Karlsruher Institut für Technologie

Runtime Effects of Memory Sharing and Deduplication

Bachelorarbeit
von

cand. inform. Michael Skinder

an der Fakultät für Informatik

Erstgutachter:	Prof. Dr. Frank Bellosa
Betreuender Mitarbeiter:	Dipl.-Inform. Marc Rittinghaus

Bearbeitungszeit: 15. Mai 2012 – 15. September 2012

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Karlsruhe, den 15. September 2012

Deutsche Zusammenfassung

Hauptspeicher ist eine knappe Ressource. Vor allem im Serverbereich ist die Anzahl laufender Applikationen durch die Menge an Hauptspeicher beschränkt. Dort werden auf einem physischen Server meist viele virtuelle Server betrieben. Da hierbei meist eine schnelle Antwortzeit, aber keine große Rechenleistung nötig ist, limitiert der Hauptspeicher, in dem die Anwendungen gehalten werden müssen.

Um den Hauptspeicher effizienter nutzen zu können sind verschiedene Techniken entwickelt worden, welche den Hauptspeicherverbrauch senken sollen. Hierbei sind die klassischen Techniken wie “shared libraries” oder der “buffer cache” zu nennen. Außerdem gibt es noch aktive Deduplizierungs-Techniken, welche den Hauptspeicher nach redundanten Inhalten durchsuchen und diese entfernen. Ohne diese Techniken kann sich, je nach Umgebung und Aufgabe des Systems, bis zu 86% redundanter Inhalt im Hauptspeicher befinden (vgl. [12]).

Bisher wurden jedoch nur die Hauptspeicher Einsparungsmöglichkeiten dieser Techniken untersucht. Diese Arbeit soll die Laufzeiteffekte der oben genannten Techniken untersuchen. Hierfür werden auf einem NUMA System verschiedene Benchmarks so ausgeführt, dass sich diese mal Inhalte im Hauptspeicher teilen können und in einem weiteren Durchlauf das “Sharing” unterbunden wird. Dies kann auf einen einzelnen NUMA Knoten beschränkt, oder so gesteuert werden, dass der Benchmark den entfernten Speicher nutzen muss. Dadurch kann sowohl “Sharing” auf einem UMA als auch auf einem NUMA System betrachtet werden.

Diese Arbeit zeigt, dass “Sharing” kaum Einfluss auf die Laufzeit von Programmen auf UMA Systemen hat. Im Idealfall wird zwar die Performanz gesteigert aber dies ist bei den meisten Applikationen kaum wahrnehmbar. Anders sieht es jedoch auf NUMA Systemen mit ihrem entfernten Speicher aus. Hier kann es zu einem Einbruch der Leistung um den Faktor zwei kommen. Eine Deduplizierung von Speicherseiten über die Grenzen einzelner NUMA Knoten hinweg, erweist sich daher in den meisten Fällen als kritisch.

Contents

Chapter 1

Introduction

Computer systems suffer from a limited main memory hence the number of running applications depends on it. Especially servers, hosting a huge number of virtual machines (VMs), could be used more cost efficiently with even more VMs running on them. But the number of virtual machines is limited the main memory.

There is enormous potential to save main memory. As described in [12], there are virtualization hosts where the amount of redundant pages is between 11% and up to 86%. The amount of redundant pages depends on the operating system and the workload of the system.

To reduce the amount of used main memory, there are methods such as shared libraries and page deduplication. They prevent the storage of same content for different applications and more programs can be held in main memory.

However, the performance impact of sharing and deduplication on applications has not been thoroughly evaluated, yet. The effects of these techniques are especially important for NUMA systems, where sharings across NUMA nodes can be established. The decreased bandwidth (33% in our test system) and the increased latency (75% in our test system) of remote memory accesses can lead to a loss of performance for applications. So, higher latency and less bandwidth must be taken into considerations if memory sharings across NUMA nodes are established.

This thesis points out the runtime effects of sharing and if there are situations, where it should be avoided. This work focuses on the performance aspect of sharing and deduplication on NUMA systems. The drawbacks of remote memory access will be pointed out. To verify that the performance loss occurs because of remote memory accesses, the results of the NUMA system are compared to measurements taken on two UMA systems. Our UMA systems have RAM with different properties such as speed and timings installed. So, it can be ensured that the loss of performance of an application, using remote memory, is the result of the less bandwidth and higher latencies.

To point out the runtime effects of memory sharing and deduplication several SPEC CPU2006 [9] benchmarks were executed on the test systems. Also, custom benchmarks, which were written to suffer directly from memory sharing, as well as benchmarks with real world applications were performed.

This work will show that there are applications with a performance loss of a factor of two, if sharing across NUMA nodes is used. Also, it points out that there are memory intensive tasks, whose performance is only decreased by around 10% and so, the benefits of remote memory have to be checked against its benefits. It is shown that the performance, of an application with sharing across NUMA nodes, depends not only the size of the working set but also on the memory access pattern.

To provide a background the next chapter mention memory sharing and deduplication techniques and how these influence the hardware of the entire system. The following chapter, Chapter 3 presents methods to evaluate the runtime effects of memory sharing and deduplication and how they can be used to determine these effects. Chapter 4 show how the additional benchmarks are implemented and describes in detail how the evaluation methods of Chapter 3 were used. The results of the evaluation are presented in Chapter 5. This chapter illustrates the performance impact of sharing on different benchmarks and applications. The work closes with a conclusion and prospect of future work in Chapter 6.

Chapter 2

Background

This chapter gives a short explanation on memory sharing and deduplication. It outlines the benefits of these techniques and how they work. In addition, effects of sharing and deduplication on the hardware are described.

2.1 Memory Sharing and Deduplication

There are several ways to share memory between processes. The following section will give a brief overview of the different sharing and deduplication techniques and how they work. For this purpose the traditional mechanisms of sharing are described first. The second part of this section explains techniques, which were introduced to extend the capabilities of these approaches.

2.1.1 Traditional Mechanisms

Major operating systems (e.g., Microsoft Windows, Linux) have already mechanisms built-in that reduce the duplication of memory during normal system operation. The most common techniques are described in the following section.

Shared Libraries

One way of sharing memory is the use of shared libraries, which contain often used code. Applications are able to use the functionality without containing the library program code itself. Only the interface is needed at program creation. During the compilation and the linkage of the program, references are made to the used functions, but they are not included in the binary. Instead, the explicit code has to be made available during the dynamic linking at runtime.

As described in [17], the dynamic linker is executed first after the initialization of a newly created process. It resolves the references and determines which libraries have to be loaded. The operating system recognizes that a library has already been loaded by checking its file control block (e.g., the inode in Linux). If the library is already in use an additional mapping is established. The application is then able to read and execute the code of the shared library. Only if the library has not been used previously or got evicted due to memory pressure, it is loaded from mass storage into main memory.

This technique offers the capability to share code and read-only data, (e.g., application resources such as constant data sets, images or embedded language files) between an arbitrary number of processes.

Shared Memory via Child Processes

Another possibility is to share the whole address space of a process. After a `fork()` system call in Linux the original process is cloned. Initially, the parent and the child processes use identical address spaces. If one of the processes tries to modify data, a copy-on-write algorithm (COW) is executed. Therefore a new page is allocated and the page, which should be modified, is copied first. Then the newly created page is mapped into the process address space and the write operation is restarted. So it is possible to share the pages of two processes until they are modified.

A slightly different way of sharing memory between child processes is employed for threads. As described in [10], Linux internally does not distinguish between processes and threads. The only difference is the level of sharing. While for processes COW is used to preserve memory isolation, threads are able to modify the shared memory. For that reason, threads are also called light-weight processes, as they effectively share the identical address space with other threads. Changes made by a thread are visible to any other thread within the same process context. So it is possible to modify data in parallel.

Explicitly Shared Memory

Another way to share memory between processes is that processes explicitly request for it. There are system calls that create a shared memory region, which can be accessed from different processes. Data that is stored in this section can be modified by all processes that have mapped the shared memory region in their own address space.

Memory-Mapped Files

Another kind of memory sharing are memory-mapped files. Operating systems like Linux have a page cache where files that have already been used are cached. Subsequent accesses to a cache file are resolved by mapping according pages into the requesting process address' space. So only one copy has to be held in main memory. If a process modifies such pages copy-on-write is performed. Shared libraries (refer to Section 2.1.1) are shared by this technique.

2.1.2 Deduplication

Traditional memory deduplication techniques determine same content in main memory by evaluating the location (e.g., file, address space) where the content is originates from. Only content from the same location can be shared by the operating system. To identify those locations the operating system depends on contextual and semantic information about operations and structures as well as their relationship in the system. Therefore, traditional memory deduplication techniques only work within a single OS domain.

This problem surfaces for example in server environments running virtual machines as the virtual machine abstraction creates a semantic gap. However, if guests with the same OS version and application stack are running the chance is high that code and data can potentially be shared between virtual machines. As described in [12], there are virtualization hosts where the amount of redundant pages is between 11% and up to 86%.

Traditional memory deduplication techniques are not able to recognize these sharing opportunities. Therefore, mechanisms were developed that allow sharing across multiple OS domains.

Kernel Samepage Merging

With Kernel Samepage Merging (KSM) it is possible to share memory between different processes [16]. The application only has to give hints to the KSM kernel module via `madvise()` to indicate which pages should be scanned. The scanner then checks if there are pages with the same content in any of the specified memory regions. If identical pages exist, only a single copy is kept in main memory.

All other pages are freed and their corresponding mappings are redirected to the single left instance. The new shared page is set read-only so when a process writes to it an exception is thrown and the page is duplicated via COW.

In contrast to traditional deduplication mechanisms, KSM does not depend on the semantic information about pages, but solely works on page contents. It is therefore often used in conjunction with KVM to deduplicate memory in virtualized environments. In that case, the virtual machines' whole physical memory is marked with `madvise()`.

ESX Server

To exploit the sharing opportunities in virtualized environments, VMWare ESX server checks on its own for pages with same content used by the guests. There is no need to modify the guest operating systems for this technique. As described in [18] the hash of a page is calculated and a lookup in a hash table is performed. Once a page with the same hash value is found, the entire content of the pages is compared to rule out hash collisions. If the pages are identical one of the pages is shared and the other one is freed. When one of the guest operating systems tries to modify a shared page, a page fault is generated and a private copy is created.

Difference Engine

Advancement over basic memory scanners like KSM and the integrated scanner in ESX Server is Difference Engine [13]. Difference Engine does not only establish sharings between identical pages, but also between similar pages. Furthermore, a compression of distinct pages is performed. Pages that are not frequently used and have similar content are represented in memory with a reference page and patches. Whereas the reference page can be accessed directly, the patched pages need to be constructed from the reference page and the corresponding patch. Pages that are not frequently used and that cannot be represented via a reference page and a patch are compressed.

Satori

Memory scanners can reduce the amount of required main memory. But for this purpose they have to run frequently to scan the main memory. Depending on the scan rate, this introduces considerable overhead, regarding memory bandwidth usage and CPU time spent for scanning.

Satori therefore takes a different approach. As described in [14] read operations of the hard disk are used to identify sharing opportunities. On each read the hash value of the data is calculated and compared to previous reads. Once the hash values are identical, a candidate for sharing is found. After a complete comparison of the content a sharing is established. With this mechanism also short-lived sharing opportunities can be found and there is no need for frequently executed memory scanners. However, Satori is limited to sharing opportunities that stem from file-backed pages.

2.2 Hardware Effects

Sharing has also an impact on the performance of the computer hardware and the runtime of applications. Not only the caches benefit from the less amount of main memory required. There are also effects because of the NUMA architecture. This section outlines which hardware features have to be taken in account when sharing is considered.

2.2.1 NUMA

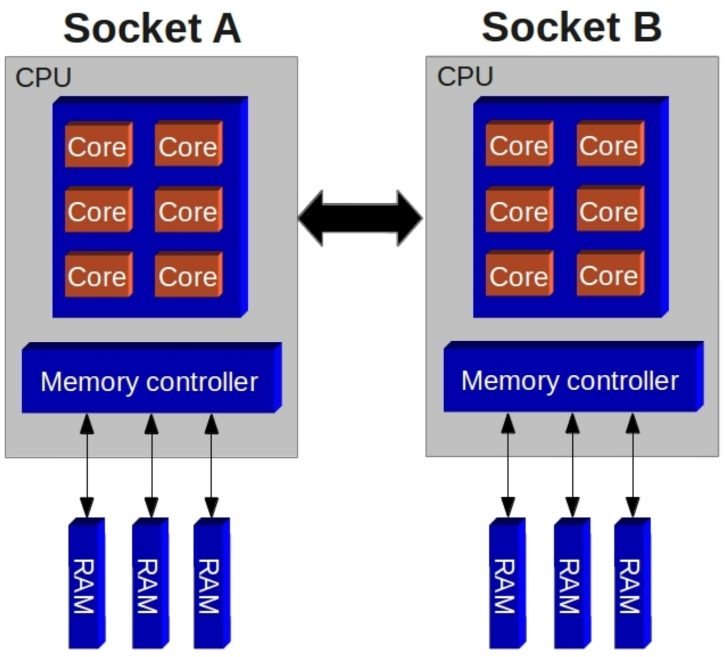


Figure 2.1: Schematic of a NUMA system. Each socket with its CPU and memory is a NUMA node.

NUMA systems play an important role when considering the runtime effects of memory sharing and deduplication. If a multi-socket system is used each socket has its own memory controller with its own memory. The general architecture of a NUMA system shows Figure 2.1. In these systems, processes are able to run on different sockets, but share code or data, which is stored in memory located on one socket. This prevents one process from direct memory access and inter-CPU communication has to be used. Sharing memory across NUMA nodes can therefore negatively impact applications performance.

There are several NUMA architectures available, which have in common that they add additional latencies to remote memory accesses. Also the bandwidth of the inter-CPU communication can be less than the bandwidth of the main memory. So the overall bandwidth can be reduced. For example there is the “QuickPath Interconnect” (QPI), which is used by Intel and our test systems. Another example is “Hyper Transport” (HT), which is used by AMD.

2.2.2 Processor Caches

As CPU caches are getting larger they can cache more instructions and data, but in comparison to the main memory they are still small. Figure 2.2 shows the memory hierarchy of our test systems. Memory sharing affects the efficiency of caching, because data and code that is shared in main memory has to be cached only once. So it is possible to cache more instructions and data. This leads to lower latencies and greater output since less main memory accesses are necessary. However, this only works on physical tagged caches, but since last level caches are typically physical tagged and are the largest of the processor caches, performance implications can be expected. An evaluation of sharing opportunities in last level caches already showed that depending on the cache size, workload and execution phase up to 45% of cache lines can be saved through memory deduplication [12].

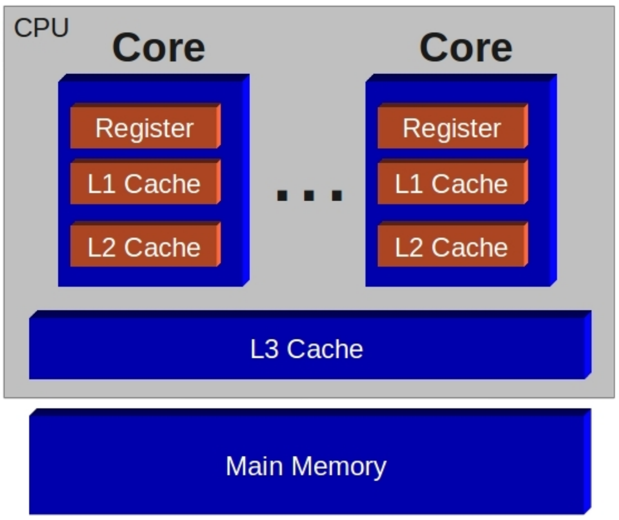


Figure 2.2: Schematic of a computer memory hierarchy. Today’s CPUs have more than one cache. The Intel processors that we used in this work, have private caches for each core and one cache that is shared by all cores of the CPU.

2.2.3 Prefetching

Prefetching is another method that reduces the cost of expensive remote memory accesses. Prefetching offers the capability that data and code is loaded by the CPU into the cache before the application requests for it. So the gap between slow memory and fast caches is reduced. As described in [15] CPUs, as used by our test systems, have several different hardware prefetchers that load data into caches.

2.3 Conclusion

The sharing techniques described in Section 2.1 have in common that they reduce the amount of physical memory needed by the system. More applications are able to run on a single computer and the efficiency of servers can be increased. Also the physical CPU cache benefits from these techniques. Fewer pages have to be stored in the last level cache, which potentially has a positive effect on runtime of applications.

However, on NUMA architectures it is uncertain that there are no drawbacks caused by memory sharing and if special caution should be taken to avoid sharing pages across NUMA nodes.

Chapter 3

Analysis

The following chapter describes the possibilities to evaluate the runtime effects of memory sharing and deduplication. First of all, the techniques to control the sharing are given with a short explanation. Also, the benefits and drawbacks are shown. The second part of this chapter describes how the runtime effects are evaluated.

3.1 Controlling of Memory Sharing

To evaluate the runtime effects of sharing and deduplication it is necessary to disable the sharing and deduplication mechanisms. This allows measuring their performance impact on UMA and especially NUMA systems. Since deduplication has to be explicitly activated there is no need to disable it. However, the traditional memory sharing techniques are embedded in the operating system. Nevertheless, there are several possibilities to prevent sharing with different benefits and drawbacks, which are outlined in the following sections. Since Linux is used for the benchmarks, the approaches focus on this operating system.

3.1.1 Static linking

The easiest way to prevent the sharing of libraries is the static linking of a binary, thereby avoiding the use of shared libraries in the first place. Instead of using references in the application, the library code and data is copied into the executable. This leads to a duplication of the original libraries' code and data in memory if multiple applications with the same library dependencies are running simultaneously. However, a drawback of this approach is that the operating system still recognizes, if the same executable is executed more than once. A second instance of the same application would not be loaded into main memory. Instead, a mapping would be established. Furthermore, this approach does not cover dynamically loaded resources and files that are still deduplicated through the file cache. Another drawback is that the application has to exist in source code to build the statically linked version. This is not possible for most of the benchmarks. Another point is that more overall memory is mapped when a dynamically linked program is executed. During the static linking only necessary code is written to the binary, whereas bigger parts of a library are mapped when the dynamic linker evaluates the references to the library code at runtime. This may also lead to another runtime characteristic, which makes comparisons to the unmodified benchmark difficult. Therefore, static linking is not an appropriate way to show the effect of sharing in a non-intrusive way.

3.1.2 Statifier

Another possibility is the use of a statifier like [2] or [5]. This program takes the binary and all needed libraries and merges them into a single executable. Although this technique was originally designed to allow the execution of an application in environments where the system does not have the libraries installed, it similarly as static linking leads to a duplication of library code and data.

However, it suffers from the same problems as described in Section 3.1.1. The operating system still recognizes the execution of the same executable and prevents further memory duplication. In addition, the statifier potentially changes the runtime characteristics of the executable (e.g., faster start) and therefore may lead to distorted measurement results. An improvement over static linking is that a statified executable contains the entire library code and not just the required functions. The address space layout and memory consumption are thus comparable to the original program. Another benefit of statifiers is that the source code of the benchmarks and libraries is not required.

3.1.3 Kernel Modification

Another way to prevent sharing is to modify the Linux kernel and disable any file-based sharing. This would ensure that libraries and the text segment of a binary would be loaded more than once and that not only additional mappings are established.

The problem with this approach is that it is not possible to determine from the inside of the kernel if a memory region should get an additional mapping or if it should be loaded into the main memory again. In consequence all sharing would have to be disabled and also operating system functions would be influenced. This would lead to incomparable runtime effects because the disabled sharing cannot be restricted to a dedicated benchmark. Also, it would not be possible to move the test setup easily. The tests could not be done with a new unmodified Linux kernel.

3.1.4 Chroot Environment

The most elegant way to disable sharing is the use of multiple *chroot environments* (change root). Each environment is a sandbox into which all dependencies of an application (i.e., libraries, dynamically loaded resources and files) including operating system libraries are installed and the application can be executed isolated from the rest of the system.

Since each chroot environment comprises its own copy of the corresponding files the operating system cannot recognize that the same binary is loaded in different chroot environments and memory sharing is suppressed. In contrast, to static linking or statifying, the whole execution stays the same, so dynamic linking and library loading takes place, but is executed for each environment separately. This ensured that the measurement results are comparable to a standard execution with full sharing.

Another benefit is that neither the operating system nor the application has to be modified. This makes it possible that the whole benchmarks can be evaluated on different systems without any modifications. Only the chroot environment has to be created.

However, compared to a complete deactivation of memory sharing through a kernel modification, the approach does not prevent files getting cached and shared within a single chroot environment.

3.1.5 Conclusion

	Static Linking	Statifier	Kernel Modification	Chroot Environment
Shared Libraries Disabled	●	●	●	●
Buffer Cache disabled			●	●[1]
No Runtime Impact				●
No Source Code Required		●	●	●
No OS Modification	●	●		●
Portable	●	●		●

[1] The chroot environment does not disable the cache itself but prevents sharing across environment borders (e.g., between multiple instances of the same application).

Table 3.1: Aspects, which have to be considered if sharing has to be disabled. Except from portable they are all necessary. For this work, only the change root environment meets all requirements.

The chroot environment is the only approach that meets all important requirements. As seen in Table 3.1, the chroot environment disables most sharing mechanisms of the operating system and the buffer cache can be freed after each run. So caching of files does not influence the runtime of subsequent runs. Also no modifications of the operating system or the binaries are necessary. Furthermore, since the unmodified binaries can be executed the comparability with the standard case (i.e., with activated sharing) is guaranteed. So the chroot environment is the most feasible technique to disable sharing.

3.2 Node Affinity

The approaches discussed so far allow preventing memory sharing between multiple instances of the same application. To evaluate the performance impact of sharing across NUMA nodes, it is also necessary to control the assignment of processes to CPUs and memory modules. This section describes which possibilities exist to accomplish this.

3.2.1 Tasktset

A way to control the usage of the CPUs is the `taskset` system call [7]. This system call sets the CPU affinity of a process. The process will then be scheduled only on the selected cores.

However, with this system call it is not possible to restrict the memory modules that are used by a process. So it is not possible to guarantee that a process has to do remote memory accesses.

3.2.2 CPUSET

Another method to control a process's CPU affinity are CPUSETs [1]. They can be used to group CPUs and memory nodes together. An application, which is assigned to such a set, is only scheduled to the grouped CPU. Moreover, all memory required by the application, is allocated from the selected memory nodes. With this method it is easy to achieve that code or data is shared by processes across different NUMA nodes by assigning CPU and memory modules from distinct NUMA nodes. All memory accesses to application private memory are then guaranteed to be remote memory accesses.

3.2.3 Conclusion

	Taskset	CPUSET
Controls Memory Affinity	●	●
Controls CPU Affinity		●

Table 3.2: Comparison of methods to set the node affinity. CPUSETs meet all requirements.

As seen in Table 3.2 only CPUSETs have the capability to control the usage of the CPU and the main memory. Therefore, they are used to evaluate the performance impacts of sharing across multiple NUMA nodes as well within a single node.

3.3 Evaluation Methods

The performance impact of sharing can be evaluated by measuring the execution time of applications with and without sharing by forcing memory accesses to local or remote nodes on a NUMA system. To guarantee representative results it is necessary to choose a wide range of applications for the evaluation. The SPEC CPU2006 benchmarks are suitable to cover various different types of applications. To evaluate effects not covered by the SPEC benchmarks special purpose benchmarks can be used. The synthetic benchmarks then need to be supplemented with benchmarks of real world applications to get a complete picture of the performance impact of sharing.

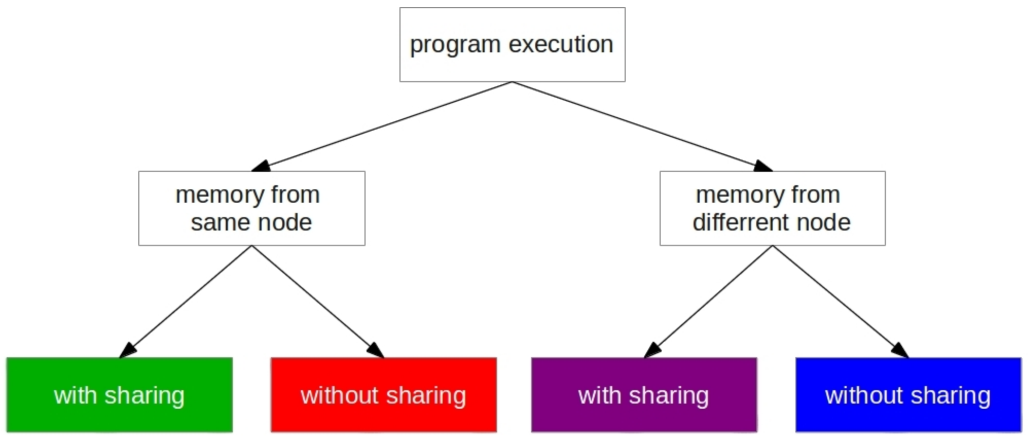


Figure 3.1: Schematic for the different test runs on the NUMA System. Local or remote memory can be used for a benchmark. A run of the benchmark can also be done with or without sharing.

Chroot environments and CPUSETs are suitable to simulate different sharing scenarios. The sharing between multiple instances of the same application can be allowed or suppressed with the chroot environments. During a single benchmark run multiple instances of the same benchmark are executed. If sharing needs to be prevented each instance of a benchmark runs in its own chroot environment. Otherwise, only a single chroot environment is used. With the help of the CPUSETs it is possible to simulate sharing within or across NUMA nodes. This is accomplished by assigning different NUMA nodes for the scheduling and memory allocation of a process.

The benchmarks can be executed with memory from a different NUMA node (CPUSET) to simulate a scenario where all the code and data is already loaded on a different NUMA node and a new instance that shares this data and code is started on another node. In consequence, the process has to do remote memory accesses.

These results can be compared to runs where the memory of the local NUMA node is used. This simulates a scenario where a new instance is started, but the required code and data is copied to the local NUMA node and thus duplicated. So, only local memory accesses are necessary.

To evaluate the performance impact of sharing on UMA systems, the CPUSETs need to be configured to keep memory allocations on the same NUMA node that dispatches the benchmark. Figure 3.1 summarizes all execution possibilities.

These are worst case scenarios since the benchmarks and applications are forced to use only the desired memory modules. Usually, the local memory is used for local variables and newly allocated memory. So, the impact of sharing across NUMA nodes can be smaller on production use systems.

To support the interpretation of benchmark results it is in addition beneficial to get information on the general sensitivity of the benchmarks to varying memory performance. This data can be gathered by measuring the execution time of the benchmarks on a system with high and low memory performance, respectively.

3.3.1 Latency and Bandwidth

To get a basic idea of the performance impact of remote memory accesses in a NUMA system the memory latency and bandwidth of local and remote accesses can be measured. This data also helps to put further results into the right context. To perform the measurement following benchmarks are suited, whereas CPUSETs (refer to Section 3.2.2) can be utilized to direct memory accesses to local or remote memory respectively.

Stream

To measure the bandwidth of main memory the Stream benchmark [6] is well suited. It is a simple benchmark that reads and writes data blocks from and to main memory. The time of these memory operations is measured and the bandwidth is calculated.

LMbench

It is not enough to know the bandwidth of main memory. Applications that read or write arbitrarily small data blocks do not suffer from a limitation of the bandwidth, but from high latencies. Therefore, the latencies of the installed memory are measured with LMBench [3]. This is a suite of simple benchmarks for Linux systems. The benchmark *bm_mem* of the suite does random reads in main memory and measures the required time for an access, thereby determining the memory latency.

3.3.2 Synthetic Benchmarks

To get meaningful results it is necessary to use wide-ranging synthetic benchmarks. Not only memory intensive, but also compute intensive applications are required to cover all impacts of memory sharing and deduplication on the runtime. The SPEC CPU2006 suite is suitable for this purpose since it consists of several benchmarks with different focuses. It used to indicate which kinds of applications are influenced by sharing.

SPEC gcc

The SPEC gcc benchmark is based on the GNU Compiler Collection [9]. For a maximal stressing of the system a compiler with many optimization flags is executed on a predefined test set. The compiler has been slightly altered by the authors of the benchmark so it spends more time analyzing the source code inputs and uses more memory. It accesses many small chunks of memory in an arbitrary way and uses around 3.3 MB for the text segment. It is expected that there are runtime effects of memory sharing on UMA systems since the text segment can be shared.

SPEC libquantum

Another benchmark with arbitrary memory accesses is the SPEC libquantum benchmark that is based on the C library libquantum, which simulates a quantum computer [9]. Quantum computers are based on the principles of quantum mechanics and can solve certain computationally hard tasks in polynomial time. Libquantum uses matrix operations to simulate a quantum computer [11]. This can lead to many memory accesses if the matrices become large and the elements are used in an arbitrary way. Therefore, this benchmark should suffer from low performance memory and thus from sharing across NUMA nodes.

SPEC mcf

To evaluate the effects of sharing on a benchmark that has a huge working set (1.6 GB), which is used during the entire benchmark run, the SPEC mcf benchmark is appropriate. Mcf is a benchmark derived from MCF, a program used for single-depot vehicle scheduling in public mass transportation [9].

SPEC bzip2

In contrast to the previous benchmarks, which access small memory chunks in an arbitrary way, the SPEC bzip2 benchmark processes data in sequential blocks. It is expected that the prefetchers of the CPU reduce the drawbacks of sharing across NUMA nodes and therefore this benchmark is suitable. The SPEC bzip2 benchmark is based on the bzip2 compression software [9]. bzip2 splits the input data into blocks and runs several algorithms on these blocks to compress them. The only difference between the benchmark and the compression software is that the benchmark performs no file I/O, but instead works entirely on a test set in memory. So only the CPU and the main memory are benchmarked.

SPEC hmmer

The previous benchmarks focused on applications with a large working set held in main memory. To evaluate also the effects of sharing on applications that focuses on computationally heavy tasks the SPEC hmmer benchmark is suitable. This benchmark is based on the HMMER sequence analysis software package [9]. HMMER is used for protein sequence analysis and the search for protein or nucleotide sequences with the help of hidden Markov models.

Sequential Code Benchmark

The benchmarks described above have only a small text segment compared to the data segment. Since code is a good candidate for sharing it has to be evaluated with a benchmark that uses plenty of code. Therefore the self-written sequential code benchmark is suitable. It consists only of increment and decrement operations. Apart from the variable, which is incremented and two time stamps no data is processed. For implementation details refer to Section 4.1.

The operating system is able to share the complete text segment of this application between several running instances. Since the binary has a size of 10.3 MB it can be completely cached if sharing is used [2].

Non-Sequential Code Benchmark

Early evaluation results showed that the CPU's prefetcher predicts the required instructions of the sequential code benchmark very well. The memory access is done

[2]Our evaluation system has 15 MB of L3-Cache

before the next instructions are needed. So no time is spent waiting for the memory access. To prevent the preloading through the prefetcher the non-sequential code benchmark also consists of many increment and decrement operations in a loop but they are grouped in code blocks that are called arbitrarily. It is expected that this is especially costly if the processor needs to perform remote memory accesses on a NUMA system. For implementation details refer to Section 4.2.

3.3.3 Real World Applications

The synthetic benchmarks described above build the base line for the considerations of the runtime effects of memory sharing and deduplication. But these benchmarks only stress the CPU and the main memory. Aspects like reads from the hard disk are not covered. These aspects can be the limiting factors and therefore sharing has maybe not the impact suggested by the synthetic benchmarks. For this purpose it is also necessary to benchmark real world applications.

Kernel Build

To evaluate an application that accesses many small memory chunks in an arbitrary way, a kernel build is suitable. In contrast to the synthetic SPEC gcc benchmark the input has to be read from the hard disk. So still a huge number of libraries and a lot of code is executed during the compilation but the file I/O of the hard disk may lead to another result.

MEncoder

In contrast to the kernel build, MEncoder [4] a tool to transcode video files, processes input data in a predictable way because the input file is read sequentially. So it is expected that the prefetchers reduce the impact of low performance memory and thus sharing across NUMA nodes. Since the same input file can be used by multiple instances a sharing can be established. Therefore, this benchmark is suitable to illustrate the runtime effects of memory sharing.

Chapter 4

Design and Implementation

The previous chapter mentioned the benchmarks and tools for evaluating the run-time effects of memory sharing and deduplication. The following chapter gives an explanation how they were used. In addition, the implementations of the sequential code benchmark and the non-sequential code benchmark are described.

4.1 Sequential Code Benchmark

The sequential code benchmark consists of a loop with 160000 increment and decrement operations. This was almost the maximum size, which the GCC 4.6.3 can process without a segmentation fault but therefore the code had to be grouped into blocks. `if()`-statements that are always true are used for this purpose. To increase the execution time the increment operations are executed in a loop. This loop is executed 25000 times and the execution time is measured with a start and an end timestamp. A code snippet of the benchmark can be seen in Listing 4.1.

```
...
gettimeofday(&tp, &tzp);                          //get the timestamp
start = tp.tv_sec                                 //store only the seconds
for(counter = 0; counter < 25000; counter++) {
  i == rand();
  number = rand() % 4;
  if(number != 10) {
    i++;i++;i++;i++;
    ...
  }
  if(number != 10) {
    i--;i--;i--;i--;
    ...
  }
  if(number != 10) {
    i++;i++;i++;i++;
    ...
  }
  if(number != 10) {
    i--;i--;i--;i--;
    ...
  }
  i == rand();
  number = rand() % 4;
  if(number != 10) {
    i++;i++;i++;i++;
    ...
  }
  if(number != 10) {
    i--;i--;i--;i--;
    ...
}
gettimeofday(&tp, &tzp);                        //get the timestamp
end = tp.tv_sec                                 //store only the seconds
...
```

Listing 4.1: Part of the sequential code benchmark. 160000 increment and decrement operations are executed in a loop and the execution time is measured.

The application was compiled without optimizations to ensure that the increment operations were not optimized to a single value assignment. This benchmark ensures a huge text segment with many sharing possibilities. The binary itself has a size of 10.3 MB.

4.2 Non-Sequential Code Benchmark

The sequential code benchmark was altered to achieve an unpredictable execution order. Therefore the `if()` blocks are altered into `if() - else if()` blocks, so only a quarter of the instructions are executed. To gain comparable results the loop is executed four times more often. So the prefetcher is not able to properly predict the execution flow. A code snippet of the benchmark can be seen in Listing 4.2.

```
...
gettimeofday(&tp, &tzp);                        //get the timestamp
start = tp.tv_sec                               //store only the seconds
for(counter = 0; counter < 100000; counter++) { // 100000 = 4 x 25000
  i == rand();
  number = rand() % 4;
  if(number == 0) {
    i++;i++;i++;i++;
    ...
  } else if(number == 1) {
    i--;i--;i--;i--;
    ...
  } else if(number == 2) {
    i++;i++;i++;i++;
    ...
  } else if(number == 3) {
      i--;i--;i--;i--;
      ...
    }
    i == rand();
    number = rand() % 4;
    if(number == 0) {
      i++;i++;i++;i++;
      ...
    } else if(number == 1) {
      i--;i--;i--;i--;
      ...
}
gettimeofday(&tp, &tzp);                        //get the timestamp
end = tp.tv_sec                                 //store only the seconds
...
```

Listing 4.2: Part of the non-sequential code benchmark. Increment and decrement operations are grouped into blocks. These code blocks are executed randomly and 40000 operations are executed during one loop iteration. The execution time is measured via a start and an end time stamp.

Like the sequential code benchmark, the application was compiled without any optimization to ensure that the program is not altered by the compiler. Therefore, the binary itself is has also a size of 10.3 MB.

4.3 Chroot Environment

As described in Section 3.1.5 chroot environments were suitable to control sharing. Ubuntu 12.04 offers the capability to create a chroot environment with a minimal installation of an operating system. The command: `debootstrap /install/dir /repository/location` was used to create a single chroot environment. Due to the dependencies of the benchmarks, additional packages (e.g., GCC, MEncoder) needed to be installed into the new environment. The final environment was then copied 6 times to allow running multiple instances of the same application without memory sharing. With 6 environments all cores of a NUMA node can execute an application in parallel.

To execute a program inside one of the chroot environments the following command line was used: `schroot -c - environment_name command`.

4.4 Execution Scripts

To automate the benchmark execution and the gathering of results multiple scripts were used. Listing 4.3 shows the script that was employed to start a benchmark in the chroot environments.

```
#!/bin/bash
schroot -c environment_1 -- sh benchmark.sh > env_1 &
schroot -c environment_2 -- sh benchmark.sh > env_2 &
schroot -c environment_3 -- sh benchmark.sh > env_3 &
schroot -c environment_4 -- sh benchmark.sh > env_4 &
schroot -c environment_5 -- sh benchmark.sh > env_5 &
schroot -c environment_6 -- sh benchmark.sh > env_6 &
```

Listing 4.3: Shell script to execute 6 parallel runs of a benchmark in 6 different chroot environments.

For each benchmark a custom start script configures and controls the execution of the benchmark or application within the isolated chroot environment. The results are written to text files and examined after the runs. Listing 4.4 shows how an exemplary run for the MEncoder benchmark is started. The execution time is measured with `time`, a command line tool to measure the execution time of a process and the required main memory during a run was measured with the command line tool `top`. The output of MEncoder is deleted after a run and the next run is started.

```
#!/bin/bash
counter=1
cd /mencoder
while [ $counter -le $1 ]; do
/usr/bin/time -a -o zeit_1 -f ’%e’ sh mencoder.sh > /dev/null
rm *.avi
rm *.log
counter=$(($counter+1))
done
```

Listing 4.4: Shell script to execute a run of MEncoder inside a chroot environment. A cleanup is performed after each run. The execution time is measured with the command line tool `time`.

In contrast to the other benchmarks performed in this work, early evaluation results of the kernel build benchmark showed that the operating system’s file cache lead to a speed up for every subsequent run. This is most probably caused by the fact that a kernel build accessed hundreds of source files and with each run improves the coverage of the cache. To avoid this effect and ensure deterministic results the file cache, dentry (directory entry) cache and the inodes were freed after each run. The command in line 9 in Listing 4.5 was used to accomplish this. It was executed 100 seconds after the last build finished. As a kernel build took around 250 seconds, this was enough time for all processes to finish.

Since the process file system (/proc) is not available in a chroot environment the cache reset had to be added to the script presented in Listing 4.3. The execution returns from the chroot environments to the script after the sixth run has finished. Once the waiting period has elapsed, the caches are freed.

```
#!/bin/bash
counter=1
schroot -c environment_1 -- sh kernel_build.sh > env_1 &
schroot -c environment_2 -- sh kernel_build.sh > env_2 &
schroot -c environment_3 -- sh kernel_build.sh > env_3 &
schroot -c environment_4 -- sh kernel_build.sh > env_4 &
schroot -c environment_5 -- sh kernel_build.sh > env_5 &
schroot -c environment_6 -- sh kernel_build.sh > env_6
sleep 100 #wait until all processes have finished
echo 3 > /proc/sys/vm/drop_caches # free page cache, dentries and inodes
sleep 20 #give some time to the OS to load OS used memory again
counter=$(($counter+1))
done
```

Listing 4.5: Shell script to execute a synchronized run of a benchmark inside a chroot environment. The `sleep` of 100 seconds guarantees that all processes have finished and the page cache is not freed during a run.

Chapter 5

Evaluation

After explaining the problem, which can occur with sharing, and how this can be evaluated, this chapter shows the results of the tests. First of all, the test systems are described. The next part shows the results of the benchmarks mentioned in Section 3.3. The last part of this chapter discusses the results and situations in which sharing should be avoided.

5.1 Evaluation Setup

To perform the evaluation three different test systems were used. Two single-socket systems were utilized to determine the impact of memory speed on the applications and benchmarks. One of the systems was equipped with a single low performance memory module. The other one had high performance memory installed. Since four memory modules were installed the dual channel interface and full interleaving was activated. The third system was a dual-socket system to evaluate the performance impact of memory sharing within a single and across multiple NUMA nodes. For further hardware details refer to Table 5.1.

To eliminate hardware runtime effects all power saving features like “Intel Speed-Step” were disabled. Also “Intel Turbo Boost” and “Hyper Threading” were disabled.

As mentioned in Section 4.3, Ubuntu was optimal to set up the test environment. Therefore, Ubuntu 12.04 64 bit was installed as operating system on all test systems. The 64 bit version of the operating system was necessary to use more than 4 GB of main memory. A minimal version of Ubuntu was used as basis for the chroot environments.

	NUMA System	High Performance Memory System	Low Performance Memory System
Installed CPU	2 x E5-2420	i7-2600k	i7-2600k
CPU Interconnect	QPI (7.2GT/s)[1]	-	-
Memory Size	12 GB[2]	8 GB	2 GB
Module Count	6	4	1
Memory Channels	3	2	1[3]
Memory Frequency	1333MHz	2133 MHz	1067 MHz
Memory Channels	3	2	1

[1] Corresponds to a maximum bandwidth of 14.4 GB/s [8]
[2] Each of the NUMA nodes in the dual socket system 6 GB.
[3] Effectively only a single channel because only one module was installed.

Table 5.1: Short overview about the Hardware installed in the 3 different test systems.

5.2 Methodology

To simulate the scenarios described in Section 3.3 six chroot environments were used, one for each CPU core of a NUMA node. So it was possible to run six single threaded instances of a benchmark in parallel without sharing. The only exception was the SPEC mcf benchmark. Because it requires 1.6 GB of main memory only three instances could be executed.

All benchmarks and applications were executed and measured 20 times, whereas the first and the last run were not taken into account. The last run was left out because it was possible that some benchmarks had already finished and the remaining instances benefit from less competitive memory accesses.

5.3 Latency and Bandwidth Benchmarks

The results of the latency and bandwidth benchmarks are shown in Table 5.2. Although the NUMA system uses much slower memory modules than the high performance memory system (1333 MHz vs. 2133 MHz), the triple channel interface is almost able to fully compensate the less bandwidth. Nevertheless, the remote memory access increases the latency by around 75% and also decreases the bandwidth by nearly 33%.

The difference between the latencies of the low performance memory system and the high performance memory system is negligible. Surprisingly, the memory in the NUMA system has a higher latency by 57 % than the low performance memory system on a local memory access, although its speed is 25% higher. This is most probably caused by the fact that the memory in the NUMA system is buffered (registered) and uses ECC for data integrity. The bandwidth of the high performance memory system is greater by over 130% compared to the low performance memory system. However, this was expected since the low performance memory system contains only a single memory module and therefore does not benefit from dual channel and memory interleaving.

	Bandwidth (MB/s)	Latency (ns)
NUMA Architecture Local Memory	10520	85
NUMA Architecture Remote Memory	7054	148
Low Performance Memory	4993	54
High Performance Memory	11565	49

Table 5.2: Measurements of the latency and bandwidth of the different test systems. The remote memory access increases the latency by around 75% and decreases the bandwidth by nearly 33%.

5.4 Synthetic Benchmarks

The latency and bandwidth benchmarks give a basic idea of the performance of memory accesses on the different systems. But it is not certain that the less bandwidth and higher latency of a remote memory access influences the performance of an application. Furthermore, the performance impact of memory sharing within a NUMA node is unknown. The synthetic benchmarks shed a first light onto these questions.

The following section presents the results of the synthetic benchmarks. The left side of each figure denotes the execution times for the parallel runs on the NUMA system with and without sharing. However, only the traditional memory deduplication techniques embedded in the OS were used to establish sharings. No deduplication techniques as described in Section 2.1.2 (e.g., memory scanners) were active during the benchmarks. The right side illustrates the benchmark's performance on the single socket systems. The error bars in the diagrams indicate the maximum and minimum execution times measured.

All synthetic benchmarks receive their input data directly from memory. No periodic I/O accesses are made during a run.

5.4.1 SPEC gcc

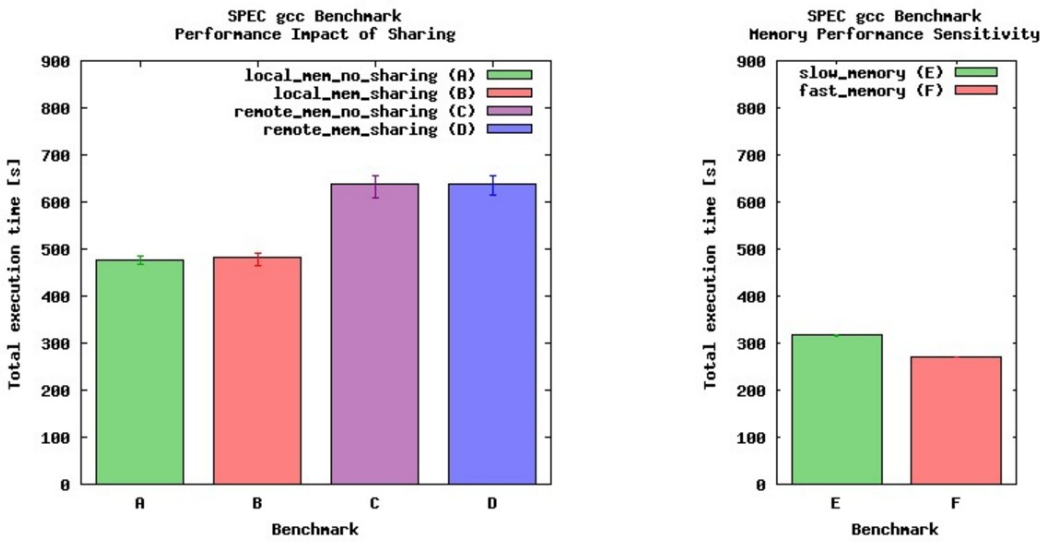


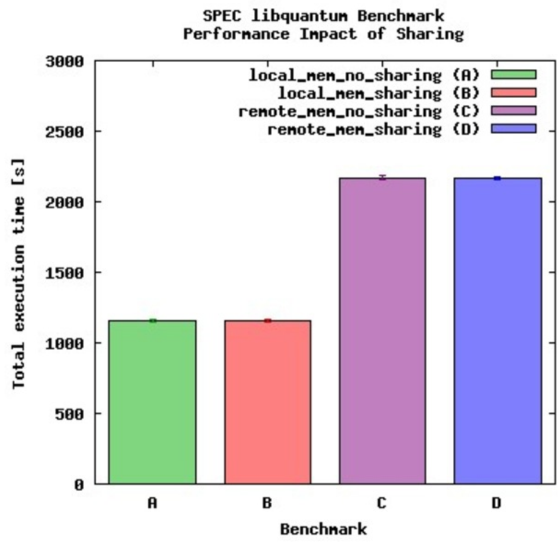
Figure 5.1: Execution time of the SPEC gcc benchmark.

The SPEC gcc benchmark was chosen because it accesses many small chunks of memory in an arbitrary way. It was expected that this benchmark suffers from remote memory accesses. The results meet this expectation. As seen on the left side of Figure 5.1, the execution time is increased by around 31% when remote memory is used. The performance impact on the single-socket systems is smaller. This comes from the fact that the memory latencies of the systems are almost equal. The benchmark accesses the memory in small chunks and therefore the bandwidth has only few impacts.

Although sharing across NUMA nodes has an impact on the performance of the benchmark, sharing on the same NUMA node does not influence the execution time. There are no positive effects of sharing since each process needs its own data. With the traditional sharing mechanisms only the code of GCC can be shared. But the 3.3 MB of code are facing 500 MB of dynamically allocated unshared data processed by each of the running SPEC gcc benchmarks.

5.4.2 SPEC libquantum

The negative impact of remote memory accesses is also visible in the SPEC libquantum results since the benchmark also accesses main memory in an arbitrary way. As seen on the left side of Figure 5.2, the effect is even greater. The



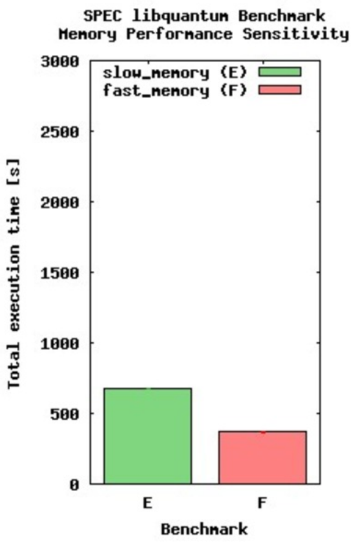


Figure 5.2: Execution time of the SPEC libquantum benchmark.

execution time is increased by a factor of two. This is also the case for the low performance memory system, which indicates that this benchmark suffers from a limitation of bandwidth.

As with the SPEC gcc benchmark, there are no positive effects of sharing on a UMA system since only code can be shared and each running benchmark needs its own data. 40 KB of code are facing 64 MB of data stored in main memory.

5.4.3 SPEC mcf

In contrast to SPEC gcc and SPEC libquantum the SPEC mcf benchmark has a huge static working set (1.6 GB). Therefore, the memory accesses can hardly be cached. Figure 5.3 shows this behavior since the execution time of the benchmark with remote memory is increased by around 28%. So, this benchmark suffers from sharing across NUMA nodes.

If sharing on an UMA system is regarded the results and the explanation are the same as above. 40 KB of code are facing 1.6 GB of data required by each of the running mcf benchmarks and thus sharing of code has no major impact.

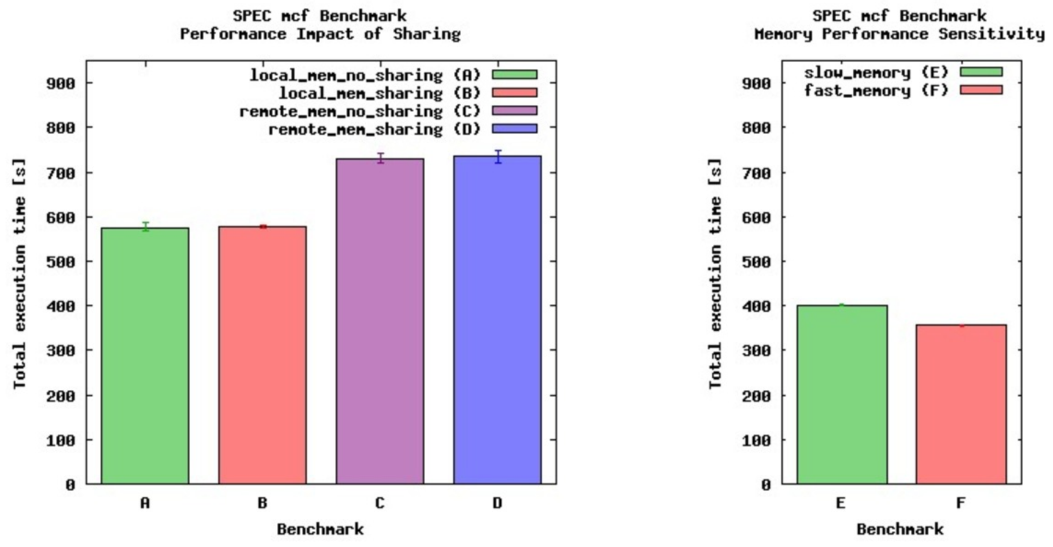


Figure 5.3: Execution time of the SPEC mcf benchmark.

5.4.4 SPEC bzip2

The results above showed that benchmarks that access memory in an arbitrarily way suffer from the high latency and the less bandwidth of remote memory accesses. The SPEC bzip2 benchmark has a slightly different result.

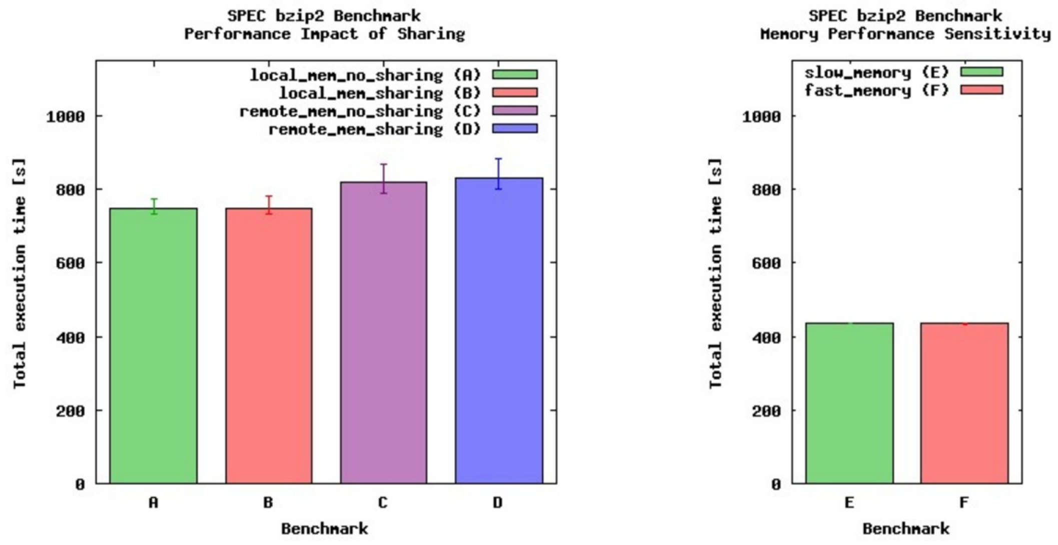


Figure 5.4: Execution time of the SPEC bzip2 benchmark.

In contrast to the SPEC gcc and SPEC libquantum benchmark sharing across NUMA nodes has only a small impact on the execution time. It only increases by around 10% if remote memory is used. Moreover, the right side of Figure 5.4 implies that the bzip2 benchmark is not suffering considerably from low performance memory. This comes from the fact that bzip2 compresses data in blocks. Each time a block is compressed the next one is read. Since the data blocks are read sequentially the prefetcher can preload the next block. The CPU thus spends only a short time waiting for data.

As seen in Figure 5.4, the benchmark does not benefit from sharing on UMA systems. Compared to the 850 MB of unshared data accessed by the benchmark the 56 KB of shared code are negligible.

5.4.5 SPEC hmmer

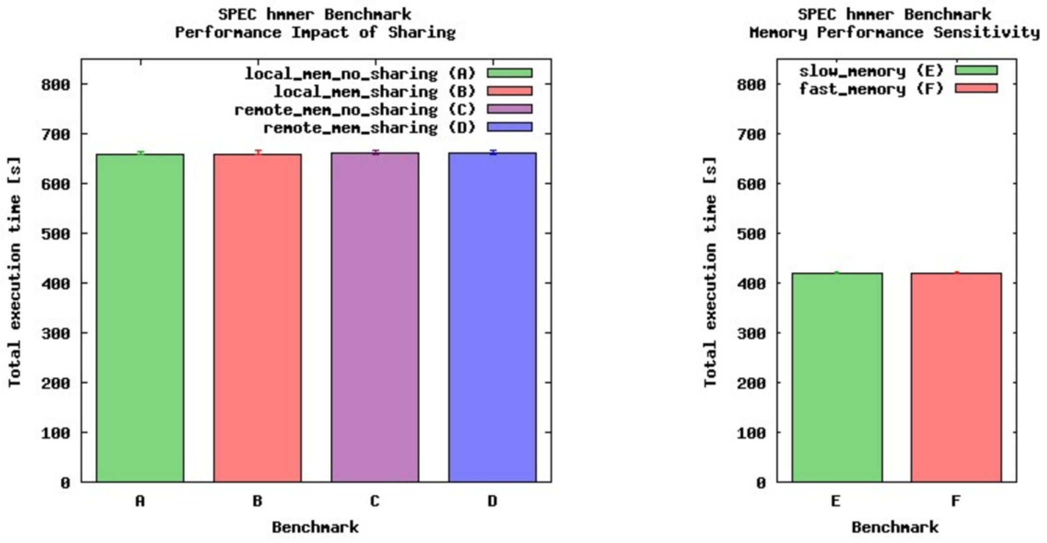


Figure 5.5: Execution time of the SPEC HMMER benchmark.

The previous benchmarks had a large working set and thus suffered from sharing across NUMA nodes if arbitrary memory addresses were read. In contrast, to these benchmarks the SPEC hmmer benchmark is compute intensive and as expected not affected by sharing at all. Figure 5.5 shows that the execution time depends only on the speed of the CPU. It does not matter whether sharing is used or not, since only 284 KB of code are used and thus can fully be cached. Also, with 20 MB the working set is small and so there is not even an impact of the execution time if remote memory is used.

5.4.6 Sequential Code

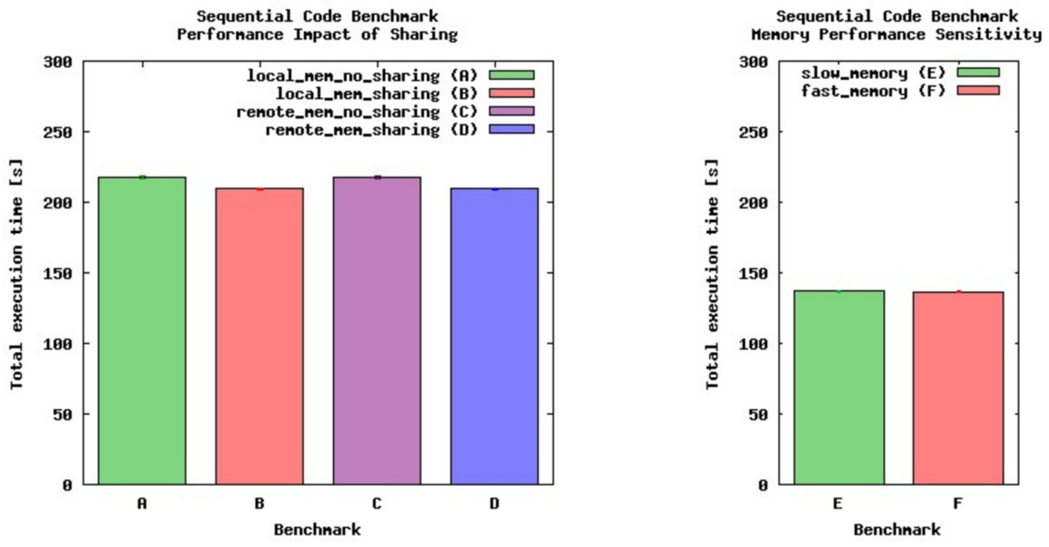


Figure 5.6: Execution time of the sequential benchmark.

All previous benchmarks had a small text segment compared to the dynamically created unshared data. Therefore, there were no possibilities to regard sharing benefits on a UMA system. The sequential code benchmark showed that there are almost not measurable benefits of sharing on UMA systems. Figure 5.6 shows that the execution time was decreased by 4% if sharing was enabled regardless remote or local memory was used.

5.4.7 Non-Sequential Code Benchmark

The prefetcher of the CPU reduced the time spent, waiting until the next instructions are loaded in the sequential code benchmark. This is not possible with the non-sequential code benchmark and the results in Figure 5.7 support this fact. The complete binary can be cached if sharing is enabled and because of that, the sequential and the non-sequential code benchmark have almost the same execution time. If sharing is suppressed the execution time increases by 11%. The benefits of sharing code on the execution time are still small compared to the results of shared data.

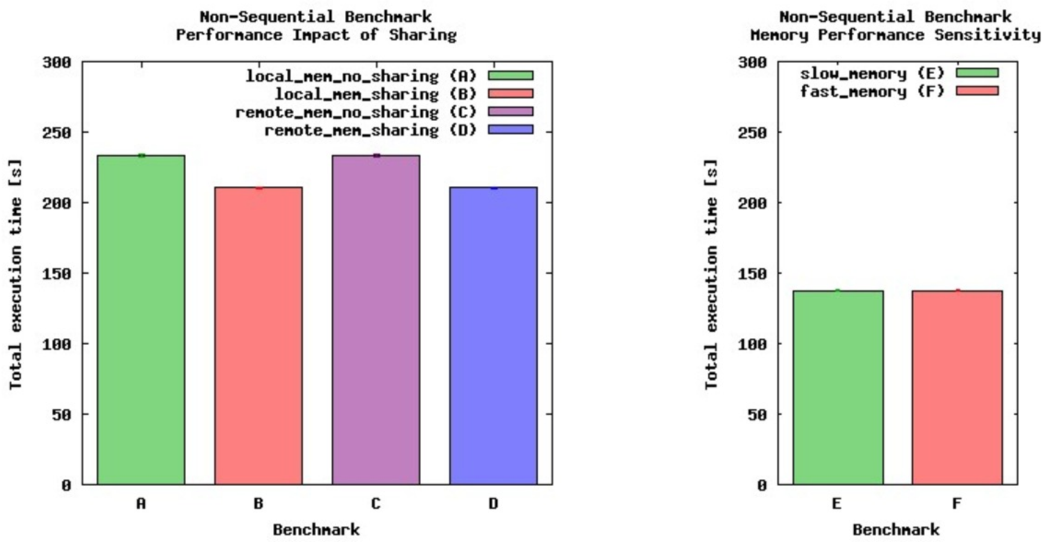


Figure 5.7: Execution time of the non-sequential benchmark.

5.5 Real World Applications

The previous section showed that among synthetic benchmarks those with random memory access patterns suffer from sharing across NUMA nodes. But also real world applications have to be considered since there are more aspects that determine the overall performance of an application (e.g., file I/O). The following section presents the results from the real world applications.

5.5.1 Kernel Build

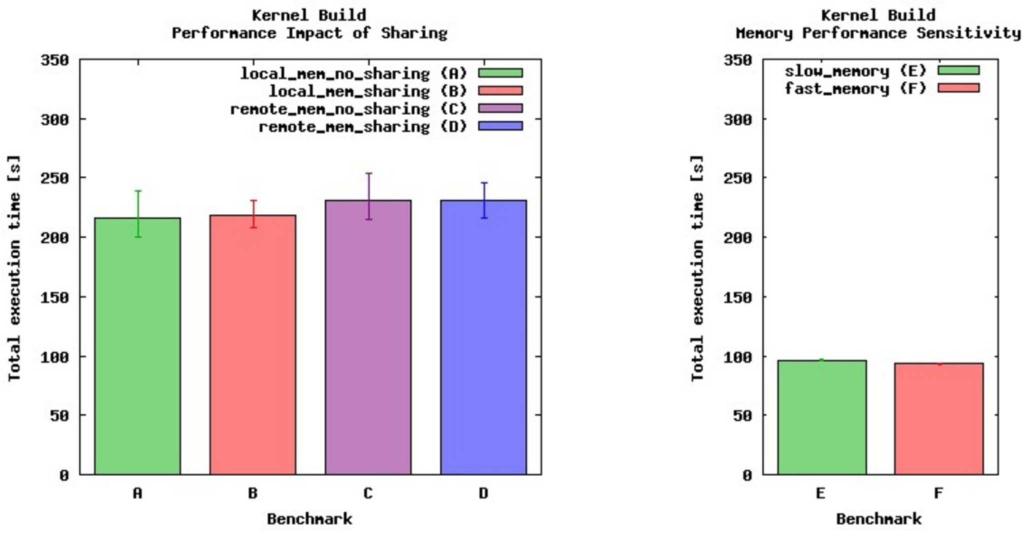
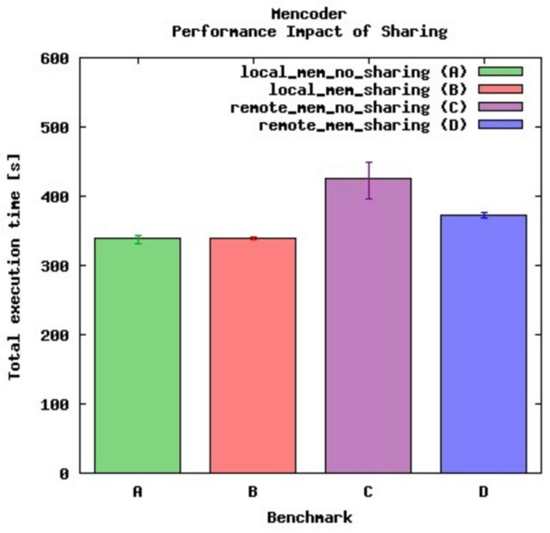


Figure 5.8: Execution time of make on a Linux kernel.

A kernel build is a pendant of the SPEC gcc benchmark and therefore the results are comparable. The Linux kernel with version 3.3.6 was built with the default config created by `make allnoconfig` and the required time was measured. Actually the results have the same trend. Sharing across NUMA nodes increase the execution time whereas sharing on a UMA system has no major impact. But, the effect is less marked on the kernel build as seen on Figure 5.8. The execution time is only increased by 6% if remote memory is used. This is quite small in contrast to the 31% of the SPECC gcc benchmark. This is because there are a lot of hard disk reads on the input files and writes on the output files. So the speed of the memory has no great impact on the execution time. This can be seen on the right side of Figure 5.8. There is no difference in the execution time if slow or fast memory is used.

5.5.2 MEncoder



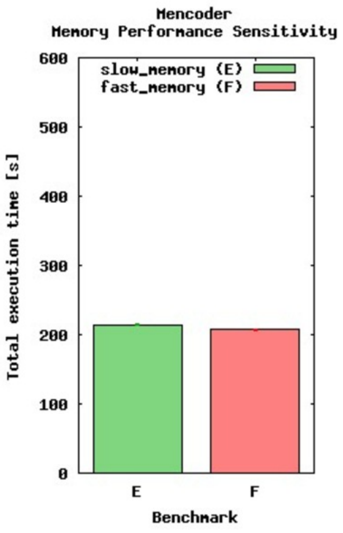


Figure 5.9: Execution time of MEncoder on a 180 MB mkv video.

In contrast to the kernel build, MEncoder processes its input data sequentially so it was expected that sharing across NUMA nodes has no major impact on the performance. To evaluate this fact a 180 MB full HD video was scaled and converted. Figure 5.9 shows that the performance is decreased by approximately 10% if remote memory is used and the input data can be shared. If sharing is disabled and the input file has to be read six times from hard disk into main memory the execution time is increased by around 25%. The prefetcher is capable to generally reduce the effects of low performance memory. The almost identical execution times on the single-socket systems suggest that. However, the competitive memory accesses of the six running instances bring the single QPI to its limit and the performance decreases in the unshared remote access configuration.

5.6 Discussion

The results above show not only that there are drawbacks of remote memory accesses, but also that synthetic benchmarks suffer from that fact. Furthermore, this effect can be regarded by real world applications, too. But, there are different results for the different types of sharing. On the one hand there is the sharing across NUMA nodes and on the other hand sharing on UMA systems. The different types of sharing have to be evaluated separately.

5.6.1 Sharing Memory on UMA

Sharing through the traditional mechanisms has no major impact in terms of execution time on a UMA system or a single NUMA node. The sequential code benchmark slightly benefited by around 4% from sharing (see Figure 5.6) because its huge text segment can be shared. However, the text segment of applications is usually so small that it can be cached and even if the text segment of an application is too big to be cached, the prefetcher of the CPU preloads the instructions. This fact is seen if the results presented in Figure 5.6 are compared to the results of Figure 5.7. The non-sequential code benchmark, where instructions cannot be preloaded, spends more time for execution compared to the sequential code benchmark. Although both benchmarks execute the same number of instructions the execution time is increased by around 7% without prefetching. So, drawbacks of unshared code are reduced by large caches and the prefetcher of the CPU and thus barely noticeable with real world applications and other synthetic benchmarks.

In our evaluation, scenarios where large data segments are shareable through traditional sharing mechanisms are rare. Only MEncoder slightly benefited from sharing the input file on a remote NUMA node, since six instances processed the video in the same order. In consequence, the cache was able to prevent many remote memory accesses which is not possible if six instances of the input file are in main memory. Hence, remote memory suffer from the same drawbacks as low performance memory (higher latency and less bandwidth), this effect can also be regarded on systems with low performance memory. The cache can prevent many memory accesses on such a system, too. If data is accessed more arbitrarily this effect is most probably smaller.

The benchmarks and applications used to evaluate the runtime effects of sharing and memory deduplication (except from the sequential and the non-sequential code benchmark) do not benefit from sharing on a UMA system. However, no drawbacks are recognizable.

5.6.2 Sharing Memory on NUMA

Sharing on UMA systems has only a marginal impact on the performance of applications. In best case the performance can be improved slightly although it is barely measurable.

However, sharing on NUMA systems cannot be rated that easily. As mentioned above sharing of code has no major impact (less than 1% for the SPEC benchmarks and the kernel build) on the runtime of a process since large caches and the

	Almost No Drawbacks (<1%)	Moderate Drawbacks (around 10%)	Heavy Drawbacks (>25%)
Computationally-intensive	●		
Memory-Intensive Sequential Access		●	
Memory-Intensive Arbitrarily Access			●

Table 5.3: Performance impact of sharing across NUMA nodes for different types of applications. Memory-intensive applications with an arbitrary access of main memory suffer most from sharing, whereas computationally intensive tasks show almost no performance loss.

prefetcher of the CPU reduce the drawbacks of remote memory. Only synthetic benchmarks that are written to suffer directly from sharing code across NUMA nodes show a decreased performance (4% up to 11% depending on the benchmark). So, it is not surprising that also the SPEC hmmer benchmark depends not on the memory characteristics at all since it is computationally intensive and has only a small data and text segment. There is no variation of the execution time measurable. Most time is spent for computations and therefore memory accesses are negligible.

However, there are applications that are affected by the higher latencies and less bandwidth of remote memory. The performance of the SPEC bzip2 benchmark was decreased by 10% if remote memory was used. Also MEncoder required 10% more time when it was forced to use remote memory. The time spent for file I/O cannot cover the less performant memory accesses. This is only the case for the kernel build where many small files are read. In contrast to the kernel build, the SPEC gcc benchmark, which performs no file I/O, suffered by 31% when remote memory was used. So, the time spent for hard disk reads can cover remote memory accesses.

But there are applications that suffer even more from sharing across NUMA nodes. The SPEC libquantum benchmark is slower by almost a factor of two if remote memory accesses are necessary, although with 64 MB the data set is much smaller than the data set of the SPEC bzip2 benchmark (850 MB). The prefetcher, which in some cases can narrow the gap between main memory and caches, is not able cover the drawbacks of the remote memory in the case of SPEC libquantum since the memory accesses are arbitrarily. Prefetching was possible for the SPEC bzip2 benchmark and MEncoder since the input is processed sequentially. Also the

SPEC mcf benchmark suffered from this fact. The huge data set (1.6 GB) with arbitrary memory accesses lead to a decreased performance of 28%. So, the size of required main memory is not the only criteria to determine if an application suffers from sharing across NUMA nodes. When the working set cannot be almost completely be cached the memory access pattern has to be taken also in account.

However, these results are from worst case scenarios. The benchmarks and applications were forced to use only remote memory when sharing across NUMA nodes was simulated. Usually, the local memory is used when new memory is allocated and thus the impact of sharing across NUMA nodes can be smaller on production use systems.

Chapter 6

Conclusion

The limitation of main memory which led to the development of techniques to reduce the amount of required main memory brought a higher efficiency for today's computer systems. More applications can be executed on a single system and thus increase the workload.

There are only benefits of memory sharing on UMA systems through the traditional sharing mechanisms. Less memory is used whereas performance enhancements are barely measurable. Even dedicated benchmarks as the sequential code benchmark benefit only by around 4% which can be increased to 11% if the prefetcher cannot prefetch instructions as done by the non-sequential benchmark. All other applications with a more realistic behavior are not affected at all.

Although memory sharing can increase the workload of a computer system, it can also reduce the performance of an application on NUMA systems. Sharing across NUMA nodes can lead to a significant performance loss for memory intensive applications because remote memory accesses typically show a considerable higher latency and smaller bandwidth. In our evaluation system the latency for remote accesses increases by around 75% while the bandwidth decreases by nearly 33%. The performance of the SPEC mcf benchmark, which requires 1.6 GB of main memory, is reduced by 28% if it is forced to use remote memory. However, not only applications with a huge demand for main memory are negatively affected by sharing across NUMA nodes, but also applications with small data sets. The execution time of SPEC libquantum increases by a factor of two, although the size of the data set is only 64 MB.

Our evaluation revealed that the performance of real world applications can be reduced, too. MEncoder requires at least 10% more time to convert a 180 MB video if remote memory is used. The runtime effects of memory sharing are even measurable although the file I/O for reading the video file covers a part of the drawbacks of remote memory accesses. Also, the kernel build requires around 6% more time to build a kernel, which is quite small in contrast to the 31% of the SPECC gcc benchmark. So, file I/O outweighs a part of the drawbacks of remote memory accesses, but they are still measurable. Therefore, especially deduplication techniques as described in Section 2.1.2 can lead to a loss of performance if memory is shared across NUMA nodes.

Also, there are applications that do not suffer from sharing at all. The SPEC hmmer benchmark depends not on memory characteristics since it has only a small text and data segment (all in all around 20 MB), memory is not arbitrarily accessed and the benchmark is computationally intensive. So, it is difficult to determine which applications suffer from sharing. Many aspects such as caches, the prefetcher, memory access pattern, required file I/O need to be considered. Table 5.3 indicates which types of applications are influenced by sharing.

6.1 Future Work

This work showed that there are runtime effects of memory sharing and deduplication within a synthetic environment. Further researches have to evaluate if these effects occur also in production use environments. If there is a performance loss too, the memory usage pattern of an application must be taken into account if deduplication is used across NUMA nodes.

For this purpose there has to be a measure, which determines if sharing decreases the performance. This must be determined at runtime. The cache misses, triggered by a process, could be a good measure. Processes, which have a huge amount of cache misses, uses the main memory in an arbitrarily way and suffer most from the higher latencies of remote memory.

Bibliography

[1] *CPUSET*. http://www.kernel.org/doc/man-pages/online/pages/man7/cpuset.7.html.

[2] *Ermine*. http://www.magicermine.com/index.html.

[3] *LMbench*. http://www.bitmover.com/lmbench/.

[4] *MEncoder*. http://mplayerhq.hu/design7/news.html.

[5] *Statifier*. http://statifier.sourceforge.net/.

[6] *STREAM Benchmark*. http://www.streambench.org/.

[7] *Taskset*. http://linux.die.net/man/1/taskset.

[8] *An Introduction to the Intel QuickPath Interconnect*. http://www.intel.com/content/www/us/en/io/quickpath-technology/quick-path-interconnect-introduction-paper.html, 2009.

[9] *SPEC CPU2006 Documentation*. http://www.spec.org/auto/cpu2006/Docs/, 2011.

[10] Greg Gagne Abraham Silberschatz, Peter B. Galvin. *Operating System Concepts*. Wiley, 2010.

[11] Hendrik Weimer Björn Butscher. Simulation eines quantencomputers. Technical report, Universität Stuttgart, March 2012.

[12] Jan-Jan Wu Chao-Rui Chang and Pangfeng Liu. An empirical study on memory sharing of virtual machines for server consolidation. *Proceeding ISPA '11 Proceedings of the 2011 IEEE Ninth International Symposium on Parallel and Distributed Processing with Applications*, 2011.

[13] Diwaker Gupta et al. Difference engine: Harnessing memory redundancy in virtual machines. Technical report, University of California, San Diego, 2010.

[14] Grzegorz Miłós et al. Satori: Enlightened page sharing. Technical report, USENIX Association, 2009.

[15] Intel. What you need to know about prefetching. In *What you Need to Know about Prefetching*. intel, August 2009.

[16] M. Tim Jones. *Anatomy of Linux Kernel Shared Memory*. IBM Corporation, http://www.ibm.com/developerworks/linux/library/l-kernel-shared-memory/, April 2010.

[17] Daniel Pierre Bovet; MarcoCesati. *Understantind the Linux Kernel*. O'Reilly, 2006.

[18] Carl A. Waldspurger. Memory resource management in vmware esx server. Technical report, VMware, Inc., December 2002.