starting at around 1 kHz—an increase in program run-time is visible. When memory is migrated instead of threads, this causes performance to degrade steadily from about 1 Hz onwards.

Previous work already identified memory migrations as more expensive in terms of run-time overhead [Dre07b, BZDF11]. In order to migrate memory (between NUMA nodes, for instance), four operations have to take place. First, the memory has to be unmapped. Then, a new memory region has to be allocated. The next step is to copy the content from the source region to the newly allocated memory. Last but not least, the memory has to be mapped again. Typically, the latter process happens lazily, i.e., only upon access. Such an access triggers a page fault that has to be resolved. This lazy mapping is also the scheme I use (see section 3.3.3 on page 51 for more details on memory-migration support in the NUMA manager).

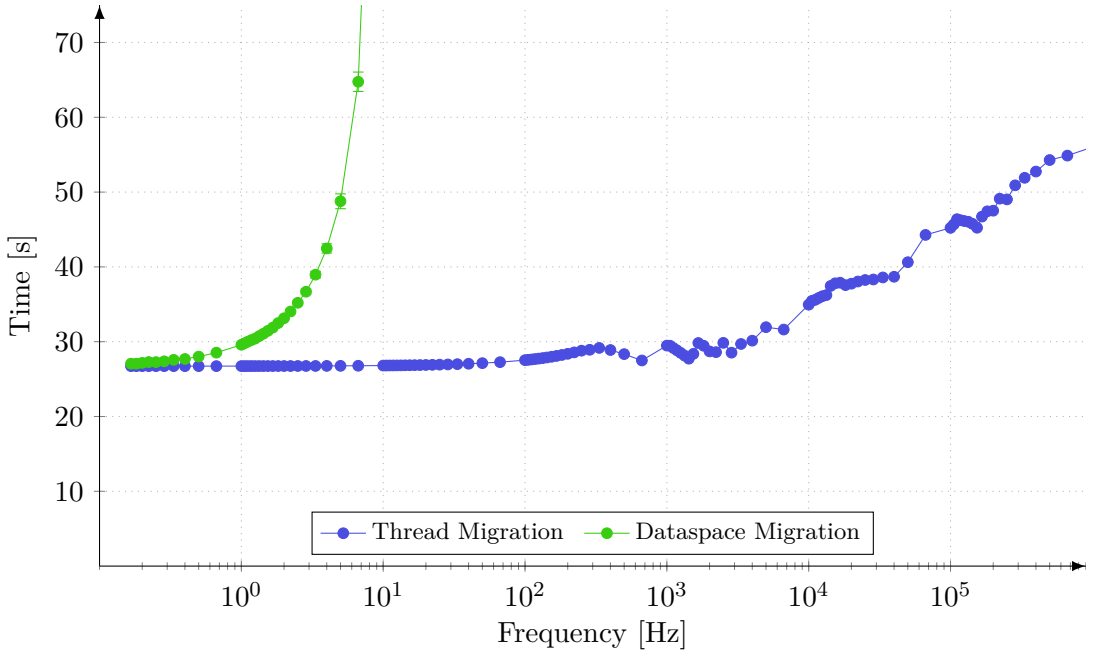


Figure 4.3: Performance Impact of Thread and Memory Migrations

A previous experiment (see section 4.1.2) already identified that page fault and mapping overhead are not negligible. Unmapping and copying the memory content is costly as well. Moreover, unmapping induces further indirect costs because of a TLB flush that has to be performed in order to remove the stale Page Table Entries (PTEs) from the cache. Afterwards, costly page-table walks are required to refill the TLB in case an access to an uncached page happens.
Furthermore, a memory migration in a microkernel such as Fiasco.OC, involves multiple switches from user space to kernel land (for unmapping as well as for the communication required to resolve page faults, for example), whereas a thread migration requires only one transition to kernel and back.

Thread migrations, on the other hand, have to—depending on the implementation—dequeue and enqueue the thread from the scheduler's CPU-local run lists and move its state to the destination CPU. This state mainly comprises the thread's registers, an amount in the order of 100 to 1,000 bytes. Copying an entire memory region of possibly several Megabytes is considerably slower.
Thread migrations also induce indirect costs. The caches on the destination CPU are likely to be cold, i.e., do not have cached entries of the data the thread is working on. For the thread, this means an increasing amount of cache misses will occur on the new CPU.

As before, a more in-depth analysis of all factors influencing performance is not possible within the available time frame and is also out of my thesis' scope. Overall, the experiment shows that the costs of memory migrations outweigh those of thread migra-

Access From	Time	Std Dev
Same Core	273 ns	$4.04 \cdot 10^{-2}$ ns
Different Core	3,373 ns	$3.01 \cdot 10^{1}$ ns
Different Node	6,183 ns	$7.73 \cdot 10^{1}$ ns

Table 4.2: Performance-Counter Access Times

tions significantly. This leads me to the conclusion that thread migrations should—if possible—be used in preference over memory migrations.

4.2.2 Performance Counters

The NUMA manager performs rebalancing to alleviate NUMA-related penalties which it detects based on performance metrics. These metrics are derived from performance events which in turn are monitored using performance counters. The counters, provided in the form of registers by the hardware (typically the CPU), can be queried and modified by software. Due to specifics of the x86_64 architecture which forms the basis of my implementation, kernel support is required for interaction with the counter registers.
My NUMA manager accesses these counters periodically. As explained before, they are multiplexed among threads, so that an analysis at the thread level is possible (see section 3.4.3 on page 62).
Reading the values requires a system call which introduces further overhead in addition to the actual read instruction. Depending on the number of managed threads, this process can introduce undesired performance penalties. In the following, I assess the overhead caused by accessing the counters.

Measurements

In order to get an impression of the overhead induced per thread, I developed a program that reads the performance counters' values in a loop. The loop executes 10,000,000 iterations. Note that each operation works on four performance registers in order to reduce the total number of system calls over the case where each counter register has to be handled separately. Four registers are read or reset with a single system call, for instance—a useful property because each counter can monitor a different event.

Table 4.2 shows the mean time it takes to access the performance counters. I distinguish between different thread-to-CPU affinities: when the accessing thread runs on the same core, when it executes on a different core but the same socket, and when it runs on a different socket. The mean is calculated based on ten runs.

Evaluation

The data illustrates that the location of the accessing thread is important for considerations of the run-time overhead induced. A thread that is running on a different core but the same socket as the thread whose counters to read requires an order of magnitude

more time to finish the operation (≈ 300 ns vs. ≈ 3,400 ns). When the accessing thread resides on a different socket, the required processing time is doubled once more with approximately 6,200 ns.
The additional overhead for accessing the registers of a thread which is executing on a different core and possibly a different NUMA node can be explained with the kernel internal mechanism used for handling a system call in that case. In simplified terms, the request and all the required arguments are placed in a queue. An IPI is sent to the target CPU to notify it about the new item in case its request queue was empty. The destination CPU dequeues the request and performs the required action. After that, the original request can be acknowledged back to user space.
Previous work determined the system call overhead on Linux to be in the order of 100 ns for entering and exiting the kernel [SS10, THWW10]. This is in line with my own experience stemming from work I have done in that area. The results indicate that Fiasco plays in the same league. With the additional work for reading the four registers and the overhead for accessing a "remotely" running thread, the measured run time is plausible.

As an aside: *reading* the performance counters is not the only operation performed by the NUMA manager. It also has to *configure* them and to *reset* their values. I performed the same experiment with these two operations. The times are comparable but not shown here for brevity.

4.3 NUMA Manager Performance Benefits

My thesis' main goal is to increase application performance by incorporating knowledge about the underlying NUMA architecture. The previous chapter described the NUMA manager component and the mechanisms it provides.
This section investigates the achievable performance of managed applications by employing different policies. First, I explain static policies used for placement of threads and dataspaces when they are scheduled or allocated, respectively. Afterwards, I elaborate on a more dynamic approach that identifies problems and bottlenecks at run time.

4.3.1 Static Policies

With the term *static policy* I refer to a placement decision that is made once and is not changed throughout the lifetime of the object of interest. For threads, this means I decide on which logical CPU to schedule a thread at the moment when it is first supposed to run. For dataspaces, the decision on which NUMA node to allocate the memory is made when the dataspace is created.

Thread-Placement Policies

In L4Re, new threads, unless explicitly stated otherwise, are always spawned on the first CPU, i.e., logical CPU 0. Specifying otherwise involves cumbersome and not very expressive specification of a CPU mask that has to be passed to the responsible `Scheduler` object.
In my opinion, this approach is neither user friendly nor very flexible. In the worst case, the thread-placement strategy has to be revised with every new task added to the setup. Moreover, even if a CPU mask is given, the default scheduler object places all created threads on the very first available CPU among the members of the CPU set.
For my experiments, I need an easier way of telling the system on which logical CPU to initially schedule a new thread. Furthermore, I want a more advanced way of automatic thread placement.
To accomplish these goals, I implemented four static thread-placement policies which provide the user with more flexible choices as to where threads are spawned initially:

`Fixed` With the `Fixed` policy, threads are executed on a specifiable logical CPU which does not change automatically at all. If a different start CPU is desired, the user has to explicitly set this CPU beforehand.

`RR Threads` Often, scheduling is performed in a round-robin fashion on the available CPUs [SGG09, Tan09]. With the `RR Threads` policy, a new thread is assigned to a certain start CPU. The CPU number is increased afterwards for every thread until it wraps around because the last logical CPU is used. Then, thread assignment starts over with the first CPU.

`RR Tasks` In order to make different tasks run on different CPUs automatically, the `RR Tasks` policy can be used. This policy places all the task's threads on one logical CPU. If a new task is created, it is assigned the current start CPU, causing new threads for that task to be scheduled there. The start CPU is increased in round-robin order as for the previous policy but on a per-task basis instead of per thread.

`Balanced` In many cases, when overall workload performance is more important than that of a single specific task, load should be distributed among the system's NUMA nodes. The `Balanced` policy uses knowledge of the previously started (and not yet terminated) tasks to decide where to place a new task and its threads. For example, when a second task is created after a first one, it is homed on the node where the first task is not running causing all its threads to be scheduled there as well.

The `Fixed` policy imitates the default behavior on our L4 system: all threads are scheduled on the same start CPU. In the case of my NUMA manager, the start CPU is easily configurable, for instance, from within the Lua start-scripts, allowing for a customizable placement of threads on logical CPUs.
The `RR Threads` and `RR Tasks` policies are more useful for a very basic load distribution. Depending on the amount of threads and tasks, either policy might prove advantageous: when there are less threads than there are CPUs in the system, `RR Threads` should be used. When the number of threads exceeds the number of CPUs, `RR Tasks` can provide better performance because threads sharing an address space are scheduled in conjunction. This aspect can benefit the locality of reference they are likely to exhibit. The former two policies potentially suffer from the problem that subsequent tasks are homed on the same NUMA node. This behavior can cause an imbalance between different nodes when not all available logical CPUs in the system are used. The `Balanced` policy tries to address this problem by allocating logical CPUs to tasks in an alternating fashion between the NUMA nodes.

Access to these scheduling policies is granted from within the Lua scripts offered by Ned. The user can provide a flag specifying which policy to use when starting the NUMA manager application.

Dataspace-Allocation Policies

Contrary to scheduling, memory-allocation decisions cannot be influenced at all from outside a task in the default system. To allow for user friendly access, I implemented the following four static policies guiding dataspace allocations:

`Default` The `Default` policy allocates dataspaces on the node specified in the `alloc` invocation on the `Mem_alloc` object.[1]

[1] A NUMA node is only accepted as a parameter to the `alloc` method on the `Mem_alloc` object after my changes for creating a node-aware allocation API, as described in section 3.4.1 on page 58.

`Local`	In order to allocate memory locally to the requesting thread, the `Local` policy can be used. This policy is comparable to the first-touch strategy provided on other operating systems, such as Linux. The main difference is the time when the decision on which node to allocate a dataspace is made: when the allocation is performed and not when the first access happens, as is the case for first touch.
`Remote`	In the case that remote memory allocation is desired, the `Remote` policy provides the necessary functionality. It is mainly devised for evaluation purposes, e.g., for investigation of remote access penalties.
`Interleaved`	I have implemented a dataspace type that interleaves its memory pages across the system's NUMA nodes. Using the `Interleaved` policy, all created dataspaces are of this type.

All presented thread-scheduling and dataspace-allocation policies can be used for a rough load and memory distribution throughout the system when starting a set of tasks. More dynamic balancing and bottleneck management can be built on top of or next to them.

4.3.2 Dynamic Rebalancing

The policies presented in the previous section allow for static placement of threads and dataspaces: the objects are placed once and their affinity is not changed afterwards. However, as stated as requirement R2 (see page 34), the NUMA manager should also perform dynamic rebalancing if the need arises. Rebalancing can happen in the form of thread or memory migrations. A scenario where such migrations are required is an imbalance due to shared resource contention (requirement R3), for instance. To satisfy these requirements, I have to implement a dynamic approach.

The Auto Policy

To satisfy the need for dynamic rebalancing, I implemented the `Auto` policy. Using this policy, the NUMA manager checks for imbalances periodically and reacts to them.
The `Auto` policy performs initial thread placement in the same fashion as `Balanced`: initially, threads are approximately balanced among the systems NUMA nodes. Furthermore, I take hardware threads into account: if enough cores exist, new threads get assigned to a full-fledged core. Only if all cores execute at least one thread, additional hardware threads on them are used for execution of more application threads.
The memory-allocation policy to use stays configurable, i.e., one of `Default`, `Local`, `Remote`, or `Interleaved` can be employed.
The frequency at which to perform balance checking is configurable as well and can be adjusted from within the Lua start-scripts. A commonly employed frequency is 1 Hz, meaning that checks are performed every second. An interval of 1s is also what Dashti et al. [DFF+13] and Majo and Gross [MG11a] use in their systems to check for potential problems.

Cache-Balancing

The NUMA manager's overall approach is to focus on locality: data should be located close to the thread working on it. When data is allocated locally, remote access penalties and interconnect contention are minimized. However, memory controllers and shared caches can become contended.
Depending on the workload—especially on the amount of tasks and threads it comprises—and on the system at hand, contention is inevitable. One approach to minimize its impact is to balance contention among the system's resources.
To reach this balance, I use a scheme referred to as *cache-balancing* [MG11a]. In short, cache-balancing tries to balance pressure on all the system's caches approximately equally.

(a) Unbalanced Caches **(b)** Balanced Caches

Figure 4.4: Illustration of Cache-Balancing

Figure 4.4 illustrates this principle. As an effect of the migration of one or more threads from the first processor (that uses "Cache 1") to the second processor (containing "Cache 2"), pressure on both caches gets balanced.

There are various definitions of the term "cache pressure". Knauerhase et al. use LLC misses per cycle [KBH+08]. Other research employs miss-rate curves [TASS09] or stack distance profiles [SEH10]. Blagodurov et al. [BZDF11] and Majo and Gross [MG11a] suggest using the LLC Misses Per 1,000 Instructions (MPKI). The MPKI is also what I use to define the *Cache Pressure* metric:

$$\text{Cache Pressure} = \frac{\#\text{ LLC misses} \times 1{,}000}{\#\text{ Instructions retired}} \tag{4.1}$$

As explained when discussing caches (see 2.1.3 on page 7), they evict older cache lines in favor of newer ones. This process can cause cache misses to occur when the previously evicted cache line is accessed in a later step. The more cache misses a cache exhibits, the more it is contended for by different threads.
Of interest for this work is only the LLC because it is commonly shared among a processor's cores. The L1 and L2 caches are typically private to each core, making balancing less useful.
In order to determine the number of LLC misses as well as the number of retired instructions, I use a processor's performance counters. Usage of these counters is a

common approach for detecting and tracing performance problems and employed by several researchers [IM03, Era08, TH08, MV10, MG11a].

Remote Load Ratio

Cache-balancing helps to distribute contention for last-level caches by migration of threads. Since each processor typically contains its own LLC that is shared among all its cores, this scheme causes migrations to happen across NUMA node boundaries. Such node-crossing migrations, in turn, can cause access penalties because previously local memory may now be remote to a migrated thread.
To alleviate these penalties, memory can be migrated to the new node as well. Since memory migrations are more costly than thread migrations, such a migration might not always be beneficial (see section 4.2.1 for a detailed analysis of either migration type).
I use the *Remote Load Ratio* metric in order to decide whether or not to migrate memory. The Remote Load Ratio is defined as the ratio of loads from remote memory to all loads:

$$\text{Remote Load Ratio} = \frac{\#\text{ remote Loads retired}}{\#\text{ Loads retired}} \tag{4.2}$$

Both required values in this fraction are hardware events and can be measured as before using performance counters. I use a machine-dependent Remote Load Ratio value to decide whether or not to migrate a dataspace to a different NUMA node if it experiences remote access penalties.

The Algorithm

In accordance with the one presented by Majo and Gross [MG11a], I developed an algorithm for mapping threads to logical CPUs. For this algorithm, every task is *homed* on one NUMA node.
In a first step, the algorithm creates a mapping from threads to logical CPUs that favors locality—threads are mapped to logical CPUs that reside on the same NUMA node the owning task is homed on.
The second step refines this mapping, possibly migrating a task's threads to a different node if this migration favors cache balance. Algorithm 4.1 on the next page depicts this process. Only when the difference in cache pressure between the system's processors is greater than a certain threshold, a task to migrate is searched (line 8). In that case, the algorithm checks whether or not the migration of a managed task's threads to another NUMA node benefits the overall cache balance (line 16). The best candidate—if a suitable one was found—is then migrated (line 22). Note that the algorithm shown is simplified over the real version: handling of corner cases and optimizations were removed to make it more comprehensible.
The third and final step decides whether or not to migrate a task's dataspaces as well, based on the Remote Load Ratio. This step does not exist in Majo and Gross' algorithm for they do not handle migration of memory at all. Algorithm 4.2 on the facing page illustrates my approach.

Input: List *threads* of all Threads
Output: Refined Mapping from Threads to CPUs

```
1  pressure_0 ← cache_pressure_on_node(0)
2  pressure_1 ← cache_pressure_on_node(1)
3  difference ← pressure_0 − pressure_1
4  if difference > 0 then
5      node ← 0
6  else
7      node ← 1
8  if |difference| ≥ THRESHOLD then
9      min_pressure ← min(pressure_0, pressure_1)
10     max_pressure ← max(pressure_0, pressure_1)
11     task_list ← tasks_on_node(node)
12     candidate ← NULL
13     foreach task in task_list do
14         min_pressure' ← min_pressure + task.cache_pressure
15         max_pressure' ← max_pressure − task.cache_pressure
16         if |min_pressure' − max_pressure'| < difference then
17             candidate ← task
18             difference ← |min_pressure' − max_pressure'|
19         end
20     end
21     if candidate ≠ NULL then
22         migrate_task_threads(candidate)
23 end
```

Algorithm 4.1: Refining the Thread Mapping

Input: List *dataspaces* of all Dataspaces
Output: Updated Mapping from Dataspaces to NUMA Nodes

```
1  foreach dataspace in dataspaces do
2      home_node ← dataspace.task.node
3      if dataspace.node ≠ home_node then
4          if dataspace.task.remote_load_ratio ≥ THRESHOLD then
5              migrate_dataspace(home_node)
6      end
7  end
```

Algorithm 4.2: Refining the Dataspace Mapping

There are limitations to this algorithm. Like Majo and Gross', it reasons about migration of entire tasks, not a single thread. However, the mapping from threads to CPUs is decided on a per-thread basis so as to better utilize the processor's resources.
Moreover, the algorithm only works for a system with two NUMA nodes. Adjusting it to work with more nodes is possible but not a goal of this thesis.
Contrary to Majo and Gross' version, I removed the limitation that only a maximum number of processes, equaling the number of cores in the system, can be managed.

4.3.3 SPEC Workload

Having discussed the basic design principles of the `Auto` policy, its contribution to the performance of managed applications is evaluated in the next step.
For this evaluation, I use the *SPEC INT 2006* benchmark suite [Hen06, Cor11]. The benchmarks provide a number of realistic workloads used in practice, such as mail filtering, compression, compilation of source code, vehicle scheduling, game playing, and video compression.
Out of the suite's twelve benchmarks, I use ten for my evaluation. I am unable to run `458.sjeng` due to a memory access error in the 64 bit version of the program on L4.[2] `483.xalancbmk` cannot be compiled for our microkernel environment because it requires deprecated STL features which are not provided by our C++ standard library.

I created a workload comprising the ten SPEC benchmarks. For the measurements, I executed this workload in various configurations: with local, remote, and interleaved memory, as well as with the different thread-placement policies presented in section 4.3.1. Furthermore, to get an impression of the overall overhead introduced by the NUMA manager, I implemented the static thread-scheduling policies presented in section 4.3.1 on page 74 directly in the Lua start-up script. This way, it is possible to achieve the same thread placement without any involvement and, thus, overhead of the NUMA manager.

Measurements

Figure 4.5 on page 82 illustrates the results of my measurements. Shown is the CPU time each of the ten SPEC programs requires to finish its work as well as the total wall-clock time required to process the entire workload (▬). Wall-clock times make up only a fraction of the accumulated CPU times due to parallel execution of the benchmarks on the available cores. Plotted is the mean value of five runs. The standard deviation is less than 5.83 % of the mean.

From bottom to top the figure depicts the `Auto` policy, with and without memory migrations enabled, the static `Balanced` and `RR Tasks` policies implemented in Lua (i.e., without NUMA manager overhead) and in the NUMA manager, respectively, as well as the static `RR Threads` policy. It is impossible to emulate the `RR Threads` policy in a Lua start-up script because from within such a script, only a *per-task* CPU can be

[2] The error is unrelated to the NUMA manager as it occurs when this component is not involved as well.

set. All the application's threads are scheduled on the same CPU. Each of the policies was executed in different memory allocation setups.
Not shown is the `Auto` policy in combination with `Interleaved` allocations because the dataspace type used in this case, `DataspaceInterleaved`, has no migration support by design.

Evaluation

The data suggests that the static `Balanced` policy performs best with a total wall-clock execution time of 718 s for the SPEC workload. With approximately 4,300 s, the accumulated CPU time for the different benchmarks is also the lowest one compared to the other configurations.
In my opinion, this result can be attributed to the aspect that not all SPEC INT 2006 benchmarks are memory-intensive. The thread and memory migrations I perform with the more sophisticated `Auto` policy, however, are all made with a connection to memory in mind: I rebalance threads as to better distribute pressure on the system's LLCs and I migrate memory to alleviate remote access penalties. If a benchmark is non-memory-intensive, all memory-related migrations require a longer time to pay off. When migrations happened too often because new imbalances emerged in the meantime, the migration overhead outweighs the benefit.

Furthermore, compared to the two static Lua policies, "R1-D-S" (wall-clock run time of 823 s) and "B-D-S" (851 s), `Balanced` performs 13 % and 16 % better, respectively. Being the best performing static Lua policy for this workload, "R1-D-S" would also be the best option to choose on an L4 system without any NUMA-specific optimizations.

The plot also illustrates that the `Auto` policy yields the second best performance out of the available configurations. However, its best result is achieved with memory migrations disabled (with 4,784 s accumulated CPU time for "A-L-M" vs. 5,491 s for "A-L-*"). The wall-clock execution time behaves analogously.
My explanation for this difference in run time is the greater influence a memory migration has on application performance than the remote access penalties induced in case no migration happens. One aspect leading to this high impact could be the coarse granularity at which memory migrations work in my scheme: I reason about migration of entire dataspaces between NUMA nodes. Although I perform migration at the granularity of a page lazily, i.e., only upon access, the generated page fault as well as the actual migration in form of a copy operation require a lot of CPU time. Compared to the case where memory stays remotely, several subsequent accesses have to occur in order to amortize the migration to local memory.

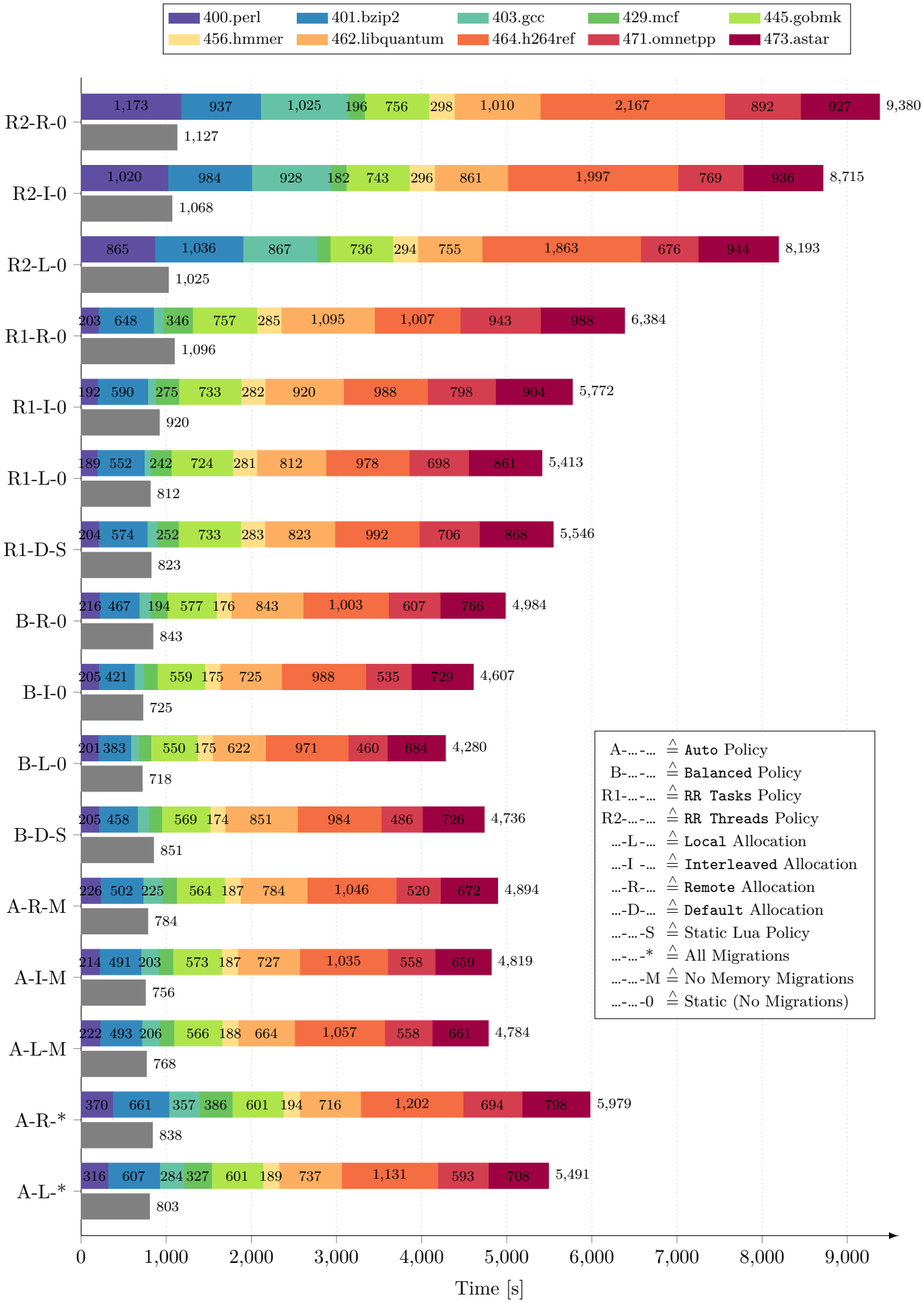


Figure 4.5: SPEC Workload Execution-Times in Different Configurations

When scheduling is load based, or better: cache-pressure based, as is the case for the `Auto` policy, the differences between `Local`, `Interleaved`, and `Remote` are less significant compared to the static policies. The overall run-time difference for the `Auto` policy (without memory migrations) between the `Local` and `Remote` case is 110 s. In the case of static policies—`RR Tasks` or `RR Threads`—the difference is in the order of 1,000 s.
I attribute this behavior to cache-balancing which helps to mask remote access latencies. With a better balance of cache-pressure, more cache hits occur. More cache hits, in turn, lead to less accesses to main memory.

Another interesting aspect is the performance of the static Lua policies: "B-D-S" and "R1-D-S". These policies perform worse than their counterparts implemented within the NUMA manager ("B-L-0" and "R1-L-0", respectively).
I believe that this difference is caused by the memory allocation scheme: as explained before, it is not possible to influence memory allocations from within Lua. This aspect leads to memory being allocated remotely in my two socket system for half of the started tasks (depending on the policy used, it can be less or more than half). With memory being remote, execution slows down. The NUMA manager ensures a local allocation of memory—thus, the overall performance is better than for the Lua policies.

The diagram also shows an order of the application run-times for the static policies with regard to the memory-allocation scheme used: the version using locally allocated memory performs best, second best is the interleaved version, and the slowest one is always the configuration using remote memory. These results are in line with the previously observed NUMA factors in section 3.2.2 on page 38.
For the static policies, the scheduling does not depend on dynamic parameters such as the load. For that reason, the thread-to-CPU mapping is always the same and, thus, causes similar contention patterns to emerge. In this case, access latencies to memory are the only characteristic that differs among the runs—and remotely allocated memory induces the highest overhead out of the three available schemes.

4.3.4 Mixed Workload

The experiment just described used the SPEC INT 2006 benchmarks to create a workload. However, the benchmarks exhibit different pressure on the CPU and the memory subsystem. Specifically, not all benchmarks are memory-bound or memory-intensive. As my NUMA manager's functionality is related to memory—be it contention on shared caches caused by frequent memory accesses or remote access penalties due to suboptimal memory placement—I decided to create a more memory-intensive workload.

In a first step, I had a second look at the SPEC INT 2006 benchmarks and executed them in a serial fashion, i.e., without any contention, one after the other, on an otherwise not loaded machine. The only aspect I changed between the runs is the type of memory: for one run, it gets allocated locally, for the other, it is remote. My goal is to pick memory-intensive benchmarks and combine them to a more memory intense workload.

Figure 4.6 on the following page illustrates the run times of the ten SPEC INT 2006 benchmarks. Shown is the mean of five runs. The standard deviation is less than 0.32 %

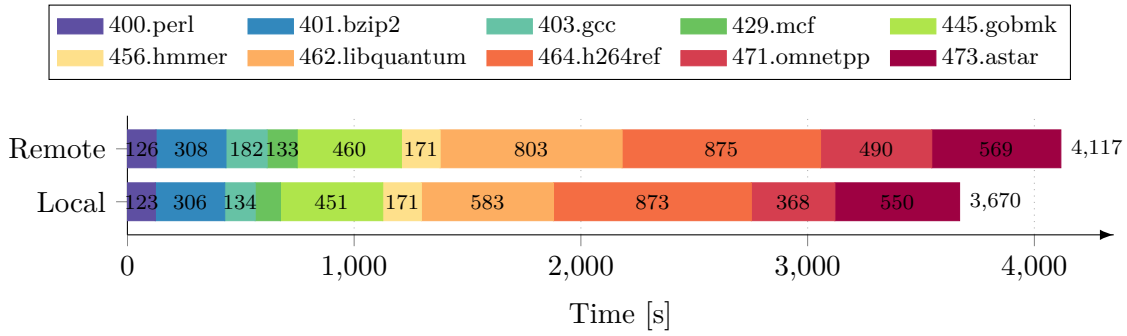


Figure 4.6: SPEC Benchmark Execution-Times with Local and Remote Memory

of the mean. Note that the wall-clock time is not plotted as a separate bar for it is equivalent to the sum of the CPU times of the solo benchmarks.
The figure shows that three programs experience a higher slow-down compared to the rest: `403.gcc` requires 36 %, `462.libquantum` 38 %, and `471.omnetpp` 33 % more time when memory is allocated remotely. The other benchmarks' run times degrade only slightly.
Based on these results, I decided to create a new, more memory-intensive workload for which I chose to use `462.libquantum` and `471.omnetpp`. I left out `403.gcc` due to its comparably short run time and because it comprises eight program invocations in itself which complicates the creation of a workload in the Lua scripts that I use.
In addition to the two mentioned SPEC benchmark programs, I use the very memory-intensive STREAM Triad benchmark.

Measurements

Figure 4.7 on the next page illustrates the execution times for the mixed workload. Shown is the mean of five runs. The standard deviation is less than 6.19 %.
Note that the workload comprises three instances of `462.libquantum` as well as `471.omnetpp`. In addition, six STREAM Triad instances are used. As before, the CPU time of each benchmark is depicted as well as the required wall-clock time (▬). Shown is only the performance of the best performing policy combinations from my previous experiment in order to keep the diagram concise.

Evaluation

This time, "A-L-*", i.e., the NUMA manager's `Auto` policy with memory migrations enabled, yields the best wall-clock run time with 938 s. Without memory migrations ("A-L-M"), the run time increased significantly to 1,262 s. The previously best policy, `Balanced` ("B-L-0"), causes the workload to finish in 1,051 s. The benchmarks' accumulated CPU times form the same order with respect to the policies.
The results indicate that memory migrations can indeed be useful to alleviate remote access penalties for memory-intensive workloads: the ability to perform memory migrations constitutes the only difference between "A-L-*" and "A-L-M". The result of which

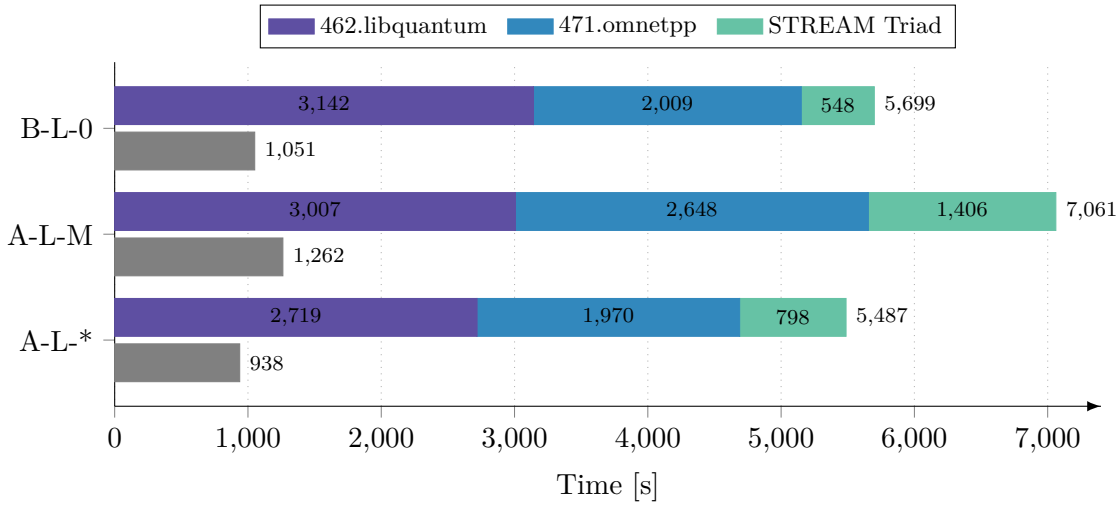


Figure 4.7: Mixed Workload Execution-Times

is a decrease of overall workload run-time from 1,262 s down to 938 s (–26 %). Compared to `Balanced`—the best static policy—`Auto` offers a 11 % performance gain.

These values indicate that it is hard, if not impossible, to create a static policy that meets the requirements of a variety of workloads. Instead, mechanisms for automatic problem detection at run time are useful. With advanced placement and migration algorithms incorporating knowledge of such bottlenecks, better performance can be achieved than with a static placement alone.

4.4 Code Size

The amount of code an application comprises can be correlated to the number of potential bugs and, thus, security vulnerabilities it contains [OW02, MCKS04, AMR05]. Particularly interesting from a security point of view is the code a program needs to trust in order to correctly fulfill its purpose: the *Trusted Computing Base (TCB)* [HPHS04, SPHH06]. My NUMA manager is involved in the management of memory, making it part of a managed application's TCB.

This section examines the amount of code the NUMA manager and its components are made up of. It is not meant to be a comprehensive security analysis but rather to provide the reader with an impression of the complexity of the involved components to reason about the applicability of the approach used.

Overall, the L4 package making up the manager application includes four main parts:

`NUMA manager` The NUMA manager—the required component for making client applications NUMA-aware—is explained in detail in section 3.3.1 on page 45.

`NUMA controller` The NUMA controller component can be used to interact with the NUMA manager using command-line options. This component offers convenient access of the NUMA manager's functionality from within Lua scripts.

`libnuma` The `libnuma` helper library provides abstractions for basic NUMA-related functionality. The library offers functions for retrieving the NUMA node for a given logical CPU, for querying whether a node is local or remote to a given CPU, and for allocation of memory, for instance. Furthermore, it contains code wrapping the communication with the NUMA manager (to query or change on which logical CPU a thread runs, for example).
This library is used by both the NUMA manager and NUMA tasks (although the latter do not require it).

`libpmu` Another library used by the NUMA manager is `libpmu` which provides support for reading and writing the system's performance counters in a flexible way. The performance counters are part of the *Performance Monitoring Unit (PMU)*, hence the name. The library relies on the system calls introduced for instructing the kernel to perform the necessary privileged operations of reading the counter registers (section 3.4.3 on page 62 provides more details on the topic of performance counters).

Table 4.6 on the facing page shows the number of physical Source Lines of Code (SLOC) each component is made up of (excluding comments). I used the *CLOC* tool for automatic gathering of these numbers [Dan13]. Note that shown is only the amount of code added to the system with these four new components. The NUMA manager application in particular relies on several libraries provided by the L4Re which are also part of its TCB but not included in this evaluation.

Component	Language	Lines of Code
NUMA manager	C++	3,725
libnuma	C++	200
libpmu	C++	192
NUMA controller	C	59

Table 4.6: Lines of Code of NUMA Manager Components

Unsurprisingly, the NUMA manager application itself makes up the major part with approximately 3,700 SLOC. The other components require more than an order of magnitude less code.
Out of the NUMA manager's 3,700 SLOC, almost 1,000 SLOC are required in order to handle dataspaces (not shown in the table). The region map, which is needed due to limitations of Moe, makes up another 400 lines (see section 3.3.3 on page 51). The dynamic rebalancing by means of the `Auto` policy comprises roughly 300 SLOC. The remaining 2,000 SLOC are used for the scheduling component, the static policies, the infrastructure for virtualization of all objects, the statistics gathering unit, and more.

From a code size point of view, the implementation is reasonable. Virtualizing almost all L4Re objects requires additional boilerplate code. However, implementing certain parts from scratch could be avoided which is also the case for the region map in particular. By providing more versatile base classes or scaffoldings, code reuse could be fostered and the overall amount of source code reduced.

4.5 Comparison to Other Work

To the best of my knowledge, the approach of providing NUMA awareness to the L4 Runtime Environment is novel. However, other research has investigated topics related to my work, such as load balancing, memory or thread migrations, and NUMA awareness in microkernels. This section compares my work to others.

4.5.1 Memory Migrations

Tikir and Hollingsworth investigate how memory migrations can be beneficial for software performance [TH08]. Like I, they use an approach that operates mainly at user level and does not require modification of applications. Their strategy is to dynamically inject code that creates additional profiling and migration threads into processes. The profiling thread periodically reads out specialized hardware performance-monitors that can be plugged into their Sun Microsystems Fire 6800 server. The monitors listen for address transactions on the system's interconnect. Based on this information, the migration thread may choose to migrate remotely allocated pages of memory in order to reduce access latencies.
With this infrastructure in place, the authors conducted experiments with the OpenMP C benchmark and the NAS Parallel Benchmark suite. These experiments show that their approach reduces the number of remote memory accesses by up to 87 %. On their system, this yields a performance increase of 18 %.
Furthermore, the researchers investigated which additional sources provide good information for guiding memory migrations. They have a look at performance counters that monitor cache and TLB misses and also develop a hypothetical hardware component that enhances TLB entries with an additional counter that is incremented on each page access for this entry—the *Address Translation Counter (ATC)*.
Using a simulator as well as real hardware, the authors compare these approaches. The evaluation shows that the ATC approach performs slightly better than all others. The dedicated monitoring hardware performs comparable to using the cache-miss performance counter. Only TLB miss-rates prove not very useful as a metric to base migration decisions on.
The paper concludes that memory migrations are a good possibility for achieving high performance on NUMA systems. Hardware support is essential for acquiring knowledge about when to migrate a page. However, a dedicated hardware component is not required with event counters available in today's processors.
Contrary to my work, the authors concentrate solely on memory migrations and are not concerned with thread migrations. They investigate the quality of different sources used for guiding migration decisions and even devise a dedicated hardware component for that matter. I do not compare different sources for basing my memory migration decisions on and rely on performance counters which they found to be suitable for that purpose.
Tikir and Hollingsworth's approach of memory migrations works at page granularity. Although I migrate memory at page granularity as well, I reason about migration of

entire dataspaces—typically a multitude of pages—and only decide whether a migration is desired or not on this level.

4.5.2 Microkernels

Not much work has been published in the context of microkernels and NUMA systems. The only publication I am aware of is by Kupferschmied et al. [KSB09]. They present an idea on how to fit NUMA-aware memory management into a microkernel environment. As typical of microkernels, they strive for a solution where changes to the kernel itself are kept to a minimum and most of the work is done at user level. Their focus lies on locality but ignores contention on shared resources. Although operating in user space, the approach covers kernel memory as well.
According to their idea, an address space is always bound to (and thus is local to) a specific NUMA node. All data, including kernel-level objects, are allocated on that node. By explicitly mapping memory from a different node, access to remote memory is possible on demand. They have an interesting approach of handling thread migrations: when a thread migrates to a different NUMA node, it actually migrates to a different address space that is local to that node. This means that even when migrated, a thread's user and kernel data-structures reside on local memory.
On the one hand, the approach of Kupferschmied et al. is more comprehensive than mine as it works for kernel-internal data structures as well. However, the kernel is not aware of the overall concept: it always allocates data locally—the different address spaces provided by user space libraries have no specific semantics to it. On the other hand, my work goes one step further towards other issues of NUMA systems as I also consider contention and dynamic rebalancing. Being L4 based, it would be interesting to see what performance a combined version of their idea and my work yields.

4.5.3 Load Balancing

Pilla et al. devised a NUMA-aware load balancer [PRC+11]. Their system—*NumaLB*—is based on *Charm++*, a "C++-based parallel programming model and runtime system designed to enhance programmer productivity by providing a high-level abstraction of the parallel computation while delivering good performance". The authors enhanced Charm++ to incorporate knowledge about the system's topology into migration decisions.
The load balancing assigns tasks (*chares* in their terminology) to cores and performs migrations if necessary. They use a heuristic that creates a cost factor based on the core's current load, communication characteristics between affected chares, and the NUMA factor. Based on this cost factor, a greedy iterative balancing-algorithm assigns tasks to CPU cores, beginning with the heaviest chare and the least loaded core.
In the evaluation, three benchmarks exhibiting different communication patterns are employed and the results are compared to those of four other load balancers (GreedyLB, MetisLB, RecBipartLB, and ScotchLB). Pilla et al. claim that NumaLB offers an average speed-up of 1.51 on the programs' iteration times compared to runs without load balancing. Furthermore, NumaLB causes a seven times smaller migration overhead than the other load balancers due to an overall reduced number of migrations.

Comparable to my work, the authors took an existing run-time system and adapted it. However, Charm++ already provides the means of chare migrations and statistics gathering out of the box. For my approach, I had to implement most of the mechanisms myself.
In addition, the researchers claim that knowledge of communication patterns helps to find better placement decisions. I did not investigate the influence of IPC on program performance and its relation to thread placement because on the L4Re system, gathering precise communication statistics is not a trivial task: communication happens over IPC gates that have to be associated with a thread. This association, however, is not exposed to other tasks.

4.5.4 NUMA-Aware Contention Management

Blagodurov et al. investigate the influence of contention for shared LLCs, memory controllers, and cross-processor interconnects on memory-intensive applications in the context of NUMA architectures on the Linux operating system [BZDF11]. Remote access latency is regarded as a fourth factor possibly slowing down execution.
The group found that performance degradation due to contention on shared resources can be severe for certain workloads, depending on the overall load on the system. They also note that previous not NUMA-aware contention-management algorithms do not work properly in NUMA contexts and may even cause a performance degradation due to shared resource contention caused when threads are migrated to a different node but memory is not.
Based on this knowledge and their previously developed UMA-based DI algorithm, they devise *DINO*, a NUMA-aware contention-management algorithm performing thread and memory migrations. DINO evaluates performance counters for deciding if intervention is required: like I, the authors use the number of LLC misses per 1,000 instructions as a guideline. The higher the miss-rate the more memory pressure a program exhibits and the more contended shared resources may become.
Focus is also laid on memory migrations: they explain why most of the time it is neither feasible nor favorable to migrate the application's entire resident set. Instead, only a certain amount of pages before and/or after the accessed page are migrated. To provide this functionality, DINO relies on libnuma (see section 2.3.3 on page 16) and *perfmon*, a tool used for programming and accessing a machine's performance counters.
Evaluating the work, Blagodurov et al. show that DINO outperforms all compared systems in most workloads. In particular, they report a performance improvement of 20 % over default Linux and of up to 50 % over DI—which, according to the authors, performs "within 3 % percent of optimal on non-NUMA systems".
The authors' approach employs thread and memory migrations to mitigate NUMA-related problems, like I do. Moreover, their implementation also resides in user space although not implemented on a microkernel.
We both agree that memory migrations are costly and that their employment should be minimized. Blagodurov et al. look more deeply into this matter: they found that migrating the entire *resident-set* is not the best option. Instead, they use a heuristic to determine migration of which pages is expected to yield the best performance. I use a

lazy migration approach that only migrates pages once they are accessed which makes such a heuristic less useful.
In terms of achievable performance, the improvements are comparable to mine: the average run-time improvement for benchmarks of the SPEC CPU 2006 suite ranges from a slight degradation to as much as 35 %, depending on the benchmark.

Majo and Gross take a more in-depth look at what causes problems on NUMA systems and identify LLC and memory controller contention as the two main reasons [MG11a]. Based on initial measurements, they argue that neither minimizing cache contention nor providing maximum locality yields the best performance—a trade-off has to be found. They devise the *NUMA-Multicore-Aware Scheduling Scheme (N-MASS)*, an extension to the Linux scheduler. N-MASS uses locality and contention metrics to decide where to place a task's threads.
Majo and Gross' evaluation uses several benchmarks from the SPEC CPU 2006 suite with differing CPU and memory-utilization characteristics in varying multiprogrammed workloads. Furthermore, they allocate memory locally and remotely in different combinations. They compare their results to solo runs, i.e., ones without contention and with locally allocated memory. Overall, N-MASS is able to increase performance by up to 32 % with an average of 7 %.
The N-MASS system resembles my work the closest. Apart from being Linux based, the authors also focus on locality and try to balance cache pressure. Basically, I use the same algorithm to achieve a balance in terms of pressure on caches. One difference is that the decision whether or not to migrate a thread in their system depends on the *NUMA penalty*—a value retrieved from a profile run of the application of interest that mirrors performance when working on local and remote memory. I do not require such a profile run because my approach does not consider a NUMA penalty in their sense but only cache pressure. The reason for this behavior is that my NUMA manager offers memory migrations in addition to thread migrations, whereas the N-MASS system concentrates solely on thread migrations. After a thread got migrated, I detect whether too many accesses happen remotely, in which case I migrate the memory as well.
Like I, they evaluate their work on a subset of memory-intensive SPEC benchmarks from which they created multiprogrammed workloads. With up to 32 %, their N-MASS component achieves performance benefits over default Linux that are comparable to those of my NUMA manager.

The evaluation of the NUMA manager has shown that it can provide performance benefits for workloads on NUMA machines. For not memory-intensive workloads, dynamic rebalancing is not beneficial and a static placement policy provides good performance. In such a case, using a static NUMA manager policy can offer a performance benefit of 13 % over the best static Lua policy. With a memory-intensive workload, dynamic problem detection and mitigation pay off and can offer 11 % speed-up compared to the fastest static NUMA manager policy.
The next chapter concludes my thesis by providing an outlook for future work and by giving a summary of the goals I achieved.

5 Conclusion & Outlook

5.1 Future Work

During my work, many ideas for future work and improvements for my implementation came to mind which could not be incorporated into this thesis due to timing constraints or because they were out of scope. These include:

- *Optimizing thread placement and memory allocation for low energy consumption.*

  Energy consumption is a major factor in embedded devices but also plays an important role for servers and desktop PCs. Previous work has shown that remote memory accesses can require more energy than local ones [Dal11]. Knowledge about energy-consumption characteristics can be incorporated into scheduling decisions made in the context of NUMA systems. For that matter, energy-consumption counters could be accessed in addition to the performance counters employed in my approach.
  My implementation is not concerned with power consumption in its current stage but it can be extended in this direction. Energy aspects were intentionally not part of my thesis assignment.

- *Query the NUMA topology at run time.*

  My prototype contains hardcoded NUMA factors which are only correct for the test system I used (see section 3.2.1 on page 37). This is not desirable for common use because L4 runs on a variety of systems which might exhibit a different NUMA topology.
  In general, information about the system's topology can be retrieved at boot time or later. ACPI tables, specifically the SLIT, contain such data [CCC+06].

- *Allowing for adaptation to different NUMA factors.*

  This work investigates the influence of NUMA properties for a system with only two nodes. Consequently, there is only one NUMA factor (see section 2.1.2 on page 5 for details on the term NUMA factor). However, NUMA architectures can have arbitrarily many and diverse NUMA factors, depending on the number of available interconnects and the interconnect network topology.
  With a broader set of test machines, future work could investigate the influence of different NUMA factors. Algorithms need to be adapted and heuristics fine tuned or completely made aware of this additional level of variety in access times.

- *Using of* PEBS *or* IBS.

  Intel and AMD provide two sampling methods that can be used to gather more exact information on *where* an event happened (and not just *that* it happened): Precise Event Based Sampling (PEBS) and Instruction Based Sampling (IBS). With these mechanisms, it is possible to correlate which instruction caused an memory event such as an last-level cache miss or a remote memory access with the memory location that was accessed. This knowledge can help to minimize memory migration costs, for example.
  The DINO system designed by Blagodurov et al. uses IBS for extracting information about which page a remote access referenced [BZDF11]. The authors argue that IBS is superior to "simpler" performance counters, however, they also state that sampling accuracy may be an issue.
  Unfortunately, as of now, support for these mechanisms has not been built into L4.

- *Utilizing NUMA awareness for increasing fault tolerance.*

  Investigations have shown that errors in DRAM modules often span multiple bits close to each other or even entire rows [SL12]. Using replication-based fault tolerance, as done in the *Romain* framework [BD13], these findings in conjunction with a NUMA API that allows for allocations to be served by a certain node, can decrease the impact of such faults.
  Typical of replication systems is a majority vote among the replicas in case an error is detected in one of them—the goal being to find a majority of replicas with the same result which are then assumed to be error-free. Due to heuristics employed in memory allocators trying to improve locality, it is likely that physical memory used for the replicated processes is located close to each other. If a row or multi-bit error occurs in such a region, it might influence two replicas at a time—a potentially fatal event for the outcome of the majority vote.
  Future work might investigate whether storing and possibly running replicas on different nodes in a NUMA system can minimize such errors on replicated processes. It might also be interesting to devise heuristics coping with numbers of replicas that exceed the number of NUMA nodes in the system.

- *Port this work to other architectures than x86_64.*

  As explained before, this thesis is only concerned with the x86_64 architecture. Our microkernel and the L4Re, however, can run on a variety of other architectures, including ARM, PowerPC, and Sparc.
  Future work can port my NUMA manager to these architectures. I estimate the effort to be not significant as the code is written at a high level of abstraction without architecture specific details playing an important part. However, due to different cache organizations, page sizes, or other CPU characteristics in general, performance hot-spots shall be investigated.

- *Consider usage of multi-threaded programs with data sharing.*

  Due to a lack of multi-threaded benchmarks, I evaluate my work using single-threaded applications although they might be executed concurrently, i.e., in a multiprogrammed context. However, more and more applications are becoming multi-threaded and data sharing within the applications increases [MG12].
  This inter-processor sharing of data causes different performance characteristics of applications. For instance, in the case that data is shared between cores on the same processor, resources like caches and the memory controller are shared as well. This can have positive and negative effects: if contention becomes too high, performance might degrade (see section 3.2.4 on page 41). If data is shared at the level of the cache, performance can increase because the slower main memory is not involved and less cache misses may occur.
  Future work could focus on multi-threaded workloads and evaluate their effects as well as the performance benefits.

- *Improve memory migration performance.*

  I found that memory migrations cause significantly more run-time overhead compared to thread migrations. At the same time, their employment is often mandatory because of the remote access penalties induced otherwise.
  Future work could further investigate the performance bottlenecks. Possible findings include that an alternative memory-migration implementation performs better, e.g., an eager instead of a lazy version, or that page-table management within the kernel is responsible for the comparably bad performance. Last but not least, Moe's dataspace implementation could be enhanced directly with migration support as to remove the object-virtualization overhead that slows down page-fault processing.

5.2 Conclusion

In this work, I made the L4 Runtime Environment NUMA-aware. I found that both remote access penalties as well as shared resource contention influence application performance in NUMA systems negatively. To that end, I devised the NUMA manager—a component for alleviating these problems by means of thread and memory migrations. The NUMA manager facilitates NUMA-aware placement of threads and memory without modifying the managed application. Furthermore, it seamlessly integrates into the L4Re system and offers an easy way to start managed applications.

The management component is an active service: it periodically checks for emerging problems by means of performance metrics. Using hardware performance counters, these metrics are gathered on a per-thread basis. The NUMA manager bases its placement decisions on two principles: locality and balance of cache pressure.
By allocating memory on the same node the accessing thread executes on, I minimize remote access penalties. Locality also helps to reduce cross-processor interconnect contention. When accesses happen locally, no traffic has to pass through this interconnect. Furthermore, contention for last-level caches is mitigated by migrating threads to a different processor if the latter shows less pressure on its LLC.
With thread migrations in place, a new problem can emerge: previously local memory is now remote to a thread. Therefore, the NUMA manager provides a mechanism for migrating parts of a managed application's memory between NUMA nodes.

For achieving its goal without modification of a managed application's source code, the NUMA manager is built around the concept of object virtualization in L4Re: threads and dataspaces are virtualized and proxy objects handed out to clients. Using these proxy objects, affinity to the system's resources can be changed from within the NUMA manager.

To implement the NUMA manager component, I furthermore:

- Designed and implemented an API for NUMA-aware allocations at user space, extending existing means of memory allocation,
- Provided Fiasco.OC with topologically fixed CPU IDs as to allow for precise thread placement with regard to the system's topology,
- Added performance-counter multiplexing support to the kernel—a requirement for reasoning about performance penalties at the level of threads.

My evaluation has shown that for not memory-bound workloads, the NUMA manager can offer a performance gain of 13 % over the best performing static Lua policy. Furthermore, by means of dynamic rebalancing and memory migrations, memory-intensive workloads can benefit from NUMA-aware contention management. In such a case, the NUMA manager decreased wall-clock run time for a managed application by 11 % compared to its fastest static placement policy.

Glossary

ABI Application Binary Interface.

ACPI Advanced Configuration and Power Interface.

AFT Adaptive First-Touch.

AMD Advanced Micro Devices.

AMP Asymmetric Multiprocessing.

API Application Programming Interface.

APIC Advanced Programmable Interrupt Controller.

ATC Address Translation Counter.

BIOS Basic Input/Output System.

ccNUMA Cache-Coherent NUMA.

CPI Cycles Per Instruction.

CPU Central Processing Unit.

DPM Dynamic Page Migration.

DRAM Dynamic Random-Access Memory.

DSM Distributed Shared Memory.

EFI Extensible Firmware Interface.

FSB Front-Side Bus.

GQ Global Queue.

HPC High Performance Computing.

HT HyperTransport.

IBS Instruction Based Sampling.

IMC Integrated Memory Controller.

IPC Inter-Process Communication.

IPI Inter-Processor Interrupt.

IT Information Technology.

KIP Kernel Interface Page.

L4Re L4 Runtime Environment.

LLC Last-Level Cache.

MMU Memory Management Unit.

MPKI LLC Misses Per 1,000 Instructions.

N-MASS NUMA-Multicore-Aware Scheduling Scheme.

NUCA Non-Uniform Cache Access.

NUMA Non-Uniform Memory Access.

PEBS Precise Event Based Sampling.

PMU Performance Monitoring Unit.

PTE Page Table Entry.

QoS Quality of Service.

QPI QuickPath Interconnect.

RAID Redundant Array of Independent Disks.

SLIT System Locality Information Table.

SLOC Source Lines of Code.

SMP Symmetric Multiprocessing.

SMT Simultaneous Multithreading.

TCB Trusted Computing Base.

TLB Translation Lookaside Buffer.

UEFI Unified Extensible Firmware Interface.

UMA Uniform Memory Access.

Bibliography

[AAHV91] Sarita V. Adve, Vikram S. Adve, Mark D. Hill, and Mary K. Vernon. *Comparison of hardware and software cache coherence schemes.* SIGARCH Comput. Archit. News, 19(3):298–308, 4 1991.

[ADC11] Thomas J. Ashby, Pedro Diaz, and Marcelo Cintra. *Software-Based Cache Coherence with Hardware-Assisted Selective Self-Invalidations Using Bloom Filters.* IEEE Trans. Comput., 60(4):472–483, 4 2011.

[AJR06] Joseph Antony, Pete P. Janes, and Alistair P. Rendell. *Exploring Thread and Memory Placement on NUMA Architectures: Solaris and Linux, UltraSPARC/FirePlane and Opteron/Hypertransport.* In Proceedings of the 13th International Conference on High Performance Computing, HiPC'06, pages 338–352, Berlin, Heidelberg, 2006. Springer-Verlag.

[AMD06] Inc. Advanced Micro Devices. *Performance Guidelines for AMD Athlon™ and Opteron™ ccNUMA Multiprocessor Systems.* `http://support.amd.com/us/Processor_TechDocs/40555.pdf`, 6 2006. [Online; accessed Friday 19th July, 2013].

[AMD13] Inc. Advanced Micro Devices. *AMD HyperTransport™ Technology.* `http://www.amd.com/uk/products/technologies/hypertransport-technology/Pages/hypertransport-technology.aspx`, 2013. [Online; accessed Thursday 4th July, 2013].

[AMR05] Omar Alhazmi, Yashwant Malaiya, and Indrajit Ray. *Security Vulnerabilities in Software Systems: A Quantitative Perspective.* In Proceedings of the 19th Annual IFIP WG 11.3 Working Conference on Data and Applications Security, DBSec'05, pages 281–294, Berlin, Heidelberg, 2005. Springer-Verlag.

[ANS+10] Manu Awasthi, David W. Nellans, Kshitij Sudan, Rajeev Balasubramonian, and Al Davis. *Handling the Problems and Opportunities Posed by Multiple On-chip Memory Controllers.* In Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques, PACT '10, pages 319–330, New York, NY, USA, 2010. ACM.

[BD13] Hermann Härtig Björn Döbel. *Where Have all the Cycles Gone? – Investigating Runtime Overheads of OS-Assisted Replication.* In Workshop on Software-Based Methods for Robust Embedded Systems, SOBRES'13, 2013.

[Ber11] Lars Bergstrom. *Measuring NUMA effects with the STREAM benchmark*. CoRR, 2011.

[BGM+08] Jairo Balart, Marc Gonzalez, Xavier Martorell, Eduard Ayguade, Zehra Sura, Tong Chen, Tao Zhang, Kevin O'Brien, and Kathryn O'Brien. *Languages and Compilers for Parallel Computing*. chapter A Novel Asynchronous Software Cache Implementation for the Cell-BE Processor, pages 125–140. Springer-Verlag, Berlin, Heidelberg, Germany, 2008.

[Bre93] Timothy Brecht. *On the importance of parallel application placement in NUMA multiprocessors*. In USENIX Systems on USENIX Experiences with Distributed and Multiprocessor Systems - Volume 4, Sedms'93, pages 1–1, Berkeley, CA, USA, 1993. USENIX Association.

[BS93] William J. Bolosky and Michael L. Scott. *False sharing and its effect on shared memory performance*. In USENIX Systems on USENIX Experiences with Distributed and Multiprocessor Systems - Volume 4, Sedms'93, pages 3–3, Berkeley, CA, USA, 1993. USENIX Association.

[BZDF11] Sergey Blagodurov, Sergey Zhuravlev, Mohammad Dashti, and Alexandra Fedorova. *A Case for NUMA-aware Contention Management on Multicore Systems*. In Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference, USENIXATC'11, Berkeley, CA, USA, 2011. USENIX Association.

[CCC+06] Hewlett-Packard Corporation, Intel Corporation, Microsoft Corporation, Phoenix Technologies Ltd., and Toshiba Corporation. *Advanced Configuration and Power Interface Specification, Revision 3.0b*. `http://www.acpi.info/DOWNLOADS/ACPIspec30b.pdf`, 10 2006.

[Cha01] Alan Charlesworth. *The sun fireplane system interconnect*. In Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM), Supercomputing '01, New York, NY, USA, 2001. ACM.

[Con04] HyperTransport™ Consortium. *HyperTransport™ I/O Technology Overview – An Optimized, Low-latency Board-level Architecture*. 6 2004. [Online; accessed Wednesday 31st July, 2013].

[Cooa] Intel Coorporation. *Intel® Quickpath Interconnect Maximizes Multi-Core Performance*. `http://www.intel.com/technology/quickpath/`. [Online; accessed Thursday 4th July, 2013].

[Coob] Intel Coorporation. *Intel® Turbo Boost Technology—On-demand Processors Performance*. `http://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html`. [Online; accessed Saturday 10th August, 2013].

[Coo09] Intel Coorporation. *An Introduction to the Intel® QuickPath Interconnect*. http://www.intel.com/content/www/us/en/io/quickpath-technology/quick-path-interconnect-introduction-paper.html, 1 2009. [Online; accessed Sunday 23rd June, 2013].

[Coo12a] Intel Coorporation. *Intel® 64 Architecture Processor Topology Enumeration*. http://software.intel.com/en-us/articles/intel-64-architecture-processor-topology-enumeration, 1 2012. [Online; accessed Saturday 27th July, 2013].

[Coo12b] Intel Coorporation. *Intel® Xeon® Processor X5650*. http://ark.intel.com/products/47922/Intel-Xeon-Processor-X5650-12M-Cache-2_66-GHz-6_40-GTs-Intel-QPI?wapkw=x5650, 4 2012. [Online; accessed Monday 29th July, 2013].

[Coo13a] Intel Coorporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html, 7 2013. [Online; accessed Thursday 18th July, 2013].

[Coo13b] Intel Coorporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture*. http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-1-manual.html, 6 2013. [Online; accessed Thursday 1st August, 2013].

[Coo13c] Intel Coorporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2A: Instruction Set Reference, A-M*. http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-2a-manual.html, 6 2013. [Online; accessed Saturday 27th July, 2013].

[Cor05] Intel Corporation. *Excerpts from A Conversation with Gordon Moore: Moore's Law*. ftp://download.intel.com/sites/channel/museum/Moores_Law/Video-Transcripts/Excepts_A_Conversation_with_Gordon_Moore.pdf, 2005. [Online; accessed Wednesday 2nd October, 2013].

[Cor11] Standard Performance Evaluation Corporation. *SPEC CPU2006*. http://www.spec.org/cpu2006/index.html, 9 2011. [Online; accessed Friday 14th June, 2013].

[CY04] Mung Chiang and Michael Yang. *Towards Network X-ities From a Topological Point of View: Evolvability and Scalability*. 10 2004.

[Dal11] Bill Dally. *Power, Programmability, and Granularity: The Challenges of ExaScale Computing.* In Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium, IPDPS '11, pages 878–, Washington, DC, USA, 2011. IEEE Computer Society.

[Dan13] Al Danial. *CLOC – Count Lines of Code.* `http://cloc.sourceforge.net/`, 2013. [Online; accessed Thursday 5th December, 2013].

[DFF+13] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. *Traffic Management: a Holistic Approach to Memory Placement on NUMA Systems.* In Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, pages 381–394, New York, NY, USA, 2013. ACM.

[Dre07a] Ulrich Drepper. *Memory part 4: NUMA support.* `http://lwn.net/Articles/254445/`, 10 2007. [Online; accessed Friday 19th July, 2013].

[Dre07b] Ulrich Drepper. *What Every Programmer Should Know About Memory.* 11 2007.

[EOO+05] Alexandre E. Eichenberger, Kathryn O'Brien, Kevin O'Brien, Peng Wu, Tong Chen, Peter H. Oden, Daniel A. Prener, Janice C. Shepherd, Byoungro So, Zehra Sura, Amy Wang, Tao Zhang, Peng Zhao, and Michael Gschwind. *Optimizing Compiler for the CELL Processor.* In Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques, PACT '05, pages 161–172, Washington, DC, USA, 2005. IEEE Computer Society.

[Era08] Stéphane Eranian. *What can performance counters do for memory subsystem analysis?* In Proceedings of the 2008 ACM SIGPLAN workshop on Memory systems performance and correctness: held in conjunction with the Thirteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '08), MSPC '08, pages 26–30, New York, NY, USA, 2008. ACM.

[Fou13] Linux Foundation. *What is Linux Memory Policy?* `https://www.kernel.org/doc/Documentation/vm/numa_memory_policy.txt`, 2013. [Online; accessed Monday 8th July, 2013].

[Gav92] Bezalel Gavish. *Topological design of computer communication networks – The overall design problem.* European Journal of Operational Research, 58(2):149–172, 1992.

[GF09] Brice Goglin and Nathalie Furmento. *Memory Migration on Next-Touch.* In Linux Symposium, Montreal, Canada, 2009.

[GHF89] Guy-L. Grenier, Richard C. Holt, and Mark Funkenhauser. *Policy vs. Mechanism in the Secure TUNIS Operating System.* In IEEE Symposium on Security and Privacy, pages 84–93. IEEE Computer Society, 1989.

[Gra12] Bill Gray. *numastat(8): numastat – Show per-NUMA-node memory statistics for processes and the operating system.* `http://www.kernel.org/doc/man-pages/online/pages/man8/numastat.8.html`, 2012. numastat(8) – Linux manual page.

[Gro12] Operating Systems Group. *L4 Runtime Environment.* `http://os.inf.tu-dresden.de/L4Re`, 2012. [Online; accessed Friday 21st June, 2013].

[Gro13a] Operating Systems Group. *Data-Space API.* `http://os.inf.tu-dresden.de/L4Re/doc/group__api__l4re__dataspace.html`, 2013. [Online; accessed Wednesday 18th September, 2013].

[Gro13b] Operating Systems Group. *L4Re Servers.* `http://os.inf.tu-dresden.de/L4Re/doc/l4re_servers.html`, 2013. [Online; accessed Monday 22nd July, 2013].

[Gro13c] Operating Systems Group. *L4Re::Mem_alloc Class Reference.* `http://os.inf.tu-dresden.de/L4Re/doc/classL4Re_1_1Mem__alloc.html`, 2013. [Online; accessed Wednesday 18th September, 2013].

[Gro13d] Operating Systems Group. *L4::Scheduler Class Reference.* `http://os.inf.tu-dresden.de/L4Re/doc/classL4_1_1Scheduler.html`, 2013. [Online; accessed Friday 20th September, 2013].

[Gro13e] Operating Systems Group. *Memory allocator API.* `http://os.inf.tu-dresden.de/L4Re/doc/group__api__l4re__mem__alloc.html`, 2013. [Online; accessed Friday 12th July, 2013].

[Gro13f] Operating Systems Group. *Moe, the Root-Task.* `http://os.inf.tu-dresden.de/L4Re/doc/l4re_servers_moe.html`, 2013. [Online; accessed Monday 22nd July, 2013].

[Gro13g] Operating Systems Group. *Ned, the Init Process.* `http://os.inf.tu-dresden.de/L4Re/doc/l4re_servers_ned.html`, 2013. [Online; accessed Monday 22nd July, 2013].

[Gro13h] Operating Systems Group. *Region map API.* `http://os.inf.tu-dresden.de/L4Re/doc/group__api__l4re__rm.html`, 2013. [Online; accessed Friday 12th July, 2013].

[GVM+08] Marc Gonzàlez, Nikola Vujic, Xavier Martorell, Eduard Ayguadé, Alexandre E. Eichenberger, Tong Chen, Zehra Sura, Tao Zhang, Kevin O'Brien, and Kathryn O'Brien. *Hybrid access-specific software cache techniques for the cell BE architecture.* In Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques, PACT '08, pages 292–302, New York, NY, USA, 2008. ACM.

[Hen06] John L. Henning. *SPEC CPU2006 Benchmark Descriptions*, 9 2006.

[HFFA09] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. *Reactive NUCA: Near-Optimal Block Placement and Replication in Distributed Caches*. SIGARCH Comput. Archit. News, 6 2009.

[HHL+97] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. *The Performance of µ-Kernel-Based Systems*. In Proceedings of the 16th ACM Symposium on Operating System Principles (SOSP), pages 5–8, St. Malo, France, 10 1997.

[HPHS04] Michael Hohmuth, Michael Peter, Hermann Härtig, and Jonathan S. Shapiro. *Reducing TCB size by using untrusted components: small kernels versus virtual-machine monitors*. In Proceedings of the 11th workshop on ACM SIGOPS European workshop, EW 11, New York, NY, USA, 2004. ACM.

[ICdF13] Roberto Ierusalimschy, Waldemar Celes, and Luiz Henrique de Figueiredo. *Lua*. `http://www.lua.org/about.html`, 3 2013. [Online; accessed Saturday 15th June, 2013].

[IM03] Canturk Isci and Margaret Martonosi. *Identifying program power phase behavior using power vectors*. In Workshop on Workload Characterization, 2003.

[Kam09] Patryk Kaminski. *NUMA aware heap memory manager*. 3 2009.

[KBH+08] Rob Knauerhase, Paul Brett, Barbara Hohlt, Tong Li, and Scott Hahn. *Using OS Observations to Improve Performance in Multicore Systems*. IEEE Micro, 28(3):54–66, 5 2008.

[KHMHb10] Yoongu Kim, Dongsu Han, Onur Mutlu, and Mor Harchol-balter. *ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers*. In Matthew T. Jacob, Chita R. Das, and Pradip Bose, editors, HPCA, pages 1–12. IEEE Computer Society, 2010.

[Kle04] Andreas Kleen. *An NUMA API for Linux*. 8 2004.

[Kle05] Andi Kleen. *A NUMA API for Linux*. Technical report, 4 2005.

[Kle07] Andi Kleen. *NUMA(3): numa – NUMA policy library*. `http://www.kernel.org/doc/man-pages/online/pages/man3/numa.3.html`, 12 2007. NUMA(3) – Linux manual page.

[KMAC03] Chetana N. Keltcher, Kevin J. McGrath, Ardsher Ahmed, and Pat Conway. *The AMD Opteron Processor for Multiprocessor Servers*. IEEE Micro, 23(2):66–76, 3 2003.

[KSB09] Philipp Kupferschmied, Jan Stoess, and Frank Bellosa. *NUMA-Aware User-Level Memory Management for Microkernel-Based Operating Systems*. 3 2009.

[KSL13] Tim Kiefer, Benjamin Schlegel, and Wolfgang Lehner. *Experimental Evaluation of NUMA-Effects on Database Management Systems*. Proceedings of the GI-Fachtagung Datenbanksysteme für Business, Technologie und Web, BTW 2013, Magdeburg, Germany, 3 2013.

[Kuz04] Ihor Kuz. *L4 User Manual — API Version X.2*. Emb., Real-Time & Operat. Syst. Program, NICTA, 9 2004. Available from `http://www.disy.cse.unsw.edu.au/Softw./L4`.

[Lam06a] Christoph Lameter. *Local and Remote Memory: Memory in a Linux/NUMA System*. 2006.

[Lam06b] Christoph Lameter. *Page migration*. `https://www.kernel.org/doc/Documentation/vm/page_migration`, 5 2006. [Online; accessed Monday 8th July, 2013].

[LBKH07] Tong Li, Dan Baumberger, David A. Koufaty, and Scott Hahn. *Efficient Operating System Scheduling for Performance-Ssymmetric Multi-Core Architectures*. In Proceedings of the 2007 ACM/IEEE Conference on Supercomputing, SC '07, pages 53:1–53:11, New York, NY, USA, 2007. ACM.

[LCC+75] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. *Policy/mechanism separation in Hydra*. SIGOPS Oper. Syst. Rev., 9(5), 11 1975.

[Lie95] Jochen Liedtke. *On μ-Kernel Construction*. In Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP-15), SOSP '95, pages 237–250, New York, NY, USA, 12 1995. ACM.

[Lie96a] Jochen Liedtke. *L4 Reference Manual - 486, Pentium, Pentium Pro*. Arbeitspapiere der GMD, Yorktown Heights, NY, USA, 9 1996. IBM T.J. Watson Research Center.

[Lie96b] Jochen Liedtke. *Microkernels Must And Can Be Small*. In Proceedings of the 5th IEEE International Workshop on Object-Orientation in Operating Systems (IWOOOS), Seattle, WA, USA, 10 1996.

[Lie96c] Jochen Liedtke. *Toward real microkernels*. Commun. ACM, 39(9):70–77, 9 1996.

[Lim07] ARM Limited. *ARM1156T2-S™ Revision: r0p4, Technical Reference Manual*, 7 2007. [Online; accessed Saturday 24th August, 2013].

[LL97] James Laudon and Daniel Lenoski. *The SGI Origin: a ccNUMA Highly Scalable Server*. SIGARCH Comput. Archit. News, 5 1997.

[LPH+05] Jeremy Lau, Erez Perelman, Greg Hamerly, Timothy Sherwood, and Brad Calder. *Motivation for variable length intervals and hierarchical phase behavior.* In IEEE International Symposium on Performance Analysis of Systems and Software, pages 135–146, 2005.

[LS12] Min Lee and Karsten Schwan. *Region Scheduling: Efficiently Using the Cache Architectures via Page-Level Affinity.* SIGARCH Comput. Archit. News, 40, 3 2012.

[LW09] Adam Lackorzyński and Alexander Warg. *Taming subsystems: capabilities as universal resource access control in L4.* In Proceedings of the Second Workshop on Isolation and Integration in Embedded Systems, IIES '09, pages 25–30, New York, NY, USA, 2009. ACM.

[MI2] Daniel Müller. *Evaluation of the Go Programming Language and Runtime for L4Re.* Belegarbeit, Technische Universität Dresden, 7 2012.

[MA12] Nakul Manchanda and Karan Anand. *Non-Uniform Memory Access (NUMA).* 1, 3 2012.

[McC] John D. McCalpin. *STREAM: Sustainable Memory Bandwidth in High Performance Computers.* `http://www.cs.virginia.edu/stream/`. [Online; accessed Monday 29th July, 2013].

[McK10] Paul E. McKenney. *Memory Barriers: a Hardware View for Software Hackers.* 7 2010.

[MCKS04] Parastoo Mohagheghi, Reidar Conradi, Ole M. Killi, and Henrik Schwarz. *An Empirical Study of Software Reuse vs. Defect-Density and Stability.* In Proceedings of the 26th International Conference on Software Engineering, ICSE '04, pages 282–292, Washington, DC, USA, 2004. IEEE Computer Society.

[MG11a] Zoltan Majo and Thomas R. Gross. *Memory Management in NUMA Multicore Systems: Trapped between Cache Contention and Interconnect Overhead.* SIGPLAN Not., 46(11), 6 2011.

[MG11b] Zoltan Majo and Thomas R. Gross. *Memory System Performance in a NUMA Multicore Multiprocessor.* In Proceedings of the 4th Annual International Conference on Systems and Storage, SYSTOR '11, pages 12:1–12:10, New York, NY, USA, 2011. ACM.

[MG12] Zoltan Majo and Thomas R. Gross. *A Template Library to Integrate Thread Scheduling and Locality Management for NUMA Multiprocessors.* In Proceedings of the 4th USENIX conference on Hot Topics in Parallelism, HotPar'12, Berkeley, CA, USA, 2012. USENIX Association.

[Moo00] Gordon E. Moore. *Readings in computer architecture.* chapter Cramming more components onto integrated circuits, pages 56–59. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[MV10] Collin Mccurdy and Jeffrey Vetter. *Memphis: Finding and Fixing NUMA-related Performance Problems on Multi-Core Platforms*. In Proceedings of ISPASS, pages 87–96. IEEE Computer Society, 2010.

[MYS03] Mark S. Miller, Ka-Ping Yee, and Jonathan Shapiro. *Capability Myths Demolished*. Technical report, Systems Research Laboratory, Johns Hopkins University, 2003.

[OA89] S. Owicki and A. Agarwal. *Evaluating the performance of software cache coherence*. SIGARCH Comput. Archit. News, 17(2):230–242, 4 1989.

[OHSJ10] Ricardo Orozco, Hai-Yue Han, Richard Schafer, and Taylor Johnson. *Memory Hierarchies for Multi-Core Processors*. ECE 570 High Performance Computer Architecture, Oregon State University, Corvallis, Oregon, 3 2010.

[OW02] Thomas J. Ostrand and Elaine J. Weyuker. *The Distribution of Faults in a Large Industrial Software System*. In Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA '02, pages 55–64, New York, NY, USA, 2002. ACM.

[PRC+11] Laércio L. Pilla, Christiane Pousa Ribeiro, Daniel Cordeiro, Abhinav Bhatele, Philippe O. A. Navaux, Jean-François Méhaut, and Laxmikant V. Kale. *Improving Parallel System Performance with a NUMA-aware Load Balancer*. 7 2011.

[RW70] Lawrence G. Roberts and Barry D. Wessler. *Computer network development to achieve resource sharing*. In Proceedings of the May 5-7, 1970, spring joint computer conference, AFIPS '70 (Spring), pages 543–549, New York, NY, USA, 1970. ACM.

[Sal94] Peter H. Salus. *A quarter century of UNIX*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1994.

[Sar99] Kanoj Sarcar. *What is NUMA?* `https://www.kernel.org/doc/Documentation/vm/numa`, 11 1999. [Online; accessed Monday 8th July, 2013].

[SEH10] Andreas Sandberg, David Eklöv, and Erik Hagersten. *Reducing Cache Pollution Through Detection and Elimination of Non-Temporal Memory Accesses*. In Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[SGG09] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. Wiley & Sons, Upper Saddle River, NJ, USA, 8th edition, 2009.

[SL12] Vilas Sridharan and Dean Liberty. *A study of DRAM failures in the field.* In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '12, pages 76:1–76:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[SLN+11] ChunYi Su, Dong Li, Dimitrios Nikolopoulos, Matthew Grove, Kirk W. Cameron, and Bronis R. de Supinski. *Critical Path-Based Thread Placement for NUMA Systems.* PMBS '11, New York, NY, USA, 2011. ACM.

[SPH+03] Timothy Sherwood, Erez Perelman, Greg Hamerly, Suleyman Sair, and Brad Calder. *Discovering and Exploiting Program Phases.* IEEE Micro, 23(6):84–93, 11 2003.

[SPHH06] Lenin Singaravelu, Calton Pu, Hermann Härtig, and Christian Helmuth. *Reducing TCB complexity for security-sensitive applications: three case studies.* In Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006, EuroSys '06, pages 161–174, New York, NY, USA, 2006. ACM.

[SS10] Livio Soares and Michael Stumm. *FlexSC: flexible system call scheduling with exception-less system calls.* In Proceedings of the 9th USENIX conference on Operating systems design and implementation, OSDI'10, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association.

[SU07] Sebastian Schönberg and Volkmar Uhlig. *µ-kernel Memory Management.* 2007.

[Sut09] Herb Sutter. *The Free Lunch is Over.* `http://www.gotw.ca/publications/concurrency-ddj.htm`, 8 2009. [Online; accessed Tuesday 30th July, 2013].

[SZ08] John E. Savage and Mohammad Zubair. *A unified model for multicore architectures.* In Proceedings of the 1st international forum on Next-generation multicore/manycore technologies, IFMT '08, pages 9:1–9:12, New York, NY, USA, 2008. ACM.

[Tan09] Andrew S. Tanenbaum. *Modern Operating Systems.* Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2009.

[TASS09] David K. Tam, Reza Azimi, Livio B. Soares, and Michael Stumm. *RapidMRC: Approximating L2 Miss Rate Curves on Commodity Systems for Online Optimizations.* In Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XIV, pages 121–132, New York, NY, USA, 2009. ACM.

[TH08] Mustafa M. Tikir and Jeffrey K. Hollingsworth. *Hardware monitors for dynamic page migration.* J. Parallel Distrib. Comput., 68(9):1186–1200, 9 2008.

[THWW10] Josh Triplett, Philip W. Howard, Eric Wheeler, and Jonathan Walpole. *Avoiding system call overhead via dedicated user and kernel CPUs.* OSDI'10, 2010.

[TMV+11] Lingjia Tang, Jason Mars, Neil Vachharajani, Robert Hundt, and Mary Lou Soffa. *The Impact of Memory Subsystem Resource Sharing on Datacenter Applications.* SIGARCH Comput. Archit. News, 6 2011.

[VE11] Frederik Vandeputte and Lieven Eeckhout. *Transactions on High-Performance Embedded Architectures and Compilers IV.* chapter Characterizing time-varying program behavior using phase complexity surfaces, pages 21–41. Springer-Verlag, Berlin, Heidelberg, Germany, 2011.

[WM95] Wm. A. Wulf and Sally A. McKee. *Hitting the memory wall: implications of the obvious.* SIGARCH Comput. Archit. News, 23(1):20–24, 3 1995.

[YS80] Jeffry W. Yeh and William Siegmund. *Local network architectures.* In Proceedings of the 3rd ACM SIGSMALL symposium and the first SIGPC symposium on Small systems, SIGSMALL '80, New York, NY, USA, 1980. ACM.

[ZBF10] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. *Addressing Shared Resource Contention in Multicore Processors via Scheduling.* SIGARCH Comput. Archit. News, 3 2010.