Universität Karlsruhe (TH)
Research University • founded 1825

System Architecture Group
Department of Computer Science

Design and Implementation of a Microkernel-Based Operating System for NUMA Machines

Philipp Kupferschmied

Diplomarbeit

Verantwortlicher Betreuer: Prof. Dr. Frank Bellosa
Betreuender Mitarbeiter: Dipl.-Inf. Jan Stöß

20.03.2008

Hiermit erkläre ich, die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Literaturhilfsmittel verwendet zu haben.

I hereby declare that this thesis is a work of my own, and that only cited sources have been used.

Karlsruhe, den 20.03.2008

_____

Philipp Kupferschmied

Abstract

Systeme mit uneinheitlichen Speicherzugriffszeiten sind die konsequente Weiterentwicklung klassischer Mehrprozessorarchitekturen. Anstelle alle Prozessoren mittels eines gemeinsamen Speicherbusses an einen gemeinsamen Hauptspeicher anzubinden, ist jede CPU über einen eigenen Speichercontroller und -bus an eigenen, lokalen Speicher angebunden. Auf diesen lokalen Speicher kann die CPU schnell zugreifen, wohingegen Zugriffe auf entfernten Speicher, also Speicher anderer CPUs, länger dauern. Dieser Ansatz hilft, die begrenzte Skalierbarkeit klassischer SMP-Architekturen zu überwinden, bei denen Speichercontroller und -bus mit steigender CPU-Zahl zum Flaschenhals werden. Die unterschiedlichen Speicherzugriffszeiten stellen eine zusätzliche Herausforderung sowohl für Betriebssystementwickler als auch für Anwendungsentwickler dar. Um bestmögliche Leistung zu erreichen, sollten Daten immer möglichst nah an der auf sie zugreifen CPU gehalten werden. Migration und Replikation sind wichtige Mechanismen zur Verbesserung der Lokalität, aber es hängt von vielen Faktoren ab, welcher der beiden Mechanismen vorzuziehen ist oder ob sie überhaupt anwendbar sind. Diese Faktoren sind unter anderem die Zugriffshäufigkeit, das Verhältnis zwischen Lese- und Schreibzugriffen und der Grad, zu dem das betroffene Objekt gemeinsam (also von mehreren CPUs gleichzeitig) genutzt wird. In dieser Arbeit stellen wir ein auf NUMA-Hardware optimiertes Betriebssystem vor, das auf dem L4 Mikrokern basiert. Es werden sowohl notwendige Modifikationen am Kern selbst als auch der Aufbau der im Userlevel realisierten Systemdienste vorgestellt. Insbesondere stellen wir ein Konzept vor, das die Replikation von Seitentabellen erlaubt und die notwendige Synchronisation vollständig im Userlevel durchführt.

Abstract

Systems with non-uniform memory access characteristics are the consequent evolution of "classical" SMP systems. Instead of all CPUs being connected to a single shared memory via a single memory bus, each CPU is assigned its own "local" memory, which it can access directly and thus fast. Access to memory of others CPUs is possible, but is slower than local access. This model overcomes the scalability problems of classical SMP architectures, where the memory controller and bus become a bottleneck with an increasing number of CPUs in the system. However, the non-uniform access latencies pose new challenges to both operating system and application developers. Data should be kept local to the accessing CPU wherever possible. Migration and replication can be used to improve data locality, but the choice for the right mechanism depends on many factors like access frequency, read-to-write ratio, and the degree of sharing of an object. In this thesis, we propose a design for a NUMA-aware operating system based on the L4 microkernel. We discuss necessary modifications to the kernel and present a design for user-level operating system services and applications. We present a concept that allows for replication of page tables with synchronization performed entirely at user-level.

Contents

CHAPTER 1

Introduction

Systems with non-uniform memory access (NUMA) characteristics are the consequent evolution of classical symmetric multiprocessor (SMP) systems. On SMP systems, all processors are connected to main memory with a single memory bus and a single memory controller. Thus, the entire memory can be accessed with the same latency by all processors. However, with an increasing number of CPUs in the system, the memory bus becomes a scalability bottleneck. In contrast, on NUMA systems, each CPU (or group of CPUs) has its own, local memory, to which it is connected via its own memory controller and memory bus. The combination of CPU(s), memory controller/bus, and local memory is called a *node*. Node-local memory can be accessed fast, while accesses to memory of other nodes can be significantly slower. However, NUMA systems still provide a contiguous physical address space. An operating system or application developed without regard for the NUMAness of the hardware will run without modifications, but might perform poorly. NUMA-systems have become popular for UNIX-based servers in the mid 1990s. Nowadays, NUMA systems are likely to become popular as well for smaller servers and workstations. For example, the AMD Opteron, a widely used x86-compatible CPU in small servers and even desktop systems, is NUMA-capable: Each CPU has its own built-in memory management logic, and can directly access a subset of the machine's main memory. It is thus not surprising that both recent versions of Microsoft Windows and Linux provide mechanisms for exploiting NUMA capabilities of the hardware [3].

Windows and Linux are, like most other operating systems used nowadays, based on monolithic kernels. All system services and most of the device drivers are linked to a single, huge kernel binary. This approach provides no isolation between components, an error in one component can thus affect the entire kernel. With increasing size and complexity of operating systems, the probability for errors increases, while finding and fixing these errors becomes more difficult. Microkernels promise a way of overcoming these problems. Instead of having a single kernel image, there is only a small, privileged code base, the microkernel. System services and device drivers run as user-level applications within their own protection do-

mains. A microkernel must therefore provide a high degree of flexibility to allow for the construction of arbitrary systems on top.

1.1 Problem Definition

To exploit the capabilities of NUMA systems, operating system and application programmers must address efficient code and data placement. Even on systems with a low remote-to-local access ratio, the performance benefits of a better memory placement can be quite high [35]. It is therefore important that the operating system itself places its own data in an "efficient" way, and that application programmers have a possibility to influence memory placement of their applications. Common solutions to provide better locality are migration and replication. Migration means that data is moved to the node from where it is accessed most frequently. Replication means that multiple copies of an object are created and placed on different nodes. The choice when and where to migrate data is not an easy one, as one has to decide when the costs for remote memory accesses outweigh the costs for migration (i.e. copying the data and updating references). In case of replication, synchronization overhead can eliminate performance advantages of local accesses, making replication only suitable for data that is more frequently read than written. Our work focuses on microkernel-based operating systems. In addition to the afore mentioned requirements, there are some microkernel-specific additional problems. A microkernel should be kept "small", with "small" referring not primarily to its code size, but meaning that the microkernel should only consist of what is absolutely necessary for correct system operation. In particular, the kernel must be kept policy-free to allow for the construction of arbitrary operating systems on top. Therefore, policies regarding migration and replication of memory or processes are to be implemented at user-level. The microkernel should not make any assumptions on memory or process placement, nor should it influence or limit the user-level policies. Additionally, in-kernel synchronization primitives should be avoided when possible. The choice for the best synchronization primitive depends on access and sharing patterns of the data that is "protected" by the synchronization primitive, but in case of NUMA hardware also on the ratio of remote to local latency. A static synchronization primitive within the microkernel thus would limit the kernel's flexibility and contradict the principle of keeping the kernel free of policies. Finally, also applications running on top of the operating system must be able to exploit the hardware characteristics of a NUMA system. The operating system must offer an API that allows applications to influence both memory and thread placement, or it must automatically choose a "good" memory placement for applications. Hence, we have to tackle the following problems:

1. All data structures used by the microkernel must be revised in order to find out if they can be replicated or migrated.

2. The kernel must be kept minimal. We therefore need to find out which NUMA-specific changes to the kernel are absolutely necessary and which can be handled at user-level.

3. Applications must be given the possibility to exploit the NUMA-capabilities of the hardware.

In the following chapters, we present our design and implementation of a NUMA-aware, microkernel-based operating system. Our work focuses on the L4 microkernel, which was not designed with a special regard to NUMA architectures.

1.2 Organization of the Document

This thesis is organized as follows: In Chapter 2, we describe both microkernel-based systems and multiprocessor architectures, with the focus on NUMA architectures. We present related work regarding different aspects of NUMA-aware operating systems. In Chapter 3, we start with an overview of our proposed system architecture for a microkernel-based NUMA-aware operating system. We then go into detail and have a closer look both on the requirements the microkernel must satisfy and on the operating system services we construct on top. Chapter 4 details our implementation. We describe the changes that were necessary to the L4 microkernel as well as the implementation of user-level operating system services. In Chapter 5, we analyze the performance of our implementation. We compare the results to the unmodified kernel. Chapter 6 concludes with a recapitulation of our work and with proposals for future work.

CHAPTER 2

Background and Related Work

2.1 Microkernel-Based Operating Systems

Most operating systems used nowadays are monolithic. That means that all OS services such as pager, scheduler, and device drivers run in a single protection domain. On the one hand, this greatly simplifies interaction between components: All data is directly accessible by all services, and a service can be requested with a simple (and cheap) function call. On the other hand, with increasing OS complexity, those systems get harder to debug. An error in one component can affect other components or even the entire system. Microkernel-based systems follow another approach: Instead of having all services running with the same privileges and within the same protection domain, the basic idea is to keep the privileged part of the operating system as small as possible. This leads to a system with only a small microkernel running with high privileges, while OS services are implemented as user-level applications (called *servers*), strictly isolated from each other. Ideally, the microkernel itself should be as flexible as possible, allowing for arbitrary operating systems to be constructed on top. This requires the microkernel to be policy-free.

In contrast to the monolithic approach, interaction between components becomes much more complicated. Data of one component is no longer accessible by all others, unless it is explicitly shared. A service call is no longer a simple function call, but requires the crossing of protection domain boundaries. Both for sharing of data between address spaces and for invoking services across address space boundaries, the microkernel must provide a suitable mechanism. Figure 2.1 shows an example of a microkernel-based operating system. On top of the microkernel run several operating system services like a pager, a memory server, and several device drivers, each within its own protection domain. User applications request services provided by these servers. For example, when a page fault occurs in an application's address space, the kernel notifies the pager. The pager might then have to request a new page of physical memory from the memory server, which it can map into the application's address space (by using an appropriate mechanism

provided by the kernel) so that the application can resume execution.

In contrast to a monolithic operating system, management information is distributed amongst the microkernel and the user-level services. For example, a pager handling a page fault requires knowledge about the layout of the faulting task's address space, i.e. it must know which data to map at the address where the fault occurred and where that data can be found. The microkernel does not have or require this knowledge. In contrast, only the microkernel can directly modify the hardware page tables. A user-level pager (or any other user-level application) must not be permitted direct access to hardware page tables for security reasons. Consequently, the microkernel must offer an API that allows a pager to install new mappings in a secure manner.

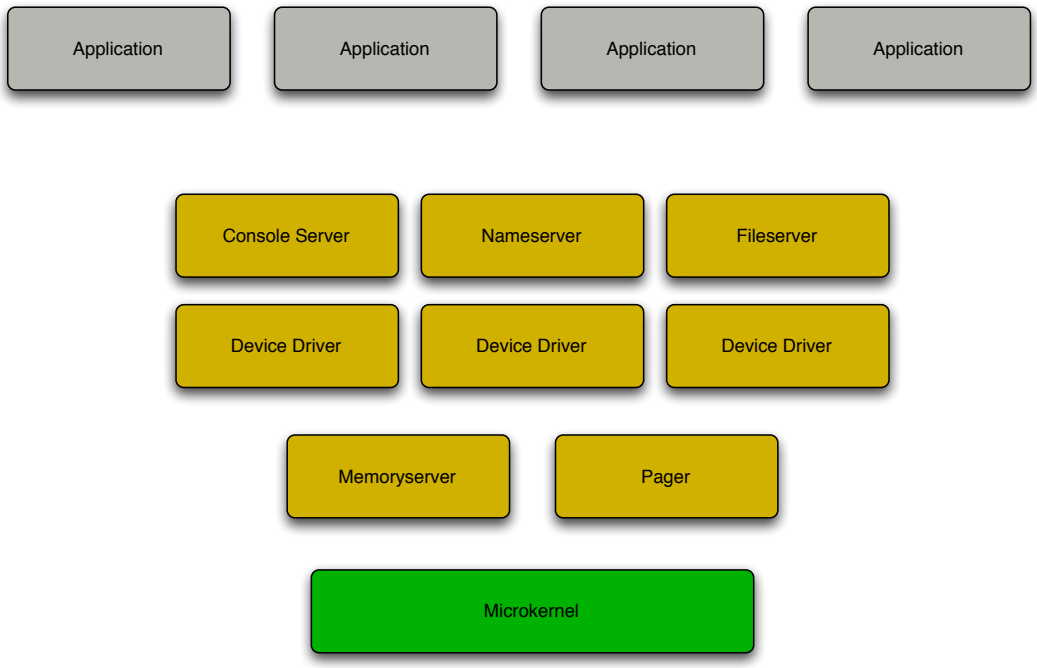


Figure 2.1: Exemplary design of a microkernel-based operating system.

The microkernel approach promises several advantages: The absence of policies in the kernel allows for reuse of the microkernel for operating systems of very different kinds. The strict isolation of components improves system stability, as errors within one component ideally do not affect the rest of the system. Furthermore, components can easily be exchanged as long as their API remains the same.

2.1.1 The L4 Microkernel

L4 is a microkernel of the second generation, originally developed by Jochen Liedtke [23–25]. Apart from some exceptions, it implements the afore mentioned absence of policies in the kernel. It offers two abstractions, threads and address spaces, where threads represent an entity of execution (and thus CPU time), and address spaces represent protection domains. Additionally, L4 offers two basic mechanisms, mapping and inter process communication (IPC).

Threads and Address Spaces

Threads and address spaces are the abstractions offered by L4. A thread is an entity of execution, identified by a unique thread ID. Threads are also the end points of L4's IPC (see below), i.e. a thread can send a message to another thread. Address spaces form protection domains, that is, data from one address space cannot be accessed from within another, unless it is explicitly shared. A thread always belongs to exactly one address space, but an address space can contain an arbitrary number of threads. Address spaces are identified implicitly by the thread IDs of their threads. Address spaces are populated by mapping memory into them. A thread running in an address space can map or grant any page of memory from its own address space to any other address space, as long as there exists a thread willing to receive mappings. This leads to a recursive model of address space construction, as shown in Figure 2.2. Originating from a root address space (called *Sigma0*), other address spaces are populated by mapping memory into them. This design also influences the behavior of unmapping memory from an address space (or restricting access rights): When a thread inside an address spaces decides to unmap a page, the page is removed from all address spaces it was mapped on to. The programmer has the choice if the page shall also be unmapped from the current address space (referred to as flush) or only from subsequent address spaces (referred to as unmap). For example, if a thread in address space 0 in Figure 2.2 performs an unmap on the red page, this page would be removed both from address space 2 and address space 4, but not from address space 0. In case of a flush, it would also be removed from address space 0.

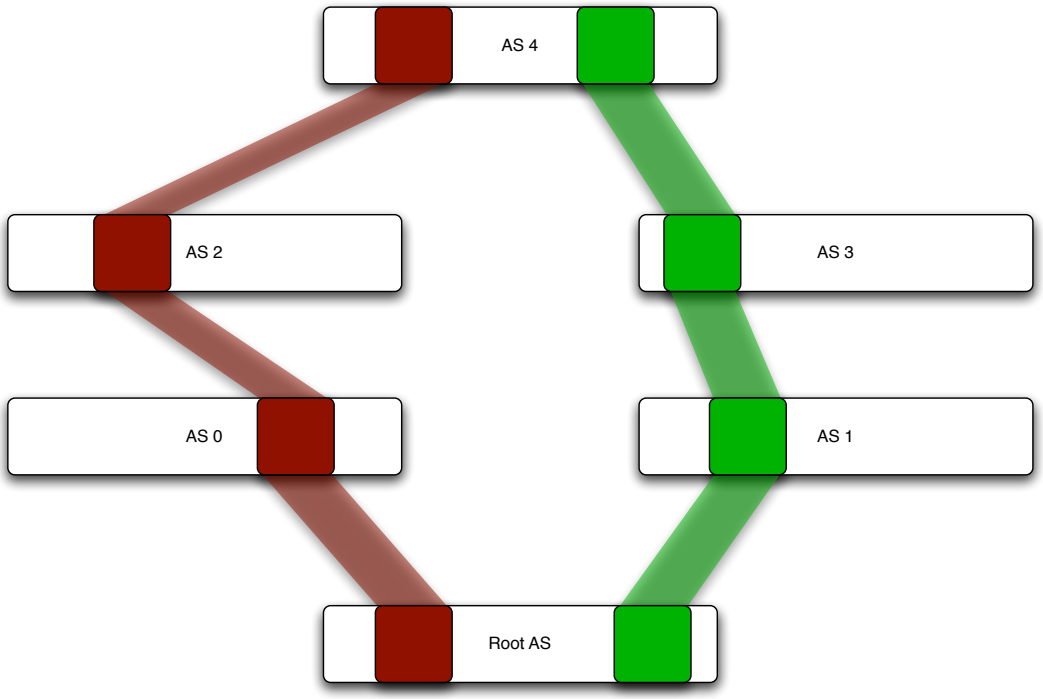


Figure 2.2: Recursive construction of address spaces.

Interprocess Communication

L4 offers an IPC primitive that allows for efficient communication between threads. L4's IPC is synchronous, meaning that both sender and receiver must explicitly invoke an IPC operation and that they are blocked until the operation completes. The kernel thus does not need to buffer any message contents. A message can consist of both untyped and typed items. Typed items can be map items, grant items, or string items. Map and grant items allow for mapping/granting of memory from one address space to another. String items can be used to transfer larger message contents, i.e. to copy arbitrary amounts of data from one address space into another. To transfer message contents and to control IPC operations, each thread has a number of message registers (MRs) and buffer registers (BRs). MRs are used to transfer the contents of the message, BRs control the receiving of string items.
IPC is also possible for threads residing on different CPUs. In that case, inter-processor interrupts are used to synchronize CPUs, making cross-CPU IPC more expensive than IPC between two threads on the same CPU.

Page Fault, Exception, and Interrupt Handling

As stated before, L4 is policy-free. Arbitrary policies can be implemented on top. Therefore, also page fault, exception, and interrupt handling is delegated to user-level servers.

Page Faults Each L4 thread is assigned a pager thread, that is responsible for handling the thread's page faults. L4 makes no restrictions regarding the pager hierarchy, the only condition that must hold true is that each thread is assigned exactly one pager. When the hardware raises a page fault exception, the kernel checks in which address space the page fault occurred. If it is valid, the kernel sends an IPC message to the faulting thread's pager on behalf of the faulting thread. The message contains the virtual address the thread faulted on, the instruction pointer of the instruction that performed the access, and the kind of access it performed (read, write, or execute, if the hardware supports this). The pager's reply specifies the page of memory that shall be mapped to the address the thread faulted on. L4 offers *fpages* to specify memory that shall be mapped. Fpages are an abstraction of hardware memory pages. The smallest possible size of an fpage is the hardware page size offered by a specific machine, but fpages can also be larger than the largest page size the hardware supports. The kernel handles the pager's reply message transparently for the faulting thread by establishing the correct mapping and then discarding the reply message.

Exceptions Exceptions are handled in a way similar to page faults. A thread can be assigned an exception handler thread. When an exception occurs, L4 generates an exception message on behalf of the thread that raised the exception. This message contains the faulting instruction pointer as well as architecture specific exception words. The handler's reply contains the instruction pointer where the thread shall be resumed, and can optionally contain additional architecture specific words.

Interrupts On L4, hardware interrupts are regarded as hardware-implemented threads, and consequently identified by a thread ID. The interrupt handler for a hardware interrupt must be registered as the interrupt's pager. When the hardware raises an interrupt, the interrupt thread sends an IPC to it's handler (i.e. pager). The corresponding hardware interrupt is disabled until the handler replies with a reenable message.

Basic System Services

In addition to the microkernel itself, two servers are always created on system startup: Sigma0, which is the root pager and thus owns all available physical memory (including device memory), and a roottask. Sigma0 hands out physical memory to applications (either by the applications explicitly requesting the memory or by handling page faults). Yet, Sigma0 hands out each free physical page exactly once. This is for security reasons: At system startup, all memory is distributed amongst the operating system services that require it (e.g. pagers). If another process attempts to request memory from Sigma0 afterwards, the request will fail.
The roottask and all threads that are created in the roottask's address space later on are privileged in that they are allowed to execute certain L4 system calls, mainly for thread and address space creation and manipulation. Threads running outside the roottask's address space are not permitted to execute these system calls. Instead, a system call server thread running in the roottask's address space can be used as a wrapper for system calls, allowing to implement additional security policies at user-level.
With this initial configuration, it is possible to create all other necessary servers and thus bootstrap the entire system.

Performance Analysis

As seen in the previous paragraphs, even elementary system services like pagers, exception handlers, and interrupt handlers are implemented as user-level applications. A thread requesting a service offered by a server must use L4's IPC primitive to call that service. The most common case of IPC is communication between two threads in different address spaces, but on the same CPU. In that case, a single IPC send operation requires to change the privilege level (i.e. to enter the kernel), to switch address spaces, and to change the privilege level again (to leave the kernel and switch to the receiver thread). The same effort is needed for the reply. With hierarchical services, the required number of IPCs can increase dramatically. Although L4 is said to be one of the fastest microkernels currently available, one must keep IPC costs in mind when designing a system architecture. Long call chains should be avoided as well as communication across CPUs. Shared memory can be an alternative in some cases.

2.2 NUMA Systems

Multiprocessor systems in general are systems with more than one CPU (or CPU core). With symmetric multi-processing, all CPUs in the system are equal in that

every CPU can execute every task. In a “classical” SMP system, all CPUs are connected to a single shared memory via a single memory controller and bus. This design ensures uniform access latencies for the entire memory, but the memory interconnect becomes a bottleneck with increasing number of CPUs. Parallel memory accesses of multiple CPUs must be serialized and thus limit scalability of SMP machines. Systems with a non-uniform memory architecture (*NUMA*) are very similar to SMP systems, but eliminate the single memory interconnect as a bottleneck. Instead of having all CPUs equally connected to main memory, the system is partitioned in so-called *nodes*, where each node consists of one or more CPUs (or cores) and an amount of node-local memory (together with a memory-controller). Nodes are connected via a fast local interconnect. This interconnect allows a CPU to access memory of another but the own node (so-called *remote memory*). However, remote accesses can be much more expensive than accesses to local memory. Costs depend on hardware and timing characteristics, but also on the distance (i.e. the number of hops) between the CPU performing the access and the memory that shall be accessed. Figure 2.3 and 2.4 show simple examples of an SMP- and a NUMA-system, respectively.

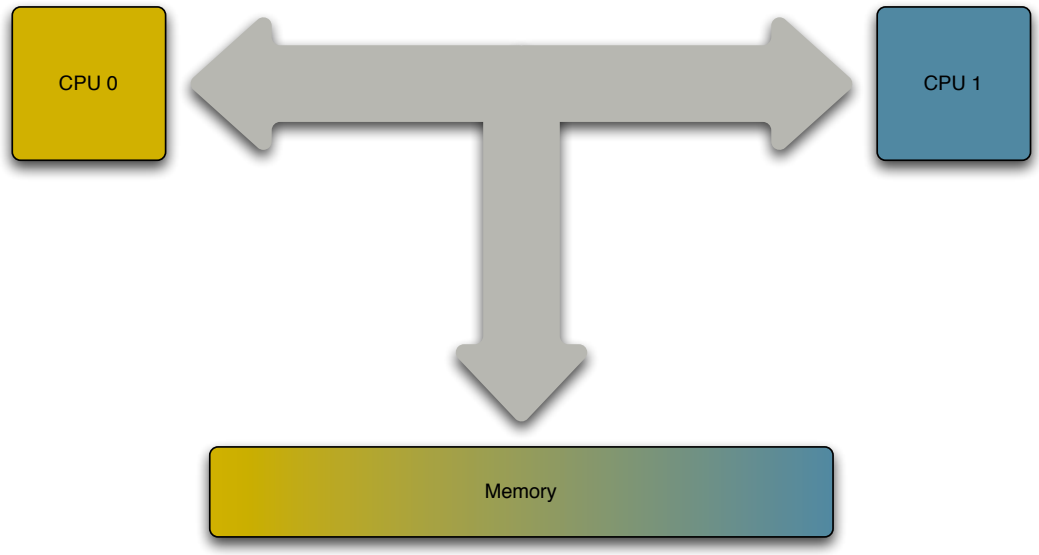


Figure 2.3: A ”classical” SMP system with uniform memory access characteristics. Both CPUs are directly connected to a single shared memory.

Nowadays CPUs normally have a hierarchy of caches to reduce the number of code and data fetches from main memory. Caches pose additional challenges to multiprocessor system: If data in memory is altered by one CPU, there might still exist copies in another CPU’s cache. These copies must be found and either be updated or invalidated. This cache coherence nowadays is commonly hardware-implemented. In case of NUMA systems, one also speaks of *ccNUMA* if cache coherence is ensured “automatically”, i.e. in hardware. In this work, we do not deal with NUMA systems where cache coherence is ensured in software. Therefore, we use the terms NUMA and ccNUMA interchangeably.

While NUMA allows for better scalability with high numbers of CPUs, it also

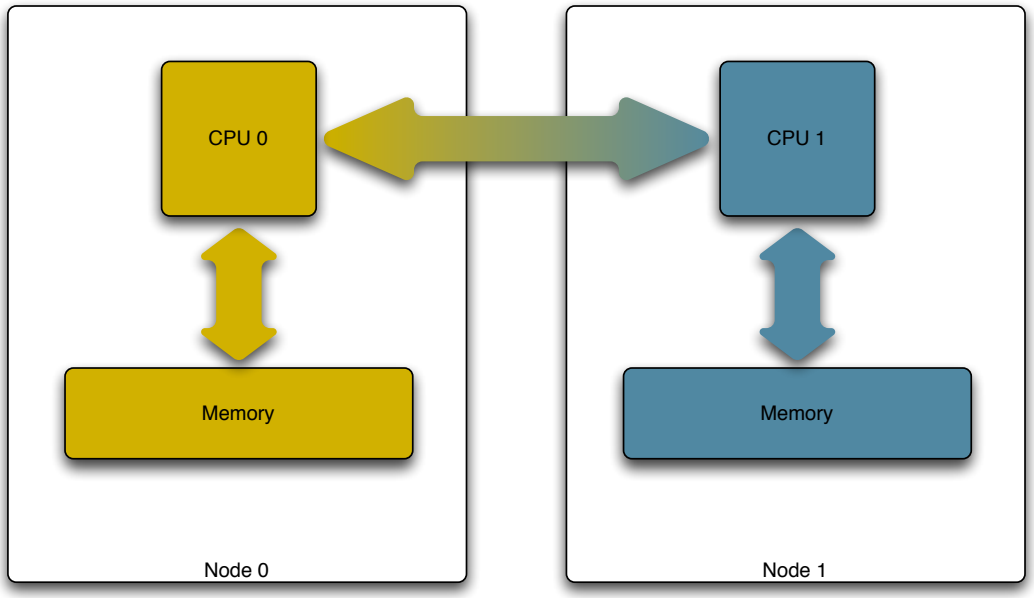


Figure 2.4: A NUMA system consisting of two nodes, each with one CPU. Accesses to remote memory are passed over the interconnect between the CPUs.

makes system- and application programming more complicated. Operating systems and applications designed only with regard to SMP machines will also run on NUMA machines, but with performance penalties, especially on large-scaled machines with a high remote-to-local latency ratio.

2.2.1 Memory Management for NUMA Systems

Optimizing data locality on NUMA systems is an important requirement to allow for good performance of both operating system and applications. Data locality means that data is placed on the node or near to the node where it is accessed from. There are two general possibilities: Manual placement, where the application programmer decides on which nodes memory is allocated and when and where to move it, and automated placement, where the operating system or a memory manager make these decisions at runtime, depending on the current system behavior. Both approaches have advantages and disadvantages, and combinations of both are possible.
Cox and Fowler implemented a coherent memory abstraction for NUMA systems, which hides most of the hardware's NUMA characteristics by transparently replicating and migrating pages [13]. The authors replicated page tables to allow for per-CPU mappings of replicated code and data. They state that replication of page tables does not limit scalability because a page table needs not contain all mappings of an address space, i.e. can be synchronized lazily when necessary. For shootdowns, the authors chose an approach that requires notification only of those processors that are using a mapping of the affected page. Additionally, only processors on which an affected address space is currently active need to be interrupted. Our approach for updating page tables is similar in that we also populate page

tables lazily. For unmaps, we are also able to determine a subset of all processors that must be notified. However, as our approach works entirely at user-level and uses only abstractions and mechanisms offered by the L4 microkernel, we are not able to tell which address space is currently active on a specific CPU.

Bolosky et al. [7, 8] implemented several NUMA memory management policies and compared them to an off-line, optimal cost policy. Their results show that a good NUMA policy is of importance, as it can improve overall program performance by as much as 25% to 50%, and that different memory architectures can require different policies. They identified false sharing as a dominant factor that negatively affects application performance.

Sandhu et al. [34] argue that a hardware page is the wrong abstraction for managing shared data, because data sharing in applications does not occur at page granularity. Instead, they group locations of shared data that are accessed together and in the same way (read and/or write) to a so-called shared region. Cache and memory coherence is then enforced at region granularity. Yet, an application programmer must manually specify regions in the source code. Before data of a region is accessed, a process must explicitly request read or write access. Such a request not only performs locking, but also interacts with the system to enforce cache and memory coherence. While the authors state that their approach has several advantages over traditional NUMA memory management schemes (like reduced false sharing and reduced costs for maintaining consistency), the disadvantage of their approach is that it relies on information either given by the application programmer or by the compiler.

The authors of [27] showed that a simple first-touch allocation policy can lead to significant performance improvements both on systems with hardware and software coherent caches. To deal with the problem that one thread initializes data that is later used by another thread on another node, the authors introduced a "done with initialization" annotation, allowing a programmer to specify the point at which the operating system should begin with dynamic placement. An additional "phase change" annotation can be used to tell the operating system to reevaluate its placement decisions. More sophisticated migration and replication strategies (including some which require additional hardware support) did not lead to much improvement for the programs used for evaluation.

In [40], the authors developed an algorithm to decide whether a page of memory shall be migrated or not. Their approach requires information about cache misses or TLB misses, both on a per-page granularity. The hardware raises an interrupt when the number of misses exceeds a predefined threshold. Their approach achieved good results, however, the required hardware support is not available on most machines.

The authors of [32] implemented a user-level solution for dynamic page migration, loosely based on the afore mentioned approach. Their approach also requires reference counters per page, which are available on the SGI Origin2000 hardware they used.

The authors of [9] focused on building an x86-based NUMA system. However, special hardware was used to realize the interconnect between nodes and to allow for monitoring of memory accesses. The gathered information was not used for dynamic page migration/replication, but only for profiling purposes, allowing

an application programmer to find and eliminate performance bottlenecks and to manually optimize a program to improve data locality.
[26] suggests another approach. The authors used x86 performance counters to capture memory accesses and logged calls used for memory allocation during a first run of a program. From the collected information, a profile is generated, which can then be used to take better page-placement decisions when the program is run a second time.
The authors of [22] compared various policies for dynamic page replication and page migration. They derived the following conclusions from their observations: The performance of programs written for UMA architectures can be improved by applying dynamic page migration and replication, but normally does not achieve the performance of "hand-tuned" programs. They didn't find a single policy that could be considered as best for all kinds of programs that were used for benchmarking.

2.2.2 Microkernels and NUMA architectures

IPC performance is of paramount importance for microkernel-based operating systems. Gamsa et al. optimized IPC for shared-memory multiprocessors [15]. Their design is based on protected procedure calls rather than on message passing. It preserves parallelism and avoids accesses to shared data and locking in the common case. A service call is always executed on the CPU of the caller, thus avoiding cross-CPU notifications. Although their implementation runs on a NUMA machine, the authors state that the non-uniform memory access time had no measurable effect on the IPC performance because their design avoids accesses to remote memory. In contrast, L4's IPC based on message passing allows for communication between threads on different CPUs (with considerably higher costs because of the required cross-CPU notifications). Our user-level design aims for minimizing the required number of cross-CPU IPCs and for preserving parallelism by explicitly replicating service threads per CPU. Yet, we cannot completely avoid cross-CPU IPC, but require it for unmapping memory.
The Raven kernel [33] is a microkernel tailored to shared memory multiprocessors. Many of its services are implemented at user-level, with the goal to minimize kernel invocations. However, the Raven kernel focuses on systems with a uniform memory architecture, not on NUMA systems.
Tornado [14] and its successor K42 [4] are both microkernel systems designed with a special regard towards NUMA machines. An object-oriented structure is used to reduce contention and to improve locality. Furthermore, Tornado introduces clustered objects, which allow multiple component objects to appear like a single object. K42 additionally allows hot-swapping of objects, i.e. to change implementations at runtime to better suit the current needs. Clustered objects are useful to replicate or migrate objects transparently for the client that uses the object. We did not require such a flexible abstraction for the changes to L4's kernel objects we made. However, although not explicitly designed as a clustered object, our abstraction of a task address space is comparable: While the task running within its task address space has the illusion of a single address space, different address spaces are used per node so that address space management information (e.g. page tables) is

replicated. These per-node address spaces are created and populated on demand.
Disco [10] chooses a different approach. The authors argue that modifying existing operating systems to better support multiprocessor architectures (including, but not limited to NUMA machines) is a difficult and resource-intensive task. Instead, they introduce an additional layer between hardware and operating system, which acts like a virtual machine monitor, but with the goal to hide the hardware specifics (such as the non-uniform memory access latencies) from the guest operating systems. Multiple instances of unmodified (and non-optimized) operating systems can run on top on a single, scalable computer. Disco also implements page replication and migration to provide local memory to the virtual machines where possible. The guest operating systems running on top thus do not need to be aware of the NUMA characteristics of the underlying hardware. Cellular Disco [16] extends Disco's approach by turning a large-scale shared-memory multiprocessor into a virtual cluster, consisting of a number of different "cells". It allows for fault containment, i.e. faults in one cell do only affect virtual machines running in that cell, while still preserving the benefits of shared-memory multiprocessors by implementing dynamic resource sharing. Both for Disco and Cellular Disco, the virtual machine monitor introduced to abstract the hardware is comparable to a microkernel, as it is intended to be kept small, and its main purpose is to abstract the hardware. However, the (Cellular)Disco hypervisor is not required to be policy free. For example, Cellular Disco implements two separate CPU load-balancing policies and a gang scheduler.
VMWare's ESX Server is also NUMA-aware [41]: Each virtual machine running on top is assigned a home node, and the virtual machine is only scheduled on CPUs on this node. The hypervisor periodically checks load distribution and migrates virtual machines between nodes if appropriate. The virtual machine's memory is transparently migrated to the new node to avoid accesses to remote memory. Memory sharing between virtual machines is also realized on a per-node basis, again eliminating the need for remote accesses. Additionally, memory and processor utilization can also be controlled manually.

2.3 Synchronization

The performance of synchronization mechanisms is crucial for the overall system performance. It is therefore not surprising that synchronization mechanisms were and still are a popular research area. The authors of [11] compared remote invocation (i.e. message passing) with direct access to remote memory. They conclude that the choice between these two alternatives heavily depends on architectural features, mainly the costs for remote invocation, the costs for atomic operations (required for synchronization), and the ratio of remote to local memory access time. It is also influenced by cache characteristics. The authors argue that for every machine there exists a break-even point regarding the length of an operation above which remote invocation is favorable.
The microkernel-based *Hurricane* operating system combines several locking techniques to improve performance and scalability [38]. The authors implemented a hybrid approach, using coarse-grained locks when held only for a short period of time and fine-grained locks to protect data for longer periods of time. They used

distributed locks to allow processors to spin locally when waiting for a lock, thus reducing traffic on the inter-connection network. Hierarchical clustering [39] was used to allow for replication and migration of system data structures. Hierarchical clustering is a technique that partitions a multiprocessor system into clusters, each of which runs an independent copy of the microkernel. Data is replicated to clusters where possible.
MCS locks are a scalable implementation of spin locks, with a constant (i.e. independent of the number of CPUs competing for the lock) number of remote accesses per lock acquisition [31].
Uhlig invented dynamic locks [36, 37], allowing the lock granularity to be adjusted at runtime, depending on access and sharing patterns of the protected objects.

CHAPTER 3

Design

Our design decisions are driven by two main requirements: preserving parallelism and locality of accesses. Preserving parallelism means that parallel, independent request must be handled in parallel by the operating system. This leads to better scalability, which is a general requirement for all multiprocessor systems. In this context, scalability means that system performance increases (optimally linear) with the number of CPUs in the system. Locality is of particular relevance on NUMA machines, meaning that data should be kept in local memory of the accessing node wherever possible.
For the sake of the following discussion, we define the abstract term of an *object*, which can refer to concrete data objects such as page tables or thread control blocks, but also to more abstract objects such as address spaces or threads. There exist several possibilities to satisfy the requirements for parallelism and locality, with none of them being perfect for all cases. Instead, it heavily depends on the access and sharing patterns of an object to decide which one to choose. The following paragraph discusses possible solutions to allow for preserving of parallelism and/or improving locality:

Replication of objects: Instead of having only a single instance of an object, one or more replicas are created. This has two advantages: First, contention can be reduced or avoided, thus preserving parallelism. Second, the replicas can be placed in local memory of all accessing nodes, making all accesses to the object local accesses. However, replication comes at some cost: If the replicated object is modified, all copies must be kept consistent by propagating the changes to all replicas. This makes replication feasible only for objects with a high read-to-write ratio, i.e. objects that are much more often read than written.

Migration of objects: In case of an object with a high number of write accesses, the overhead for synchronization of replicas is too high. In that case, instead of replicating the object, it can be a better choice to migrate it to the accessing node. Of course, migration also comes at some cost: The object has to be

copied to the destination node's memory, the memory on the source node has to be freed, and references to the object might need to be updated. Furthermore, migration only makes sense if the object is accessed mainly by a single node. In case of multiple nodes accessing an object in parallel, pingpong-effects (i.e. the object being migrated between nodes all the time) must be avoided. In contrast to replication, migration only improves locality, but not scalability.

Pinning of data objects: If an object cannot be replicated because of frequent write accesses and cannot be migrated because it is accessed by different nodes simultaneously, the only remaining possibility is to keep a single instance of the object in local memory of one node. This solution of course neither preserves parallelism nor does it favor local accesses over remote accesses (the object is in remote memory for all but one node). It can, however, be useful if an object is known to be accessed mostly by a single node. Pinning thus can avoid accidental migration or unnecessary replication of such an object.

When an object can be accessed from different CPUs simultaneously, accesses must be serialized to avoid race conditions. Serialization can either be performed lock-based or message-based. The decision which is favorable depends on the access and sharing patterns of the data object as well as on the characteristics of the underlying hardware, mainly the costs of remote memory accesses and the costs of inter processor interrupts (IPIs). Uhlig suggests dynamic locking to adapt lock implementation and granularity at runtime, depending on the degree of sharing [36]. Yet, dynamic locks don't take hardware characteristics into account, which also affect the choice for an "optimal" synchronization mechanism. For example, on a machine with low IPI-costs, message based synchronization might be favorable over lock base synchronization, especially if the costs for remote accesses are high.

3.1 System Architecture Overview

A microkernel provides only a minimal abstraction of the underlying hardware and does not implement operating system services like pagers, file system handling, and so on. These services must be implemented as separate tasks, running at user-level. The implementation of these services must follow the afore mentioned principles: Preserving parallelism and locality of accesses. Therefore, each service must offer at least one thread per CPU that can handle incoming service requests. Each of these threads must work on local data where possible. In this way, parallelism is preserved, as independent, parallel requests to the same server can be handled in parallel. Furthermore, the amount of cross-CPU communication (which is far more expensive than communication between threads on the same CPU) is minimized, because a thread requesting a service can always call a server thread on the same CPU. Figure 3.1 shows a simple, microkernel-based operating system, running on a system with two nodes (and one CPU per node). In this example, two servers run on top of the microkernel: a memory server and a pager. Each of these servers consists of two threads, i.e. one per CPU. On top of the servers run different applications, both single-threaded applications (assigned to exactly one node

at a time) and multi-threaded applications (that might be active on both nodes simultaneously). We now have a look at the required operating system servers.

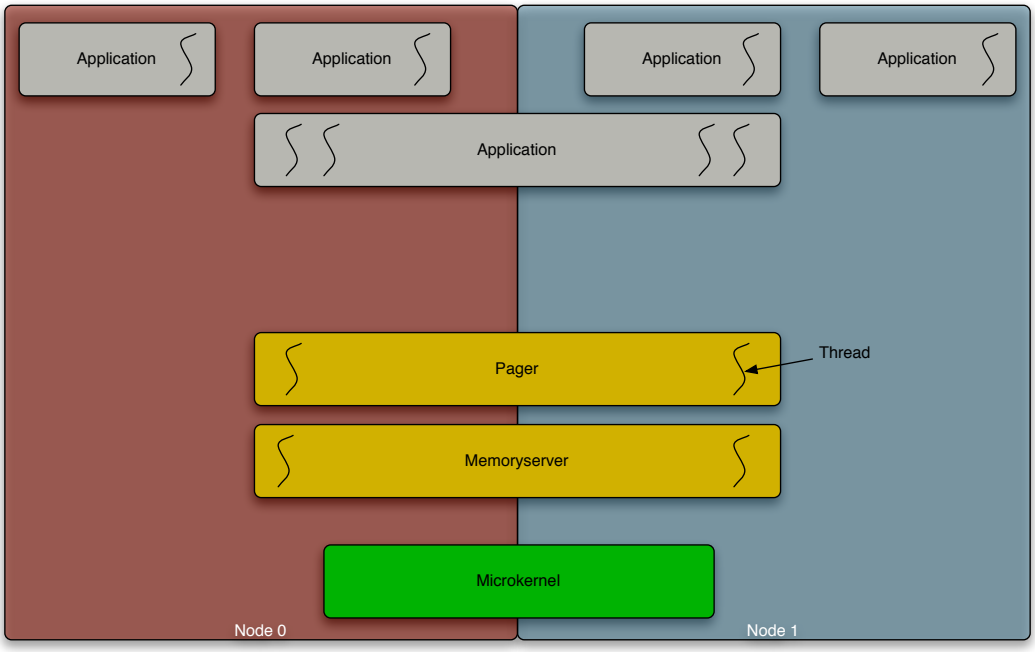


Figure 3.1: Architecture of a microkernel-based operating system for NUMA hardware.

3.1.1 Pager/Memory Manager

A pager and a memory manager are needed to satisfy an application's demand for memory. If a paged thread touches a region of virtual memory that is currently not mapped to physical memory, the pager must handle the resulting page fault in cooperation with the memory manager, most likely by mapping some memory into the thread's address space (if the thread performed an illegal access, the pager might initiate to kill that thread instead). In our design, a pager is responsible for paging all threads that run on the same CPU. If a thread migrates to another CPU, one pager must hand the thread over to the other. While this handoff itself is straightforward, there remains the open question of how the new pager gets the required information about the migrated thread's address space layout, i.e. which pages are currently mapped in, etc. If all pager threads run within the same address space (as shown in Figure 3.1), this is not a problem - in that case, management information for all address spaces is visible to all pager threads. However, there are some good reasons to place each pager thread into its own address space, which we will discuss later. In that case, data from one pager is not easily accessible by other pagers, but must instead be shared explicitly. It is either possible to copy the entire management information upon migration, or to query the originating pager on demand, i.e. when a page fault occurs. For both solutions, one has the choice between using IPC or shared memory. We discuss thread migration in more detail in Section 4.2.4.

3.1.2 Load Balancer

Load balancing is an important task in multiprocessor systems. If the load on one CPU becomes too high, threads or processes must be migrated to other CPUs with less load. For NUMA systems, CPUs in the system should not be treated equally when considering load balancing: It is generally favorable to migrate threads or tasks to another processor of the same node than to an arbitrary processor in the system, at least if the thread/task has large parts of its working set in local memory of the current node. To be able to model the system topology for load balancing decisions, we propose a design with one (potentially multi-threaded) load balancer per node, which is responsible only for surveying the load within its own node and to migrate threads between CPUs of its node. When the overall load of the node is too high, it can notify a master load balancer which keeps track of the load distribution of the entire system. This solution is similar to the scheduling domains of recent Linux versions, which are also used to model the system's topology to be able to take better scheduling/balancing decisions [6].

3.2 Node-Local User-Level Data

Parallel applications consisting of multiple threads are a likely workload on a NUMA machine. With only a small number of CPUs per node, the application's threads must be distributed amongst nodes to execute the application in parallel. In that case, the application's address space is active on different nodes simultaneously. This scenario raises a number of questions to deal with:

- Where to place page management information both for the microkernel and for the user-level pagers? With L4's current implementation of address spaces, the layout of an address space is represented with a single kernel page table hierarchy which lies in physical memory of a specific node. If the corresponding address space is active on multiple nodes in parallel, page table accesses are remote accesses for all but one node, which contradicts the principle of locality. A similar problem arises for the address space information the user-level pagers have to keep.
- How to provide node-local data for applications? An application should be given the possibility to allocate data on and migrate data to a specific node, either by offering an API that allows the application to influence memory allocation or by an OS subsystem that performs automated allocation, replication, and migration of memory.

In the following, we present different approaches to tackle these problems.

3.2.1 A First Approach

An application can create a designated region within its virtual address space which is partitioned into different areas, each backed by physical memory of a specific node, as depicted in Figure 3.2. Those areas can then be used as a "pool" of node-local data. All data that is important only or mostly on a specific node can be

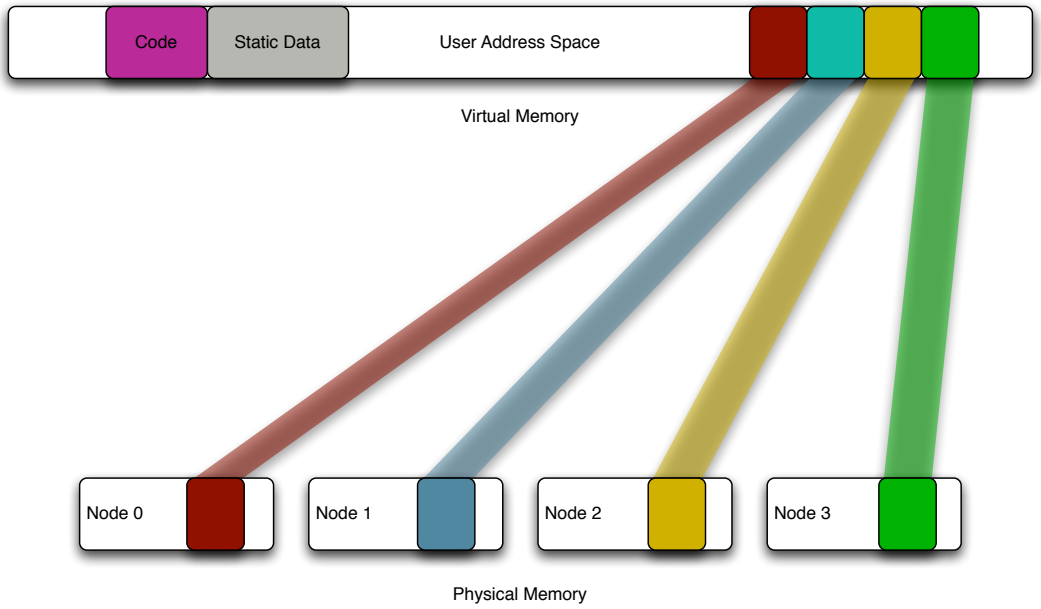


Figure 3.2: Pools of memory from each node can be mapped to different locations within the application's virtual address space, allowing for dynamic allocation of node-local memory.

placed inside the corresponding node pool. Thread stacks can be allocated in the pool of the node the corresponding thread belongs to. However, this approach lacks some flexibility: Replication of the application's code is difficult: Unless relative addressing is used, code is expected at a specific virtual address and cannot easily be copied to different locations. To do this, relocation would have to be performed. More generally spoken, it is not possible to map the same range of virtual addresses to different physical addresses on every node, which is a fundamental requirement for dynamic page replication. Any form of node-local data thus requires indirection. A more flexible solution is desirable, which allows an address space layout as shown in Figure 3.3: While the gray, global region of the virtual address space is backed by the same physical memory on every node, the local regions are mapped to different physical memory on every node.
Additionally, while this first approach does not require any changes to L4 or the way L4's address spaces are used, it also leaves the question how to handle address space management information open.

3.2.2 In-Kernel Replication of Page Tables

Instead of having a single page table hierarchy per address space, L4 could be modified to have a page table hierarchy for every node (or even for every CPU) in the system. These hierarchies can be placed in local memory of the node they belong to, thus improving data locality. Additionally, this approach allows for a flexible form of node-local data: As the replicated page table hierarchies need not necessarily be consistent, different mappings can be established for different nodes, allowing the same region of the virtual address space to be mapped to different physical locations, depending on the node on which the access is performed. This

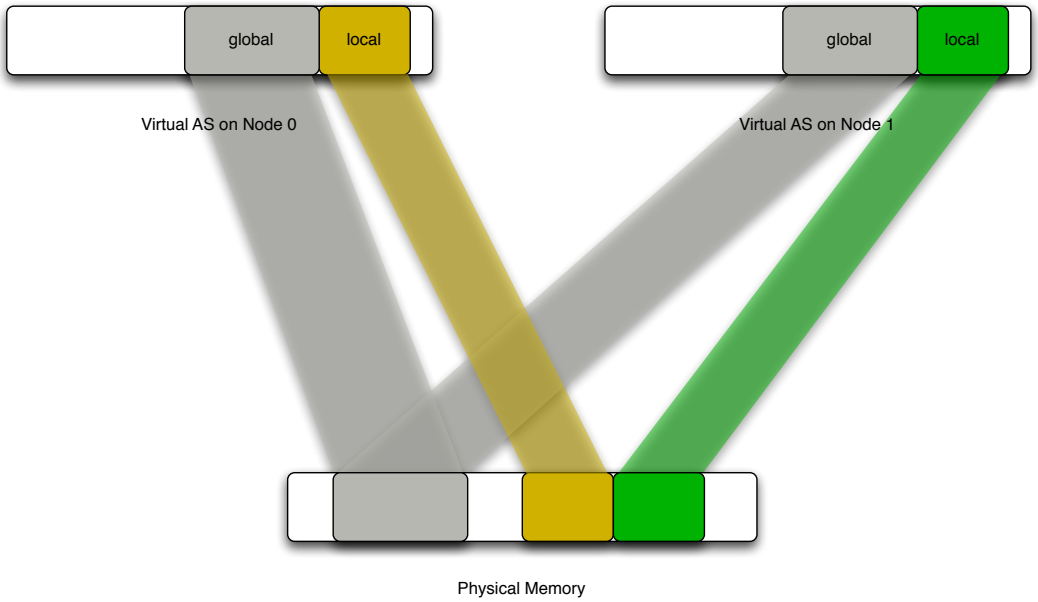


Figure 3.3: The local region of the virtual address space is mapped to different physical addresses on every node.

greatly simplifies code replication, because code can always be mapped to the "correct" virtual addresses, but backed by different physical memory on every node. As the kernel cannot know whether a newly established mapping shall be made global or local, changes to the API are necessary. With an additional bit in each map or grant item, the user can specify whether a mapping shall be made local or global, i.e. equally visible on all nodes. Global mappings must be propagated to all other page table replicas, i.e. synchronization must be performed. The choice for the right synchronization primitive is not a trivial one: If lock-based synchronization is used, one has the choice between different locking granularities (the possibilities range from locking of an entire page table hierarchy to locking of single entries) and lock implementations. Additionally, lock-based synchronization requires accessing remote memory to perform modifications. In contrast, message-based synchronization avoids or at least reduces both the need for locking and accesses to remote memory, but one has to take into account the costs for notifications across CPUs, i.e. inter processor interrupts. Hence, the decision which synchronization primitive is favorable does not only depend on the access and sharing patterns of the page table hierarchies, but is also influenced by the costs for IPIs and for remote accesses [11]. While locking granularity can be adjusted at runtime, depending on the access and sharing patterns [36], it is difficult to also take hardware characteristics into account. Manually modifying the kernel to suit the hardware platform it is designated to run on contradicts the idea of a minimal, flexibly useable microkernel. Additionally, propagating a newly established mapping to all page table replicas might be unnecessary, as it might never be accessed on all nodes. Thus, a lazy approach that performs synchronization only when and where necessary is preferable. In case of address space organization, there exists an alternative that avoids eager, explicit in-kernel synchronization of page table replicas, while still

providing the desired flexibility. We present this approach in the next section.

3.2.3 Using Different Address Spaces

The approach we suggest can be realized entirely at user-level, without requiring changes to the kernel or the kernel API. Instead of explicitly replicating the page table hierarchy of an address space in the kernel, we use different, unmodified address spaces for each node. From the viewpoint of the kernel, these different address spaces are completely unrelated, the kernel thus does not perform any form of synchronization between them. Instead, synchronization is performed entirely at user-level between the pagers of the address spaces. Figure 3.4 shows an example with an application running on two nodes (each with one CPU). A different address space is used on each node, and each is managed by its own pager thread. Synchronization of the two address spaces is the pagers' task. As Figure 3.4 also shows, this approach allows both for node-local mappings (green) and global mappings (gray).

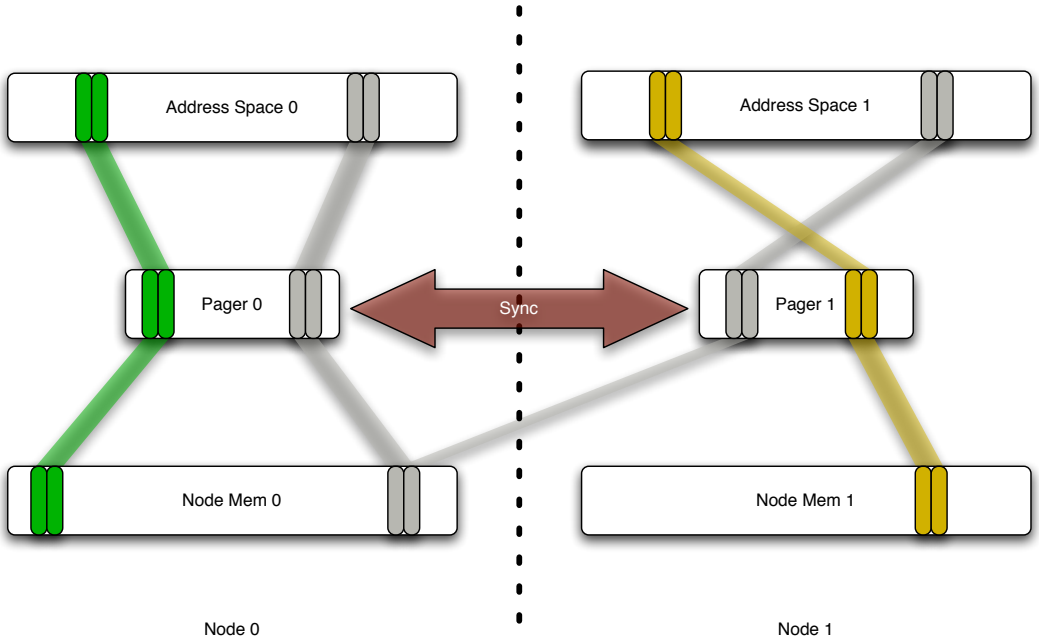


Figure 3.4: Per-node address spaces allow both for global and node-local mappings and avoid introducing an additional synchronization primitive in the microkernel.

The required synchronization between the pagers affects the way mappings are established and removed. While the pagers perform explicit synchronization at user-level, the kernel page tables are implicitly synchronized. In the following, we discuss how the usage of per-node address spaces affects both establishment and revocation of mappings.

Establishing Mappings

Establishing a node-local mapping is straightforward: If a pager receives a page fault IPC on a virtual address where a node-local mapping is expected (e.g. an address where the application code lies), it can simply reply with a map item to map the correct physical memory. No form of synchronization is required. Establishing a global mapping is a more complex operation. When a pager is notified of a page fault on a virtual address that shall be global and for which no valid mapping exists yet, it also replies with the correct map item, but this time, the other pagers must be made aware of that newly established mapping. As for the in-kernel approach, there exists a multitude of synchronization possibilities between the pagers, which we discuss later. When the virtual address where the new mapping was established is accessed on another node, there will also occur a page fault because the kernel page tables of the address space on that node are not yet updated: From the viewpoint of the kernel, the different address spaces are completely unrelated, it does therefore not perform any form of synchronization. However, because of the synchronization between the pagers, this pager now knows what physical memory must be mapped to that address and can reply with the correct map item. Every pager must therefore be able to access and map all physical memory.
In conclusion, arbitrary synchronization strategies can be applied between the user-level pagers, while the synchronization of kernel page tables for the different address spaces works implicitly and lazily, as new global mappings are established on a page fault basis.

Revoking Mappings

In this section, when we talk about revocation of a mapping, this can mean both that the mapping is completely removed (i.e. the corresponding page(s) are unmapped) or that the access rights of the mapping are restricted (e.g. from read-write to read-only). Revocation of global mappings is a more complex task than establishment of global mappings, as this time, the in-kernel page tables cannot be updated lazily. If a global mapping is revoked, this change must be made visible to all affected address spaces immediately. Otherwise, it can happen that the mapping can still be accessed in another address space. The recursive nature of L4's map and unmap operations allows for different approaches to tackle this problem. If a thread calls the *unmap* system call on a page within its address space, the page is unmapped from all address spaces to which it was mapped onward from the current address space. In Figure 3.4, if pager 0 would call *unmap* on one of the gray pages, this paged would be unmapped only from address space 0. To unmap a page from all affected address spaces, there exist various approaches:

1. The originating pager notifies all other pagers to call *unmap* for themselves. An IPC-based protocol can be used for this. The disadvantage of this solution is that the number of required IPCs scales linearly with the number of remote pagers that must be notified. This number is influenced by the number of tasks to which a page was mapped and by the parallelism of these tasks, i.e. on how many nodes a task is active simultaneously. The total number of nodes in the system is an upper bound for the required number of cross-CPU notifications.

2. A root pager is used that donates memory to and revokes memory from the pagers running on top. In that case, the originating pager can call the root pager to perform the *unmap*, and the corresponding page will be unmapped from all address spaces it was mapped to, including the address spaces of the pagers. For example, if the root pager in Figure 3.5 calls *unmap* for one of the gray pages, this page will be removed from the application's address spaces (i.e. address space 0 and address space 1), but also from the address spaces of pager 0 and pager 1. In contrast to solution 1, only a single IPC call to the root pager is required. However, revoking the mapping also from the pagers' address spaces is an unwanted behavior.

3. The *unmap* system call is modified to allow a "directed unmap", i.e. allowing a pager to directly unmap memory from another but its own address space. Such a modification would impose great security problems, as a pager must not be permitted to unmap arbitrary pages it is not responsible for. One would have to introduce an additional security model into the kernel, e.g. by grouping pagers together that are allowed to unmap each other's memory. Even if this was done, there remain open questions. For example, a pager that wants to unmap a page handled by another pager would have to know at which virtual address this page is mapped in the other pager's address space. To revoke a global mapping from n address spaces, n calls to the directed unmap system call are needed, thus the costs for this solution also are in $O(n)$.

From these three possibilities, we favor the first, but the second can be an alternative if the overhead of the first turns out to be too high. In Chapter 5, we discuss the expected costs and potential optimizations.

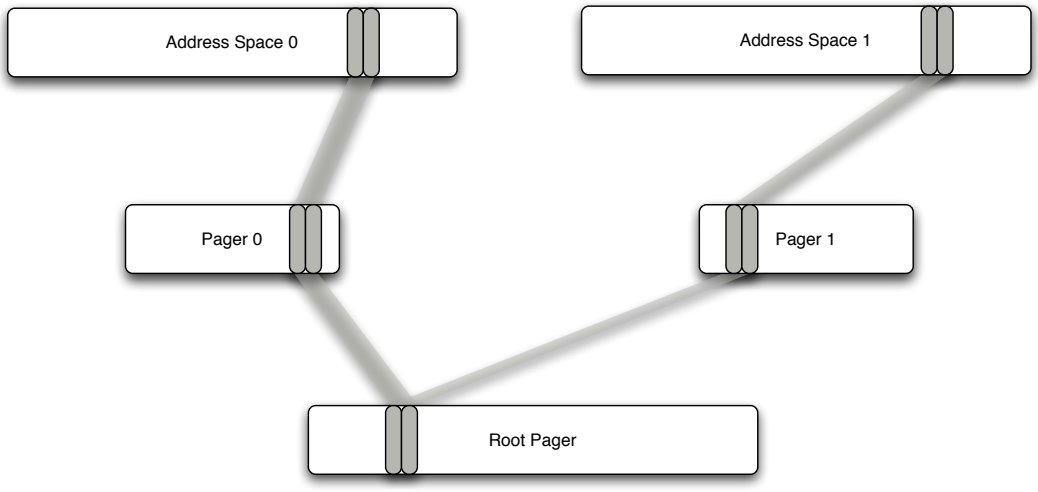


Figure 3.5: Recursive mapping hierarchy with a root pager donating memory to all other pagers.

Synchronization between Pagers

Any changes made to a global mapping (including the creation of a new global mapping) by one pager must become visible in the other pager's data structures, too. The easiest solution for this is to let all pagers work on the same data structures by mapping them into every pager's address space. This, however, does not only require locking, but also contradicts the principle of locality, as the management information would be in remote memory for all pagers except for one. Additionally, a single instance of management information makes it difficult to store information about node-local mappings. As the pagers need the management information simultaneously and frequently, migration also is not an option. We therefore decided to replicate the information, so that each pager can work on its own data. Yet, the problem how these replicas are synchronized remains. As we stated in Section 3.2.3, in case of revocation of a mapping, changes must be propagated to other pagers eagerly. This requires an eager, push-based approach, i.e. the originating pager must notify all other pagers to revoke the mapping and to update their management information. Therefore, an IPC-based solution is favorable: The originating pager calls an appropriate function of the other pagers, triggering remote revocation of the mapping. In the opposite case, if a new, global mapping is established by one pager, other pagers will still suffer a page fault on the corresponding virtual addresses afterwards, allowing synchronization of pager management information to be performed lazily. Alternatively, the originating pager (i.e. the one that established the new mapping) can also eagerly update the other pagers' data structures. The disadvantage of the eager approach is that it might cause unnecessary updates, in case the newly established mapping will not be accessed on all nodes. With a lazy approach, a pager receiving a page fault at a virtual address that is not explicitly marked as local must query all other pagers to find out if there already exists a valid, global mapping. This solution scales bad, as the costs grew linearly with the number of pagers (and thus CPUs or nodes) in the system. We decided to use a combination of eager and lazy propagation to avoid the necessity of querying all other pagers: One of the pagers is designated as a "master pager". Whenever one of the other pagers establishes a new mapping, it eagerly notifies the master pager of the changes. Upon a page fault on a global mapping, a pager now only needs to query the master pager to find if there already exists a valid mapping. To avoid that the master pager becomes a scalability bottleneck, a different master pager can be used for every task, for example the pager on the node where the task was created on. This reduces contention on a single pager. Both the update of the master pager's management information and querying the master pager to find valid global mappings can be done via IPC or via shared memory. In the latter case, locks are required to ensure consistency. The same arguments as for the in-kernel solution influence the choice for a synchronization primitive: Access and sharing patterns as well as latencies of remote accesses. With our user-level based solution however, the kernel is kept free of such a primitive and remains flexible, while user-level servers can be adapted to better suit a specific hardware, if necessary.

Thread Migration

Migrating a thread to another CPU on another node now becomes more complicated, as it is no longer sufficient to only put the thread on the new CPU with an appropriate system call. Instead, it must also be migrated into the corresponding address space on the target node. If no address space exists yet, one must be created. The pagers therefore must be aware of which address spaces belong to the same task. We discuss implementation-depended problems of thread migrations in Section 4.2.4.

3.3 Kernel Objects

For all kernel objects, such as page tables and thread control blocks (TCBs), it must be decided how locality can be improved, and how parallelism can be preserved. Because L4 is a microkernel, the number of objects we need to revise is small. We identified page tables, thread control blocks, and the entries of the mapping database.

3.3.1 Page Tables

Page tables are potential candidates for replication. Write accesses are rare compared to read accesses and normally only affect a small part of the table (i.e. a few entries). Dynamic locks as suggested by Uhlig [36] could be used to serialize accesses to page tables, with the lock granularity being adjusted depending on the access pattern of a page table (which depends on the number of nodes on which the address space is active). Additionally, L4 already uses different page directories for each CPU to allow for CPU-local kernel data, as shown in Figure 3.6: Most entries of the per-CPU page directories point to the same, shared subtables (depicted in green), but some entries point to CPU-local subtables (depicted in red). The virtual addresses that are occupied by these “CPU-local” entries are mapped to different physical addresses on every CPU. This allows for CPU-local data without having to use indirection. L4 uses this concept for some statically defined in-kernel data, e.g. the thread control block of the idle thread. However, the concept currently does not support dynamic allocation of CPU-local memory, nor is it exported to user-level applications. Yet, it could be further enhanced in that the entire page table hierarchy is replicated instead of only the CPU-local tables. In that case, each page table hierarchy can be placed in node memory of the node the page tables belong to, thus making all page table accesses local. This leads to the problems we described in Section 3.2.2: Synchronization of page tables must be performed across node boundaries, requiring a synchronization primitive in the kernel that also depends on remote memory latencies. However, as described in Section 3.2.3, we use per-node address spaces instead of a single, node-spanning address space for each application. Therefore, we decided to optimize our solution for that case and not for the case of address spaces spanning across multiple nodes. Each address space is assigned a ”home node“, on which all shared page tables are allocated on, while “CPU-local” tables are allocated on the node of the corresponding CPU. With this model, an address space can still become active on every node (and on different

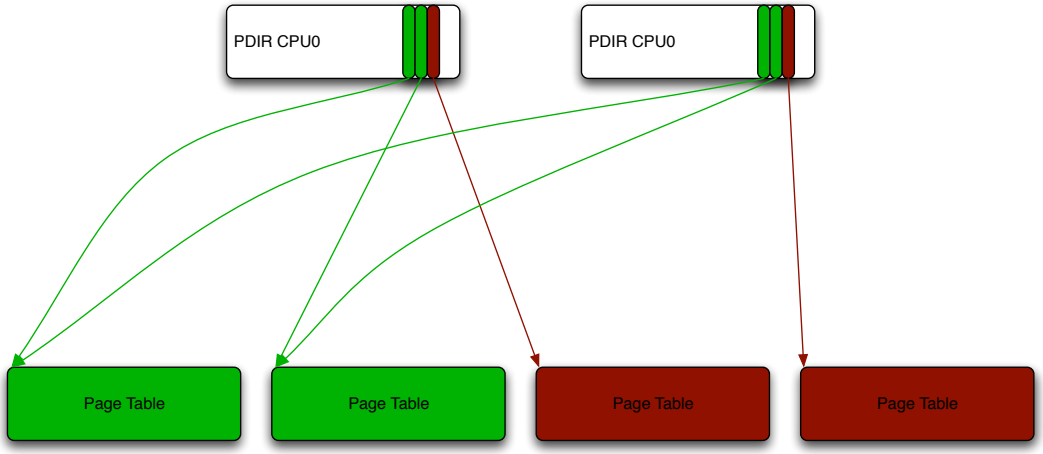


Figure 3.6: Exemplary page table hierarchy for a system with two CPUs. The green entries in both page directories point to the same page tables and thus to non-CPU-local data. The red entries point to different page tables. The corresponding virtual addresses can thus be mapped to different physical addresses on each CPU.

nodes simultaneously), but it is favorable that it is active only on its home node. In that case, the entire page table hierarchy will be placed in node-local memory, and no in-kernel synchronization of page tables must be performed across node boundaries. It us up to the user-level to ensure that address spaces are active on exactly one node.

3.3.2 Thread Control Blocks

A thread's TCB is used to store relevant information for that thread, including its state and its kernel stack. As a thread is always assigned to a single CPU (and thus to a single node), it seems to be a good idea to always keep a thread's TCB on the thread's node and migrate it whenever the corresponding thread is migrated. However, a TCB is not only accessed by its own thread, but also by other threads. For example, if a thread wants to send an IPC to another thread, it has to check the destination thread's state and thus its TCB. To avoid remote accesses in that case, every TCB would have to be replicated to every node's memory. We decided not to do this, but to migrate TCBs instead for two reasons: First, we consider the overhead for synchronization of the replicas too high - a thread's TCB is written on every IPC operation. Second, we favor IPC between threads on the same CPU over cross-CPU IPC, as the latter requires inter-processor interrupts and thus is more costly. If both threads run on the same CPU, accesses to the sender's and receiver's TCB are both local accesses. In case of a cross-CPU IPC, the additional penalty caused be accessing a TCB in remote memory is not significant compared to the overall costs for a cross-CPU IPC.

L4 currently addresses TCBs via a virtual linear array with a fixed size. A thread's ID is used to calculate the virtual address of its TCB. No memory is mapped into the virtual array initially. On a read-fault, a zero-filled "dummy TCB" is mapped in. On a write fault, memory for a new TCB is allocated and mapped at

the desired location. Figure 3.7 shows an example. Theoretically, migration of a

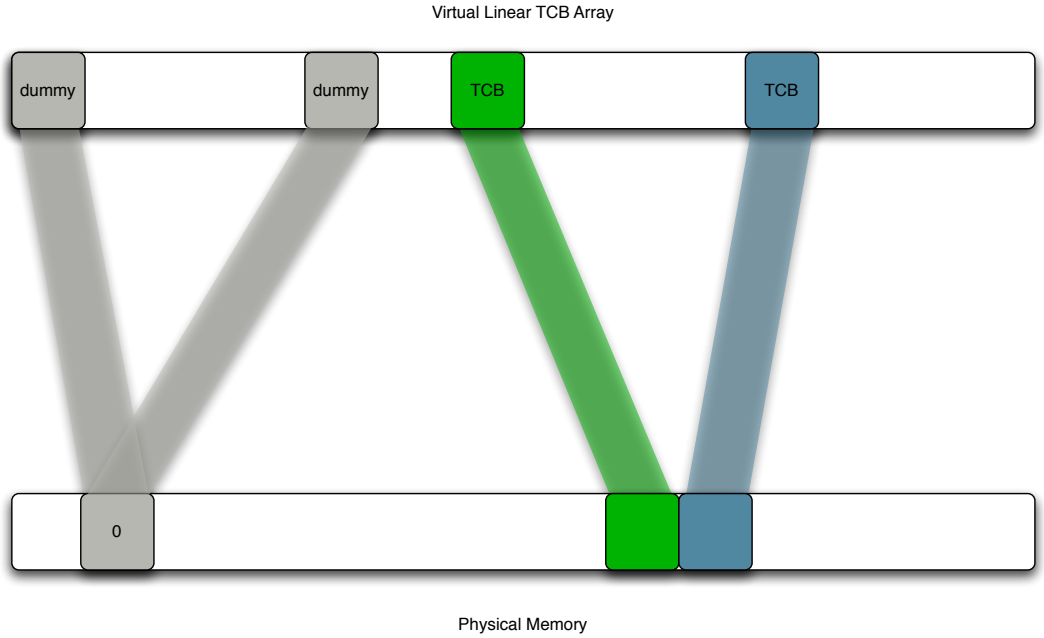


Figure 3.7: Virtual linear TCB array. The two dummy TCBs (gray) are both mapped to a zero-filled page. "Real" TCBs (green and blue) are mapped to different pages of physical memory, allocated on demand. In this picture, the mapping granularity is fine enough to allow for mapping on a per-TCB basis.

TCB is straightforward: Allocating a new physical page for it, copying the old page to the new page, and finally changing the mapping to the new physical address. Furthermore, the affected entries must be flushed from all CPUs' TLBs. However, a problem arises when more than one TCB occupies a single page (which is the case for the IA32 kernel): As mappings can only be changed on a per-page basis, all TCBs on a page have to be migrated. However, it cannot be ensured that the corresponding threads run on the same node. By migrating one TCB from remote to local memory, another TCB lying in the same page might be migrated away from its "home node". A more fine-grained solution is required, which allows for migration of one TCB without side effects for other TCBs. One approach is to expand the size of a TCB so that only a single TCB occupies a page. This either requires to reduce the maximum number of allowed threads or to increase the size of the virtual TCB array. The former approach might lead to a system running out of thread IDs too early, the latter approach is a problem on 32 Bit architectures, where the kernel area currently is limited to a size of 1 GB. Furthermore, also physical memory is wasted because it is questionable if there are any benefits from larger TCBs, even though the additional space could be used for the thread's kernel stack. L4 does not perform any recursion in the kernel, and in-kernel function call chains are short, thus there is no need for larger kernel stacks. Additionally, larger TCBs lead to a higher occupation of the TLB, leading to a higher TLB-footprint of the kernel, with negative effects on the overall system performance, at least when the number of existing threads is high.

Another approach is to discard the scheme of direct addressed TCBs and to use indirect addressing instead. A thread's ID can be used to index into an array of pointers to TCBs. When migrating a TCB, new memory is allocated from the destination node's memory pool, the TCB is copied, and the pointer is changed to point to the new location. The memory on the source node can then be freed. This approach introduces an additional level of indirection, but provides the desired flexibility without limiting the number of threads or wasting both virtual and physical memory. There are two problems one has to take care of with this approach: First, when a thread is going to be migrated, it must be ensured that its TCB is not accessed by any other thread (running on a different CPU) in this moment. Read-Copy Update (RCU) [28–30] can be used to ensure that no other threads are currently accessing the TCB. Second, there might be threads which are currently not active but still have direct references to their own or another TCB on their kernel stack. As threads can be sleeping for an arbitrary amount of time, one cannot make any assumptions if and how long such references exist. Finding and updating all existing old references to a TCB when this TCB is migrated is not possible, as those references are stored on a thread's kernel stack, the layout of which is not known. Instead, a thread has to reload all TCB references it keeps when it is reactivated. This requires to identify all points within the kernel code at which stale TCB pointers might be accessed, which is not a trivial task. The pointer array itself is a candidate for replication, as write accesses are only required in case of a thread migration. Additionally, as a thread migration is always performed by the source CPU of the to-be-migrated thread, concurrent modifications of the same TCB pointer can not occur. Yet, it must be ensured that a TCB pointer is not accessed and dereferenced while migration is in progress.
Mainly due to time constraints, we decided to favor the first approach for our implementation.

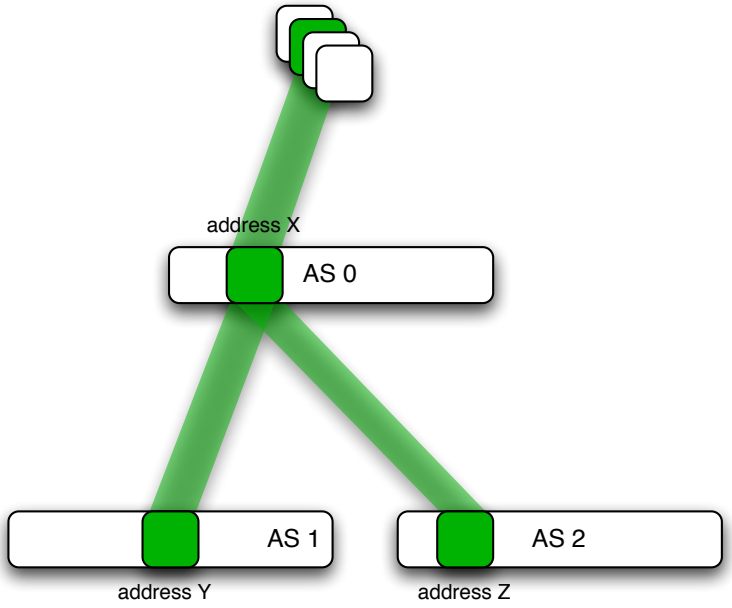
User-Level Thread Control Blocks

Parts of a thread's TCB are mapped into the user area instead of the kernel area. These so-called UTCBs contain information that can be exposed to user applications without affecting security. In particular, a thread's message and buffer registers are part of its UTCB. As for kernel TCBs, there also arises the question where to place a thread's UTCB, and how to migrate it when the thread migrates. In contrast to TCBs, however, increasing the size of a UTCB to allow for migration of single UTCBs is even a bigger problem: For example, on IA32, a UTCB is only 512 bytes in size. Increasing the size to 4 kilobytes (i.e. eight times the size it actually requires) is not reasonable. On the other hand, changing the addressing scheme of UTCBs is even more complicated as it is for TCBs: The address of a UTCB is used as the thread's local ID (i.e. an ID that is only unique within its address space), and the kernel API specification ensures that the location of a thread's UTCB will not change during the lifetime of that thread [21]. If indirect addressing was used, this assertion would no longer hold true, causing problems with applications that might rely on that assumption.
However, the design decisions we made for the organization of address spaces, i.e. using different address spaces on each node, also offer a simple solution to this

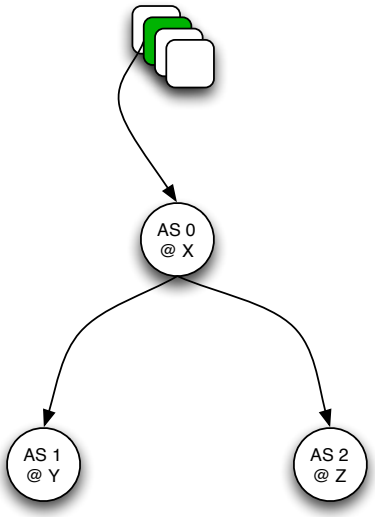
problem, allowing for migration of UTCBs without indirection and without increasing their size. The reason why this is possible is that all threads within an address space run on the same node. Therefore, even if numerous UTCBs occupy a single hardware page, all of them can be backed with memory from the local node. If a thread is going to be migrated to another node, this implies migrating it into another address space. Only two conditions must hold true to allow for migration of a thread into another space: First, there must not exist a thread with the same local thread ID (i.e. UTCB location) in the target address space. Second, the UTCB area in the target address space must occupy the same region as it does in the source address space, so that the migrated thread's UTCB can be placed at the same address as it has been before.

3.3.3 Mapping Database

The kernel's mapping database (MDB) keeps track for each page of physical memory to which address spaces and at which virtual addresses it is mapped. Figure 3.8 shows an example: A physical page (green) is hierarchically mapped into three different address spaces, as seen in Figure 3.8(a). Figure 3.8(b) shows the corresponding tree of map nodes. We decided to allocate memory for map nodes from the home node of the corresponding space, i.e. the space to which the page is mapped. This is an optimization tailored to the design we described in Section 3.2.3, with different address spaces per node. The most frequent reason for accessing the mapping database is a pager mapping memory into or unmapping memory from an address space it pages. With our design, a pager only pages threads/address spaces that run on the same node as the pager itself. Therefore, both the mapnode that describes a physical page within the pager's address space and the mapnode that describes the page within the application's address space are node-local.



(a) Address space structure



(b) Corresponding MDB tree

Figure 3.8: Hierarchy of address spaces and corresponding mapping database tree.

CHAPTER 4

Implementation

The design we presented in the last chapter affects both kernel objects and the user-level architecture. We identified TCBs, page tables, and the mapping database as the most important kernel objects. We decided to migrate TCBs whenever the corresponding thread is migrated to another node. Page tables are allocated on the home node of the corresponding address space, i.e. the node on which the address space is created. The map nodes of the mapping database are allocated on the home node of the corresponding address space. In this way, accesses to the mapping database for mappings between two address spaces on the same node access mostly local memory. Our user-level architecture requires each service to be replicated per CPU (i.e. each service must offer at least one thread per CPU), so that concurrent requests can be handled in parallel. Additionally, instead of using a single address space for an application, we propose to use one address space per node. This allows for an efficient, kernel-transparent replication of page tables, where synchronization is performed entirely at user-level. Replicated page tables do not only improve locality, but also allow for different virtual-to-physical mappings of the same virtual addresses on every node, which is a fundamental requirement for any form of dynamic page replication (e.g. for code replication). This chapter describes our implementation, which closely follows the proposed design. We start with the changes that we made to the L4 microkernel and then detail the implementation of our user-level services.

4.1 Kernel Objects

This section describes the necessary changes to kernel objects. The basis of all further changes to the kernel is a modified version of the kernel memory allocator, which we describe first. Our modifications allow for memory allocation from a specific node. We then detail our changes to the implementation of address spaces, the replication of kernel code, and finally our implementation of TCB migration.

4.1.1 A NUMA-Aware Kernel Memory Allocator

The basis of making kernel objects such as page tables or TCBs node-local (i.e. by replicating or migrating them) is the possibility to allocate memory on a specific node. Therefore, we had to modify the kernel memory allocator so that one can specify from which node memory shall be allocated. L4's non-NUMA-aware kernel memory allocator works as follows: On system startup, Kickstart (that is L4's boot strapper) parses the BIOS memory map to find free physical memory. A configurable amount of that memory is marked as reserved for the kernel memory allocator. The information where this memory is located is passed to the kernel via the kernel interface page (KIP). L4 evaluates the informations in the KIP, remaps the memory to the upper end of the kernel area, and donates the reserved memory to the kernel memory allocator, which handles memory requests from that memory pool. A static, fixed offset is used for remapping the memory pool, which allows for an easy translation between physical and virtual addresses, as the following holds true for all memory of the pool:

$$\text{addr}_p = \text{addr}_v - \text{offset}$$

That is, an arbitrary virtual address from the pool can be translated to the corresponding physical address by subtracting the fixed offset, and vice versa. With a single fixed offset however, it is only possible to map one contiguous block of physical memory into the kernel area. For a NUMA system, one pool per node is required, so using only a single block of physical memory for the kernel memory allocator is not sufficient. Therefore, we made some changes to Kickstart as well as to the kernel memory allocator itself. Instead of setting up a single pool of physical memory, Kickstart now creates one pool per node, each of which can consist of a contiguous block of free physical memory at an arbitrary location. Thus, the memory can no longer be remapped with a fixed offset, instead, the offset differs from pool to pool. When translating a virtual address to its corresponding physical address and vice versa, it must be checked in which pool the address falls to be able to choose the correct offset. This makes translations between physical and virtual addresses more expensive: Given a memory address, a minimum of two comparisons is required to find out in which pool the address falls, and an additional memory access is needed to load the corresponding offset. This solution is still cheaper than walking page tables (which could only be used for virtual-to-physical translations anyway). Figure 4.1 shows an example for a system with four nodes. The reason for having the memory of the first node (node 0) at the uppermost virtual address is only simplification of the implementation: The uppermost address to which a memory pool can be mapped is statically defined, and the first pool the kernel finds in the KIP (i.e. the pool of node 0) is mapped directly below that address. Each following pool is then mapped below the previous one.

The memory pools allocated by Kickstart are passed to the kernel via the KIP, and then donated to the kernel memory allocator, which keeps a free list for each of the pools. Each list is protected with its own lock so that requests to different pools can be handled in parallel. We extended the *kmem_alloc* function with an additional parameter to specify the node to get the memory from. The *free* function does not need such an additional parameter, instead, it checks in which node pool the virtual address to be freed falls.

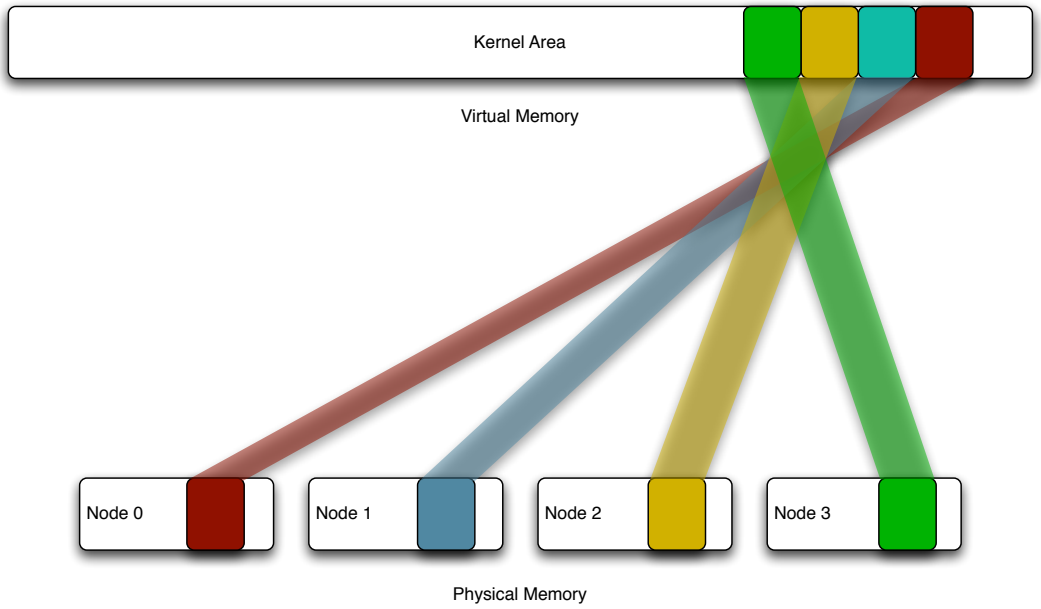


Figure 4.1: The NUMA-aware kernel memory allocator: For each node, a pool of physical memory is mapped into the kernel area of each virtual address space.

4.1.2 Address Spaces

The *space_t* Object

L4's *space_t* object on IA32 used to represent an address space is basically the page directory itself, plus a so-called shadow table of another 4k, used to store pointers to the corresponding mapping database entries. Additional information, like the number of threads in that address space, is stored inside some page directory entries marked as invalid. On SMP systems, L4's implementation for CPU-local data requires a separate page directory per CPU. In that case, the *space_t* object is expanded to hold the additional page directories, which lie contiguously in memory. Thus, it includes a page directory per CPU, no matter if the corresponding space ever gets activated on all CPUs. Additionally, modifications in one of the CPU-local page directories (e.g. a newly established mapping) are eagerly propagated to all other CPU-local page directories. It is obvious that this implementation does not scale well with an increasing number of CPUs, as the size of the *space_t* object increases linearly with the number of CPUs the kernel supports. We therefore adapted a recent implementation of a new *space_t* object [36]. Instead of having the CPU-local page directories allocated contiguously and all at once (namely upon address space creation), we keep an array of pointers to these page directories, which allows for arbitrary allocation and placement. Furthermore, all additional information that was stored in invalidated page directory entries is moved out and now stored on a separate page, together with the pointers to the per-CPU page directories. Figure 4.2 shows the old *space_t* object, Figure 4.3 the new one. A page directory for a specific CPU is allocated when the corresponding space becomes active on that very CPU for the first time. Memory is taken from the node to which the CPU belongs. The newly allocated page directory is initialized by copying all

entries for user mappings from a reference page directory belonging to the same address space (i.e. the page directory that was allocated upon address space creation), while the kernel mappings are copied from the kernel address space's page directory from the corresponding CPU (the kernel address space's per-CPU page directories are not allocated on demand, but eagerly when it is created). When a thread is migrated to another CPU for which no page directory exists yet, we do not allocate it immediately, but set a bit in the thread's resource bitmap instead. Resources of a thread are loaded whenever a switch to this thread is performed. While allocation could also be performed whenever a thread is going to be migrated, the advantage of using a resource bit is that allocation is always performed on the CPU the thread is going to run on next. Thus, no remote memory must be touched. Additionally, if a thread is migrated but never becomes active for some reason, no unnecessary allocations are performed. This solution fits well to our user-level

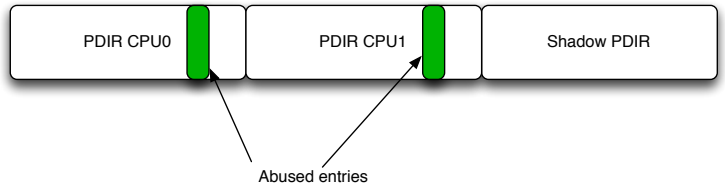


Figure 4.2: L4's old *space_t* object. Page directories for each CPU are allocated contiguously and all at once upon address space creation. Some entries are marked as invalid and used to store additional information.

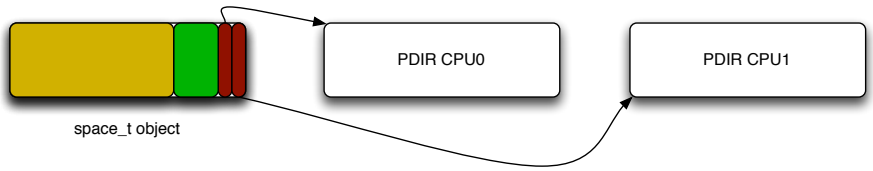


Figure 4.3: The new *space_t* object. A 4k page is used to store mapnode pointers for superpages (yellow), additional information for each space (green), and pointers to the page directories for each CPU (red). Page directories do not longer need to lie contiguously, and can be allocated on demand.

architecture: As long as an address space is active only on a single CPU, the kernel does not have to allocate additional per-CPU page directories or subtables. Memory consumption is drastically reduced compared to the old solution, and there is also no need to perform synchronization between per-CPU page table hierarchies.

Address Space Creation

When a new address space is created, its "home node" is set to the node on which the corresponding system call was performed, and memory for the *space_t* object as well as for all page tables (except those for CPU-local data) is allocated on the home node. It is not necessary, but desirable that an address space in only active on its home node. It is not possible to create an address space on a remote node, nor is it possible to change the home node of an address space once it has been created. This approach makes address space creation on a specific node more complicated, as one must ensure that the address space is created by another thread already running on that node. The alternative would have been to modify L4's API to allow for specifying a home node when a new address space is created. However, this would have introduced a special case to the API that is only relevant on NUMA systems, which we wanted to avoid.

4.1.3 Replication of Kernel Code

We use L4's already existing implementation of CPU-local data to also allow for replication of kernel code. Initially, L4's binary is placed in memory of node 0. When the kernel space (including CPU-local mappings) is initialized, we copy the code pages to every node's memory pool and map the corresponding virtual addresses to the replicated code on every node. The code section must be aligned at the hardware page size (i.e. 4k on IA32) so that it is ensured that code replication does not affect other objects within the address space, e.g. writable global variables. As L4 does not contain self-modifying code, no synchronization between the code replicas is necessary.

4.1.4 TCB Migration

To be able to migrate TCBs and retain the scheme of direct addressed TCBs (via a virtual linear array), the TCB size must be increased to equal the minimal hardware page size. We therefore doubled the size from 2k to 4k on the IA32 kernel. We did not change the overall size of the virtual linear array, thus halving the maximum number of threads in the system. The migration itself is straightforward: When a thread is migrated, we check if the destination CPU is on the same or on another node. In the former case, no TCB migration has to be performed. In the latter, we allocate a page from the destination node's memory pool. We then lock the TCB, copy its contents to the new location and finally adjust the mapping so that the TCB's virtual address is mapped to the new physical location. Before the lock is released, we must ensure that the old mapping is no longer cached in any CPU's TLB. We therefore flush the mapping (a single TLB entry) from the local CPU and invoke a remote handler on all other CPUs that flushes the corresponding entry of the remote CPUs' TLBs. The originating CPU waits for a reply from all other CPUs before it proceeds, ensuring that all TLBs are consistent when execution resumes. Thread migration is always performed by the CPU to which the thread belongs, thus ensuring that the thread that shall be migrated is currently not running (and consequently not accessing its TCB).

4.2 User-Level Services

In this section, we describe the implementation of basic operating system services of our NUMA operating system. We construct a two-level pager hierarchy on top of L4's root pager, Sigma0. Additionally, we implement a system call server that allows non-privileged threads to execute system calls. We also present the interface which each of these services has to offer, defined in IDL^4 [17]. Particular attention is paid to the mechanisms of synchronization between pagers, which are elementary to preserve applications the view of a single, node-spanning address space, although multiple address spaces are used. First, we clarify some terms and definitions we use in the following sections: When we talk of a *task*, we mean all threads and address spaces belonging to the same application. This definition differs from common definitions insofar as a task normally consists only of a single address space. To the collectivity of all address spaces that belong to a task we refer to as the *task address space*. In other words: A task address space consists of one or more (L4) address spaces, depending on the number of nodes on which the task is active. A task is *active* on a specific node or CPU when at least one thread of the task resides on that node/CPU (this definition does not require that the thread is actually running).

4.2.1 User-Level Pagers

L4's address spaces are constructed recursively. Initially, all free physical memory is owned by Sigma0, which is the root pager of all address spaces. Arbitrary address spaces and pager hierarchies can be constructed on top by mapping pages from Sigma0 into other address spaces (and from there again to other address spaces, and so forth).

As stated in Chapter 3, our design of user-level services requires one pager thread on each CPU. These pager threads then manage the address spaces of applications running on top. It is their responsibility to perform necessary synchronization between address spaces belonging to the same application. For the reasons of more flexible node-local data (e.g. to allow for code replication) and for node-local page table hierarchies, it is desirable that our application pagers run in different address spaces, too. To be able to handle page faults on global mappings originally established by another pager, each pager must be able to hand out all physical memory, not only physical memory from the node to which it belongs. As we described in 2.1.1, Sigma0 hands out every page of memory only once. It is thus not possible to map all physical memory into every application pager's address space by directly requesting the memory from Sigma0. Instead, we introduce an additional root pager, leading to the pager hierarchy depicted in Figure 4.4. The root pager acquires one memory pool from each node from Sigma0. These pools are mapped onward into every application pager's address space. With this design, the amount of physical memory supported is limited by the size of the application pagers' virtual address spaces. On IA32, virtual address spaces are 4 GB in size, of which one GB is used by L4. Therefore, our solution cannot handle more than 3 GB of physical memory. With the trend towards 64 bit architectures, we consider this not to be a serious problem. Alternatively, data spaces [5] can be used to over-

come limitations caused by too small virtual address spaces and to offer a more powerful virtual memory abstraction.

To ease synchronization between application pagers, the address space layout is the same for every application pager, i.e. the node pools are mapped to the same virtual addresses in every application pager's address space. The root pager is not only responsible for handing out node memory to all application pagers, but can additionally back the application pager's code with memory from different node pools, allowing the application pager's code to be node-local. Yet, the root pager itself is running in a single address space, thus using only a single page table hierarchy, and lacking the possibility to replicate its code. In contrast to the application pagers, the root pager is mainly required on system startup. Before starting any applications, the application pagers request all their memory from the root pager. Performance penalties caused by accesses to remote memory are negligible in this startup phase. Once initialized, the application pagers will suffer no more page faults in their own address space.

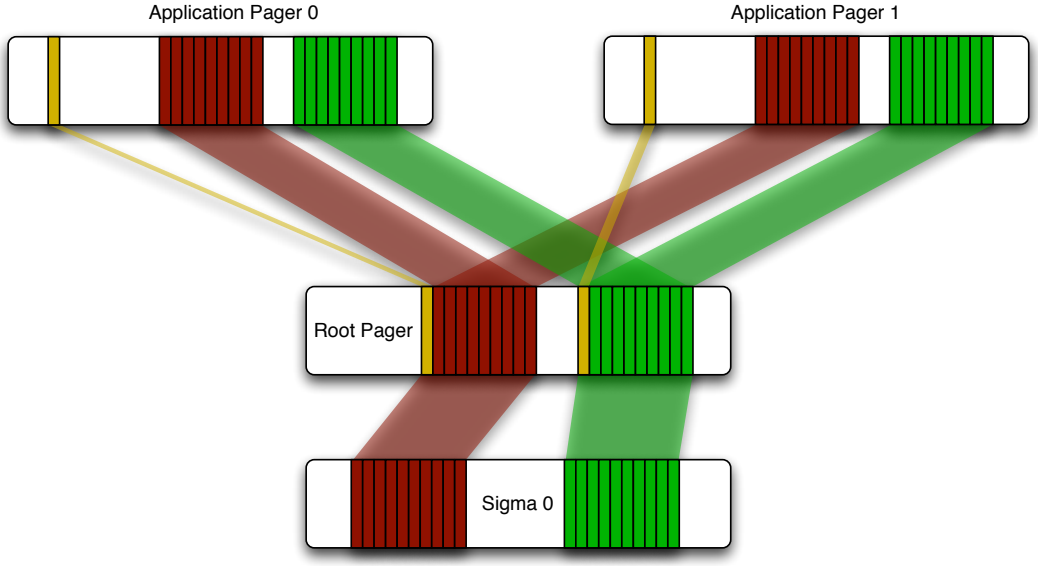


Figure 4.4: The pager hierarchy of our operating system. Example with two nodes. The application pagers' code (yellow) is replicated. The two node pools (red and green) are mapped into the address spaces of both application pagers.

The Root Pager

In addition to the kernel itself, L4 initially starts two system servers: Sigma0 and a roottask. Our root pager is part of the roottask. On startup, its master thread spawns additional worker threads and places one on every CPU in the system. The master thread then acquires a pool of memory from each node from Sigma0 and maps these pools to predefined locations within its own address space. Additionally, it acquires the memory in which the application pager module resides, ELF-loads the binary and places a copy of it in each node pool within its own address space. Afterwards, the master thread (which is set as pager of the other

worker threads) starts each of the worker threads by sending a message, as defined by the L4 API [21]. Each of the worker threads (as well as the master thread) then create an address space and a thread for the application pager on their node. It is important that these address spaces are created on the node they will reside on, so that the home node of each space is set correctly, and kernel page tables are allocated in local memory. Finally, each worker thread (including the master) sends a startup IPC to the corresponding application pager thread and enters a pager loop do be able to handle page faults of the application pagers.

Root Pager API The root pager offers only a simple interface to the application pagers:

```
interface pager
{
    [kernelmsg(idl4::pagefault)]
    void pagefault(in long addr, in long uip,
                   in long access, out fpage fp);
};
```

That is, it does nothing else but to handle page faults from the application pagers.

Application Pagers

As stated before, the application pagers set up their address spaces before any applications are started. Because the root pager offers only an interface to handle page faults, this is done by touching all regions within the virtual address space. The layout of each application pager's address space is statically defined, with fixed-sized regions for the node pools, thread stacks, and the heap. When all regions are touched, it is ensured that no further page faults will occur on the application pagers' virtual memory. In our system, application pagers are also used to create and start threads and to migrated threads between CPUs/nodes. Application startup is described in Section 4.2.3.

Application Pager API Due to the additional duties of the application pagers, i.e. managing thread creation and thread migration, their interface is more complex than the root pager's interface:

```
interface app_pager
{
    [kernelmsg(idl4::pagefault)]
    void pagefault(in long addr, in long uip,
                   in long access, out fpage fp);

    void make_local(in long start, in long size);

    void create_thread(in long ip, in long sp,
                       out L4_ThreadId_t new_thread);
```

```
    void delete_thread(in L4_ThreadId_t thread_id);

    void migrate_thread(in L4_ThreadId_t thread,
                        in long dest_cpu);

    void unmap( inout L4_Fpage_t fpage1,
                ...,
                inout L4_Fpage_t fpage63 );
};
```

The *make_local* function allows a task to explicitly mark a region of its task address space as node-local. All page faults that will occur in this region are handled by establishing a local mapping. That is, the pager that handles the page fault will map in memory from its own pool, and the newly established mapping is not inserted into the master pager's data structures. *create_thread* allows an application to create a new thread on the current CPU, which belongs to the same task as the caller. The function chooses a unique ID for the thread, inserts the thread into the data structures of all pagers on which the corresponding task is active and invokes the system call server (see 4.2.2) thread on the same CPU to finally create the new thread. The thread ID is returned to the caller.
delete_thread is used to delete a thread. Deletion implies that the thread is removed eagerly from the task structures of all pagers on which the corresponding task is active. Otherwise, a pager might come to an incorrect migration decision, i.e. trying to migrate a thread into a target address space that is no longer existent.
migrate_thread is invoked when a thread shall be migrated to another CPU. The pager ensures that threads can only be migrated by either other threads belonging to the same task or by a designated load balancer.
unmap[1] is a function used for synchronization between pagers. A pager revoking a global mapping must call all other pagers' *unmap* function. The function takes up to 63 parameters, each specifying an fpage of "physical"[2] memory that shall be unmapped (each L4 thread has 64 message registers, but one of these registers is required for the message tag). If the limited number of parameters turns out to be not sufficient, string IPC can be used to pass larger numbers of fpages, but with a higher overhead for the message itself.
The called pager can then update its own data structures and call L4's *unmap* system call with the corresponding fpages and then return the result (i.e. the fpages as returned by L4's *unmap*) to the calling pager. It is important to note that the parameters of our *unmap* function are not map items, i.e. they are ignored by the kernel. Therefore, *L4_Fpage_t* is used as data type instead of the *fpage* type, which would lead to the pages being handled as map items. For security reasons, only pagers must be permitted to call another pager's *unmap* function.

Required Data Structures The application pagers have to keep track of three kinds of information: First, they must know which threads and address spaces belong

[1] This function definition is given in pseudocode and is meant that one can specify up to 63 fpages to unmap. As IDL[4] doesn't support function overloading, a more complex interface with an appropriate C++ wrapper must be used "in reality".

[2] In this context, physical memory is actually virtual memory from the application pagers' own address spaces.

together, i.e. form a task. This knowledge is required both for the synchronization of mappings and for migration of threads between CPUs. Second, the application pagers must also know about the layout of each address space they page, i.e. which physical memory is mapped to which virtual addresses. Third, they must keep a list of free physical memory so that they can allocate new memory when necessary. To describe a task, we propose the following object:

```
class task_t
{
    L4_Word_t task_id;
    pdir_t * pdir;
    thread_t * thread_list[NUMBER_OF_CPUS];
};
```

The *task_id* is used to uniquely identify tasks. *pdir* is a pointer to the pager's management information of the task's address space, which we describe below in more detail. The *thread_list* array is an array of pointers, each pointing to a list of thread objects (which are also described below). There is one thread list per CPU, holding objects for all threads that run on this CPU. With this information, it is easy for a pager to find out if a task already has threads (and thus an address space) active on a specific CPU.
To describe threads, we use the following thread object:

```
class thread_t
{
    L4_ThreadId_t thread_id;
    task_t * task;
    thread_t * next;
    thread_t * prev;
};
```

thread_id is the global ID of the thread, under which it is also known to L4. *task* is a pointer to the task object of the task to which the thread belongs. *next* and *prev* are used to be able to form lists of thread objects, as required to describe tasks. With this structure, a pager can easily find all threads of a task: If a function provided by the pager's API is called, the pager needs to find the corresponding thread object. With that, it can retrieve the task object of the task to which the thread belongs, and from there all other threads, and thus address spaces. To quickly retrieve a thread object, given a thread ID, we propose a linear array of thread objects, with the position of a thread object derived from its thread ID.
To be able to keep track of the layout of address spaces, we decided to implement two-level page tables similar to those that are used by the IA32 hardware.
To keep track of physical memory, a simple free list can be used, with functions to allocate and free memory.
Additionally, each pager must keep track of which physical memory it mapped to which task(s). This knowledge is important in case of an unmap: If a physical page must be unmapped, all affected page table entries must be invalidated so that the pager does not reestablish the old mapping when receiving a page fault on a page that was previously unmapped. A list-based structure per physical page can be

used, which stores the task ID and the virtual address within the task's address space to which the physical page is mapped.

Synchronization of Task Management Information

Pagers only need to be aware of those tasks that have threads residing on their CPU/node. However, a pager requires a complete view of a task, i.e. it must know about all its threads and on which nodes the threads reside. Therefore, no synchronization has to be performed when a new task is created on a node. Yet, thread creation, deletion, and migration require synchronization between the pagers of those nodes on which the task is active. The pager that invokes such an action (e.g. a thread migration) not only updates its own data structures, but also all data structures of other pagers on which the same task is active. We propose a lock-based approach for this, i.e. the originating pager directly modifies the other pagers' data structures in remote memory.

Synchronization of Free Lists

Application pagers must keep track of which physical memory is free and which is in use. If a page is allocated by one pager, the other pagers must not hand out the same page for another purpose. To avoid synchronization of free lists between pagers, we decided to use a different approach: Pagers are only allowed to allocate new memory from their own node, and must therefore only keep track of free and used memory within their own pool. In case of multiple CPUs (and thus pager threads) per node, it is either possible to partition each node pool further into CPU pools, or to share free lists between pagers of the same node. This solution leads to a *first touch* allocation policy: Global regions of a task address space are always backed by memory from the node on which they are accessed for the first time.

Synchronization of User-Level Page Management Structures

If a new, global mapping is established, removed, or modified by one pager, the other pagers must be made aware of this. In case of a newly established mapping or when access rights are widened, this synchronization can be performed lazily. As we stated in Section 3.2.3, we define a master pager and keep its data structures always up-to-date. The master pager can thus be queried to find out if a valid mapping exists if a page fault on a global mapping is handled. Due to the low remote-to-local latency ratio of our test system, we decided to perform lock-based synchronization instead of message-based synchronization between pagers, i.e. the pagers can directly modify each others data structures via shared memory. If a pager receives a page fault message at a virtual address that shall be global (which is the default), it acquires a lock on the master pager's page table and checks it to find out if there already exists a valid mapping. If there is one, it inserts the mapping into its own data structures, releases the lock and answers with the physical page of memory that is expected at this location. If no mapping exists yet, the pager allocates a new page of memory from its own node pool and inserts the corresponding mapping both in its own page table hierarchy and in the master pager's page table hierarchy (the lock must be held while this is done). After

that, the lock can be released and the newly allocated page can be mapped to the application. Locking is required to avoid race conditions, that can occur otherwise when multiple pagers have to handle a page fault at the same virtual address of the same task simultaneously. To avoid the master pager becoming a bottleneck, different master pagers can be used for each task, for example by assigning the pager of the CPU on which the task was created as master pager, or by assigning pagers in a round robin fashion to newly created tasks. Additionally, instead of using a simple spin lock, more sophisticated locking techniques like MCS locks [31] can be used, which allow each processor to spin on local memory and can thus help to reduce cache coherence overhead and remote accesses.

In case of an unmap (including restriction of access rights), synchronization must be performed eagerly. We therefore decided to implement an IPC-based protocol. If a pager unmaps a globally mapped page from a task, it must notify all other pagers by calling the other pagers' *unmap* function with appropriate parameters. The called pagers can then invalidate the mapping within their own page tables (so that they will query the master pager when the receive a page fault on the corresponding virtual address the next time), update their mapping information and call L4's *unmap* by themselves. The originating pager must not resume execution until all other pagers have completed the unmap. An unmap should be initiated by the pager to the pool of which the page of memory that shall be unmapped belongs. As we stated above, a pager never hands out memory from another but its own node, except for already existing global mappings. Therefore, two conditions hold true for pages that belong to a pager's own pool: Firstly, there exist no local mappings of this page established by other pagers. When the originating pager itself unmaps the page, all existing local mappings are removed. Secondly, all global mappings of that physical page were originally established by the pager to the pool of which the page belongs. This means that the pager is aware of all tasks to which the page is mapped. It is thus sufficient to only call the *unmap* function of those remote pagers where such a task is active. It is not necessary to query the master pager or any other pager in the system. The called pagers must return the status bits (i.e. *accessed* and *dirty* in case of IA32) for each page they unmapped to the calling pager. With this information, the calling pager can find out if a page that was unmapped was modified on any node. This knowledge is important e.g. if a page shall be swapped out: There is no need to write the contents of a physical page back to the disk if there already exists an older copy and the page was not modified since it was swapped in for the last time.

If each application pager consists only of a single thread that handles all incoming requests, there exists the risk that the IPC-based unmap will deadlock. Consider a system with two application pagers that try to unmap a page of their own pool simultaneously. Each of these two pagers will first unmap the page locally and then try to call the other pager's *unmap* function. As each pager now waits for the remote call to complete, none of them is able to handle the pending call issued by the other. If no IPC timeout is specified, both pagers will be blocked forever. Even with timeouts, it is difficult to find a solution that guarantees successful completion of all unmaps and predictable time to completion. A better solution is to use an additional worker thread per application pager that is responsible for nothing but handling remote unmaps. The pager that triggers the unmap (i.e. the one that

owns the memory that shall be unmapped) calls all other pagers' worker thread to perform the unmap, while it uses its "master" thread to issue these calls. Therefore, the originating pager's worker thread is still able to handle unmap requests triggered by other application pagers. This additional parallelism within each application pager requires additional locking of pager-local data structures if they are possibly accessed by both threads. For example, the worker thread has to modify page tables to invalidate mappings, while the master thread also must modify page tables if it handles a page fault caused by a client.
A separate thread for handling incoming unmap requests issued by other pagers not only avoids deadlocks, but can also help to reduce latency and thus improve performance of an unmap.

4.2.2 System Call Servers

In L4, some system calls can only be executed by privileged threads, including both *SpaceControl* and *ThreadControl*, which are required to create new threads and address spaces. Privileged threads are all threads that run in the same address space as the initial privileged threads, i.e. Sigma0 and the roottask. If other threads shall be able to perform these system calls, a system call server must be implemented that performs the calls on behalf of the calling thread. In addition to the root pager threads we create in the roottask's address space, we therefore also spawn a system call thread per CPU. These threads offer the following API to applications:

```
interface syscall
{
    void ThreadControl( in L4_ThreadId_t dest,
                        in L4_ThreadId_t space,
                        in L4_ThreadId_t scheduler,
                        in L4_ThreadId_t pager,
                        in void* UtcbLocation );

    void SpaceControl( in L4_ThreadId_t space_spec,
                       in L4_Word_t control,
                       in L4_Fpage_t kip,
                       in L4_Fpage_t utcb_area,
                       in L4_ThreadId_t redirector );
};
```

The implementation of these functions does nothing but calling the corresponding L4 system calls with the accordant parameters. Of course, additional security policies are possible, like restricting the number of threads that are allowed to perform these system calls. As the system call server threads run in the roottask's address space, neither their code nor the page tables can be replicated.

4.2.3 Application Startup

For the sake of simplicity, our system provides only a minimal set of services. In a "real" system, one would also have a task server, which is responsible for

the creation and management of new tasks, i.e. threads and the corresponding address spaces. In our system, task creation is also part of the pagers' work. When a new task is going to be created, the pager that creates a task loads the task's binary, creates a new thread and a new address space on its own node, and sends a startup message to the application's thread. If the application wants to create more threads, it can call its pager, which in turn calls the roottask (as only privileged threads can create other threads). The application must not bypass its pager (i.e. call the system call server directly), because the pager must be made aware of the new thread. Each newly created thread is placed on the CPU on which it was created, and can be migrated afterwards.

4.2.4 Thread Migration

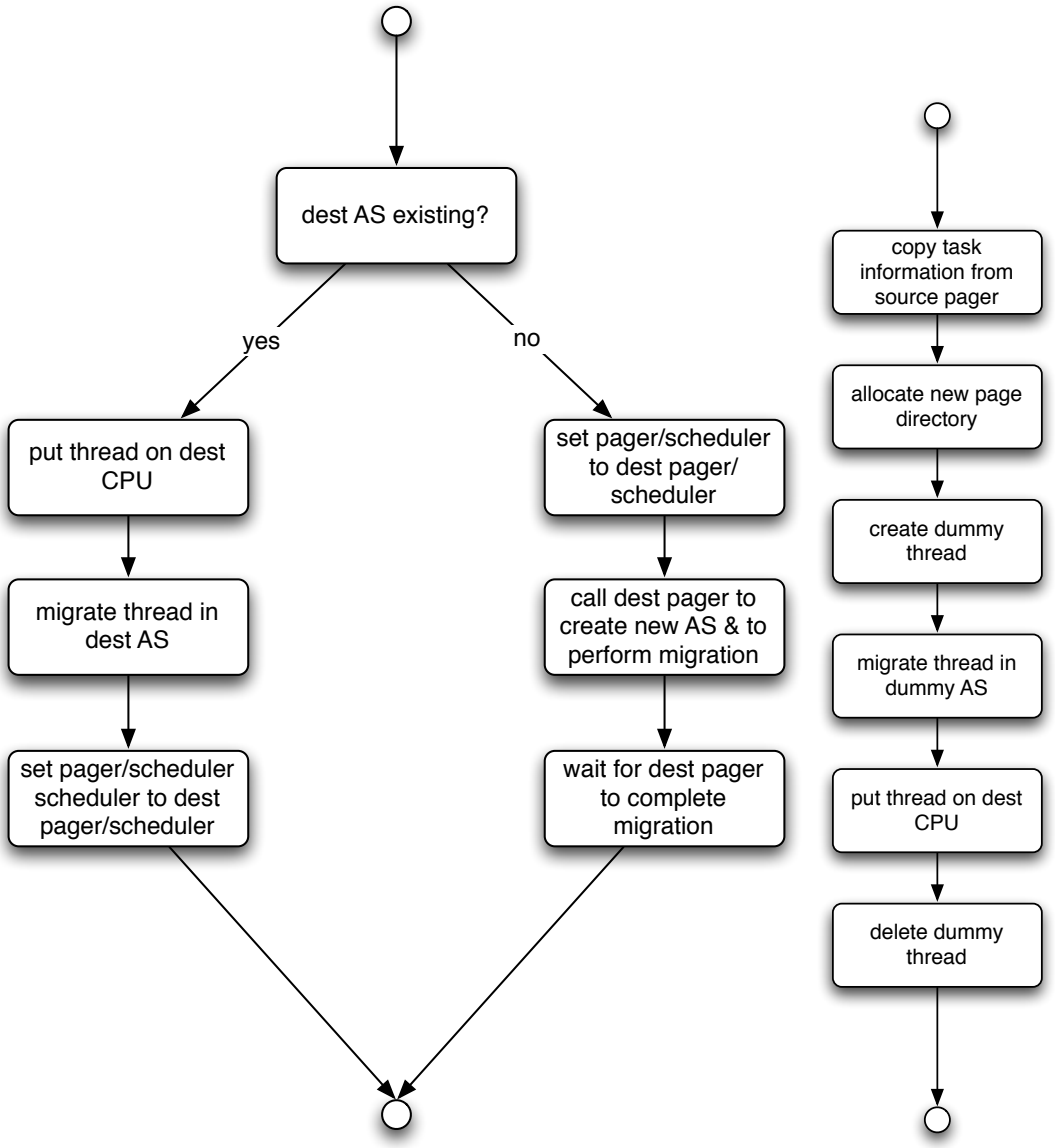
Migrating a thread from one to another node implies not only to migrate it to a new CPU, but also to migrate it into another address space. The pagers therefore need to be aware of which address spaces belong together, i.e. to the same task. L4's API poses some additional challenges here, as an address space is identified only implicitly by the threads that reside within it. When a thread of an application shall be migrated to a node where no other threads of this application exist yet, a new address space must be created. To create a new address space on L4, the *ThreadControl* system call must be used, with the first parameter (the new thread's id) being equal to the second parameter (the address space specifier). Calling *ThreadControl* in this way creates a new thread and a new address space in which the newly created thread is placed. Our scenario, however, would require to create only a new address space on the target node (i.e. with no threads associated to it), and then to migrate an already existing thread into this new address space. This is not possible with L4's API. To work around this without having to change the API to explicitly identify address spaces, there exist two solutions:

1. The thread that shall be migrated is deleted on the source node and recreated on the destination node (together with a new address space) by the destination node's pager. The thread must be recreated with the same global thread ID it had had before. The problem with this solution is that the thread's state is lost when it is deleted. Thus, important state (like the thread's user-level stack pointer and instruction pointer) must be explicitly saved and restored.

2. Instead of deleting and recreating the thread, a "dummy thread" is created on the destination node, together with a new address space (implicitly identified by the dummy thread's id). The thread that shall be migrated can then easily be placed in the newly created address spaces and afterwards be put on a CPU on the destination node. After the thread is migrated into the new address space, the dummy thread can be deleted - without ever becoming active.

To avoid the problems of explicitly saving important thread state when the thread is deleted and restoring the state when it is recreated, we favor the second solution, i.e. creating a dummy thread on the destination node, together with a new address space. Once the target address space exists, the pager on the destination node can put the to-be-migrated thread into the newly created address space and migrate it to the target CPU. The dummy thread can be removed after the migration

is complete. Figure 4.5 summarizes the steps necessary to migrate a thread. The destination pager is only invoked when there exists no thread of the affected task on the destination node. In that case, the destination pager does not have any in-



(a) Steps for thread migration performed by the source pager.

(b) Steps for thread migration performed by the destination pager.

Figure 4.5: Necessary steps to migrate a thread between CPUs/nodes.

formation about the task's address space layout. It creates an empty page directory and queries the master pager with every page fault it receives. Alternatively, it can eagerly copy the master pager's page table hierarchy after the thread is migrated. Note that copying the master pager's tables does not avoid page faults in the newly created address space, but only avoids querying the master pager when handling a page fault on an already valid mapping. In addition to the mapping information

(i.e. page tables), the pager requires the task management information (i.e. which threads belong to the task and on which CPUs these threads currently reside). The destination pager copies this data from the source pager. Of course, it has to adapt the *pdir* pointer of the copied task object so that it points to the destination pager's newly allocated page directory.

A special case of thread migration is when the last thread of a task on a specific node is migrated away to another node. In that case, the address space on the source node will automatically be destroyed (L4 destroys an address space when the last thread in that address space is killed or moved away). If the pager on the source node is the root pager, it either must not delete the address space management informations (so that the other pagers can still update it), or a new root pager must be chosen. This change must be propagated to all other pagers so that they use the new root pager from now on. Currently, our API does not support to change the root pager. If the pager on the source node is not the root pager, it must discard all address space management information. Storing the informations for the case that the task will become active on the corresponding node again in the future is not possible, because these informations cannot be guaranteed to be kept up-to-date. For example, if a page is unmapped from a task, only pagers of those nodes where the task is active will be notified. Thus, mappings that are no longer valid might remain in a page table hierarchy that is "cached" for future use. Alternatively, more complex synchronization strategies can be used to ensure that cached page tables are also kept up-to-date. Yet, the expected benefits of such a solution are rather small: Caching page tables only avoids querying the master pager in case a task is reactivated on a node. It does not reduce the number of page faults: As stated before, L4 implicitly destroys an address space once no more threads reside in it. Thus, in case of reactivation, a new L4 address space must be created and populated (by means of page faults) anyway.

Our design does not allow to change the home node of an address space. This could be a helpful feature if a complete task shall be migrated from one node to another node. If it was possible to change the home node of the corresponding space on the source node in that case, one would not have to pay the costs for creating and repopulating a new address space on the destination node. However, allowing to change the home node would require to modify L4's API (either by modifying an existing system call or by introducing a new one) and introduce a NUMA-specific special case. Additionally, the kernel would have to migrate the entire page table hierarchy to the destination node, which also causes additional overhead.

Frequent migrations of the last thread of a task or of an entire task are thus a worst-case migration scenario for our design: In that case, the corresponding address space on the source node is always destroyed and must be recreated and repopulated in case the task becomes active on this node again, making migration more expensive than it would be if only a single address space per task was used. One has to keep this fact in mind when designing a load balancer for the system. In addition to the fact that too frequent thread migrations should generally be avoided (mainly because of the overhead due to "cold" caches on the destination CPU), a load balancer for our system should also avoid migrating the last thread of a task on a specific node when possible. Additionally, thread migrations between CPUs of the same node are preferable for locality reasons.

4.2.5 Local and Global Mappings

As we explained above, synchronization of address space management information is only necessary when global mappings are affected. Yet, there remains the questions which mappings can be treated as local and which must be global. If an application is not "tuned" towards a NUMA system in that it explicitly marks regions of its address space as local, a pager must treat all virtual addresses as global. Global mappings ensure that the application has the impression of running in a single address space that spans across multiple nodes. An exception from this rule is all data that is known or expected to be read-only: In that case, the pagers can transparently replicate the data amongst nodes and map it node-local into every address space. An example for such read-only data is the application's code itself: The binary can be loaded into different node pools, and all code faults can be handled locally. However, there remains the possibility that code is written for some reasons: For example, the application might contain self-modifying code, or writable, static data might occupy parts of a page of memory that also contains code. To handle with such cases, we implemented a mechanism that is similar to *copy on write*: When a pager receives a page fault with read access on a code page of an application, it first checks the master pager if there is already mapped a page with write access permitted. If so, the pager answers with this page, thus making the mapping global. If not, it answers with the corresponding code page from its own pool, with the access rights set to read-only. Whenever a pager receives a write fault on such a read-only mapping, the mapping must be made global. This implies that all pagers are notified to unmap existing local mappings. Additionally, the new, global mapping must be inserted into the master pager's data structures.

4.2.6 Static and Dynamic Page Placement

Our current implementation with a pager handing out a page of its own memory when no valid global mapping exits yet leads to a first-touch allocation policy, i.e. a virtual address is always backed by physical memory from the node on which it was accessed for the first time. Such a simple policy is expected to be sufficient in most cases. However, there are some scenarios where a first-touch allocation policy will lead to a suboptimal memory placement. For example, consider a parallel application where a single thread first initializes all memory required by the application and then spawns additional worker threads and distributes these worker threads amongst all nodes. In that case, memory initialized by the initial thread will be backed by memory on this thread's node, although it is only accessed by the worker threads, residing on remote nodes. Therefore, the operating system should either offer an API that allows for allocation of memory on a specific node, or it should be able to detect whether a physical page is accessed mainly from remote nodes and migrate or replicate the page dynamically. While our current implementation neither offers an API for manual placement (apart from the possibility to mark a region of the virtual address space as node-local) nor supports any form of dynamic migration and replication, our design influences the implementation of such techniques. For example, our application pagers are not permitted to allocate memory from another but their own node. Static placement on a specific node as well as dynamic migration and replication however require to allocate memory on

a remote node. One possibility to tackle this problem is to discard our allocation scheme so that an application pager is allowed to allocate memory on arbitrary nodes. This in turn requires locking of the memory management information to ensure consistency. We expect allocation of memory on the local node to be much more frequently required than allocation of memory on a remote node. Therefore, locking would induce significant overhead for the common case, which is why an IPC-based solution is favorable. This requires to extend the application pagers' API with a function that allows for remote allocation. The called pager has to do nothing but to allocate the desired amount of memory on its node, to insert the mapping in its data structures (the calling pager must therefore specify to which task the memory is mapped) and to return the corresponding virtual address within its own address space to the calling pager. The calling pager can then map this memory as if it was memory from its own node. Additionally, dynamic migration or replication must be carefully implemented so that consistency is ensured. First, the page that shall be migrated/replicated must be globally unmapped from all address spaces to which it is mapped. Next, it must be ensured that no application pager reestablishes the old mapping while the migration/replication is in progress, i.e. while the contents of the page are copied to the new location. Therefore, the application pager performing the migration/replication must hold the corresponding master pager lock during the entire operation. The lock must not be released before the new mapping is inserted into the master pager's page table. Once this state is reached, all other pagers will see the new mapping when querying the master pager's page table.

CHAPTER 5

Evaluation

In this chapter, we evaluate our work. We analyze the effects of TCB migration and kernel code replication on the performance of L4's inter address space IPC. We compare our user-level solution with a non-NUMA-aware operating system. Special attention is payed on the overhead caused by the additional number of page faults that can occur when multiple address spaces are used for a single application. We also analyze the impacts of our design on the performance of an unmap operation.

5.1 Evaluation Environment

We used a system of two AMD Opteron CPUs to evaluate our work. The AMD Opteron is NUMA-capable, i.e. each CPU has a built-in memory controller. The processors are connected with a coherent HyperTransport link. A detailed description of the Opteron northbridge architecture can be found in [12]. In our system, each CPU was attached to two GB of main memory.

5.2 NUMA Memory Latencies

We started with evaluating the "raw" access latencies for both local and remote memory. To avoid page faults, we ran our benchmark in protected mode, but with paging disabled. The benchmark consisted of two phases: An initialization phase and the actual benchmarking phase. In the initialization phase, a linked list was created by writing a pointer to the next memory location (determined by a fixed offset) to the current memory location. In the benchmarking phase, the time required to walk this list was measured. Walking the list is simply a sequence of consecutive memory reads. Reading the location for the next memory access from the list (i.e. from memory) instead of calculating it by adding a fixed offset is necessary to eliminate effects caused by hardware optimization techniques such as prefetching. With the list approach, the CPU has to wait for a read operation to finish before the next operation can be carried out. The results of this benchmark

are shown in Figure 5.1. As one can see, the remote-to-local ratio is very low, i.e.

$$r = \frac{c_{\text{remote}}}{c_{\text{local}}} \approx 1.4 \tag{5.1}$$

AMD even states a remote-to-local ratio for one-hop read accesses of $r < 1.1$ [3]. We therefore expected only modest performance improvements caused by better locality of kernel objects.

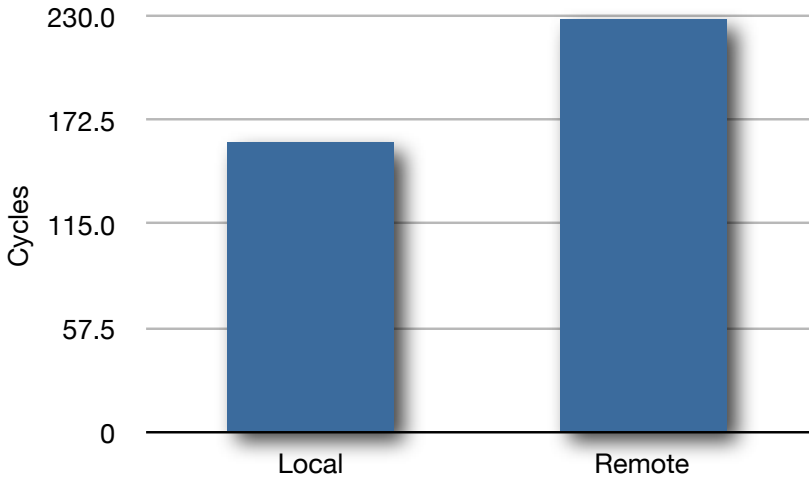


Figure 5.1: Average costs for read accesses to local and remote memory, respectively.

5.3 TCB Migration and Code Replication

Next, we analyzed how TCB migration and kernel code replication affect the performance of L4's IPC primitive. We therefore put two threads on node 1, each within its own address space. These threads send IPC messages to each other. If TCB migration and code replication are turned off, TCB accesses and kernel invocations require accesses to remote memory in case the accessed code/data is not found in the caches. Both threads (together with their address spaces) were created on node 0 and migrated to node 1 before starting the measurement. In this way, both the kernel page tables for the address spaces and the UTCBs of the two threads lie in memory of node 0, i.e. in remote memory. This is because we wanted to evaluate the effects of node-local UTCB placement separately. We measured the performance of inter address space IPC with both code replication and TCB migration disabled, with either code replication or TCB migration enabled, and finally, with both enabled. Figure 5.2 shows the results. Neither TCB migration nor kernel code replication have a significant effect on the IPC performance. To find out the reasons for this, we counted the number of accesses to remote memory during an IPC by configuring a performance counter of the Opteron CPU. Figure 5.3 shows the results for the same kernel configurations as above. Again, almost no differences can be measured. For further analyses, we disabled caching on both CPUs

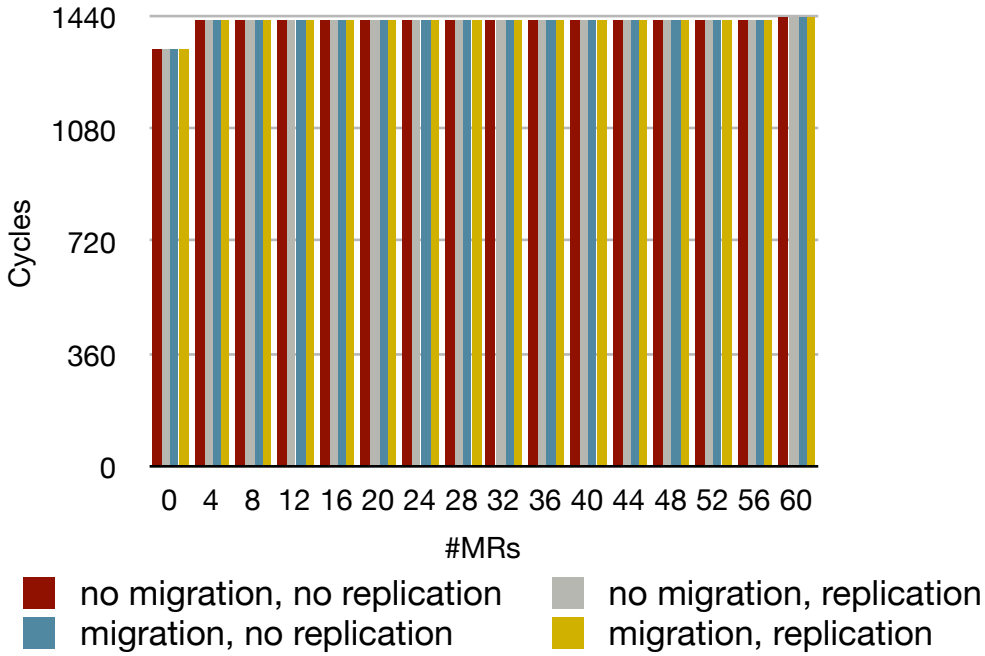


Figure 5.2: Average number of cycles per IPC, caches enabled

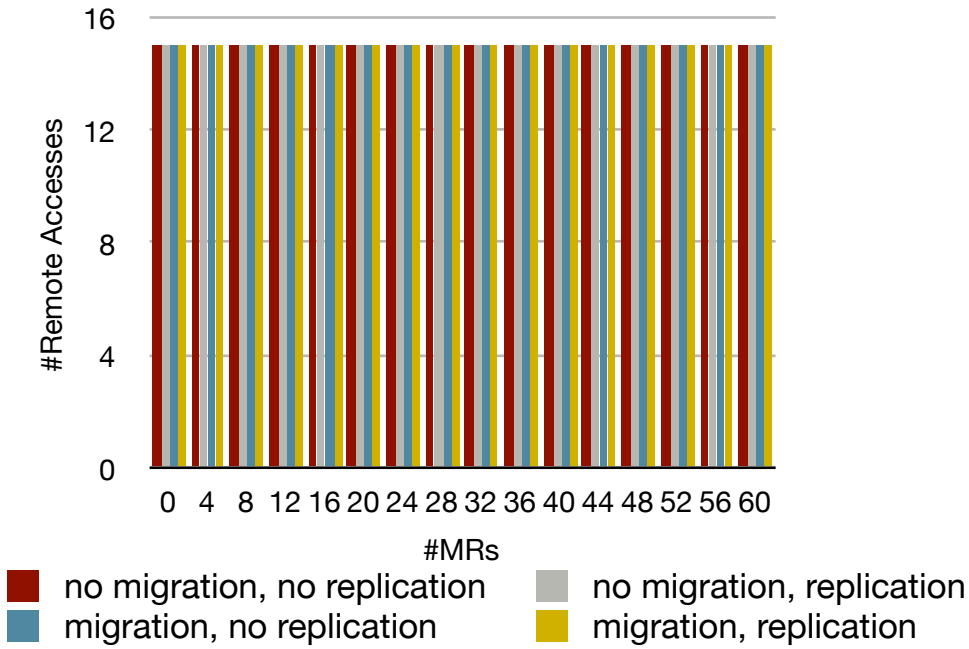


Figure 5.3: Average number of remote accesses per IPC, caches enabled.

and reran our benchmarks. The results can be seen in Figure 5.4 and Figure 5.5. This time, one can see a clear performance improvement caused both by TCB migration and kernel code replication, whereas TCB migration is more beneficial than code replication. The number of accesses to remote memory during an IPC also decreases drastically. One has to keep in mind that both the application code and the UTCBs lie in remote memory, so that there are still remote accesses necessary. Obviously, our ping pong scenario suffers only very few cache misses. This is not surprising: PingPong is a very small program, and the same code path is executed several times. Additionally, also L4's IPC code path for inter address space IPC on the same CPU is very short. We were surprised to find out that TCB migration is more beneficial than code replication: As every access to a TCB requires the execution of some code, one can assume that the number of code fetches is at least as high as the number of TCB accesses. However, other hardware optimizations like instruction prefetching are still enabled and can reduce the penalty for instruction fetches from remote memory.

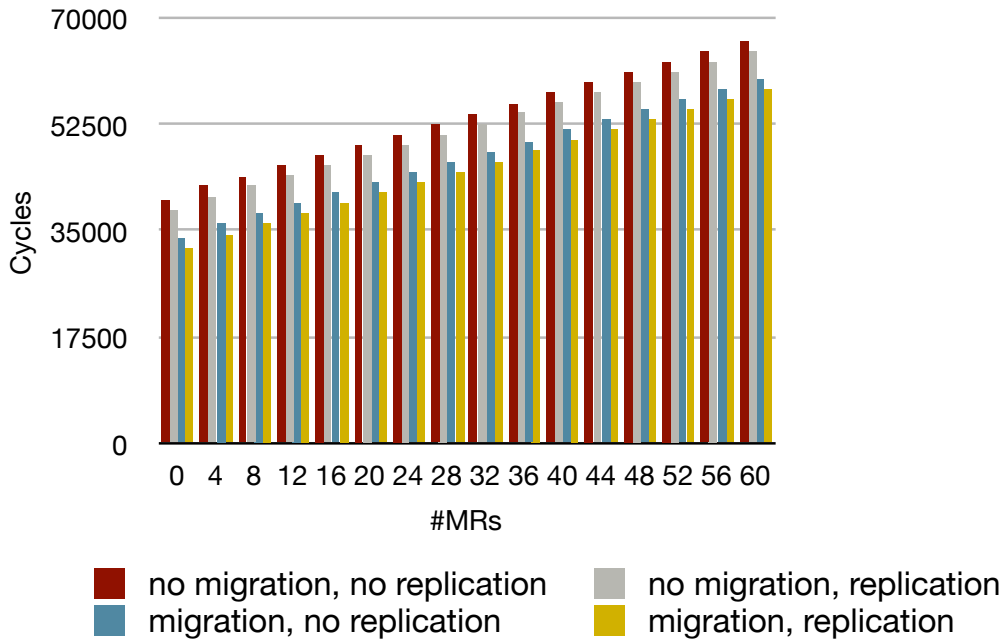


Figure 5.4: Average number of cycles per IPC, caches disabled.

As said above, for this benchmark both threads and address spaces were created on node 0 and afterwards migrated to node 1, leading to the fact that both UTCBs and page tables (except for the page directory) lay in remote memory. We modified our benchmarking scenario to also have UTCBs and page tables node-local: Instead of a master thread creating both threads and then migrating them, we migrated the master thread to node 1 and created the two threads and address spaces for the measurement there. This way, node 1 is set as home node for both spaces, and also the UTCBs are allocated on node 1. We then compared the performance of this solution with the initial configuration, where UTCBs and page tables

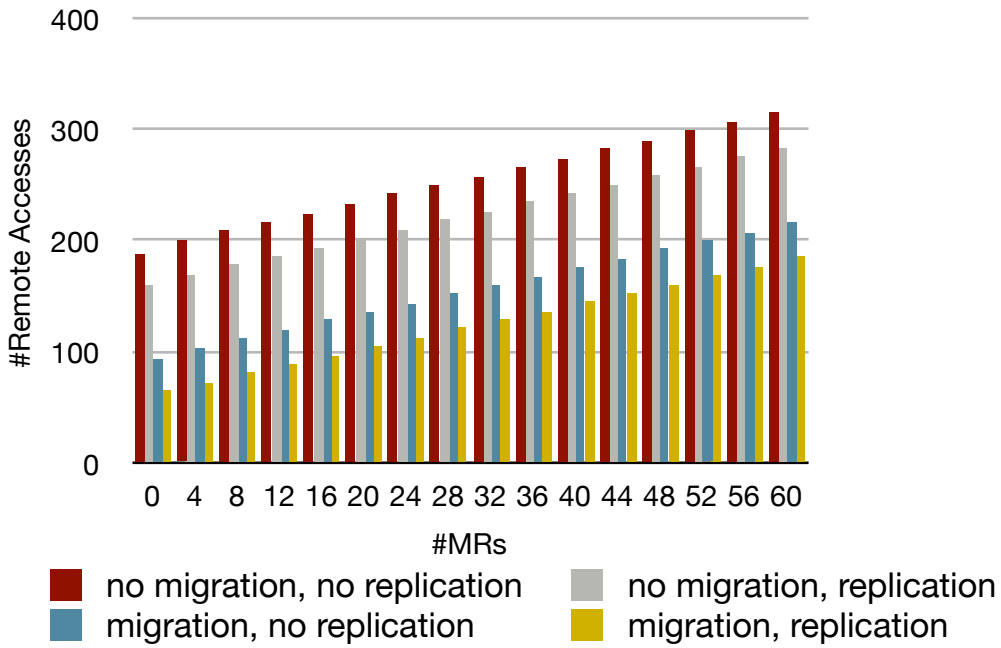


Figure 5.5: Average number of remote accesses per IPC, caches disabled.

were in remote memory. For this benchmark, caches were enabled. Figure 5.6 compares the IPC performance in both cases, Figure 5.7 the number of accesses to remote memory. While the overall performance (i.e. the number of cycles per IPC) is not improved, the average number of remote accesses per IPC is clearly reduced. As the working set of PingPong is rather small, we expect all virtual-to-physical translations to be cached in the TLB. Therefore, local placement of page tables is not of importance in this scenario, and the reduction of remote accesses is caused by local UTCB placement. As a thread's UTCB holds the thread's message registers, it is frequently accessed during an IPC.

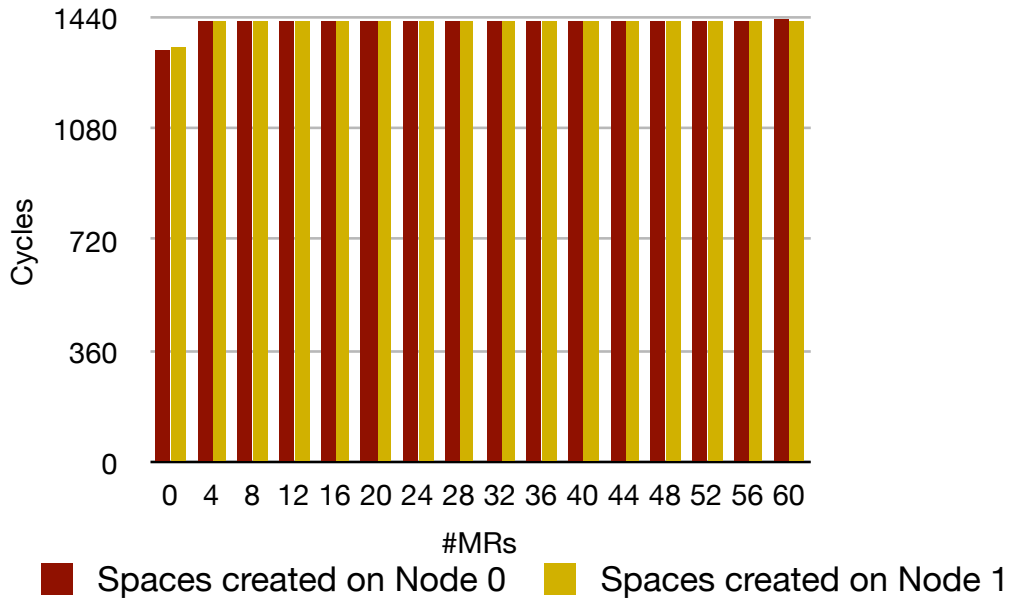


Figure 5.6: Influence of chosen home node the on average number of cycles per IPC.

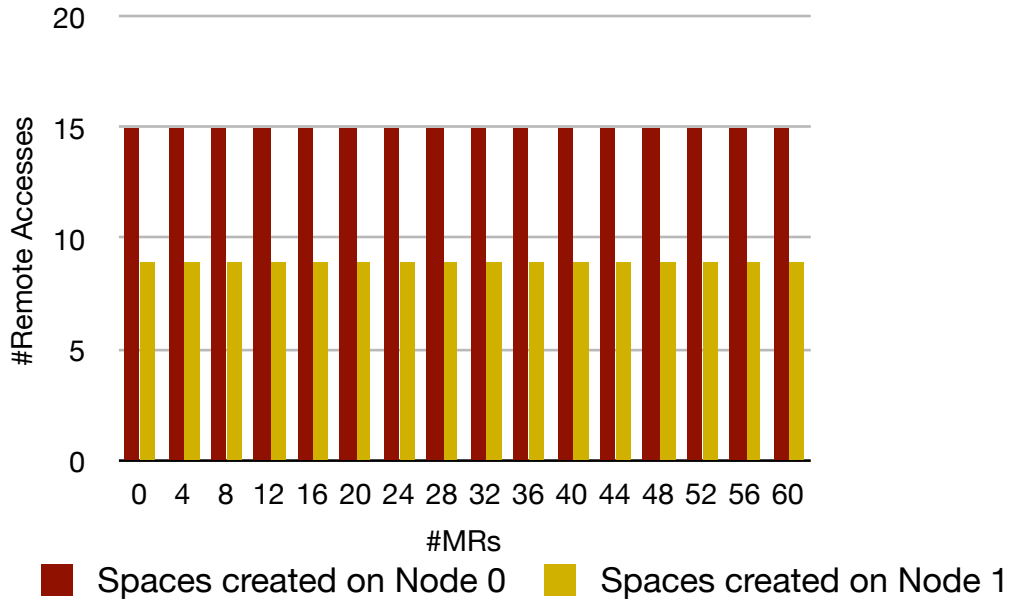


Figure 5.7: Influence of chosen home node on the average number of remote accesses per IPC.

5.4 User-Level Architecture

5.4.1 Establishment of Mappings

Next, we wanted to analyze the impacts of our design on a memory-intensive user-level application. We therefore implemented the pager hierarchy depicted in Figure 4.4 for our two-node system, i.e. two application pagers running on top of a master pager. Each application pager thread resides in its own address space, with the home node set to the node to which the pager belongs. Application pager code is replicated where possible (i.e. as long as it is not written). Each application pager uses its own management information, the pager on node 0 was designated as master pager. The pagers synchronize via shared memory, with a simple spin lock used to serialize accesses to the master pager's data structures when necessary. Each of our two application pagers consists only of a single thread, therefore, no locking was required within the second (i.e. non-master) pager. Additionally, we implemented a similar, non-NUMA operating system, that also consists of two application pager threads (one per CPU), but within the same address space. In contrast to our NUMA system, a single L4 address space was used for a task, and only memory from node 0 was given out. We implemented a small, memory-intensive sample application that consists of two threads, one per CPU. Each of these threads works on a large array within the application's task address space by walking it in 4 byte steps. The first thread starts at the bottom of its array and works upwards, the second starts at the top and works downwards. If the application is configured to walk the arrays several times, the threads walk back in the opposite direction once they have reached either the upper or lower end of their array. As the application's address spaces are initially empty, walking the array causes page faults on the first iteration. No unmaps are performed at application runtime, so that subsequent iterations cause no additional page faults. We measured the time until both threads completed the iterations over their array. This application was executed on top of both operating systems. We modified the number of iterations over the arrays as well as the degree at which the arrays overlap.

We began with the worst case scenario for our design, which is when both arrays occupy the same region of memory, i.e. overlap completely. In that case, our NUMA OS will suffer twice the number of page faults compared to any solution that uses only a single address space for a task. Figure 5.8 depicts the duration to complete against the number of iterations the threads performed, both for the NUMA and the non-NUMA system. As one can see, the NUMA system performs worse than the non-NUMA system when the array is walked only a few times. With an increasing number of iterations however, the NUMA system outperforms the non-NUMA system. This is the result we expected: With only a few iterations, the higher startup costs caused by the additional number of page faults are dominant, with a higher number of iterations, the improved locality in our NUMA system compensates this additional penalty. Figure 5.9 shows the overall number of remote accesses. Again, the NUMA OS suffers higher startup costs, caused by twice the number of page faults and thus twice the synchronization overhead (each page fault requires access to the master pager's page tables and thus to remote memory), but then outperforms the non-NUMA OS. However, this figure is representative only

to a limited degree: During our observations, we ran the same benchmark with a slightly different test application (which performed the same iterations over the array and only lacked an additional remote function call after the iterations were finished), this time, the total number of remote accesses caused by the NUMA OS was always higher than the number of remote accesses caused by the non-NUMA OS (at least up to ten iterations). Nevertheless, we got similar results for the overall runtime: With only a few iterations, the non-NUMA OS was faster, with an increasing number of iterations, the NUMA OS outperformed the non-NUMA OS. To understand why the NUMA OS can complete faster even when the total number of remote accesses is higher, one has to have a closer look at our test application: Both the NUMA and the non-NUMA OS suffer no additional synchronization overhead once all mappings are established. The increasing number of remote accesses with every iteration is caused by accessing the data itself, i.e. by the threads walking the array. In case of the NUMA OS, approximately one half of the array is backed by memory from node 0, the other half by memory from node 1. In case of the non-NUMA OS, the entire array is backed by memory from node 0. Thus, the sum of remote accesses caused by the two application threads is approximately the same in both cases. With the NUMA OS however, the remote accesses are distributed amongst both CPU, while with the non-NUMA OS, the thread on CPU 0 has to perform no remote accesses, and the thread on CPU 1 has to perform only remote accesses. Our sample application completes when both threads finished processing the array, i.e. the time to completion is determined by the speed of the slowest thread. Thus, not only the overall number of remote accesses affects application performance, but also the distribution of remote accesses amongst the CPUs. Figure 5.10 shows that the total number of remote accesses in case of the non-NUMA OS (i.e. the number of remote accesses on CPU 1) is always higher than the number of remote accesses per CPU in case of the NUMA OS.

As said before, it is a worst-case scenario for our operating system when both arrays overlap completely. On the other hand, it is the best case when both threads work entirely on their own data, i.e. the arrays do not overlap. Even if the application pagers have to treat all mappings as global (and therefore need to perform synchronization of their page tables), our solution will not suffer more page faults than a solution with only a single address space. Additionally, due to the first-touch allocation policy of our system, each thread can work on memory local to its own node. Figure 5.11 shows the results if the two arrays do not overlap, both for the NUMA and the non-NUMA system. Figure 5.12 shows the total number of remote references in that case. The NUMA OS clearly shows better performance and better locality than the non-NUMA OS.

If the application programmer knows which data is completely local to a thread, a further optimization is possible by marking the appropriate region(s) within the application's address space as node-local. For page faults on node-local addresses, the application pagers do not have to perform synchronization, thus reducing the number of accesses to remote memory and improving concurrency. Figure 5.13 compares the local with the global approach. In both cases, the application threads access only their own array, and the array is backed by memory of the local node. As one can see, the advantage of local mappings is rather small, i.e. the costs for synchronization are relatively low in our scenario.

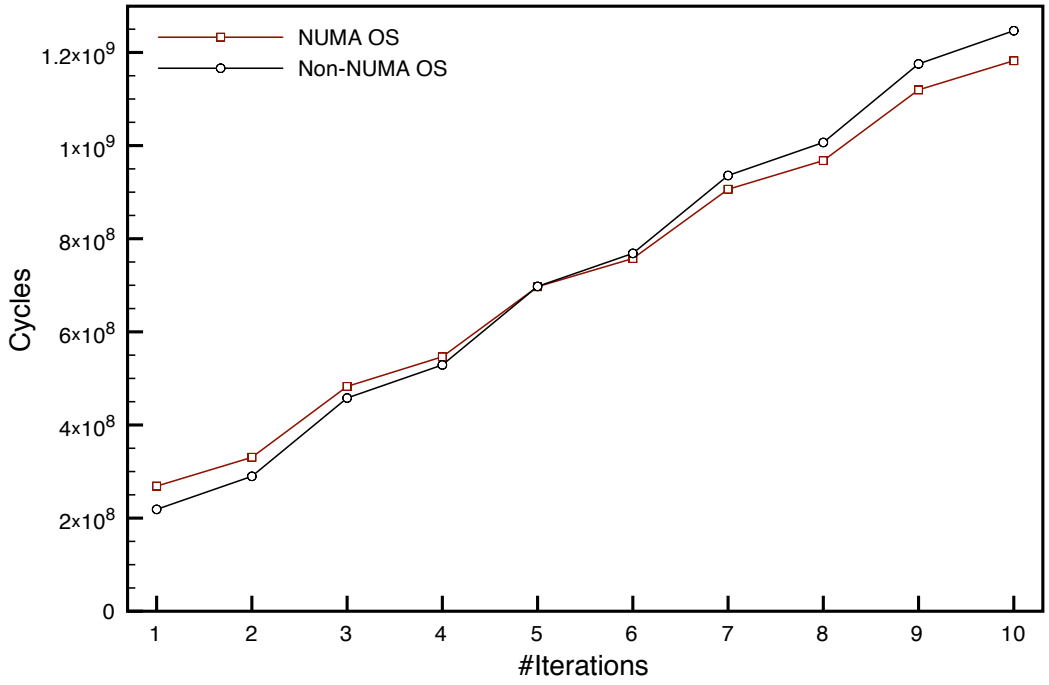


Figure 5.8: Duration to walk a 50 MB array n times in parallel.

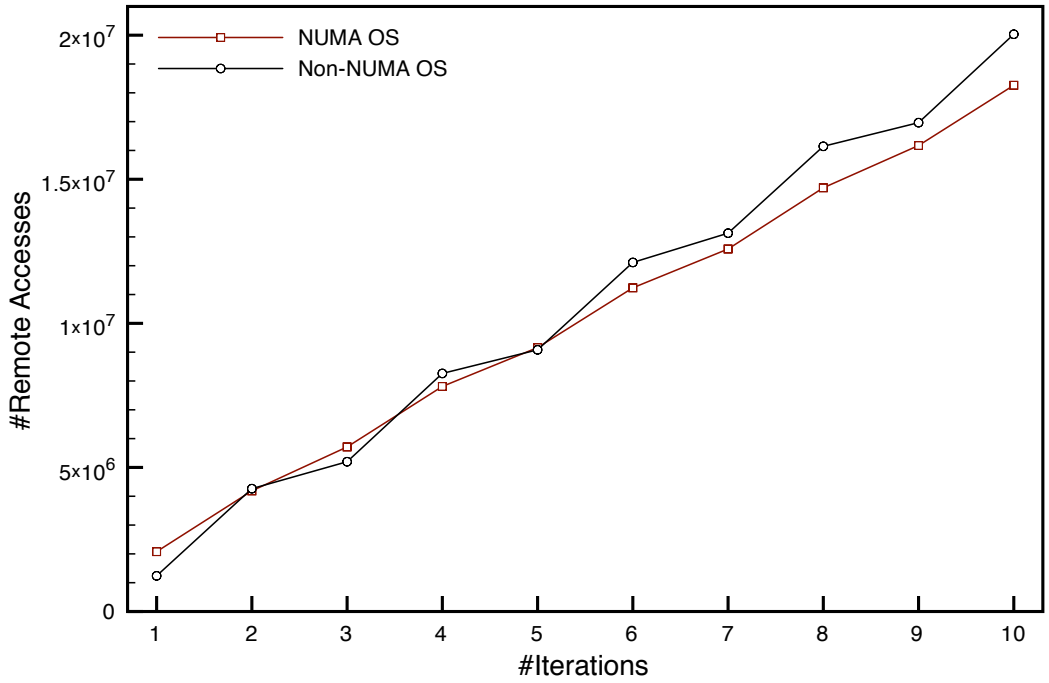


Figure 5.9: Number of remote accesses when walking a 50 MB array n times in parallel.

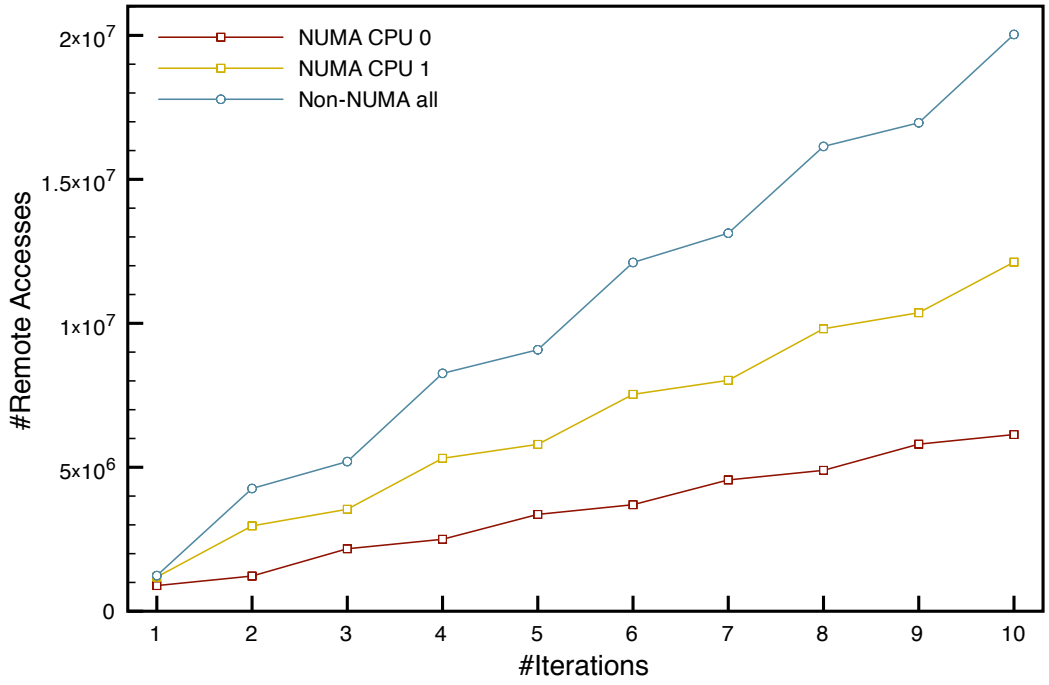


Figure 5.10: Number of remote accesses on both CPUs for the NUMA OS. In case of the non-NUMA OS, only CPU 1 has to access remote memory.

Now we wanted to analyze the impacts of different degrees of overlapping, i.e. configurations between the worst-case (complete overlapping) and the best-case (no overlapping) scenario. For the following benchmarks, we increased the degree of overlapping from 0 (no overlapping) to 1 (complete overlapping) in steps of 0.1. We did this with the threads performing only one iteration over their array, and with the threads performing four iterations over their array. Figure 5.14 and Figure 5.15 show the results. With only one iteration, the NUMA system suffers from the additional page faults even with a small degree of overlapping, i.e. 30%. With four iterations, the improved data locality of the NUMA OS is beneficial even when the arrays overlap to a degree of 90%.

Finally, we measured the effect of page table replication. Therefore, we adapted our NUMA OS in that only a single address space was used for the test application. Thus, the number of page faults on shared data is reduced, i.e. a mapping is automatically established on all nodes when a fault on one node is handled. Yet, in-kernel page tables are not replicated (except for the page directory, which is replicated per CPU to allow for CPU-local in-kernel data). For this test, the two arrays overlapped completely. Figure 5.16 compares the configuration with only a single L4 address space for the test application with the configuration with one address space per node. The configuration with per-node address spaces has higher startup costs due to the additional page faults. The improved locality of page tables is not sufficient to compensate this startup penalty, even with a high number of iterations. This is because the number of page table accesses is very low compared

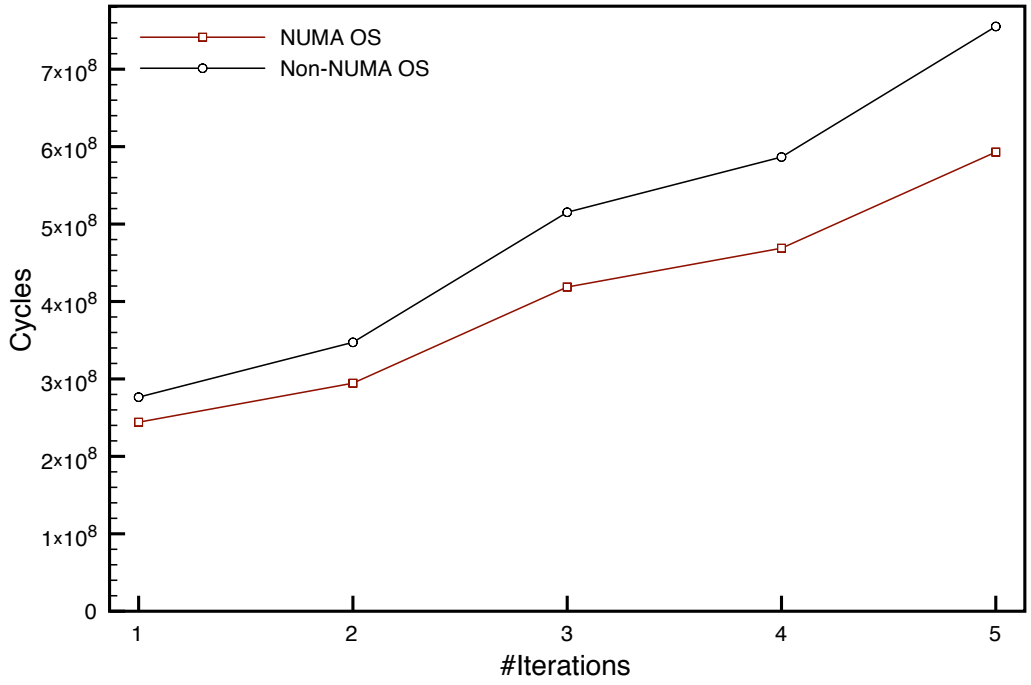


Figure 5.11: Duration to walk two non-overlapping 50 MB arrays in parallel.

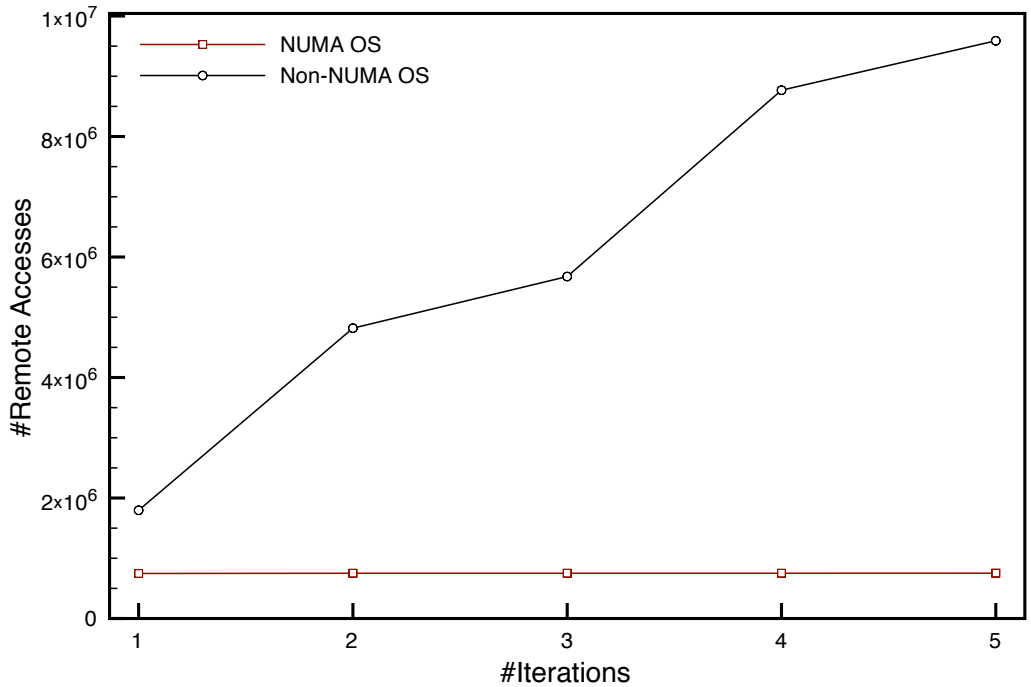


Figure 5.12: Number of remote accesses when walking two non-overlapping 50 MB arrays in parallel.

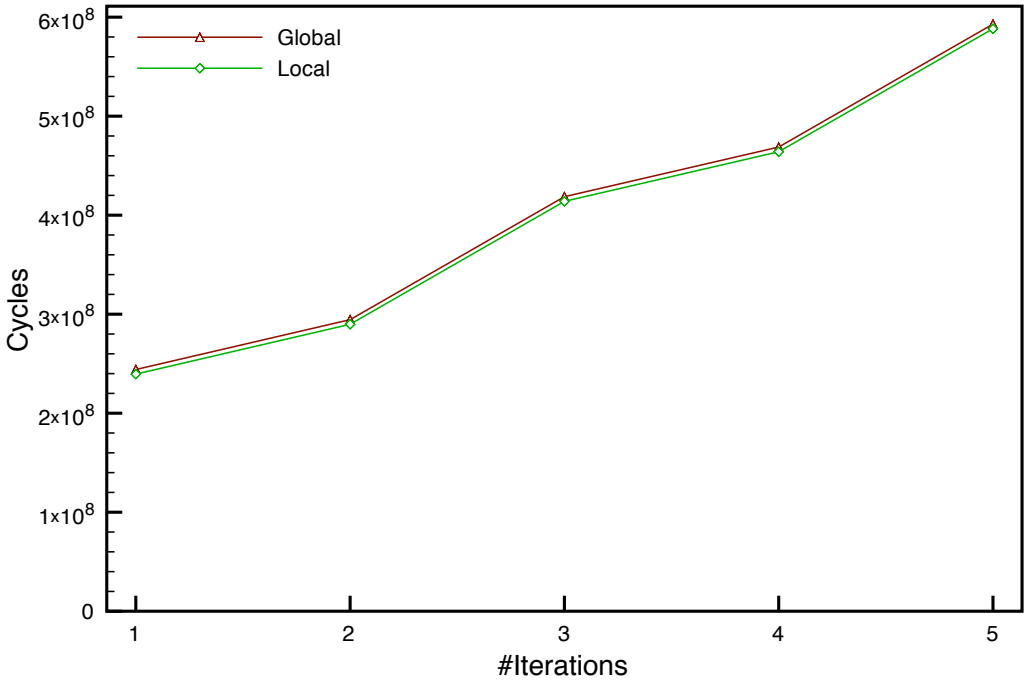


Figure 5.13: Local mappings vs. global mappings

to the overall memory accesses performed by the application: In the worst case, one page table access is required whenever a new page of the array is accessed (realistically, the TLB is expected to reduce the number of page table accesses even further), but reading the contents of the page in 4 byte steps requires 1024 memory accesses (assuming that each memory access fetches only 4 bytes). We did not implement in-kernel replication of page tables, but expect this solution to perform similar than the single address space solution shown above. In-kernel replication will offer better locality of page tables compared to the single address space scenario from above, but will also suffer from additional in-kernel synchronization overhead. Yet, our investigations have shown that better locality of page tables does not lead to a performance improvement in our benchmarking scenario.

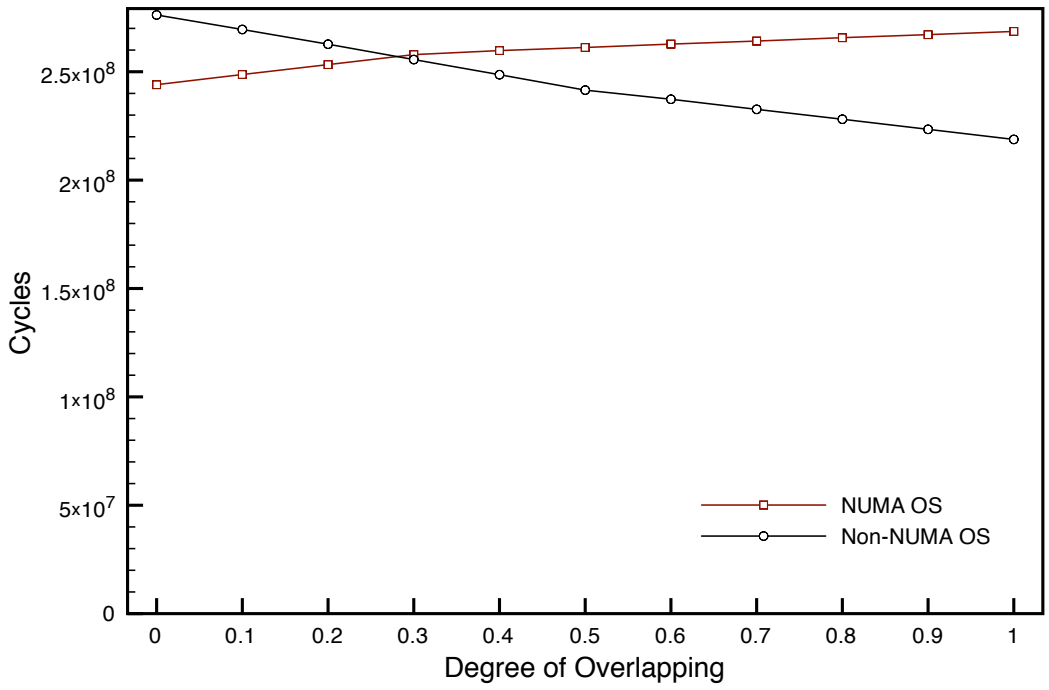


Figure 5.14: Dependency of execution time on the degree of overlapping between the two threads' arrays, one iteration.

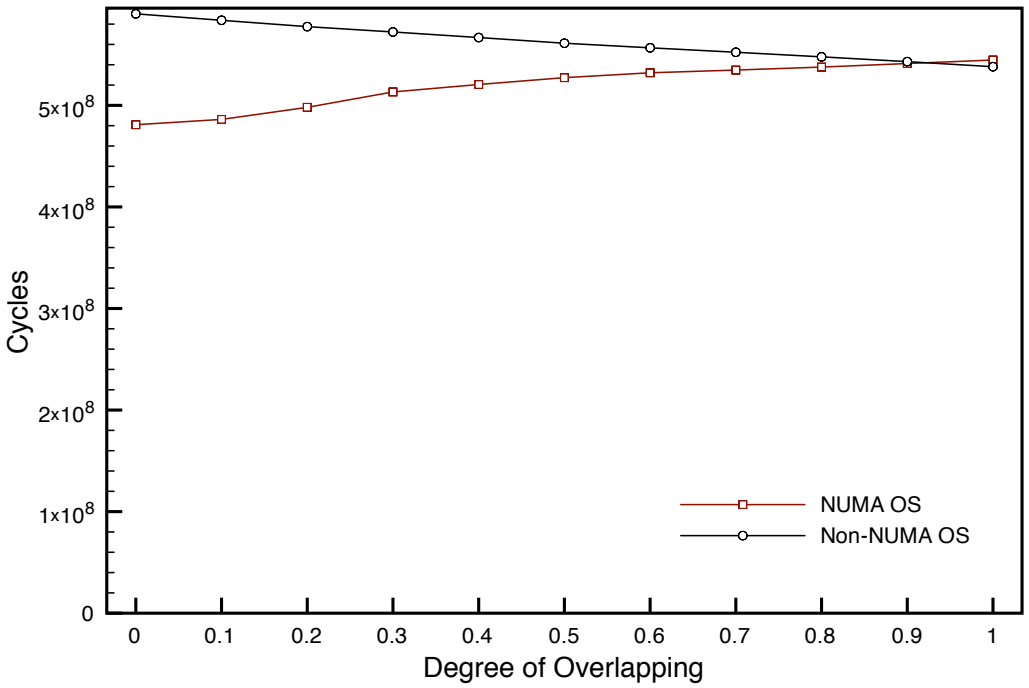


Figure 5.15: Dependency of execution time on the degree of overlapping between the two threads' arrays, four iterations.

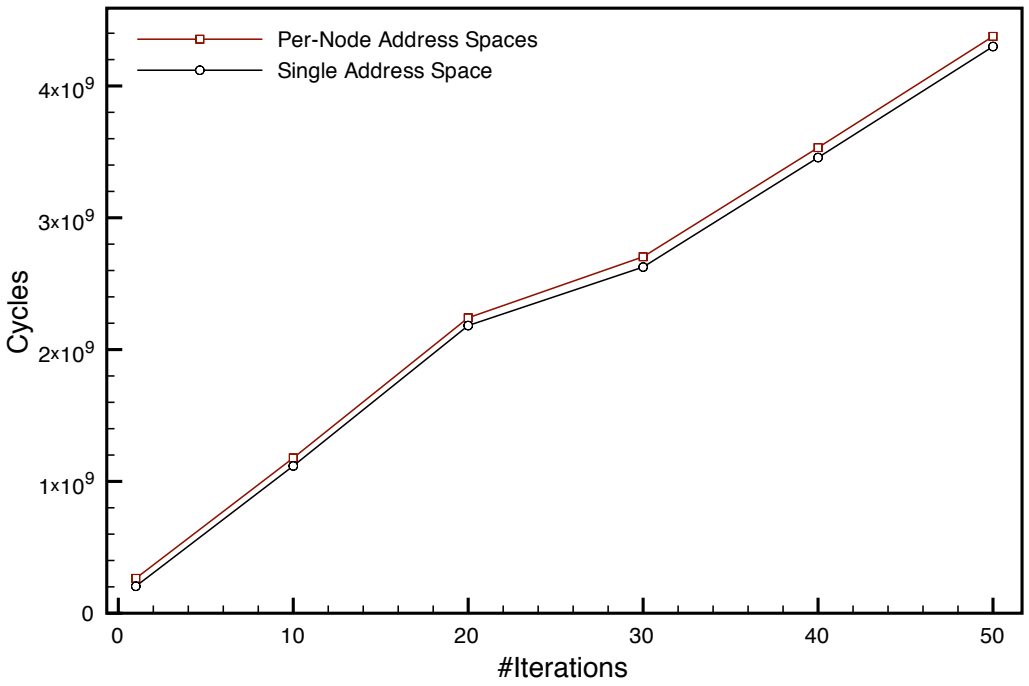


Figure 5.16: Single address space vs. per-node address spaces.

5.4.2 Revocation of Mappings

Our design not only affects the expected number of page faults, but also influences the performance of unmapping pages. If a page must be globally unmapped, it is not sufficient when only one pager calls L4's *unmap* system call. Instead, the other pagers must be notified to perform an *unmap* by themselves. Therefore, the costs for unmapping increase with the number of pagers that must be notified. We measured the time to unmap a single page of memory from the application's task address space, both for the non-NUMA and the NUMA OS. In our benchmarking scenario, an application thread running on node 0 touches a page of memory (to ensure that it is mapped) and then calls its pager's *unmap* function. The pager translates the virtual address passed by the application to a "physical" address and then calls L4's *unmap* system call. In case of the NUMA OS, it additionally invokes the *unmap* function of the other application pager, i.e. the pager on node 1. We measured the time from the application thread calling its pager's *unmap* until the call returns, i.e. the unmap completes. For the NUMA OS, two additional cross-CPU IPCs are required. It is therefore not surprising that the costs in case of the NUMA OS are about three times as high as in case of the non-NUMA OS, as can be seen in Figure 5.17. Due to the synchronous RPC semantics of the remote invocation, the originating pager will wait for the first remote pager to complete before calling the next (if any). Therefore, one can expect that the costs for a global unmap increase linearly with every additional pager that must be called. This additional overhead might be critical in some scenarios, especially with an increasing number of nodes and highly parallel applications (i.e. applications that span across multiple nodes). To optimize performance, it it possible to discard the synchronous RPC semantics when invoking the other pagers' *unmap* and use asynchronous RPC instead[1]: Instead of the originator waiting for the completion of each remote procedure call before calling the next, it can notify all pagers and then wait for all pagers to call in at a barrier. With that approach, the called pagers can perform their local unmap in parallel. Yet, the invocation of the other pagers' *unmap* would still have to be performed sequentially, as L4's IPC does not support any form of broadcast. If this optimization is not sufficient, there remains the possibility that the root pager of the system is used to perform a global unmap, as described in Section 3.2.3. Because the root pager owns all physical memory and passed it to the application pagers at system startup, an unmap performed by the root pager will revoke the corresponding mapping from all task address spaces and additionally from all application pagers' address spaces. The latter is the biggest disadvantage of this solution, because the application pagers must rerequest the memory from the root pager (by means of page faults). The condition that the root pager is only invoked at system startup will thus no longer hold true.

[1] IDL^4 offers *oneway functions* for that purpose.

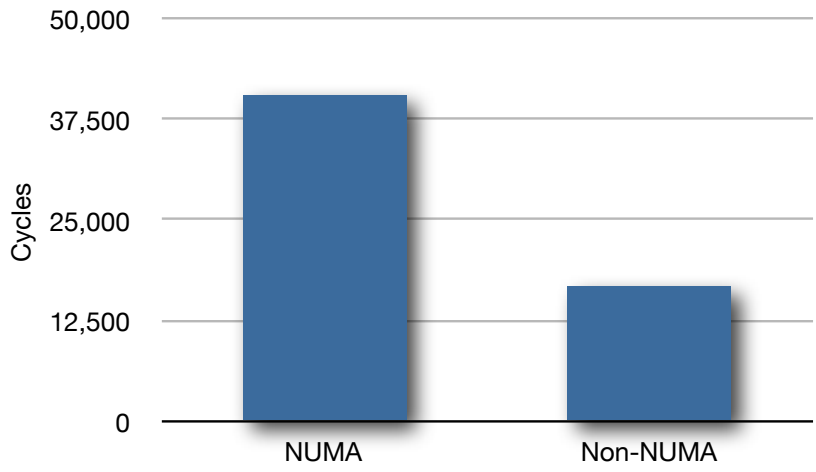


Figure 5.17: Duration to unmap a single global page, both for the NUMA and the non-NUMA OS.

CHAPTER 6

Conclusion

In this work, we designed and implemented an L4-based operating system tailored towards NUMA hardware. Our design decisions were driven by the requirement to improve locality of objects (i.e. to place objects on nodes they are accessed from) and to preserve parallelism (i.e. to design the OS so that it can handle independent, parallel requests in parallel). Migration and replication are techniques that can improve locality and parallelism, but the choice which one to choose is object-depended: Only objects with a high read-to-write ratio are candidates for replication, and only objects that are mainly accessed from a single node are candidates for migration. We therefore revised the most important data structures of L4, namely thread control blocks, *space_t* objects (including page tables), the mapping data base, and the kernel code itself. For TCBs, we favored migration over replication, as this optimizes for IPC between threads on the same CPU. We decided to keep the scheme of direct TCB addressing via a virtual linear array and therefore had to increase the TCB size so that a single TCB occupies a complete hardware page. We assigned a home node to each address space, set to the node on which the address space is created. The object describing the address space as well as all page tables (except those for in-kernel CPU-local data) are allocated on that home node. Additionally, also the memory backing the UTCB area is allocated on the home node of the corresponding space. We modified the mapping database so that memory for a map node is allocated on the home node of the space the mapnode belongs to. This makes all mapping database accesses node-local as long as memory is mapped between spaces with the same home node. Additionally, we replicated the kernel code to physical memory of each node by using L4's pre-existing implementation of CPU-local kernel data. For user-level applications, we developed a novel approach to allow for flexible, node-local data and for replication of in-kernel page tables. Instead of keeping all threads of a task within the same address space, we propose to construct one address space per node (or even per CPU), with the user-level pagers keeping track of which address spaces belong to the same task. By doing so, in-kernel page tables are synchronized implicitly by means of page faults. This avoids the introduction of an additional synchronization

primitive within the kernel, the choice of which would not only depend on access and sharing patterns of the page table it protects, but also on the latency ratio between remote and local memory accesses. Having such a primitive in the kernel would constrain the kernel's flexibility and contradict the principle of a minimal, policy-free microkernel. In addition to the fact that our user-level design avoids the introduction of a new in-kernel synchronization primitive, it also allows migration of UTCBs without having to change their size or to discard the scheme of direct addressing.

Our user-level approach requires a tight interaction between application pagers. Threads migrating to another node must implicitly be migrated into another address space. Global mappings must be made visible in all address spaces belonging to the same task. Unmapping of global pages requires special precaution, as the mapping must be revoked immediately from all address spaces. We developed an IPC-based protocol for synchronization between pagers in case of an unmap.

We evaluated the effects of our kernel modifications by running a microbenchmark to measure IPC performance between two threads on the same CPU, but in different address spaces. Our evaluation on a small AMD Opteron system with two nodes showed that the modifications we made to kernel data structures did not show much of an effect. Only with caches disabled, one can see a clear improvement in IPC performance and a reduction of accesses to remote memory with our NUMA kernel configuration enabled. Obviously, caches, TLBs, and other hardware techniques like instruction prefetching can compensate the additional costs caused by accessing kernel objects in remote memory. Additionally, the remote-to-local latency ratio of the Opteron system is relatively low. However, even with caching enabled, local placement of UTCBs leads to a reduction of the number of remote accesses per IPC. Finally, we compared our operating system with an implementation not tailored towards NUMA systems by running a memory-intensive parallel application on top. For data shared amongst multiple threads on multiple nodes, our NUMA OS suffers more page faults than an OS that uses only a single address space per task. Yet, our results show that these additional startup costs are compensated by the improved data locality provided by the NUMA OS when the number of data accesses (i.e. the number of iterations over the threads' arrays in our scenario) increases. Our design is most beneficial when each thread works only on its own data and never accesses data from the other. In that case, the number of page faults is as high as for the non-NUMA OS, but data locality is significantly improved: Each thread's data is placed in local memory. We found out the the overall performance of our application is not only influenced by the total number of accesses to remote memory, but also by the distribution of remote accesses amongst the CPUs.

6.1 Suggestions for Future Work

The system we used for evaluation was a small-scale NUMA machine, consisting of only two nodes, and with remarkably low access latencies to remote memory. While these characteristics allow for good performance also for applications and operating systems that are not tuned towards NUMA machines, they also make it hard to evaluate any benefits of a better object placement. Therefore, it would be interesting

to evaluate the effects of our changes on a system with a higher remote-to-local ratio. Additionally, our system consisted of only two nodes and was thus very small. In case of larger-scaled systems, the performance of inter-pager synchronization, especially the performance of a global unmap operation, might lead to more severe performance penalties. Alternative solutions for globally unmapping a page (like the proposed use of asynchronous RPC) have to be investigated, if our IPC-based solution turns out to be too slow. Our application used for benchmarking was fairly synthetic and only aimed for analyzing the effect of our design on a highly parallel, memory-intensive application. The impacts of our design on a more "realistic" workload still need to be investigated. Our modifications might also be useful for L4-based virtualization solutions. A NUMA-aware virtual machine monitor can hide the NUMA characteristics of the hardware from a non-NUMA-aware guest operating system.

Bibliography

[1] AMD. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*, February 2005.

[2] AMD. *BIOS and Kernel Developer's Guide for AMD Athlon 64 and AMD Opteron Processors*, October 2005.

[3] AMD. *Performance Guidelines for AMD Athlon64 and AMD Opteron ccNUMA Multiprocessor Systems*, June 2006.

[4] Jonathan Appavoo, Marc Auslander, Dilma Da Silva, Orran Krieger, Michal Ostrowski, Bryan Rosenburg, Robert W. Wisniewski, Jimi Xenidis, Michael Stumm, Ben Gamsa, Reza Azimi, Raymond Fingas, Adrian Tam, and David Tam. Enabling scalable performance for general purpose workloads on shared memory multiprocessors. Technical report, IBM Research, 2003.

[5] Mohit Aron, Luke Deller, Kevin Elphinstone, Trent Jaeger, Jochen Liedtke, and Yoonho Park. The SawMill framework for virtual memory diversity. In *Proceedings of the 8th Asia-Pacific Computer Systems Architecture Conference*, Bond University, Gold Coast, QLD, Australia, January 29–February 2 2001.

[6] Martin J. Bligh, Matt Dobson, Darren Hart, and Gerrit Huizenga. Linux on NUMA systems. 2004.

[7] William J. Bolosky, Robert P. Fitzgerald, and Michael L. Scott. Simple but effective techniques for NUMA memory management. *ACM*, 1989.

[8] William J. Bolosky, Michael L. Scott, Robert P. Fitzgerald, Robert J. Fowler, and Alan L. Cox. NUMA policies and their relation to memory architecture. *ACM*, 1991.

[9] B. C. Brock, G. D. Carpenter, E. Chiprout, M. E. Dean, P. L. De Backer, E. N. Elnozahy, H. Franke, M. E. Giampapa, D. Glasco, J. L. Peterson, R. Rajamony, R. Ravindran, F. L. Rawson, R. L. Rockhold, and J. Rubio. Experience with building a commodity intel-based ccNUMA system. *IBM Journal of Research and Development*, 45, 2001.

[10] Edouard Bugnion, Scott Devine, and Mendel Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. 1997.

[11] Eliseu M. Chaves, Jr., Prakash Ch. Das, Thomas J. LeBlanc, Brian D. Marsh, and Michael L. Scott. Kernel-kernel communication in a shared-memory multiprocessor. *CONCURRENCY: PRACTICE AND EXPERIENCE*, 5(3), Mai 1993.

[12] Pat Conway and Bill Hughes. The AMD Opteron northbridge architecture. *IEEE Micro*, 27(2):10–21, 2007.

[13] A. Cox and R. Fowler. The implementation of a coherent memory abstraction on a NUMA multiprocessor: experiences with platinum. *SIGOPS Oper. Syst. Rev.*, 23(5):32–44, 1989.

[14] Ben Gamsa, Orran Krieger, Jonathan Appavoo, and Michael Stumm. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, 1999.

[15] Benjamin Gamsa, Orran Krieger, and Michael Stumm. Optimizing IPC performance for shared-memory multiprocessors. In *ICPP (1)*, pages 208–211, 1994.

[16] Kinshuk Govil, Dan Teodosiu, Yongqiang Huang, and Mendel Rosenblum. Cellular disco: Resource management using virtual clusters on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 18, 2000.

[17] Andreas Haeberlen. *IDL4 1.0.0 User's Manual*, April 2003.

[18] Intel. *IA-32 Intel Architecture Software Developer's Manual Volume 2A: Instruction Set Reference, A-M*, January 2006.

[19] Intel. *IA-32 Intel Architecture Software Developer's Manual Volume 2B: Instruction Set Reference, N-Z*, January 2006.

[20] Intel. *IA-32 Intel Architecture Software Developer's Manual Volume 3A: System Programming Guide, Part 1*, January 2006.

[21] The L4Ka Team. *L4 Kernel Reference Manual (Version X.2)*. Universität Karlsruhe, 2004. Available from http://l4ka.org/.

[22] Richard P. Larowe, Jr. and Carla Schlatter Ellis. Experimental comparison of memory management policies for NUMA multiprocesors. *ACM Transactions on Computer Systems*, 9(4):319–363, November 1991.

[23] Jochen Liedtke. On μ-kernel construction. pages 237–250, Copper Mountain, CO, December 3–6 1995.

[24] Jochen Liedtke. Toward real μ-kernels. 39(9):70–77, September 1996.

[25] Jochen Liedtke, Uwe Dannowski, Kevin Elphinstone, Gerd Liefländer, Espen Skoglund, Volkmar Uhlig, Christian Ceelen, Andreas Haeberlen, and Marcus Völp. The L4Ka vision, April 2001.

[26] Jaydeep Marathe and Frank Mueller. Hardware profile-guided automatic page placement for ccNUMA systems. *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 90–99, March 2006.

[27] Michael Marchetti, Leonidas Kontothanassis, Ricardo Bianchini, and Michael L. Scott. *Using Simple Page Placement Policies to Reduce the Cost of Cache Fills in Coherent Shared-memory Systems*. University of Rochester, Dept. of Computer Science, 1994.

[28] Paul E. McKenney, Jonathan Appavoo, Andi Kleen, Orran Krieger, Rusty Russell, Dipankar Sarma, and Maneeesh Soni. Read-copy update.

[29] Paul E. McKenney, Dipankar Sarma, Andrea Arcangeli, Andi Kleen, Orran Krieger, and Rusty Russell. Read-copy update. In *Ottawa Linux Symposium*, pages 338–367, June 2002.

[30] Paul E. McKenney and John D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, pages 509–518, Las Vegas, NV, October 1998.

[31] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, 1991.

[32] D.S. Nikolopoulos, T.S. Papatheodorou, C.D. Polychronopoulos, J. Labarta, and E. Ayguade. User-level dynamic page migration for multiprogrammed shared-memory multiprocessors. *Parallel Processing, 2000. Proceedings. 2000 International Conference on*, pages 95–103, 2000.

[33] Stuart Ritchie. The Raven kernel: A microkernel for shared memory multiprocessors. Technical Report TR-93-36, 1993.

[34] Harjinder S. Sandhu, Benjamin Gamsa, and Songnian Zhou. Region-oriented memory management in shared-memory multiprocessors. Technical report, University of Toronto, April 1992.

[35] Mustafa M. Tikir and Jeffrey K. Hollingsworth. Using hardware counters to automatically improve memory performance. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 46, Washington, DC, USA, 2004. IEEE Computer Society.

[36] Volkmar Uhlig. *Scalability of Microkernel-Based Systems*. PhD thesis, University of Karlsruhe, 2005.

[37] Volkmar Uhlig. The mechanics of in-kernel synchronization for a scalable microkernel. 2007.

[38] Ronald C. Unrau, Orran Krieger, Benjamin Gamsa, and Michael Stumm. Experiences with locking in a NUMA multiprocessor operating system kernel. In *Operating Systems Design and Implementation*, pages 139–152, 1994.

[39] Ronald C. Unrau, Orran Krieger, Benjamin Gamsa, and Michael Stumm. Hierarchical clustering: A structure for scalable multiprocessor operating system design. *The Journal of Supercomputing*, 9(1–2):105–134, 1995.

[40] Ben Verghese, Scott Devine, Anoop Gupta, and Mendel Rosenblum. Operating system support for improving data locality on CC-NUMA compute servers. 1996.

[41] VMWare. VMWare ESX Server 2 NUMA support. Technical report, VMWare.