Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Master's Thesis Nr. 89

Systems Group, Department of Computer Science, ETH Zurich

NUMA Migration on the Barrelfish OS

by

Reto Lindegger

Supervised by

Kornilios Kourtis, Timothy Roscoe

27.8.2013

inf | Informatik
Computer Science

Abstract

A multiprocessor system with uniform memory access is difficult to scale due to the increasing contention on the memory bus and the complexity of the connections between CPUs and memory modules. Non-uniform memory access (NUMA) offers a solution for these problems by introducing so called nodes. Each node consists of a set of processors and local memory. Even though every processor can access memory on all nodes, the access delay differs for local and remote memory. Applications spanning over multiple cores might suffer from a high access delay for ill-placed memory. In this thesis, we introduce an extension for the Barrelfish operating system capable of detecting local and remote memory access and migrating memory between nodes. To achieve automatic NUMA optimization, we implement a migration policy which analyzes the access pattern and tries to find an optimal memory placement. Furthermore, we present a benchmark for measuring the impact of NUMA optimization on memory access performance.

Acknowledgements

First and foremost, I would like to thank Prof. Timothy Roscoe and Kornilios Kourtis for providing me with the opportunity to explore Barrelfish and contribute to it. This thesis was a challenging, exciting and incredibly rewarding experience. Moreover, I would like to thank Kornilios for his assistance and valuable feedback, Simon Gerber and many other members of ETH Systems Group for their input and ideas and Remi Meier, Lukas Humbel and Florian Köhl for interesting discussions and constant motivation. Last but not least, I would like to thank my fiancée Elena Teunissen for keeping me from panicking when I most needed it, my family for their support and my friends for providing time away from work.

Contents

1 Motivation

With increasing CPU speed and the ongoing trend of striving towards machines with a large number of cores, the main memory becomes more and more a bottleneck. Heavily storage oriented applications such as in-memory databases are especially sensitive to memory-induced delays. A multiprocessor system with uniform memory access is difficult to scale, since the contention on the memory bus increases with a higher number of processors. The CPUs have to stall while waiting for data to arrive from main memory. One possible solution to address this problem is the use of non-uniform memory access (NUMA).

Multiprocessor systems with non-uniform memory access divide the main memory into disjoint region called nodes. Each node has a separate memory bus, which allows the load for read/write instructions to be distributed onto different buses, thus reducing the overall bus congestion. One possible setup is one processor per node, so that every CPU has its own local memory. Note that one CPU can have more than one core, hence it is still possible for several cores to share one NUMA node. The assignment of processors to nodes depends on the design and layout of the hardware. An example of such an architecture is depicted in Figure 1.1.

Although memory access is not limited to the CPU's assigned node, the difference in the access time can be significant. While local memory can be accessed relatively fast, store and load requests to other nodes are slower. Ideally, each task (or process) on a NUMA system would run on one CPU and use exclusively local memory. Of course this scenario is oversimplified and rarely encountered in practice. Imagine a multi-threaded program that operates on a large data set (database, compiler). You could run all threads on one node and if you are lucky, there are several cores on this node's CPU. But from the load-balancing point of view, it is not the wisest thing to do. If the target system consists of many processors and nodes, you probably want to distribute the load equally on all nodes. However, this also means the threads are moved away from the data, increasing the average distance from the processing units to the memory and therefore the access time. To reduce the loss of performance caused by the long access paths, a careful placement of the data is desired. An operating system can help the programmer by providing functionalities to allocate memory on certain nodes. This enables the programmer to allocate memory based on the thread distribution and application characteristics. Of course, this technique is very static and demands the programmer to know the memory access pattern at compile time as well as in-depth knowledge of the program flow. A more elegant solution lets the operating system deal with smart NUMA allocation and memory placement. A NUMA aware OS can keep track of the memory regions and

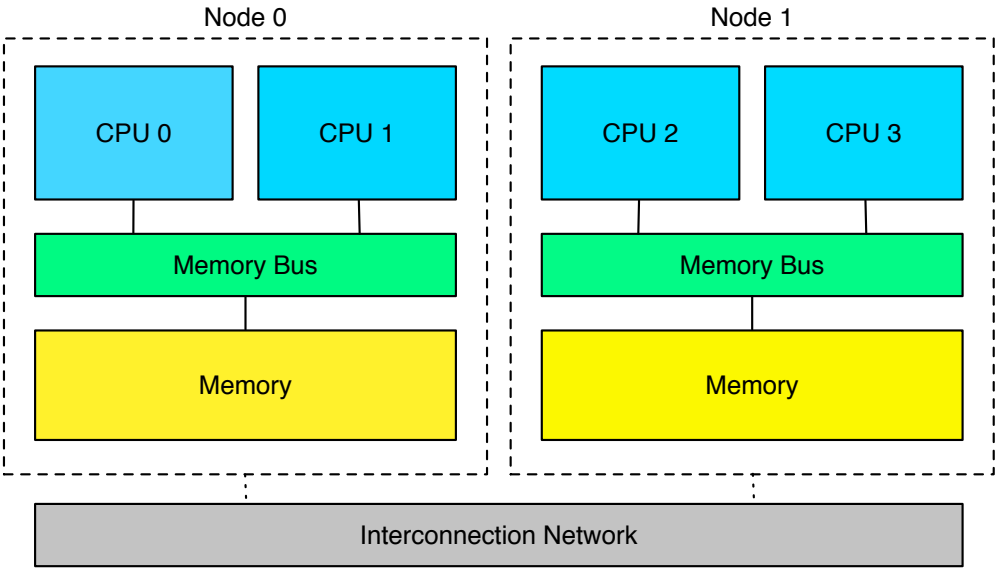

Figure 1.1: Architecture of a possible NUMA system

the cores accessing them. When multiple cross-node memory requests are detected, the system can minimize the overhead of cross-node traffic by moving the corresponding pages to the accessing node. The conditions under which pages are moved to different nodes are determined by the operating system. The basic idea is that the time saved by reduction of the average memory access distance outweighs the overhead of access detection and move operations. The difficulties of this enhancement are the reduction of the overhead for access detection and page migration to a minimum, hiding the move operation from the application such that the application can run continuously and finding an appropriate migration policy.

Outline

This thesis focuses on the functionalities required to detect local and cross-node memory access and to migrate memory from one NUMA node to another. First, we analyze how access detection and memory relocation is achieved by the new Linux kernel. Second, we introduce an extension to the Barrelfish operating system that allows for NUMA aware memory allocation, access detection and memory migration. In the third part, we analyze the impact of this enhancement on application performance by using a memory benchmark with changing access patterns. Finally, we discuss the advantages, disadvantages and problems of the described techniques.

2 Non-Uniform Memory Access in Linux

Support for NUMA awareness is highly desirable for modern and widely used operating systems. For example, Linux introduced a library for NUMA related functionalities several years ago. In 2004, the features were released for the kernel 2.6 [10]. Amongst other things, the new control tool (numactl) and the new API allowed allocating memory on specified nodes and setting memory affinity or allocation policies. The possibility to relocate memory was later added to the library, but the migration had to be manually triggered by the programmer [11]. Since automatic NUMA optimization is a really nice thing to have, there has been quite some activity in the Linux community lately regarding this task. In spring 2012, two different solutions for NUMA aware scheduling and memory placement have been presented.

2.1 Enhanced NUMA Scheduling

The first published NUMA optimization was a large patch set contributed by Peter Zijlstra in March 2012 [7],[15]. According to the author's description, the kernel is currently doing a reasonably good job at keeping short running tasks on a single node. On the other hand, long running tasks with a large memory footprint can get scheduled on other nodes (for the sake of load balancing) and so diverge from their corresponding memory. Since by default memory allocation is done on the node the task currently runs on, this can lead to an unfortunate memory placement with chunks of memory scattered over all nodes. In order to prevent memory scattering, Zijlstra introduces the concept of a *home node*. Every process has its home node, and the scheduler will try not to move the process away from it, but will do so if the system became unbalanced otherwise. However, memory is always allocated from its home node, even if the process is currently running on another node. At some point, the system might get highly unbalanced with many processes and little free memory on one node. So the scheduler checks if too many processes are forced away from their home nodes, and if that is the case, it migrates the process and its memory to another node. Memory migration is done by using a revised

method, called *lazy migration*, published by Lee Schermerhorn in 2010. The main idea of this method is to move pages strictly out of necessity transparently from one NUMA node to another. That is, a page scheduled for migration will not be relocated unless it is accessed again. When a page migration is triggered for one or more pages, the memory is unmapped from the page table and marked in a specific manner. The next memory access to this region will cause a page fault and trigger the actual page migration for the corresponding page. This technique is an efficient way of moving a large set of pages, since only the ones that are actually used get migrated.

Furthermore, Peter Zijlstra introduced two new system calls. One call can be used to form so called *NUMA groups* by binding several threads to one group ID. The second system call binds a specified range of memory to such a group. A NUMA group will always reside on the same node, so when one member of the group is migrated to another node, all other members follow.

As opposed to the other NUMA enhancement techniques presented here, this method does not detect or keep track of the local and cross-node memory accesses. It tries to keep threads and memory on the same NUMA node and avoids situations where cross-node memory access is even possible.

2.2 Automatic NUMA Balancing

Another approach to enhance NUMA performance was presented by Andrea Arcangeli at around the same time [5],[1]. The objective stays the same: process and memory should end up on the same NUMA node to increase performance. He does not use a home node, but instead tries to determine on which node the task should run and where the memory should be migrated to. In Zijlstra's solution, the programmer has the possibility to create NUMA groups and associate certain memory regions with a group of threads. Arcangeli wants to hide all the optimization. Everything should run in background to reduce the work for the programmer.

For this purpose, he introduces a mechanism to detect memory access and gather access statistics. To support memory access detection, a new kernel thread is created, with the purpose of scanning through each process's address space, marking anonymous pages with a special flag and clearing the *present flag* in the page table entries. Upon load or store instruction to this memory region, the address translation done by the hardware will fail, since the corresponding page is not marked as present. A page fault is triggered and the operating system's page fault handler is executed. The handler function sets the present bit in the page table entry, and the interrupted task can continue to run. Furthermore, the page fault handler can determine which NUMA node the accessing process resides on and what page it tried to read or write. This information is used to gather statistics about the process and the memory page. Every process maintains an array with every entry holding a counter for one NUMA node. The length of this

array also equals the number of nodes. For every accessed page, the counter of the corresponding node is increased. On the other hand, every page also keeps track of where it is accessed from. Creating an array for every page would require too much memory, hence the statistics for the pages is coarser. Every page only remembers the last node it was accessed from. At some point, the scheduler has to decide whether it should migrate a certain process or not. Thus, the scheduler analyzes the accessed NUMA nodes for every process. The process is migrated to the node targeted by the most read or write operations. Since the memory of one process can potentially be scattered all over the (physical) address space and thus reside on different nodes, it is not enough to just relocate the processes. Therefore memory migration is also performed for achieving an ideal memory/process placement.

Automatic NUMA balancing, or short AutoNUMA, has a quite simple policy for relocating pages. As already mentioned, the NUMA node ID of the last accessing CPU is stored for every page. Should a cross-node access happen, the page is queued for migration. If the following access is initiated by another node, the migration is canceled. After the first cross-node access, two faults from the same node will eventually trigger the migration. For the migration, there is one worker thread per NUMA node for the actual memory relocation. Obviously, not every read or write operation will be intercepted and added to the statistics. The pages are periodically unmapped, so only the first access directly after the unmapping will be caught.

One downside of AutoNUMA is the memory consumption. Linux keeps one struct for every mapped page in a special data structure. AutoNUMA extends this struct to maintain the access and migration stats. But since there are a many pages in a system, the struct should be kept as small as possible. Adding new field leads to a much larger memory footprint for the kernel.

2.3 Current State

After none of the two competing approaches prevailed, Mel Gorman also published a patch set in the hope of building a basis for NUMA optimization where different policies could be added to [6],[9]. Eventually, his patches ended up in the Linux main tree[1], though with changes and addition of many contributors including Zijlstra and Arcangeli. Gorman called his patch set "Foundation of automatic NUMA balancing". The patch set should, as the name suggests, be a foundation for further approaches and policies. He tried to use concepts from both previously presented change sets and merge them as far as possible. The Linux kernel[2] currently supports memory access detection using soft page faults [3]. According to this principle, pages or memory regions can be marked and unmapped to intercept write or read operations. If automatic NUMA balancing is

[1] since version 3.8
[2] www.kernel.org
[3] by clearing the present bit of the page table entries

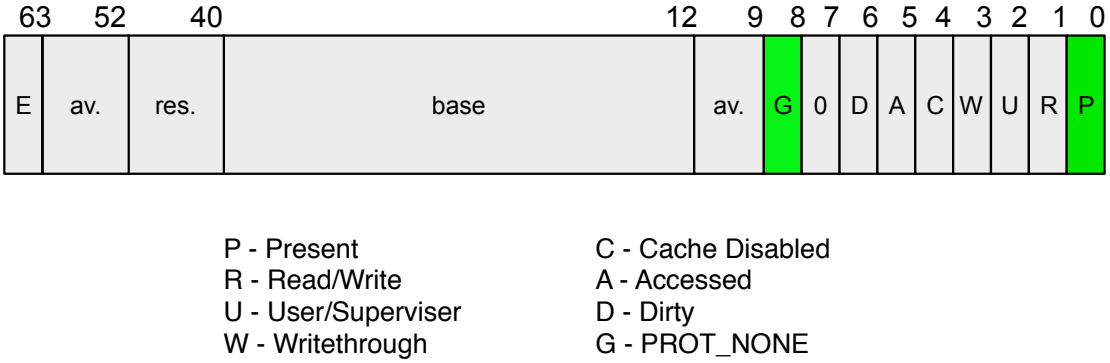

Figure 2.1: Page Table Entry

enabled for a process, the kernel periodically scans through that process's virtual address space and tags its pages as NUMA pages. To speed things up, the periodically called scan function first checks whether a process has ever been scheduled on another node than it was originally started on. If this is not the case, the process's memory probably still resides on the same node, so access detection is not necessary and would not bring any improvements. The new kernel also supports lazy migration, i.e. a page scheduled for migration will only be relocated after another (provoked) page fault.

Additionally, Mel Gorman implemented a simple NUMA policy called "move on reference of pte_numa node" (MORON). MORON serves as a temporary placeholder until a more sophisticated approach replaces it in the future. The main idea is to move a page as soon as a cross-node access is detected. Despite its simplicity, the policy still offers a performance improvement for simple access patterns.

2.3.1 Marking NUMA pages and catching page faults

In the following, we explore how the access detection is currently implemented in Linux. The detection can be done on page-level, meaning that every page table entry (PTE) can be marked individually. If the page was simply unmapped, the page fault handler will not know if it is a provoked page fault or if it is an unexpected fault. Therefore, a special flag is needed to mark the entry as NUMA-PTE. Rendering a page table entry NUMA-aware occurs it two steps. Since page table entries (Fig. 2.1) possess only a limited number of usable flag bits, the kernel developers reuse the 8th bit, known as *prot_none* and indicating that a page is present but not accessible from user space. Confusion with actual protected areas is not possible, since the flags of the virtual memory region (*struct vm_area_struct*) are checked for the *prot_none* bit prior to checking for a NUMA fault.

In the second step, the present bit in the PTE is cleared, triggering a page fault upon a subsequent read or write operation to this page. The fault handler can check whether the NUMA bit (*prot_none*) is set, and reset the present flag if that is the case.

3 Non-Uniform Memory Access in Barrelfish

Barrelfish is a multi-kernel operating system and designed to be run on many-core machines. With an increasing number of CPUs and cores, uniform memory access becomes more unlikely and performance penalties due to non-uniform memory access becomes an issue. As a result, Barrelfish might suffer from a performance drop should it run on a NUMA architecture without further optimization. To avoid this problem, Barrelfish should support NUMA-aware memory allocation, memory migration and other enhancement for increasing the performance on NUMA systems. So far, Barrelfish offers functionality to set the affinity for allocation. By providing a base address and an address limit, the desired memory region for future allocations can be defined. Since memory of different NUMA nodes is distinguished by its physical address, selecting an address range for memory allocation equals selecting a NUMA node for the allocation. However, finding the appropriate memory range for a NUMA node as well as finding the CPUs node ID is still a cumbersome task. Therefore, creating a simple and useful interface for these NUMA-related queries is a goal of this thesis.

Currently, Barrelfish lacks support for memory relocation. Once allocated, a block of memory cannot be moved to other NUMA nodes for optimization or other purposes. The additional functionality for moving pages to other physical addresses is desirable and is another objective of the proposed changes for Barrelfish.

The third enhancement of Barrelfish presented in this thesis is a mechanism to detect memory access, allowing spotting cross-node memory operations and therefore potentially ill-placed pages.

3.1 Barrelfish in a Nutshell

Prior to describing our proposed NUMA optimization, we give a brief overview of Barrelfish's architecture with emphasis on its memory subsystem. For detailed information we recommend referring to the Barrelfish website[1]. Unlike traditional operating systems, Barrelfish runs one kernel per core, called cpu-driver. The cpu-driver is responsible for scheduling dispatchers (nearest equivalent to UNIX processes), handling or forwarding page faults, traps and exceptions and maintaining and modifying capabilities. Capabilities are OS resources and can only be directly accessed and manipulated by the cpu-driver. Capabilities can for example represent a block of typed memory. In this thesis, we will mainly use frame capabilities, which are a subtype of RAM capabilities. A frame capability is basically a handle to a block of physical memory that can be mapped into a domain's virtual address space. Other capability types include device frames (access to memory-mapped devices), cnodes (region of memory containing capabilities), vnodes (memory for page tables, page directories etc.), and others. The capability types, their usage and further information about capability management is described in Mark Nevill's Master's thesis "An Evaluation of Capabilities for a Multikernel" [13].

A user application can be referred to as user domain. Usually, a domain has one dispatcher running on one core. The dispatcher is the unit of kernel scheduling and manages the domain's threads. If a domain is spanned across multiple cores, one dispatcher per core is deployed. Each dispatcher is responsible for the threads running on its core.

A relevant subsystem for this thesis is the virtual memory. It contains several components and data structure which we describe here briefly. Detailed information about Barrelfish's virtual memory can be found in Simon Gerbers Master's thesis "Virtual Memory in a Multikernel" [8].

The vspace is an object representing the virtual address space. It contains a list of all mapped vregions and a reference to the pmap. The pmap is the architecture dependant part of Barrelfish's virtual memory subsystem. Amongst other content, it contains functions for manipulating the page mappings (map, unmap, modify flags), information about the address space and the root of the vnode tree.
Memory regions containing a page table or page directory are called vnodes. It is important to differentiate between the vnode capability and the vnode struct used in the user domain. Vnode capabilities are the memory blocks which contain the architecture dependant data structure [2] used by the memory management unit to translate virtual to physical addresses. Each vnode capability has a corresponding vnode struct in the user domain. This struct contains a reference to the capability and is part of the vnode tree rooted at the pmap. A level in this tree corresponds to a level in the page table. Inner vnodes represent the page tables, page directories et cetera. Leaf vnodes on the

[1] `http://www.barrelfish.org`
[2] for example PML4, PDPT, PDIR and PTABLE in X86

other hand represent the entries in the page table. Hence using a 4-level page table leads to a 5-level vnode tree. Instead of creating one vnode for every page table entry, there is one leaf vnode per mapping. Mapping a 64 KiB frame would result in a leaf vnode covering 16 page table entries[3]. Leaf vnodes contain a reference to the mapped (frame-) capability as well as meta-information about the mapping like mapping flags and number of mapped PTEs.

Architecture independent concepts of Barrelfish's memory system are vregions and memory objects. A contiguous block of virtual memory is described by a vregion. It is associated with exactly one vspace and one memory object. A memory object also represents a block of virtual memory, but can consist of several vregions. Moreover, a memory object can contain a reference to the capability that is mapped at its covered virtual address range. Memory objects are divided into types, e.g. a type *one_frame* containing exactly one frame respectively its capability. Another type is *anonymous*. Memory objects of this type contain a list of frames that are mapped contiguously into the virtual address space. For this thesis, we will mainly use anonymous memory objects.

Figure 3.1 depicts Barrelfish's memory system and the components used in this thesis. While the diagram shows the most important objects and their relation, some connections were omitted for simplification.

3.2 Memory Allocation

In order to allocate a block of memory on a specific NUMA node, it is essential to find the corresponding physical memory range assigned to this node. Moreover, the CPU affinity should be known so as to be able to allocate local memory. Both the memory ranges and CPU affinities are stored in Barrelfish's system knowledge base (SKB) which mirrors the concept of a database about the hardware, populated at startup.

3.2.1 Initialization

Since querying the facts from the SKB for every NUMA-related operation is inconvenient and slow, we implemented an initialization method for the NUMA library. The core idea is to fetch the relevant information and store it for future use. An internal data structure is created containing all NUMA nodes, their physical base addresses and their sizes. Furthermore, the associated cores are stored in form of a bitmap for every node. An additional function allows for querying the NUMA node ID of a given core. Taking the core ID as argument, the function compares the bitmap of every node against the core number and returns the matching node's ID.

[3] assuming 4 KiB pages

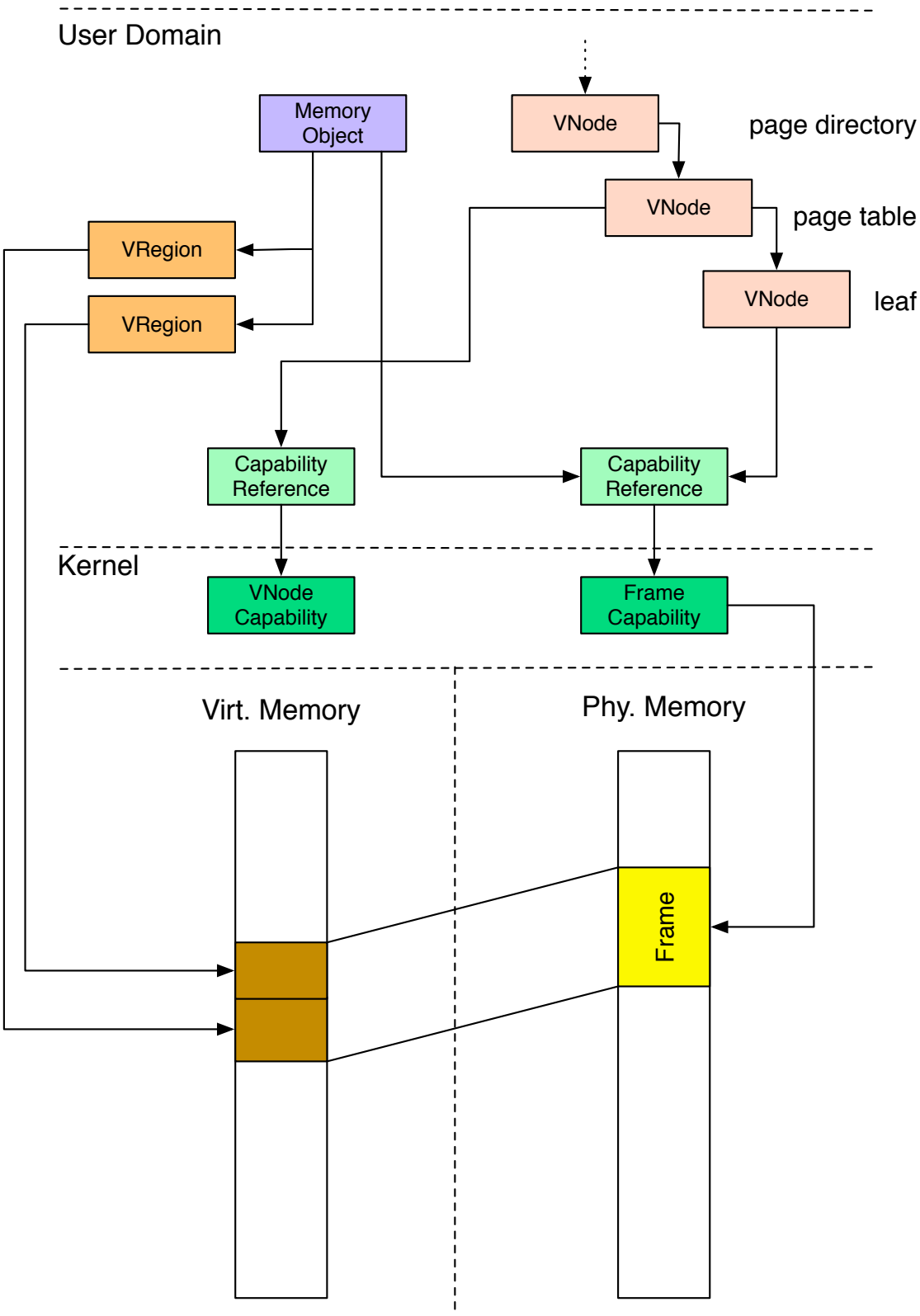

Figure 3.1: Barrelfish's Virtual Memory System

3.2.2 Allocation

The memory allocation logic is embodied within two new functions, which can be used to get a block of physical memory on a specified node or on the local node respectively. Both functions return a frame capability for the newly allocated block.

First, the current values for the memory affinity are stored so they can be reset later. As mentioned previously, a function for setting the affinity for memory allocation already exists. This function is used to set the allocation affinity according to the desired NUMA node. Next, a frame capability is obtained by calling the standard ram allocation method. Finally, the previous values for the ram affinity are restored to prevent any side effects.

3.2.3 Wrapper Functions

With this additional functionality described above, a basis for the new NUMA library is created, and NUMA-aware memory allocation is feasible. For convenience reasons, two wrapper functions allowing for allocating and already mapping a block of memory are also included in the library. The virtual address of the newly obtained and mapped block is returned. The wrapper function serve an additional purpose besides merely simplifying the process of NUMA-aware memory allocation. As we will discuss later, memory migration can only be done on the granularity of capabilities. Allocating a large portion of memory leads to a large capability. Migrating such an entire memory block to another NUMA node is slow, inefficient and unsuitable for NUMA optimization. Memory migration should be done on a reasonably small scale. As the memory access detection mechanism described later in this thesis works on the granularity of leaf vnodes, the movable capabilities should not exceed the maximal leaf vnode size.

To meet these requirements, we decided to split the requested memory block into smaller chunks of a defined size. The exact chunk size leading to the best performance is yet to be analyzed and may be subject to further investigations. So as to allocate a large block of memory, the wrapper function splits the request into smaller frames which are then allocated and combined into an anonymous memory object. To identify vnodes which were mapped with this wrapper function, we introduce a new vnode flag *movable* (Fig. 3.2).

Furthermore, the NUMA node ID of the allocated memory is stored in the mapping of each chunk. This information is later used for cross-node access detection and especially for the NUMA optimization policies.

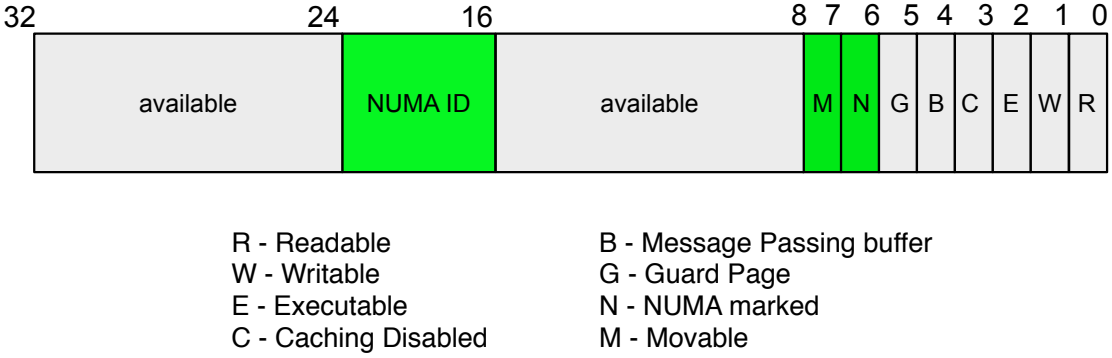

Figure 3.2: VRegion/VNode flags

3.3 Memory Migration

To improve performance on a NUMA system, migration of memory from one node to another is sometimes necessary. The reason for migration might be a relocated process or a changing access pattern of a multi-threaded and over several core spanned application. For our NUMA library, we want to provide the possibility of migrating a given memory region. In theory, the steps for relocating a block of memory are as flollows:

1. Allocate a new block of memory on the desired node
2. Copy the data from the old to the new location
3. Map the newly allocated memory on the old location's virtual address
4. Delete (free) the old memory region

However, in reality the process is more complex, especially when the already existing memory subsystem is not designed to support these actions. To elaborate how the migration is done in Barrelfish's new NUMA library, some details and difficulties are described in the following section.

3.3.1 Preparation

As previously explained, we only support the migration of one capability. Allowing relocation of a fraction of the capability would have undesirable implications. Since a frame capability needs to be a contiguous block of memory, migrating only a part of the frame is not an option. Splitting a capability into smaller chunks and migrating only one of them would work, but has some ugly side effects. For the scope of this thesis we therefore allow only migration of memory which was allocated by the above described wrapper function.

The first step is the easiest and straight forward. To obtain a new memory location for the relocation, we request a new frame capability on the target node using our previously implemented allocation function. Since we have to copy the pages from the old

to the new location, the new frame capability has to be mapped somewhere in the virtual address space. Although the frame will eventually be mapped on the same virtual address as the source, we need to find another address for now, so that both the source and the target frame are accessible in the current domain.

3.3.2 Duplication

Copying the pages from one to another location is achieved by the regular memcopy function. Yet some possible issues have to be considered in the context of memory migration. When some pages are migration from node A to node B, the data on these pages must not be altered from the point where they are copied until the point where the pages are remapped on the same virtual address as before. In other words, we cannot allow write access to the memory region we are about to relocate. If this requirement is neglected, pages on the old NUMA node might get changed after they are copied and the write operation gets lost. Therefore, we have to remap the source capability read-only. A write access by any thread to this region will cause a page fault.

However, while solving one problem we created another: The accessing thread will trigger a page fault and might not know how to resolve it. Let us explore Linux's solution to this problem. Keep in mind that Linux keeps a struct for every mapped page in an internal data structure. Upon copying a page, the corresponding page struct is locked, and the page is unmapped. The lock is released when the migration procedure is completed. On the other hand, a page fault handler also tries to acquire the lock for the faulting page before handling the fault. By the time it acquires the lock, the migration is already completed, and the page fault handler can resolve the fault by mapping the page.

An equivalent to Linux's page struct are Barrelfish's leaf vnodes. One main difference is that a leaf vnode can cover more than one page. With a carefully chosen capability size at allocation time, we end up with exactly one leaf vnode for each movable capability. A page fault hander can therefore resolve the fault induced by a write access to read-only page by comparing the movable flag. If a vnode is set read-only and the movable flag is set, the handler waits until the migration procedure is finished. The migration procedure indicates its termination by resetting the writable flag. This method can be improved, especially since movable read-only memory is not allowed in this scenario. However, it is sufficient for our test.

In the first step, we mapped the newly allocated frame somewhere in the virtual address space in order to be able to duplicate the pages from the old location. Since eventually we want to map the new frame to the same virtual address the old frame is currently mapped to, we delete its previously established mapping again.

3.3.3 Mapping

In summary, the situation so far is as follows. We have two frame capabilities, one located on the source NUMA node A and one on the destination node B. Once the data from the source frame is copied, the memory looks identical on both frames. The source frame is still mapped to its original virtual address, exactly where the user application would expect it to be. The second frame on node B is not mapped at all. It is currently not accessible, neither by the user application nor by the migrating process. Therefore, the goal of this next step is to replace the old frame with the new frame. This is done by changing the page table in such a way, that entries which pointed to the old frame's physical address now point to the new frame's address.

Let us go a bit more into detail. In order to replace a frame capability, we need to locate the corresponding memory object. Since we currently allow only memory relocation for anonymous memory objects, we have a simplified situation. Nevertheless, extending this function to support additional memory object types would be easily possible. The anonymous memory object contains a list with all frame capabilities it covers. In this list, we need to find and replace the old frame located on node A with the new frame. Once this is accomplished, we can call the mapping function for this memory object, which finally invokes the kernel's functionality for changing the page table entries.

One last problem has to be solved for the seamless exchange of frame capabilities. Barrelfish does not allow overwriting an existing page mapping in the page table. The mapping function in the kernel checks the page table entries and refuses to change a valid entry. Since until now there is no use case for remapping virtual addresses, an attempt to do so is seen as error by the calling user task. In order for the migration to work, we changed this behavior and allowed modification of page table entries. In the future, a distinction between intended and unintended modification could be implemented, possibly by introducing a dedicated function for remapping.

3.3.4 Cleanup

The data is copied to the new NUMA node and the virtual address of the migrated region now points to the new frame. The user application can already access the memory on the new node and profit from a performance improvement.
As seen in Figure 3.2, every vnode holds the identifier of the NUMA node where the mapped frame is located on. Since the location of the underlying physical memory has changed, the identifier has to be updated. Furthermore, the old frame on the source node is still allocated, albeit it is not mapped and therefore not accessible anymore. In order to prevent a memory leak, the capability for this frame needs to be destroyed and the memory has to be freed for later reuse.

After updating the vnode and destroying the old capability, the migration is fully completed. Any application accessing the moved memory region continues to work normally

without observing further changes. Yet, depending on the NUMA node the application is running on, the memory access time has increased or decreased respectively.

3.4 Memory Access Detection

The functionality for page migration sets a foundation for memory access optimization. However, programmers should not have to manually optimize their programs for better NUMA performance. The mechanism should work automatically and remain hidden from the running applications. When a domain spans over several NUMA nodes with threads distributed on many cores, it is difficult to decide which node a shared data structure should be allocated on. Moreover, during the application's runtime the access pattern might change. A previously ideal memory placement might become highly problematic due to an increasing number of cross-node accesses.

To investigate the application's behavior, statistics about memory access has to be gathered. This can be achieved by using the hardware mechanism responsible for memory translation. For every memory access, the virtual address used by the software has to be translated to a physical memory address. This translation procedure is an obvious point for intercepting read or write operations. The memory translation hardware and its associated data structure, the page table, can be misused for memory access detection. We came up with two different alternatives for this task. Due to time restrictions we were only able to implement one of them, but we describe the idea of the second method in this thesis as well. The implemented mechanism is similar to Linux's method described in section 2.3.1. Since Barrelfish's architecture is quite different from Linux's monolithic kernel, some changes and adaptations were inevitable.

3.4.1 Granularity

While Linux employs access detection on the level of pages, we choose to use a leaf vnode granularity. Unlike Linux, Barrelfish does not hold information about every page in its internal data structure for the virtual memory. In Barrelfish, the lowest level in the mapping data structure is a leaf vnode. It can contain only one page, but can also consist of a set of pages. The range of a leaf vnode is determined by the size of the mapping it represents. For example if a frame of the size of 512 KiB is mapped, one leaf vnode containing 128 pages is created[4]. However, we did not want to add additional leaf vnodes for all mapped pages since it would increase the memory footprint for every application. Currently, we support only access detection for memory blocks allocated with the wrapper allocation function described in section 3.2.3. This function already splits the allocated block into reasonably small chunks and therefore leads to small leaf vnodes. The size of these blocks fixes the granularity for access detection and for memory migration. An optimal block size can be determined in the future or alternatively could be made configurable by the application.

[4] Assuming 4 KiB pages

3.4.2 Detection by Page Fault

The implemented mechanism uses intended (or provoked) page faults. The major steps for access detection are the following:

1. Mark a page as 'not present'

2. Access to this page triggers page fault

3. Identify accessing NUMA node

4. Mark page as 'present'

Since we do access detection on leaf vnode granularity, the unmapping and mapping is actually applied to a set of pages instead of a single page. First, we mark a vnode for access detection by introducing a new vnode flag, for simplicity here called *numa flag*. We set this flag in the vnode for which we want to activate access detection. Furthermore, we clear the *present* flag. To propagate these changes to the page table, we use the capability invocation for modifying the mapping flags on the corresponding frame capability. The kernel then performs the actual modification in the page table.

The memory management unit *(MMU)*, responsible for translating virtual addresses to physical addresses, assumes a page is not present if the *present* bit in the page table entry is not set. When an address inside this page is accessed, the MMU cannot translate the address and triggers a page fault. Depending on the accessed address (e.g. an invalid address), the type of fault (e.g. write on a read-only page) and also the implemented behavior, a page fault handler usually maps the missing page or terminates the faulting application. We patched Barrelfish's page fault handler to locate the vnode corresponding to the accessed address and check for the *numa* flag. If said flag is found, it means we purposely triggered this page fault for the sake of access detection. The core responsible for this access and thus the accessing NUMA node can be easily determined. What this information can be used for will be discussed in a later section. By resetting the *present* bit in the page table entry, the interrupted application can continue its execution.

Since access detection should run automatically in the background, manually marking vnodes is not a common use case. Similar to Linux, the virtual address space should be scanned periodically for misplaced memory and, if necessary, pages should be relocated. In our implementation, a periodically executed function loops through all vregions and marks the associated vnodes for access detection. However, we currently support only access detection for memory allocated with our NUMA library. Therefore we only mark vnodes with the *movable* flag.

3.4.3 Cross-Core Page Faults

The difficulty of this approach lies in the architecture of Barrelfish. As a multi-kernel operating system, Barrelfish runs one kernel on each core. If a domain is spanned over several cores, a dispatcher for this domain runs on each core and is responsible for scheduling the domain's threads on this core. While the virtual address space is shared among the threads of the domain, each dispatcher maintains a tree with vnodes. At the current state, when a mapping is created on one core and a vnode is inserted into the local vnode tree, the other dispatchers for the same domain are not updated. There is no mechanism yet for synchronizing the vnode trees. Even though the threads on all cores can access the same virtual address space, the vnode for a specific mapping can only be found on the core where this mapping was established. A mapping created once only to remain unchanged does not cause any problems. The page table entries for this mapping are set, and the MMU can translate virtual to physical addresses for all cores using this address space. However, this implementation leads to challenges and limitations for our detection mechanism.

The first limitation affects the unmapping or marking of vnodes for NUMA access detection. As explained earlier, the vspace is periodically scanned for movable memory and the corresponding vnodes. Since vnodes are not synchronized over all dispatchers of one domain, they can only be found and marked on the node the mapping was created on. Therefore the access detection is limited to the vnodes that were created on the same core as the scanning thread is running. A possible solution would include running scanning and marking threads on all cores. However, for our proof of concept we only have one marking thread on core 0. Deploying such threads on all cores could be a part of further improvements of the NUMA library. In our test cases, we use the memory allocation function only on core 0 to ensure access detection on all relevant frames.

Another problem that arises with unsynchronized vnodes concerns the page fault handler. If a set of pages is marked for access detection (*present* bit cleared), a page fault occurs on the core that tries to access a memory address inside these pages. Theoretically, this is necessary for access detection. For page faults occurring on the same core the page was previously unmapped (in our case only core 0), this works well. On the other hand, a page fault on any other core is more difficult to handle. Since the vnodes are not synchronized among the cores, the corresponding vnode for a faulting address cannot be found if it was created and marked by another core. Without this vnode, the page fault handler can neither determine the location of the memory (i.e. its NUMA node ID), nor can it remap the page and thus handle the fault. The solution we came up with includes sending a message to the core which unmapped the pages in the first place (in our case always core 0). In the future, a broadcast-like request to all cores could be used to identify the core responsible for handling the fault. Another approach would be synchronizing the vnodes across all cores, so that every core can handle the page faults, regardless of which core unmapped the pages. While sending a message for finding the responsible page fault handler adds some overhead for the access detection and page

fault handling, synchronizing the vnodes adds overhead and complexity to the memory mapping and all vnode modifications. A suitable solution for this problem might be subject to future work.

3.5 Memory Placement Policies

NUMA optimization should run completely transparent and autonomous. No input or hint from the programmer or the application should be necessary. Therefore, the NUMA library should decide automatically if an application would benefit from a memory migration. Whether memory relocation would lead to an increased performance depends strongly on the application's memory access pattern. For example, if a thread on node A accesses a local data structure and a another thread on node B simultaneously accesses the same data structure, memory migration is useless or even harmful. On the other hand, if a node A finishes traversing a local data structure, and a thread on node B continues to access the same data, migration improves the application's performance.

For different scenarios, the same migration decision has different consequences. Defining one static migration policy can be counterproductive. We decided to offer an interface for selecting and registering different policies instad of forcing a single policy. Utilizing the above described mechanism for access detection, we can gather useful access statistics and act accordingly. For every tracked access, the registered NUMA policy is called and provided with the accessed address, the corresponding vnode and the accessing NUMA node. A clever approach could comprise counting the local and remote memory operations and deciding whether migration is beneficial based on the percentage of remote access.

3.5.1 NEMO

In the scope of this thesis, we implemented a simple memory placement policy called *NUMA enabled memory optimizer (NEMO)*. It is similar to the MORON policy implemented in Linux (section 2.3). Whenever a cross-node access is detected, the accessed frame is migrated to the accessing NUMA node. This behavior works well with sequential access to a data structure, but fails for concurrent access from different nodes. The impact of this policy can be seen in the benchmark results and evaluation later in the thesis. While it is clearly not an optimal solution for many cases, it demonstrates the usage of the policy interface and the consequences for the running application. NEMO can be replaced later with a much more sophisticated and tested standard policy.

4 Evaluation

4.1 Benchmark

To test the access detection, the migration mechanism, the NUMA optimization policy and their influence on the performance, we developed an extendable memory benchmark. The goal was to simulate a realistic memory access pattern comprising several threads and local as well as remote read and write instructions. The benchmark consists of two parts: the coordinator and the actual tests. We refer to the tests as sequences, since each test is actually a sequence of different access patterns.

Sequences can be extended as desired and new sequences can be added to the benchmark easily. The controller or coordinator is the centerpiece of the benchmark. It runs the defined sequences by executing each phase consecutively, and measures the elapsed time for each phase. The gathered measurements are then added up to provide an overall time as well as detailed measurements per phase. For our performance tests, we implemented two sequences with different behavior. The benchmark can be extended at a future date if needed.

4.1.1 Sequential Access

This test consists of one initialization phase and two access phases. We measure only access phase time, since the initialization phase does not contribute any information about NUMA performance. First, a thread on the NUMA node A allocates a predefined amount of local memory. On this allocated region, a linked list is initialized and shuffled using the Knuth shuffle [12] in order to force random instead of sequential access. The access phases traverse the list twice: First, the list is traversed by reading the pointer to the next element for every element in the list. Second, the list is traversed and one byte is written to each list element. The first phase is executed on node A, so the load and store operations access the local memory. The second phase runs on node B but still accesses the same data structure, which without further optimization leads to cross-node memory access.

This simple test reveals two important conclusions. First of all, without any NUMA optimization or cross-node access detection, we demonstrate the different access times for local and remote memory. Second, with NUMA optimization enabled, the test shows the impact of the access detection and the migration and reveals the overall performance boost.

4.1.2 Cross-Node Access

The first test explored a simplified scenario targeted only at specific circumstances ignoring concurrent cross-node access. In reality, a data structure would most likely be accessed from different threads on different nodes rather than in a sequential fashion. The second test sequences attempts to include some more realistic access patterns and uses simultaneous local and cross-node memory access. Therefore, we assess not only the impact of access detection and page migration, but also the usability of the NUMA optimization policy. A policy that initiates multiple migration operations will not reduce the performance of the simple sequential access test above, since it will only trigger one memory relocation. For a more accurate and realistic access pattern with several threads and multiple of cross-node memory access, such a policy might generate some overhead caused by numerous page migration operations.

There is one initialization phase, which allocates a block of memory of fixed size on every NUMA node. A linked list is created on each node, similar to the one in the first sequence. Four access phases follow the initialization:

Local Memory Access:
All threads run on the first node and access only local memory. This should not trigger any NUMA optimization.
Shared Memory Access:
The treads are evenly distributed among all cores. All thread access memory on the first NUMA node. Impact of NUMA optimization should be minimal.
Mixed Memory Access:
The threads are evenly distributed among all cores. Every thread starts with local memory access only and increases the amount of remote memory access up to 100%.
Unfavorable Memory Placement:
The threads are evenly distributed among all cores. Every thread reads memory from its neighbouring NUMA node. Page migration can establish an ideal memory placement.

4.2 Results and Analysis

To compare Linux's NUMA optimization and the implementation presented in this thesis, we used the same machine for running the benchmark on Linux and Barrelfish. We executed the two sequences with and without NUMA optimization extension for both operating systems. Additionally, we also tested AutoNUMA on Linux. Each test was executed 10 times. The measurements presented in this section are the averaged values.

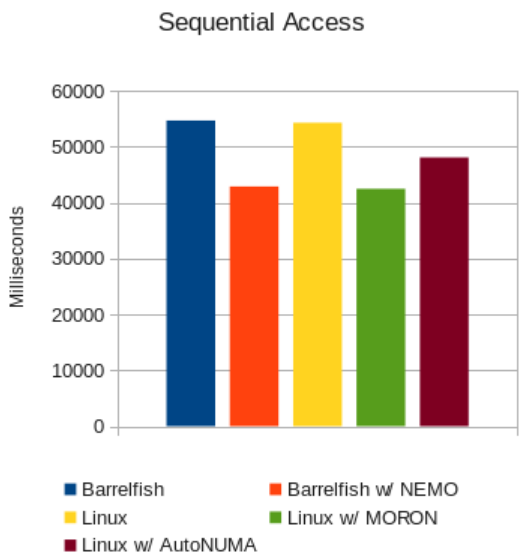

Figure 4.1: Overall results of Sequential Access benchmark sequence

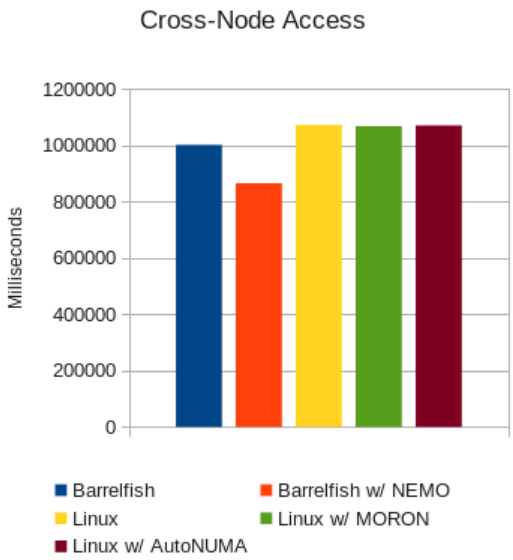

Figure 4.2: Overall results of Cross-Node Access benchmark sequence

4.2.1 Overall Performance

First, we analyze the overall performance for each sequence. The elapsed time was measured in milliseconds for every phase of a sequence, excluding the initialization and setup phase. The overall performance of a sequence is given by the sum of the execution time for each phase. The results for the two sequences can be found in Figure 4.1 and Figure 4.2. A shorter execution time denotes a better performance.

As seen in Figure 4.1, the first test sequence benefits from the optimization. In this test, the memory is first accessed by a local thread and then by a tread on a different node. Since the access pattern does not include concurrent access from different nodes, the memory is moved exactly once. Barrelfish's NEMO policy as well as Linux's MORON policy increase the sequence's performance by roughly 20%, while AutoNUMA increases the performance by 10%. The difference between MORON and AutoNUMA is due to the slower reaction time of AutoNUMA. Three consecutive page accesses are required to initiate a page migration. With a timer interval of 5 seconds, there is a delay of 10 seconds after the first cross-node access.

Figure 4.2 shows the results of the second benchmark sequence. The sequence consists of 4 phases with different access patterns. Since several threads compete for the memory, NUMA optimization is not as simple as in the first test. This test shows the usability of a NUMA migration policy in a more realistic scenario. As depicted in the diagram, the test runs slightly faster on Barrelfish than on Linux. A big difference to

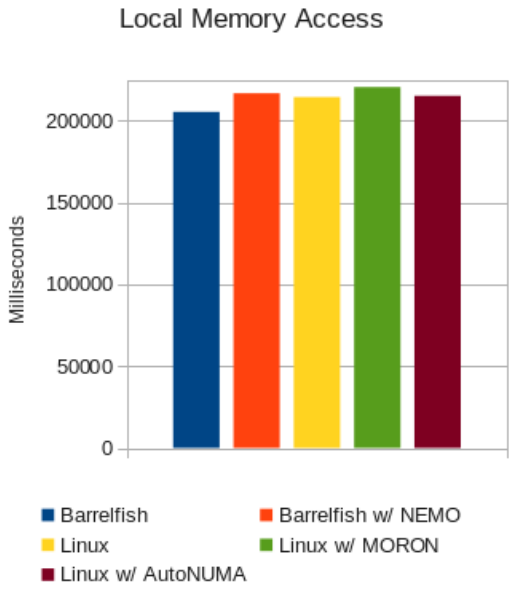

Figure 4.3: Results of Local Memory Access

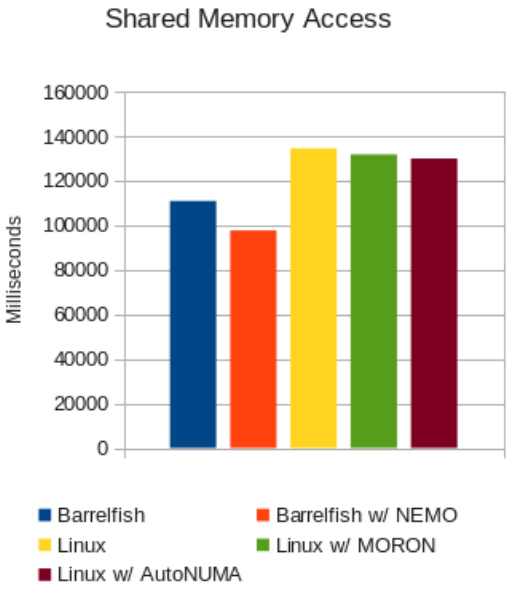

Figure 4.4: Results of Shared Memory Access

the previous test is the increased number of threads running simultaneously on every core. We suppose the reason for the increased runtime is the different algorithm for thread scheduling in Barrelfish and Linux. Because it is not relevant for our evaluation, we did not investigate the issue any further. In the following, we examine each phase of the second test sequence individually and analyze how the different NUMA policies influence its performance.

4.2.2 Performance per Phase

Figure 4.3 shows the measurements of the first benchmark phase. Each core runs multiple threads that are reading a local data structure. Since all data structures are local to the threads, no cross-core accesses are performed. This is a best-case scenario for an application; therefore no memory migration is required. Nevertheless, it is an interesting phase, because it shows the impact of access detection on the performance. The overhead introduced by the access detection in Barrelfish is roughly 5%. Linux's detection mechanism adds an overhead of 3% while AutoNUMA increases the runtime by less than 1%. The relatively large overhead in Barrelfish is induced by the page fault handling, especially on cores where messages are required (see section 3.4.3). Barrelfish's detection mechanism clearly leaves room for improvement.

Figure 4.4 displays the results for the Shared Memory Access phase. We did not expect much improvement in this test phase, since the same data structure is accessed

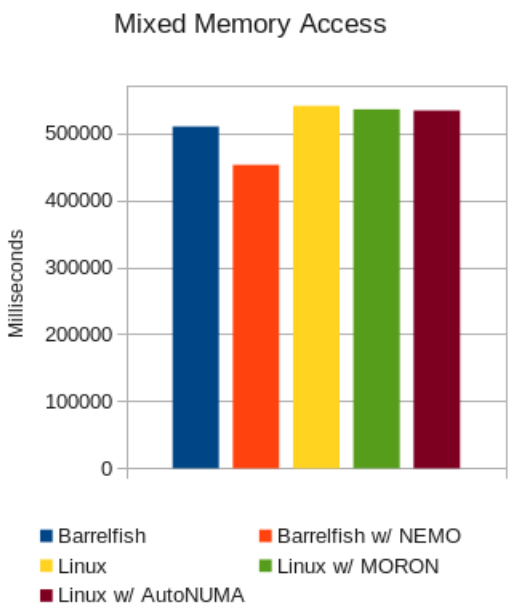

Figure 4.5: Results of Mixed Memory Access

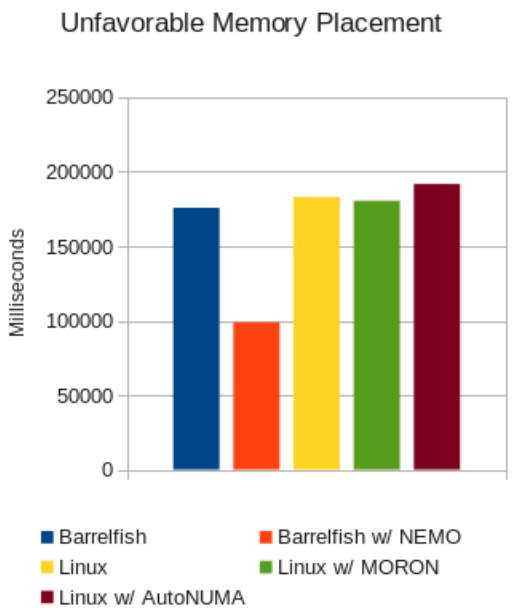

Figure 4.6: Results of Unfavorable Memory Placement

by multiple threads on different cores. Due to many cross-node accesses, we expected the memory to be copied recurrently from one node to another and back, leading to an increased phase runtime. However, the tests show that the performance actually increases with NUMA optimization. Even though the memory is copied repeatedly, the application benefits from the relocations. Our interpretation of the situation is as follows. Assume two nodes A and B, one of them, say A hosting a data structure. While the threads on node A can access the data structure locally, the threads on B have a longer access delay. Therefore, all threads on node A finish presumably earlier than the threads on node B. Since the benchmark measures the time until all threads are finished, the threads on B are decisive. If the memory is copied back and forth on the nodes, each thread has partially local memory access and partially remote memory access. This behavior leads to a longer runtime for threads on node A, but a shorter runtime for threads on node B. Moreover, our implementation of the memory migration never completely unmaps the frame until it is copied. Since the benchmark only reads the data structure, no thread has to be stopped for the memory relocation. Overall, every thread has roughly the same runtime, thus the measured time for the phase decreases by approximately 10%. Linux on the other hand unmaps a page completely for relocation, therefore the memory is inaccessible for the duration of the migration. This might be the reason for the smaller performance boost on Linux.

Figure 4.5 shows the impact of NUMA optimisation on the Mixed Memory Access bench-

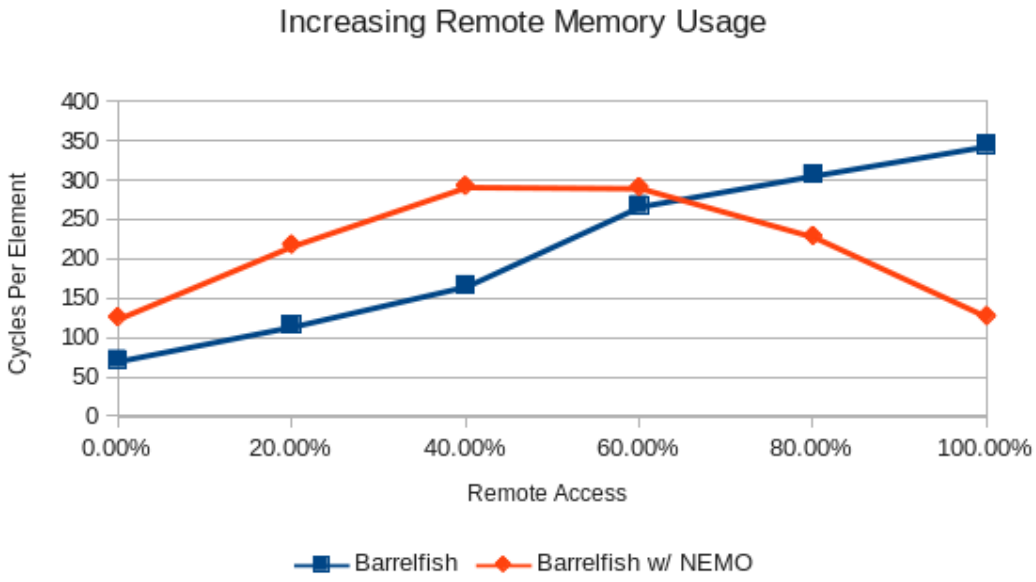

Figure 4.7: CPE for Unfavorable Memory Placement

mark phase. In this phase, there is a data structure stored on each node. Threads start with local access only and increase the percentage of remote memory reads over time. In the beginning, there are no cross-node reads, thus no memory relocation is required. In the end, there is only remote memory access, so one memory relocation per node is enough to establish a situation similar to the initial condition. If the access is not limited to the local node, optimization is more difficult and similar to the Shared Memory phase described above. In Figure 4.7, the measured cycles per element (CPE)[1] for the Mixed Memory Access phase are illustrated. Without optimization, the CPE increases with an growing amount of remote memory reads. The CPE can be reduced when NUMA optimization is active. The worst case scenario comprises all CPUs accessing both remote and local memory with an equal probability of 50%.

[1] Element in the list which is traversed by the benchmark

Figure 4.6 shows the results for the test phase with unfavorable memory placement. In this phase, threads on every core access memory on a neighbouring NUMA node. While this is a suboptimal situation, the memory placement can easily be fixed by the NUMA policy. NEMO can increase the performance of this test by approximately 40%. As depicted in Figure 4.8, the overhead induced by remote memory access is roughly 40%. Therefore, this last phase demonstrates the best possible performance improvement.

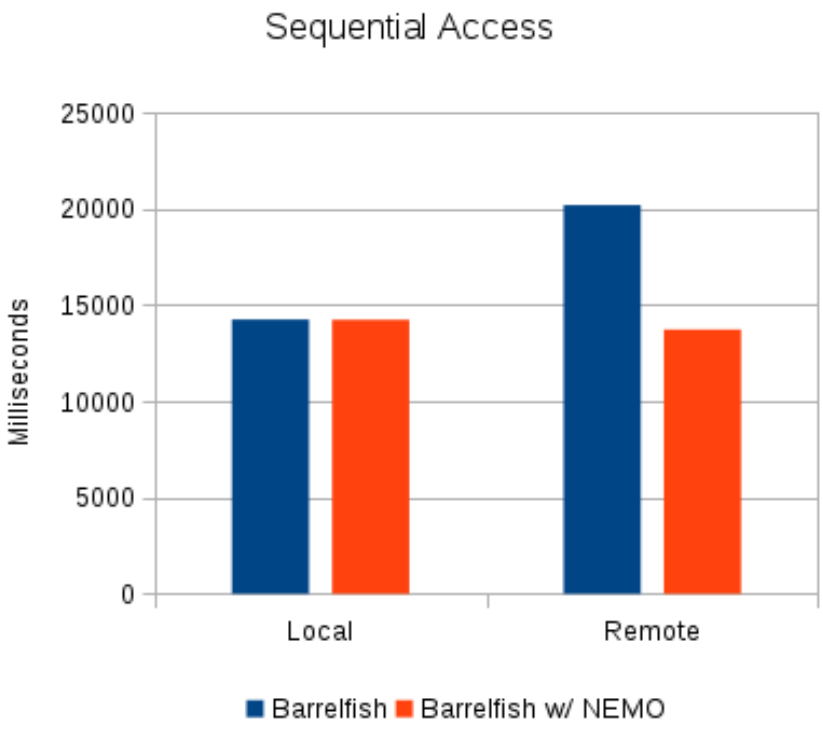

Figure 4.8: Difference for local and remote memory access

4.3 Known Issues and Limitations

The tests show a reasonable improvement in performance for memory access on a NUMA system. Nevertheless, the current implementation of the access detection and memory migration has some restrictions. For the benchmark, some assumptions had to be made which would not be guaranteed in a real-world application.

4.3.1 Limitations of Access detection

Access detection by intercepting page faults works well and does not add much overhead. Yet, the usage of the mechanism currently implemented is limited. First of all, as already mentioned earlier, there is only one thread for marking the vnodes. Moreover, only vnodes created on the same core as the scanner thread runs on are marked. This restricts the access detection to memory allocated by the first core only (or whatever core is running the scanner thread). The second limitation concerns the page fault handler. When no vnode for a faulting address can be found on the local core, the page fault handler lets core 0 handle the page fault and waits until the corresponding pages are mapped again. To avoid any problems with the access detection mechanism, the NUMA memory allocation function should be called on the same core as the scanner thread is running.

4.3.2 Limitations of Memory Migration

Memory migration suffers from the same problem as access detection. For relocation a block of memory, access to the corresponding vnodes is essential. Since the vnodes can only be found on the core the mapping was created on, the migration function may only be called on the same core as the original allocation was done. Furthermore, migration only works on capability granularity. It is not possible to migrate a fraction of a frame or a memory range spanning over several frames. To enforce the block size limitations, only memory allocated with the wrapper function described in section 3.2.3 is allowed for migration.

5 Future Work

Currently, there are some limitations for using the access detection and memory migration mechanisms. The core for scanning and allocating memory has to be chosen carefully. Moreover, the function for memory migration is also tied to the core where the memory was allocated. There are several possible improvements or optimizations for the implemented library:

Page Fault Broadcast:
A technique for finding the core which unmapped a certain page would allow to unmap pages on all cores. A page fault handler would for example send a broadcast to all cores and one core would eventually handle the fault and notify the sender. This is a necessary condition for running scanner threads for access detection on all cores.

Scanner Threads:
Deploying a thread on each core for scanning and marking leaf vnodes would allow to monitor all potentially movable frames. With this setup, every core is capable of unmapping pages. Therefore it is essential that page faults can be directed to the core which is responsible for unmapping the page (for example by finding the responsible core using a broadcast).
Another optimization includes the timer interval for the scan. At the moment, a scan is performed every 3 seconds. By experimenting with different intervals, an optimal trade-off between page fault overhead and access time improvement may be found.

Core-Independent Migration:
In the current implementation, the core that established a mapping can move the memory affected by this mapping. Allowing every core to migrate any memory would require a similar technique like the page fault broadcast. The core calling the migration function would request the appropriate core responsible for this specific memory block to execute the migration.

Memory Block Size:
At the moment, NUMA memory is split into 1 MiB blocks. This is the granularity for access detection and memory migration. Modifying this block size may change the overall performance and memory consumption of an application. Smaller blocks lead to a bigger memory footprint, since more capabilities and leaf vnodes are created and stored. It also leads to more page faults, thus the page fault handling overhead increases.

On the other hand, larger memory blocks would allow more fine grained access detection, accurate memory placement and presumably fewer move operations. Experiments could reveal a convenient size. Another possibility would be an adaptive block size.

Migration Policies:
Our implementation for access detection allows to register a custom policy handler. The policy would decide at what point a frame should be migrated. In the future, more policies could be added to find a good standard policy and to give the chance to adapt the policy to different use cases.

Lazy Migration:
Similar to Linux, Barrelfish could offer a lazy migration policy. Memory blocks scheduled for migration would not be moved instantly but rather unmapped and marked for migration. Upon the first access originated from the target node, the frame would be migrated. With this technique, memory would only be migrated if it is ever used again by the target node.

5.1 Page Table Access in User Domains

In 2012, a research group from Standford University presented a paper about a project called Dune [3]. The goal of Dune is to provide applications with access to hardware features in a safe manner. It enables an application for example to access the page table directly, without having to make a call into the kernel. This is made possible by using the virtualization hardware in modern processors. The access detection mechanism described in this thesis could benefit from such a direct and safe access to the paging hardware. Marking vnodes for detection and especially clearing the *present* bit in the page table entries would not require an invocation to the kernel anymore. It could be done by the application directly, without having to ask the kernel to do it. Of course the same is true for remapping the marked pages in the page fault handler. The overhead introduced with the access detection by page faults could be reduced to a minimum.

5.2 Alternative Detection Mechanism

In the following section we present a different approach for access detection. It also uses the page table, but avoids page faults completely. Due to time constraints we were not able to implement this method. However, we shall describe here the basic idea behind it, since it would be an interesting project for the future. Avoiding page faults has several advantages. First of all, page fault handling takes time and the application (or at least the thread that caused the page fault) is stopped for that duration. Without any page faults, the running thread does not get interrupted for access detection. Second, a method for requesting other cores to handle page faults as described above is not needed. The amount of messages necessary to track memory operations is minimized.

Instead of soft page faults, the page table entry's *accessed* bit is used. This bit indicates, that the page was accessed at least once. The page table would be scanned periodically and the *accessed* bit would be read and reset for the relevant entries. This would provide information about what pages were recently (during the timer interval) accessed. With the above suggested technique for providing user domains (restricted) access to the page table, reading and clearing this bit could be done quite fast. Since the bit only reveals if a page was accessed or not, but does not allow to determine the accessing core or NUMA node, further adaptations would be needed. Identifying the origin of a memory operation is one of the challenges in this approach. It could be done by duplicate the page table. Currently, there is one page table per domain, even if the domain is spanned across several cores. By duplicating the page table, each core could have its own page table for memory translation. This would allow to detect page access for every core separately. The per core information would be accumulated by using an echo-algorithm as described by Ernest Chang in "Echo Algorithms: Depth Parallel Operations on General Graphs" [4]. The collected data would provide statistics about what core (or NUMA node) accessed what pages.

Aside from the fact that the memory consumption would be increased by using multiple page tables per domain, one problem remains yet to be solved. All page tables of the same domain need to be consistent. So synchronizing the tables would be necessary. When one core changes an entry, either by mapping or unmapping a block of memory or by changing the permission flags, all other cores need to adapt these changes.

6 Conclusion

In the scope of this thesis we introduced a new approach to improve the performance of applications running on Barrelfish on a system with non-uniform memory access. We extended Barrelfish with three different functionalities, and provided an extendable benchmark for testing the performance impact.

First, we introduced a method for allocating memory on a given NUMA node. Provided with a node ID, the implemented library determines the matching memory range for this node, and allocates a frame of desired size in this range. Moreover, the library allows for allocating memory on the local NUMA node by identifying the caller's node ID and executing NUMA allocation on this node. Second, we introduced a technique for relocating a block of physical memory. The virtual addresses pointing to this address region remain unchanged for this relocation, thus the procedure is completely transparent for the user application. In order to migrate a memory frame to another node, the content of the frame is copied to the new location, the page table is modified to map the new instead of the old frame, and the old frame is freed. Third, we implemented memory access detection for Barrelfish. The detection mechanism allows for tracking an application's memory operations. Additionally, the mechanism provides a hook for placement/migration policies. The policy gathers information about local and remote (cross-node) memory operations and decides when a frame should be moved. We implemented a policy called NEMO which relocates a memory range whenever a cross-node access is detected.

To test access detection, memory migration and our policy, we implemented a benchmark with a changing memory access pattern. The benchmark is extendable, so that different access patterns and tests can be added in the future. The benchmark may also help to develop and test new, more sophisticated placement policies. As our benchmark has shown, the policy introduced in this thesis can improve the performance of memory accesses by a significant amount. While the overhead induced by the access detection is small, it could be reduced even more by improving the page fault handling used by the detection mechanism or avoiding the page faults altogether.

A Benchmark Data

A.1 Hardware used for Testing

All tests and benchmarks were executed on a X86_64 machine running a 64 bit version of Barrelfish or Linux respectively. The system has two NUMA nodes with one dual core CPU on each node.

Hardware
CPU: 2x Dual-Core AMD Opteron^{TM} 2220
Frequency: 2.8 GHz
L1 Cache (per core): 64+64 KiB (i-cache and d-cache)
L2 Cache (per core): 1 MiB
Main memory: 8 GiB (4 GiB per node)
Linux: Kernel 3.9 (MORON), Kernel 3.3 (AutoNUMA)

A.2 Benchmark Results

The execution time per phase and sequence is measured in milliseconds. The following table lists the averaged measurements and the standard deviation of all tests.

	Barrelfish		NEMO		Linux		MORON		AutoNUMA	
	avg.	σ	avg.	σ	avg.	σ	avg.	σ	avg.	σ
Local Read	14260	3	14236	3	13502	189	13993	945	13356	260
Remote Read 1	20200	6	14891	8	20378	334	14396	284	19707	390
Remote Read 2	20199	4	13732	4	20381	338	14061	180	15000	772
Total	54659	12	42859	12	54260	860	42449	485	48063	1181
Local Phase	205262	374	216654	3492	214300	1611	220415	2418	215129	3872
Shared Phase	110882	548	97666	1493	134532	1269	131781	947	129940	4339
Mixed Phase	509580	4187	452319	7392	540287	4917	535165	2919	533360	11553
Unfavorable	175763	6968	98907	2740	183069	1149	180505	578	191803	11020
Total	1001487	9078	865546	7819	1072188	8669	1067866	4761	1070233	24117

B Glossary

The structures, concepts and terms of the Barrelfish OS used in this thesis are listed below. Information about the operating system's architecture can be found on the official homepage[1].

Capability: An object representing OS resources (e.g. physical memory or interrupt routing tables). Capabilities are located on protected memory and can only be accessed or modified by the cpu-driver. Since capabilities are managed by trusted code, they are safe from unauthorized access or malicious modification.

Capability Reference: An object used in user domains for referencing a capability. The Capability itself can only be accessed and modified by the cpu-driver.

CPU-driver: Kernel in Barrelfish. Each core runs a cpu-driver, responsible for dispatcher scheduling, capability management and interrupt, trap and exception processing. No state is shared between the cores.

Dispatcher: The unit of kernel scheduling. It manages the threads of its domain on one core. A domain has usually one dispatcher. If the domain is spanned over multiple cores, it has one dispatcher per core.

Domain: An application or service in Barrelfish. Threads in a domain typically share the same virtual address space.

Frame: A block of contiguous physical memory of various size.

Memory Object: An object representing a block of virtual memory. It consists of one or more vregions. Unlike vregions, memory objects are typed. Two examples of memory types are frames and anonymous memory.

MMU: Memory Management Unit; Hardware component which translates virtual addresses into physical addresses by using a page table. The MMU also checks access permissions for memory pages and page flags.

NUMA node: A part of a NUMA system containing one or multiple processor(s) and some memory. A processor can access all memory addresses on the same node equally fast. The access time differs between memory on the local node and memory on a remote node.

Page: A block of contiguous physical memory with a fixed size (4 KiB for normal pages). Virtual memory can be mapped with the granularity of pages.

[1] http://www.barrelfish.org/

Page Table: Data structure used for address translation. The table comprises one entry for each mapped page in the virtual address space. Additionally to the translated address, the entry also contains flags for access permissions, user/superviser protection and cache options. The page table usually has multiple levels for fast and space-saving translation of addresses in a large and sparsely populated address space.

PMap: An object representing the mapping of virtual to physical addresses. It contains the vnode tree's root node and various functionalities for mapping and unmapping virtual addresses or modifying the mapping flags.

SKB: System Knowledge Base; A database for information about the system (e.g. number of cores, memory affinity,..). It runs as a system service on one of the cores and can be accessed by all applications.

VNode: Either a level of the page table (page directory, page table,...) or a set of page table entries.

VNode Tree: A data structure containing all vnodes of a virtual address space. Each level of the tree represents a level in the page table. The Leafs of this tree represent one or more page table entries and contain additional information about the mapping and a reference to the mapped physical frame.

VRegion: An object representing a block of contiguous virtual memory. The vregion does not comprise information about the mapping of virtual memory.

VSpace: An object representing the virtual address space. A vspace holds a list of all vregions in the address space and has reference to a pmap.

Bibliography

[1] Andrea Arcangeli. Autonuma alpha10. `http://lwn.net/Articles/488686/`, March 2012.

[2] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 29–44, New York, NY, USA, 2009. ACM.

[3] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe user-level access to privileged cpu features. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation*, Stanford, CA, USA, 2012.

[4] Ernest J.H. Chang. Echo algorithms: Depth parallel operations on general graphs. *The IEEE Transactions on Software Engineering*, SE-8:391–401, 1982.

[5] Jonathan Corbet. Autonuma: the other approach to numa scheduling. `http://lwn.net/Articles/488709/`, March 2012.

[6] Jonathan Corbet. Numa in a hurry. `http://lwn.net/Articles/524977/`, November 2012.

[7] Jonathan Corbet. Toward better numa scheduling. `http://lwn.net/Articles/486858/`, March 2012.

[8] Simon Gerber. Virtual Memory in a Multikernel. Master's thesis, ETH Zurich, 2012.

[9] Mel Gorman. Foundation for automatic numa balancing. `http://lwn.net/Articles/523065/`, November 2012.

[10] Andi Kleen. libnuma/numactl and NUMA API release notification. `http://lwn.net/Articles/67005/`, January 2004.

[11] Andi Kleen. A numa api for linux. `http://www.halobates.de/numaapi3.pdf`, August 2004.

[12] Donald E. Knuth. *The art of computer programming, volume 2 (2nd ed.): seminumerical algorithms*, pages 139–140. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[13] Mark Nevill. An evaluation of capabilities for a multikernel. Master's thesis, ETH Zurich, 2012.

[14] Andrew S. Tanenbaum. *Modern Operating Systems*, pages 529–531. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2007.

[15] Peter Zijlstra. sched/numa. `http://lwn.net/Articles/486850/`, March 2012.