Belegarbeit

Evaluation of migration costs in multi-core CPU scheduling

Stefan Wächtler

March 7, 2012

Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Professur Betriebssysteme

Betreuender Hochschullehrer: Prof. Dr. rer. nat. Hermann Härtig
Betreuender Mitarbeiter: Ing. Michal Sojka Ph.D.

Erklärung

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

Dresden, den 7. März 2012

Stefan Wächtler

Contents

List of Figures

1 Introduction

Up to the middle of the last decade, the common approach to improve the performance of a CPU was to increase its clock rate. However, a higher clock rate correlates with the production of heat and, therefore, at some point in time it was necessary to come up with another principle to increase the overall performance: multi-core CPUs. Having independently working cores enables to run more than one task in parallel but brings new challenges to operating-system schedulers: Optimal real-time scheduling algorithms that provide scheduling guarantees in multi-core systems are NP-complete. Hence, they cannot be used in practice.

Optimal real-time scheduling algorithms for single cores are well studied and available (e.g. see [Liu00]), so a widely used approach for systems with n cores is to divide the set of all tasks into n distinct subsets. Each core in the system runs its own (single-core) scheduling algorithm with its own subset of tasks. This method is called *Partitioned Scheduling* [BBA10b]. Tasks are statically assigned to single cores and are not allowed to migrate. Another principle is called *Global Scheduling*: Tasks can freely migrate between all cores of the system. Mixtures of both principles are called *Clustered Scheduling*: Tasks and cores are divided into distinct subsets. Each subset of tasks is assigned to a specific subset of cores that runs its own scheduling algorithm.

However, when comparing the different approaches, the migration costs are often neglected, which is for practical purpose unacceptable. When searching for more efficient ways of multi-core scheduling, it is important to analyze the costs of task migrations. The costs consist of two parts – *direct* and *indirect*. Direct costs describe the necessary effort to transfer the values of the registers of the source CPU to the registers of the target CPU. The costs consist of a fixed number of copy operations to/from the memory and are independent of the migrated task. Consequently, direct costs can be estimated quite easily.

In contrast, the indirect costs are hard to quantify because they are influenced by many factors like the size of the task's memory working set, memory access pattern, whether the task was running or was preempted for some time, the cache/memory architecture of the system, the overall memory load in the multi-core system, etc.

The goal of this work is to experimentally evaluate the migration's indirect costs in a multi-core system. The costs are compared to those of a preemption using two different system architectures. The factors described above are varied to find their influences on migration costs. Having comparative values of preemption and migration costs, statements about the consequences of different scheduling decisions are possible. Furthermore, the prediction of these costs might help the system's scheduler to minimize the penalty of its decisions.

The next chapter of this work introduces the necessary basics and terminology that is important to understand the work. Special attention is turned to caches in addition to the used measurement systems. Furthermore, the operating system used in the experiments, called *NOVA*, is introduced.

Chapter 3 gives an overview of how the indirect migration costs can be measured and describes the design of one method in depth. An exemplary task capable of measuring preemption and migration costs is designed. Additionally, threads able to run as background load to simulate a loaded system are examined: One approach is able to generate reproducible results, the other one generates a realistic load. The last part of this chapter describes the design of the experiments that were carried out.

Some implementation details are given in Chapter 4. NOVA has no built-in support for migrations, so the implementation of a reliable mechanism and its limitations are presented. Especially the setup and execution of threads being able to migrate between cores of the system is in the scope of interest.

The evaluation of the measurements presented in Chapter 5 is one of the major parts of this work. The background loads are classified and cache-related costs that are introduced by a preemption or migration are discussed.

The last chapter compares the results of the evaluation with other published works and draws conclusions about possible scheduling decisions in multi-core systems.

2 Fundamentals

The first section of this chapter defines basic terms that are important within this work. Caches are introduced to define cache-related preemption and migration delays (CPMD). Afterwards, the architectures used during the measurements and some of their technical details are presented.

The third section deals with NOVA and some of its basic components. This section is of particular importance for the next chapters because terminology is introduced that is frequently used during this work.

Miscellaneous terms appearing from time to time like the time stamp counter (TSC) are covered in the fourth section. The last section of this chapter describes observations about CPMD made by Bastoni et al. [BBA10a].

2.1 Terms and definitions

2.1.1 Caches

This section gives an introduction into caching techniques that are necessary to understand the principles of cache-related preemption and migration delays. Due to the complexity of caches not all details can be presented.

Definition

In general, caches are memories that store recently used data to serve future requests faster. The values in the cache are either earlier computed results or copies of values that are also stored elsewhere. This definition holds for software as well as for hardware caches.

For example, a software cache is used in a web browser to store the images of web pages a user has already visited. If the website is opened again, the cached images need not be retransmitted and, thus, the page will load faster.

The *CPU cache* is a typical hardware cache. It is a small and fast memory that stores data of main memory to speedup the accesses to that data on future reads.

To determine cache-related preemption and migration delays, CPU caches are in the scope of interest. Therefore the term *cache* will be used as a synonym for CPU caches throughout the work.

Originally, CPU caches were introduced to reduce the performance gap between the processor and main memory in von Neumann architectures. In these architectures the memory controller puts the memory address of a requested word on the address bus and the memory responds with the corresponding data on the data bus. Over the years, the clock rates of CPUs grew and the system's performance was determined by the

clock rate of the system bus that in turn determined the speed of interaction between memory and CPU. To find a solution for this imbalance, one came up with the idea to place a fast accessible memory – the cache – as near as possible to the CPU. The cache transparently saves data that comes from main memory. When later the CPU wants to reload data that was previously fetched from main memory[1], the cache can handle the request. Hence, several requests to the same memory address gain a speedup of multiple orders of magnitude because the access time to the cache is several times faster than main memory accesses.

Whenever the CPU writes to a memory address, it is sufficient to update the value in the cache and to update main memory later on. This approach hides the latency of main memory accesses and increases the overall performance.

An additional advantage of caches is a lower bus contention: If parts of the data are served by the cache, the total number of accesses to main memory will decrease.

A data request handled by the cache without accessing main memory is called a *cache hit*. The other case is called a *cache miss*.

Types of CPU caches

Based on the kind of data saved in the cache, the following notations are used: Caches which hold instructions or data are called *Instruction Cache* or *Data Cache*. The term *Unified Cache* is used for caches that save instructions and data at the same time.

During this work the term cache will be used as a synonym for data or unified CPU caches (unless explicitly stated otherwise).

Cache hierarchies

Modern computer architectures build up a hierarchy of caches. Recent systems mostly possess three different cache levels: The *Level 1 (L1)* cache is split into a separate instruction and data cache. Both are located like the *Level 2 (L2)* and the larger *Level 3 (L3)* or *last level* cache directly on the CPU die (Fig. 2.1).

If the CPU requests data that is not contained in L1 but in a higher level, the data will be moved into the lowest cache level.

Additional terminology is used in multi-core systems: A cache that is used by more than one core in the CPU is called a *shared* cache. Otherwise, it is a *private* cache.

The designer of a system with a hierarchically arranged topology of cache levels must specify whether data held in one cache level is also saved in higher levels. Such caches are called *inclusive* caches. Otherwise, the cache is called *exclusive*. As a consequence of the latter, the same data can never reside in two exclusive caches at the same time.

Cache lines

Caches are organized in *cache lines*. A cache line is the unit of data that is interchanged between the cache and main memory and between caches of different levels. Further-

[1] The property that a resource referenced some time ago will be accessed again in the near future is called *temporal locality*.

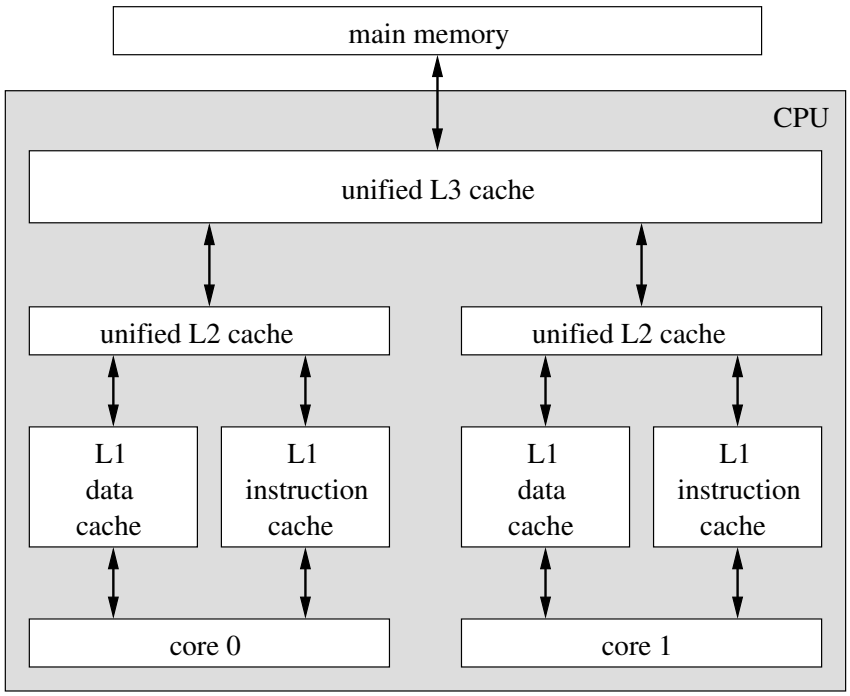


Figure 2.1: Example of the cache hierarchy in recent systems.

more, a cache line is the smallest unit that can be placed in or evicted from a cache at once.

The size of a cache line differs between system architectures and can also vary in caches located at different levels of the hierarchy. A common size in recent systems is 64 bytes.

The disposition into cache lines has several benefits compared with maintaining the units in byte granularity. On the one hand, the principle of *spacial locality* is supported: If the CPU fetches some data from main memory, the load of a whole cache line will increase the probability that later read requests of data near to the first one can be satisfied by the same cache line. Hence, the performance arises. On the other hand, several flags and additional information like the corresponding address in memory are needed to keep track of every cache line and, therefore, some overhead is unavoidable. Larger cache lines reduce this overhead because the number of control information per byte decreases.

Each cache line must contain additional information about its state that describe the actuality of the saved data:

- Invalid - the cache line must not be used as it is out of date
- Valid - the cache line holds valid data that can be used by the CPU
- Modified - the cache line holds data modified by the CPU, but the corresponding data in main memory is stale

Invalid cache lines hold no valid data and, thus, must not be used to transfer data to the CPU. After the computer is powered on all cache lines are in the invalid state. Also special CPU instructions like *WBINVL* on x86 mark all cache lines in the cache as invalid.

In contrast, valid cache lines hold actual data that can be used by the CPU without accessing main memory. Cache lines enter a valid state when the content was previously loaded from main memory and when both the cache and main memory contain the same data.

Modified cache lines cannot exist if the system uses a *write-through* strategy. In this case, *STORE*-instructions of the CPU save data directly in main memory, the effected cache line becomes valid.

Otherwise, STORE instructions result in modified cache lines without updating the content of main memory immediately. The content is written back when the cache line is evicted from the cache. Therefore, the strategy is also named *write-back*. The advantage is obvious: The time needed to accomplish a STORE instruction decreases and the total number of memory requests using the system bus decreases, too.

The next paragraph deals with new problems that arise when using the write-back strategy in multi-core systems.

Cache coherence

Caches introduce new problems in multi-processor systems as the following example shows: Two processors use a common memory but do not share any cache level. One of them reads some data from a certain memory location. After some calculation, it writes an updated value back to the same memory address. When using a write-back strategy, the new value is not placed in main memory immediately. Instead, it resides as a modified cache line in the private cache of the processor. If later the other processor reads from the memory location, it will receive an outdated value even if the access of both processors will be synchronized.

New mechanisms like *cache coherency protocols* (e.g. the *MESI* protocol) were introduced to prevent such inconsistent views on data (see e.g. [amd11, Chapter 7.3]).

Cache accesses

When the CPU reads some data, there has to be a procedure to detect whether the requested data is contained in one of the cache lines. However, the cache line must be found and its state must be either valid or modified to use its data. Otherwise, a cache miss occurs.

Every cache line is identified by a *tag*. The tag is determined by a function that takes the physical address of the requested data as an input. Usually this function extracts the n the most significant bits of the address. Depending on the kind of the cache, an index that is part of the physical address specifies a subset of all cache lines that have to be inspected in order to decide about the existence of the requested data in the cache.

If this subset is equal to the set of all cache lines, the cache will be called *fully associative*. Every cache line might contain the requested data. Depending on the size

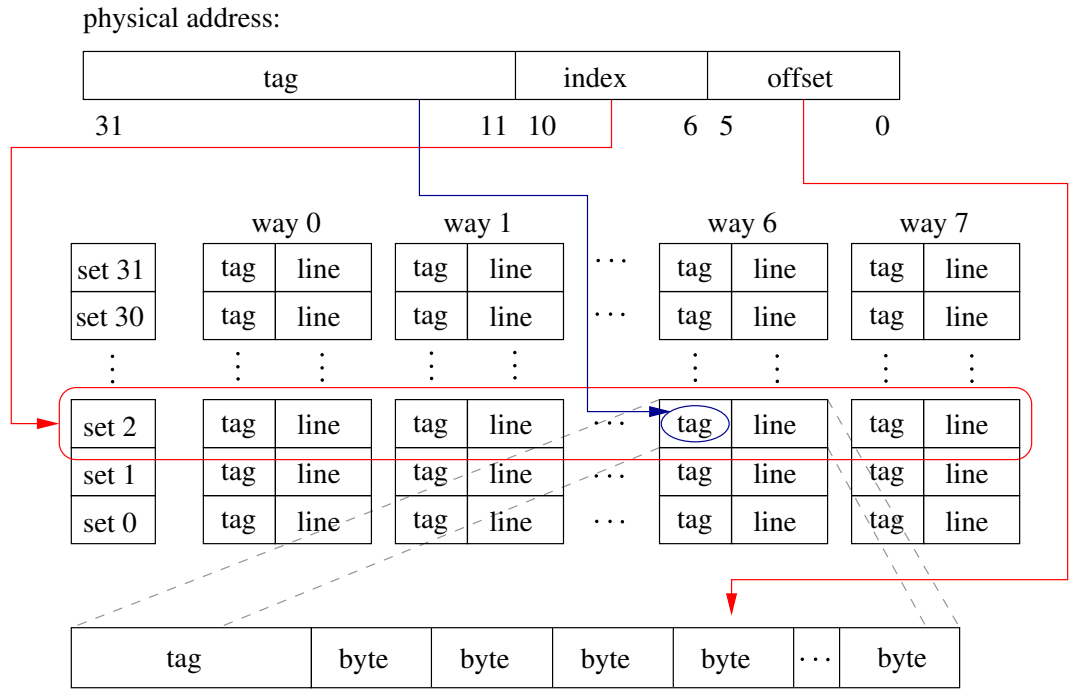


Figure 2.2: Example of an 8-way set associative cache. The index of the physical address specifies the set. If the tag of any valid or modified cache line within this set matches the tag of the physical address, the offset will determine which data to load from the cache line.

of the cache and the size of a cache line, a large number of comparisons is necessary to decide about the existence of the data in the cache. In practice, these comparisons cannot be done in parallel because the circuit would be too big, so other strategies are in use.

An *n-way set associative* cache has subsets containing n cache lines whose tags can be compared in parallel in order to decide whether the cache holds the requested data. Fig. 2.2 shows an example.

A *direct-mapped* cache has subsets containing exactly one cache line. Therefore, only one comparison is necessary to decide about the existence of the requested data.

It is easy to see that the effort to determine whether a cache line is available decreases with the number of comparisons: a fully associative cache needs as many comparisons as cache lines are in the cache, an n-way set associative cache needs n, and a direct-mapped cache needs only one comparison.

But the simplicity has its costs: The ratio between cache hits and cache misses becomes worse with decreasing associativity.

An example helps to make this clear: In a direct-mapped cache even two cache lines with the same index are problematic: The first cache line is accessed and, thus, loaded into the cache. If the other cache line is accessed afterwards, the first one will be evicted in order to make space for the second one. If the first one is then accessed again, the second cache line will be evicted and so on. The total number of available cache lines

does not matter. The example can be easily extended to an n-way set associative cache where $n + 1$ cache lines are needed. To have the same example in a fully associative cache with a total number of m cache lines, at least $m + 1$ cache lines are needed to achieve the same effect.

In practice the n-way set associative cache is often used as a compromise.

Reasons for cache misses; cache interference and cache affinity

Cache-related preemption and migration delays originate from cache misses. Therefore, this paragraph deals with the different kinds of cache misses and their corresponding reasons [BBA10a]. In addition, the terms cache interference and cache affinity are defined.

- *Compulsory misses* – When a job starts its execution for the very first time, this kind of misses occur because there is no valid data of the job in the cache so far.
- *Capacity misses* – Useful cache lines are evicted in order to make space for new ones. This happens if the number of accessed cache lines of a job exceeds the total number of cache lines.
- *Conflict misses* occur only in direct-mapped or n-way set associative caches if the mapping constraints of a cache line are not fulfilled. Therefore, useful cache lines are evicted even if not the whole cache is filled with valid or modified cache lines.
- *Coherency misses* occur only in multi-core systems. When a core modifies a cache line that is also contained in private caches of other cores, the cache coherency protocol has to invalidate the cache line there to prevent the usage of stale data.

The term *cache interference* describes the occurrence of frequent capacity and conflict misses in caches when multiple threads are running in parallel and the caches cannot hold the combined working set of the involved jobs.

A job incurs a bundle of compulsory misses at the beginning of its execution. During computation more and more cache lines of the working set are loaded into the cache and, thus, the overall rate of cache misses decreases over time. This property is also called *cache affinity*.

Cache replacement strategies

When a valid or modified cache line must be evicted in order to make space for a new one, a mechanism called *cache replacement strategy* must decide which cache line to remove.

In a direct-mapped cache the position of the new cache line is exactly specified. Therefore, the decision is obvious.

In an n-way set associative or fully associative cache the physical address of the new cache line determines the set of possible replacement candidates. The replacement strategy has to decide which cache line within this set should be chosen for replacement.

An overview about different replacement strategies and their properties can be found in [ZPL01, Chapter 2].

2.1.2 Cache-related preemption and migration delay

Cache-related preemption and migration delays (CPMD) are delays caused by the use of CPU caches. The terms preemption and migration will be defined in order to specify CPMD.

Preemption and migration

A system is characterized by a set of tasks $T = \{T_1, ..., T_n\}$. Each task T_i consists of an arbitrary number of jobs. For simplicity, all jobs of a specific task T_i have a static assigned priority p_i.

A preemption occurs when job j of an arbitrary task running on core x with the priority p_n is paused by the OS scheduler in order to execute job k of another task with the higher priority p_m on core x. After l units of time job k finishes, and job j resumes on core x. Job j incurred a preemption of length l on core x.

A preemption of a job occurs when an interrupt is triggered on a core: The currently running job on that core is interrupted for a short time to handle the interrupt and the scheduler might be invoked thereafter. The scheduler either resumes the interrupted job or it selects another job for execution.

Especially on single-core systems this approach improves the system's responsiveness and achieves a more interactive system. Imagine a long running job that is not preempted by the scheduler in favor of another job with a short execution time: The system might be unusable until the first job stops its execution. By preempting the long running job periodically, it is possible to simulate the parallel execution of multiple jobs even on single-core systems.

Besides the preemption of a job, multi-core systems allow also migrations: The migration of job j is the action of disrupting the computation of j on core x to resume the computation on core y where $x \neq y$.

The scheduler of a system decides when to migrate a job from one core to another core, e.g., if the load of the initial core becomes too high.

Both a preemption and migration should be completely transparent for the running job. In reality, this is hard to achieve especially when resources like time are involved: If a job measures points in time periodically by the usage of a fine grained clock source, every preemption will result in a shift of the next measured point in time depending on the preemption length.

Also migrations influence the execution of jobs. The usage of fine grained time stamp counters may be problematic: A thread is going to measure the execution time of an operation. Therefore, it reads the *time stamp counter* (TSC) and starts the operation. During the execution the scheduler is invoked and the thread is migrated to another core. After the operation has finished, the thread reads TSC again to measure the length of the time interval. The result might be wrong due to the lack of synchronized clocks of the involved cores.

Definition of cache-related preemption and migration delay (CPMD)

In order to describe CPMD accordingly, it is necessary to introduce some more terminology.

The execution time of job j that does neither experience a preemption nor a migration during its execution is denoted as e. Let t_1 be the point in time when job j starts its execution and let t_2 be the point in time when job j finishes its computations. The execution time e can be defined as $e = t_2 - t_1$.

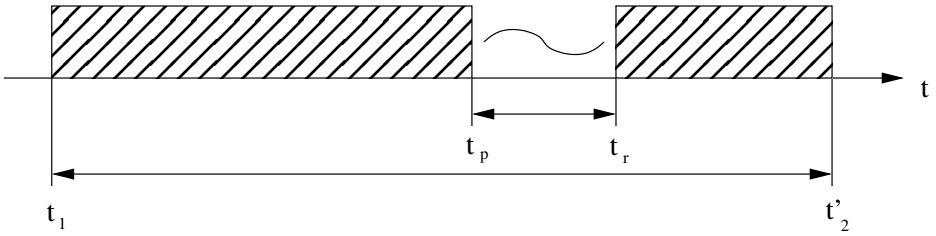


Figure 2.3: Job j starts to run at time t_1 and is preempted at time t_p in favor of another job that runs from t_p to t_r on the same core. After it has finished, job j resumes until t'_2. The length of the interval between t_p and t_r is also called the *preemption length* of job j.

Now assume that the same job j starts its execution again at t_1, but it is preempted in favor of a higher prioritized job k at point t_p (Fig. 2.3). After k has finished its computations at time t_r, job j is resumed again until it stops at time t'_2. The execution time e' of the preempted job can be determined by calculating $e' = (t'_2 - t_1) - (t_r - t_p)$.

The execution time e' of a migrated job is calculated as $e' = t'_2 - t_1$ where t_1 is the start of execution and t'_2 defines the end of execution.

CPMD describes the dependency between the execution time e of a continuously running job and the execution time e' of the same job that incurred a preemption or migration during its execution. The following property holds: $e \leq e'$.[2] The costs of CPMD can be determined by calculating $CPMD = e' - e$.

In case of a preemption, the increased execution time of a job is caused by a loss of cache affinity during the preemption and, thus, more cache misses when the job is resumed. Therefore, the execution time e' grows, which results in the cache-related preemption delay (CPD). Similar reasons explain the costs of the cache-related migration delay (CMD): If not all caches are shared between the involved cores, it will be necessary to transmit the cache lines of the job's working set from the private caches of the initial core to the caches of the target when these are accessed again.

If a migration takes place between cores sharing all caches, the costs of CMD will be low. Zero costs are unlikely because the overhead introduced by the implementation of the migration does also evict useful cache lines of the job's working set.

[2] There are rare situations, in which $e > e'$. Further information are presented in Sec. 5.3.3.

2.2 Architectures used for measurements

This section describes details of the architectures used to measure CPMD. A special focus is set on the different cache levels.

2.2.1 Intel Xeon X5650 (Dell system)

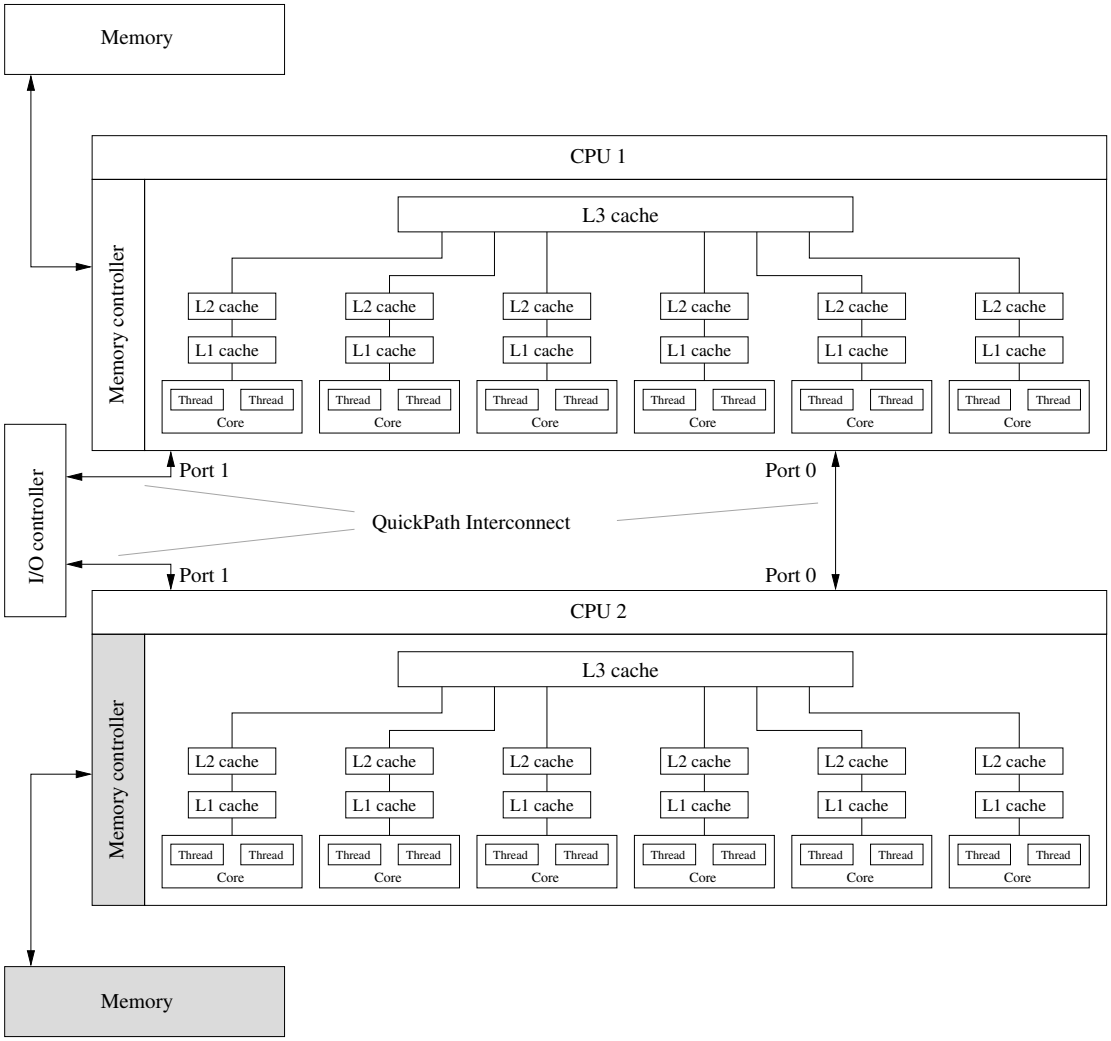


Figure 2.4: Architecture of Dell Precision T7500. The gray boxes are not used due to constraints of the NOVA operating system.

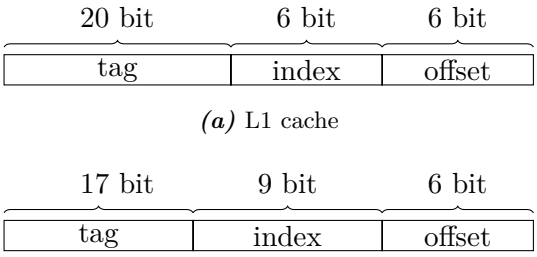
One system used to determine the costs of CPMD is the Precision T7500 produced by Dell. Fig. 2.4 gives a brief overview of the system.

Design of the CPU - Simultaneous Multithreading

The system contains two identical Intel Xeon X5650 CPUs comprising of 6 cores per processor. Intel adapted the technique of *Simultaneous Multithreading (SMT)* [TEL98] calling it *Intel Hyper-Threading*. If SMT is enabled, each core will behave like two independent *logical cores*. Each logical core can execute one thread that is also called *hardware thread*. If SMT is disabled, each physical core consists of only one core.

Requirements to use Intel's implementation of SMT are: a capable CPU, support by the chipset, and BIOS. The operating system should be also aware of SMT in order to optimize the allocation of threads [int11b, Chapter 2.2.8].

Design of the caches



(a) L1 cache

(b) L2 cache

Figure 2.5: Structure of physical addresses to determine cache line.

Every CPU possesses a hierarchy of three different cache levels. Each of them has a cache line size of 64 bytes. One physical core has two 32 KB L1 caches, one for instructions and one for data only. The corresponding 512 cache lines are organized as an 8-way set associative cache. Therefore, 64 sets of cache lines exist and eight comparisons are necessary to decide whether a data request can be fulfilled by the cache. The least significant 6 bits of a physical address (Fig. 2.5(a)) are used as an offset to specify the byte or word within a cache line. Bits 6 to 11 specify the set of cache lines that might contain the requested data. The most significant 20 bits describe the tag that is used as the comparison criterion to determine the existence of a cache line.

The L2 cache is also an 8-way set associative unified cache with a total size of 256 KB unique to each physical core. The number of index bits that determine the corresponding set are three bits larger in comparison to the index bits of the L1 cache. Therefore, the tag of each cache line is three bits shorter (Fig. 2.5(b)).

The L3 cache is a 16-way set associative unified cache that is shared between all cores of the CPU. The total size of 12 MB does not allow a straightforward mapping of the index bits into a specific subset: Because each cache line has a size of 64 bytes, there are 196,608 cache lines. Having an 16-way set associative cache, there must be 12,288 sets of cache lines. In contrast to the L1 and L2 cache, it is not possible to have a direct mapping of the physical addresses to the cache line set because the total number

of cache lines is no power of two and, thus, an uniformly distribution is not possible. Instead, a hash function is necessary that takes at least 14 bits as input to decide which cache set should be used. Implementation details by Intel about the hash function are not available.

Whereas L1 and L2 are exclusive caches, the L3 cache is inclusive [int11a, Chapter 2.3.4]. Therefore, data that is saved in the L1 or L2 cache is also contained in the L3 cache. Constructing L3 as an inclusive cache reduces snoop traffic between the cores on a processor that is necessary for cache coherence: If a core accesses some data that is not contained as a valid or modified cache line in L3, one can be sure that the data is also not in one of the private caches of any other core on the same CPU. It is the task of the cache coherency protocol to check whether the L3 cache of another processor holds the actual data.

QuickPath Interconnect

Each processor possesses two *QuickPath Interconnect (QPI)* [qpi09] ports. QPI was introduced by Intel in the second half of 2008. It replaces the Front Side Bus and connects processors in a distributed shared memory system. Intel advertises QPI as a high bandwidth, low latency interconnect that provides good scalability. It is a point to point connection that uses packets for information delivery. When QPI was introduced, the memory controller moved from the northbridge directly onto the processor die. Therefore, a main memory access must be distinguished between a direct and remote access. If the addressed memory is local to the processor, the access will be handled by its own memory controller. Otherwise, QPI is used to send a packet to the processor that has direct access. In the latter case the access time to main memory is higher due to the use of QPI. A cache coherency protocol preserves inconsistent memory views. Therefore, Dell's system can also be called a *cache coherent non uniform memory access* (ccNuma) system.

One of the QPI links connects the processor to the other one. The second port attaches each processor with the I/O controller. The ordering of the memory modules on the main board influences whether a memory access is local or remote [int11d, p. 28].

Memory accesses

The system has 24 GB of main memory that is uniformly distributed between both CPUs. Because the total amount of memory each CPU can access is larger than 4 GB, the usage of a 32 bit operating system that is not aware of the Physical Address Extension (PAE) results in a specific characteristic of the system: All addressable memory is connected locally to exactly one CPU and all main memory accesses on this CPU are directly handled by the memory controller on the die[3]. Main memory accesses of the other core always use the QPI link to access remote memory.

[3] The Dell system can be configured in a NUMA or SMP mode. All measurements were executed in NUMA mode. In SMP mode the entire address space is split into chunks that are distributed among the memory banks. This technique improves the performance of operating systems that are not aware of the characteristics of main memory.

2.2.2 Phenom 9550 (AMD system)

The AMD Phenom 9550 is a quad core processor with three cache levels. Each core possesses two identical 64 KB L1 caches, one for instructions and one for data only. L1 is a 2-way set associative cache with 512 sets. Additionally, each core has an 16-way set associative unified 512 KB L2 cache also comprising of 512 sets. All cores share a 2 MB 32-way set associative unified L3 cache. It has got 1024 different sets. In contrast to Intel's architecture, these caches are exclusive. Thus, the overall amount of data which can be saved at a time is the sum of the size of all caches.

The exclusive design of the last level cache adds additional problems: A read miss in the L3 cache produced by a core does not necessarily mean that the cache line is not in one of the private caches of another CPU. Therefore, in contrast to Intel's Xeon, the Phenom has to check if the requested data is contained in one of the private caches of another core. If the data is found there, the cache line will be transferred directly between the caches.

2.3 NOVA

NOVA[4] is a recursive acronym for *NOVA OS Virtualization Architecture*. It consists of a micro-kernel written and developed by Udo Steinberg and *NUL*, the *NOVA userland*, an unprivileged multi-server user environment providing some basic functionality. Most parts of NOVA are written in C++.

The micro-kernel is also called a micro-hypervisor because it supports recent virtualization technologies. NOVA can be used to run general applications in user mode or to host unmodified guest operating systems in *virtual machines (VMs)* by using a *virtual machine monitor (VMM)*. The aim of the micro-kernel is to design a virtualization technology with improved security. This should be achieved by reducing the trusted computing base (TCB) of the system and by lowering the communication overhead between the components running in user mode through a careful design [SK10].

The following subsections specify some important basics of NOVA that are necessary to describe the design and implementation details.

2.3.1 Capabilities

NOVA handles five different types of kernel objects: *protection domains*, *execution contexts*, *scheduling contexts*, *portals*, and *semaphores*. If a kernel object is created, NOVA will prepare a so called capability that refers to the newly created kernel object. This capability is used like a key to gain access to the corresponding kernel object by the holder of the capability, e.g., an application. Hence, a capability is opaque and immutable to the user, so it is not possible to inspect, modify, or address the capability directly [SK10]. Capabilities are used to achieve a fine grained control mechanism to specify what a application is able and allowed to do.

[4] `http://www.hypervisor.org/`

2.3.2 Protection Domains

A protection domain (PD) is a unit of protection and isolation. Like other kernel objects it is referenced by a capability. Protection domains consist of a set of capabilities to other kernel objects and platform resources like memory or I/O ports. The protection domain keeps track of the address space through capabilities to page frames [Ste11].

2.3.3 Execution Contexts

An execution context (EC) is the unit of execution within NOVA and can be compared to threads in a traditional system. The EC is referenced by a capability and is permanently bound to a specific PD and CPU. The EC consists of a scheduling context that is described in the next section, a User Thread Control Block (UTCB), an event selector, CPU and FPU registers, and a reply capability register [Ste11].

One UTCB is associated with each execution context. The UTCB is used for communication and to transfer capabilities within protection domains through the usage of communication objects called portals.

Each hardware exception in NOVA has an associated positive number that is added to the event selector to get a capability selector. To handle a specific exception, a capability to a portal with the corresponding index has to be created and bound to a user-defined function. Thus, an exception results in a call to the specified function handler.

2.3.4 Scheduling Contexts

A scheduling context (SC) is a unit of dispatching and prioritization [Ste11]. It is permanently bound at any point in time to exactly one EC and, therefore, to exactly one CPU. To start the execution of an EC, a SC must be created and assigned to the EC. In order to create such a SC a priority and time quantum must be defined. The priority has to be in the range from 0 to 255, where 255 is the highest available priority. The time quantum defines the time of execution before the scheduler of the system is invoked to select the next SC.

The NOVA scheduler handles different priority queues per CPU. Whenever the time quantum of the SC of a running EC exceeds, the EC is stopped and the time quantum of the SC is replenished. Additionally, the SC is enqueued to the end of its associated priority queue. The scheduler determines the queue with the highest priority that contains a SC whose associated EC is ready to run. The SC waiting to run for the longest time is taken from the queue by the scheduler. Consequently, scheduling decisions within a priority queue are chosen round robin.

2.3.5 Portals

Portals are used as communication end points in NOVA. They are also used to define exception handlers as described in 2.3.3.

2.3.6 Semaphores

A semaphore (SM) is a mechanism to synchronize the execution of ECs. A semaphore uses an internal counter that is initialized with a user-defined value during creation. The semaphore provides two operations: *down* and *up*.

When calling the down-operation through the capability referencing the SM, the internal counter of the SM is inspected. If it is larger than zero, the counter will be decremented by one, and the execution will proceed. Otherwise, the EC blocks on the semaphore.

If the up-operation of a semaphore is called and there are blocked ECs waiting for the semaphore, one of them will be unblocked in order to resume its operation. If there are no blocked ECs waiting for the SM, the internal counter of the SM will be incremented by one.

2.3.7 Threads

There is no thread kernel object available in NOVA. In fact, a thread as known from other operating systems can be seen as the compound of a SC and EC. However, a SC is permanently bound to a specific EC that is in turn permanently bound to a specific CPU. Consequently, by using the term thread in this context, it is not possible to migrate a thread between cores since the composition of SC and EC, and EC and CPU is permanently fixed.

A migration of a thread to another core in this context mean that the execution of an EC is interrupted and its internal state is transferred into another EC on the target CPU. A SC has to be assigned to the target EC to keep the execution of the job ongoing. It might also be possible to change the priority of a computation during the migration because the priority of both involved SCs might be different. However, in this work the priorities of the involved SCs stays always the same and does never chance during execution.

2.4 Miscellaneous

2.4.1 System Management Mode

System Management Mode (SMM) is an operational state of recent processors. It is enabled by activating a particular CPU pin or by a special interrupt caused by the *advanced programmable interrupt controller (APIC)*. SMM is used as a transparent mechanism to interrupt the currently running task on the CPU to perform platform specific functions like power management and system security (see e.g. [int11b]).

From the perspective of the operating system SMM results in jumps of the internal clocks that cannot be avoided easily. If measurements in a system are accomplished, SMM will lead to imprecise results.

A tool called *hwlatdetect* is available on Linux that can be used to determine the influence of SMM. The Dell system was traced for 5 minutes, but no occurrence of SMM

was measurable[5]. Therefore, SMM should not influence the measurements on the Dell system.

2.4.2 Time stamp counter

Beginning with the Pentium, Intel introduces a 64 bit *time stamp counter (TSC)* that is reset to zero on power on. The value of the counter is incremented with every processor cycle and allows applications to measure the time at a fine grained granularity. The duration of arbitrary operations can be estimated by reading TSC before and after the operation and by calculating the difference between the two values. The result divided by the clock rate of the CPU determines the period of time.

Caution is recommended in multi-core systems because the counters of the cores need not to be synchronized perfectly. In addition, power saving technologies that throttle the CPU speed can result in wrong values because the values of the counters are incremented with a constant rate.

TSC is used to determine all periods of time within this work, but it is never used to measure any duration in which a migration might occur.

2.5 Related Work

Bastoni et al. already analyzed the influence of CPMD in relation to scheduling decisions [BBA10a]. The costs of preemptions and migrations under different scenarios were evaluated and resulted in five different observations:

1. The predictability of CPMD depends heavily on the size of the different cache levels.
2. There are *no substantial differences* between the costs of a preemption and the costs of a migration in a loaded system.
3. Preemptions cause always less delay than migrations in an idle system. The costs of L3 and memory migrations are comparable.
4. The costs of CPMD are strongly related to the preemption length unless cache affinity is completely.
5. The number of tasks in a system does not significantly influence CPMD.

The measurement system used in [BBA10a] comprised of four Intel Xeon L7455 processors each one with six cores and uniform memory access (UMA). All cores on a processor share an unified 12 MB L3 cache. Groups of two cores share an 3 MB L2 cache, every core possesses an 32 KB L1 data cache and an identical instruction cache.

[5] A threshold that is used as an indicator for long interrupts for example caused by SMM can be passed as an argument to hwlatdetect. Even the lowest possible number of 1 μs was unable to find these interrupts caused by SMM

3 Design

The first section of this chapter presents two different methods to measure the costs of CPMD and discusses their advantages and disadvantages.

The second section describes the design of a working set that is usable to determine the costs of CPMD. Two different approaches to simulate background loads are discussed in the third section.

At the end of this chapter a detailed characterization of the experiments that were carried out and evaluated later on is presented.

3.1 Measurement of CPMD

In section 2.1.2 the sources of CPMD in recent systems were discussed. In order to determine realistic costs of it, a reliable mechanism to detect and to measure the influences of CPMD has to be designed. This section describes how an application can act to achieve this goal.

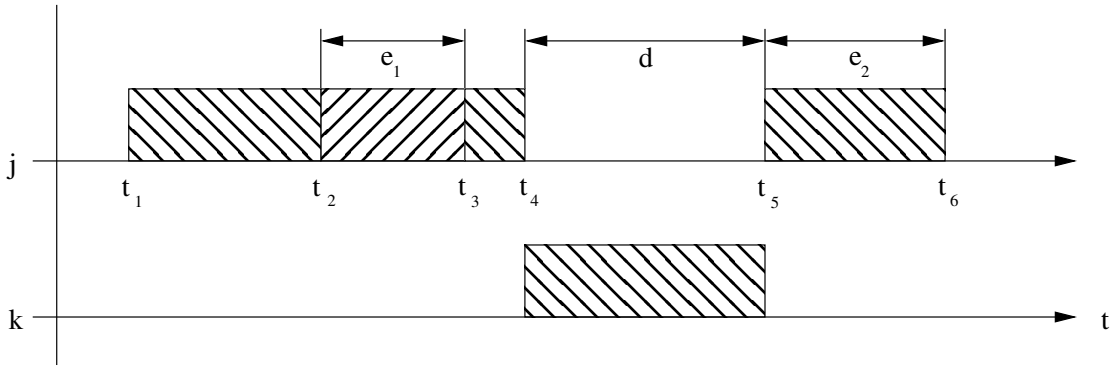


Figure 3.1: Plan of execution to measure the influence of CPMD.

A job j repeatedly accessing its working set is released in a system at time t_1 and is chosen by the scheduler to execute (Fig. 3.1). At the beginning, the job incurs compulsory cache misses. At t_2 the job accessed its working set for the first time and most parts of it are loaded into the cache. Between t_2 and t_3 the job accesses its working set for the second time. At t_4 the scheduler is invoked and preempts job j that currently accesses its working set for a third time. It might be that job j exhausted its time quantum or that the priority of another job k is higher than j's priority. When job j is rescheduled again at t_5, the cache is now loaded with the working set of job k. Parts of j's working set are no longer available in the cache and must be reloaded from

main memory, resulting in a larger total execution time. At t_6 job j finishes its execution after accessing its working set for three times.

According to [BBA10a], it is possible to determine the cache-related preemption and migration delay in two different ways.

Let us assume that a job has neither a possibility to trigger a preemption nor a migration on demand, but it can detect whether it experienced one of these events. In recent systems this is normally not the case because a preemption or migration is done by the system transparently. A job j can run as described before from t_1 to t_3 (Fig. 3.1). After accessing its working set for two times, the job checks whether it experienced a preemption or migration. If so, it has to start again from the beginning by measuring t_1.

Otherwise, one can calculate the time necessary to access the working set by subtracting t_2 from t_3. This period of time presents the *cache warm* access time to the working set. After t_3 the job waits to be preempted or migrated in order to continue. While waiting, the job randomly accesses parts of its working set to achieve cache affinity. After detecting a preemption or migration, the job record t_5 and accesses its whole working set again. Thereafter, the measured point in time t_6 allows to calculate the time necessary to access the working set after a preemption or migration by subtracting t_5 from t_6. Again, the job must be aware of a preemption and migration between t_5 and t_6. If it experiences such an event, it will have to start from the beginning by measuring t_1 again.

The delay d caused by the preemption or migration can now be determined by calculating: $d = (t_6 - t_5) - (t_3 - t_2) = e_2 - e_1$.

This method is called *schedule-sensitive* [BBA10a]. It has one major drawback: A long time is needed to get a large number of valid samples because a preemption or migration is neither allowed between t_1 and t_3 nor between t_5 and t_6. The occurrence of such an event cannot be influenced by the job and depends on the utilization of the system. Furthermore, the time necessary to wait for a preemption or migration after t_3 might be quite long in case of a system with a low utilization.

The second approach to measure CPMD is called *synthetic method* [BBA10a]. It requires the possibility to let an application decide by itself when it will be preempted or migrated. If a preemption or migration is only accomplished on explicit requests, there will be no need to check for it during runtime. According to Fig. 3.1, the job starts its execution at t_1 and measures t_2 and t_3 as described before. Thereafter, it asks some service for a preemption or migration. When the job starts again either after a preemption or on another core after a migration, t_5 and t_6 are determined. The delay d can be calculated as shown before.

The explicit request of a preemption or migration used by the synthetic method allows to get one valid sample on each round. However, this methodology has two drawbacks: It is not possible to examine dependencies between scheduling decisions and CPMD, and it is not possible to measure the influence of the total number of tasks in the system on CPMD[BBA10a, Chap. 3.2].

NOVA has no built-in support to inform a thread about a preemption during execution, but the schedule-sensitive method requires such a mechanism to work. Therefore, changes in the kernel would be indispensable. Furthermore, we are interested in as many

valid samples of CPMD as possible, so the schedule-sensitive method might not be a good choice.

The synthetic method allows to collect a huge number of valid CPMD samples within a short period of time. In addition, the preemption length can be freely chosen by the application. This property might be very useful in order to study the influences on CPMD.

After weighing the advantages and disadvantages, one has to conclude that the synthetic method is the better choice for the purpose of this work. Nevertheless, the question arises whether the synthetic method can be implemented in NOVA efficiently and with less effort.

NOVA is designed as a micro-kernel and implements only a few primitives in the kernel as described in Sec. 2.3. The design does not provide a mechanism to enable migrations out of the box. Anyway, it is possible to implement thread migrations in the user land by using ECs and SCs in combination with event handlers and semaphores. In this way, it is possible for a thread to trigger a migration of itself to a specified core. More information about the implementation of the migration functionality is provided in Sec. 4.1.

An application is able to trigger a preemption for a specified time by using a timer functionality provided by a service of the NOVA userland (NUL). Thereby it is possible to block the execution for a specified time and to resume it thereafter. During blocking other threads are scheduled.

Let us recall Fig. 3.1 in combination with the synthetic method and NOVA: An application starts its execution at t_1. It accesses its working set for the first time to bring it into the cache and measures the time needed to access its working set again. Some properties of the application's SC have to enforce that there is neither a preemption nor migration up to t_3. Otherwise, conclusions about e_1 might be wrong. Since a migration does only occur on explicit request by the job, it will not happen unexpected.

In contrast, a preemption is handled by the scheduler of NOVA and must be prevented: On the one hand, the time quantum of the SC could be chosen sufficiently long to prepare the SC with enough computation time. On the other hand, the priority of the scheduling context can be specified cleverly as follows: If all other SCs in the system that are combined with user defined ECs have a lower priority, they will not be scheduled because the time quantum of the measurement thread will be replenished directly after it runs out of time. The second approach is more convenient because determining the length of the time quantum is hard.

If the measurement thread has determined the time needed to access its cache warm working set after t_3, it will preempt itself for a specified period of time l. The thread blocks thereby on a semaphore and the scheduler chooses any other ready thread to run. If there is no such thread available, the system will keep idle. Otherwise, a special-purpose thread is scheduled that has by construction a lower priority than the measurement thread.

This design allows the usage of lower prioritized threads for creating arbitrary simulated background loads to detect the costs of CPMD. Threads running as background load access different working set sizes and load their data into the cache systematically.

From the perspective of the measurement thread these threads pollute the cache lines. Therefore, they will be called *cache polluter threads* or simply *cache polluters*.

After l time units the measurement thread is ready to run again. However, l is used as a parameter to control the time the cache polluter threads are able to work. The influence of a preemption or migration can now be determined as described before.

In single-core systems e_1 is determined without the influence of other threads running in parallel, because at any point in time only one thread can be executed in the system. Hence, multiple measurements of e_1 result in similar values.

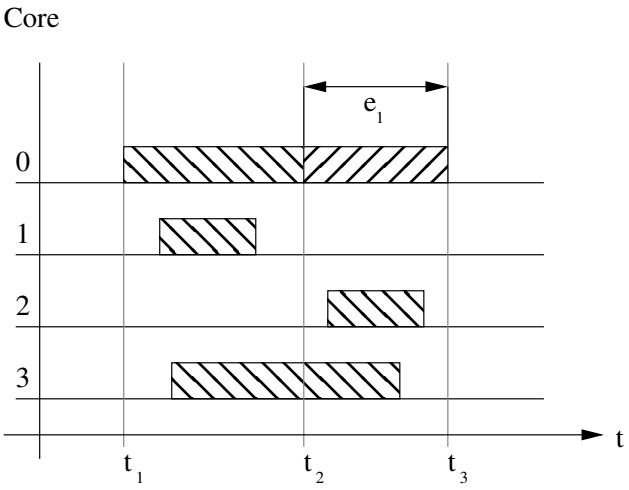


Figure 3.2: Measurement problems of e_1 in multi-core systems.

In multi-core systems this cannot be guaranteed because other threads are running on other cores in parallel. Fig. 3.2 shows an example where the cache is shared between all cores. The measurement thread on core 0 accesses its working set between t_1 and t_2 to bring it into the cache. Another thread on core 1 does its calculation in parallel and, thus, loading its own working set also into the cache. It cannot be guaranteed that parts of the working set of the measurement thread does not get lost because of cache interference. The same holds for the thread on core 3.

The measurement of e_1 might also be influenced by the thread on core 2: Data that will be accessed by the measurement thread could be evicted from the cache through loads by the other thread beforehand.

Therefore, it is necessary to remeasure e_1 whenever the background load is changed. Otherwise, the costs of CPMD would not be well defined due to e_1's missing relationship to the background load.

3.2 Data structures and access patterns of the measurement thread

In order to find the dependency of different factors on CPMD, the working set accessed by the measurement thread has to be defined. The goal of this section is to design

a proper data structure that can be used to construct this working set. It must be sensible to CPMD on the one side, and its properties must be easily adaptable to study the influences of CPMD on the other side.

In fact, desired properties of the data structure are:

1. Simple design
2. Dynamic in size
3. Low utilization of the CPU besides memory accesses
4. Enforcement of sequential accesses
5. Sensitive to CPMD
6. Possibility to modify data

A linked list is a common data structure that fulfills all of these properties: It is simple in design and easily extendable in size by increasing the number of list elements dynamically. This property is needed to study the influence of CPMD on the working set size. Furthermore, the elements of the list are sequentially accessible without complex calculations, so the CPU only waits most of the time to receive data from the caches or main memory. This property enables to measure additional characteristics like the cache and main memory access times.

Enforcing a sequential access by design prevents the compiler from optimizing the code in an undesired way. Otherwise, it would be hard to study the influencing factors of CPMD. If a list element contains a pointer to the next list element, as it is the case in a linked list, it will not be possible to get the address of the next list element without accessing the previous one beforehand.

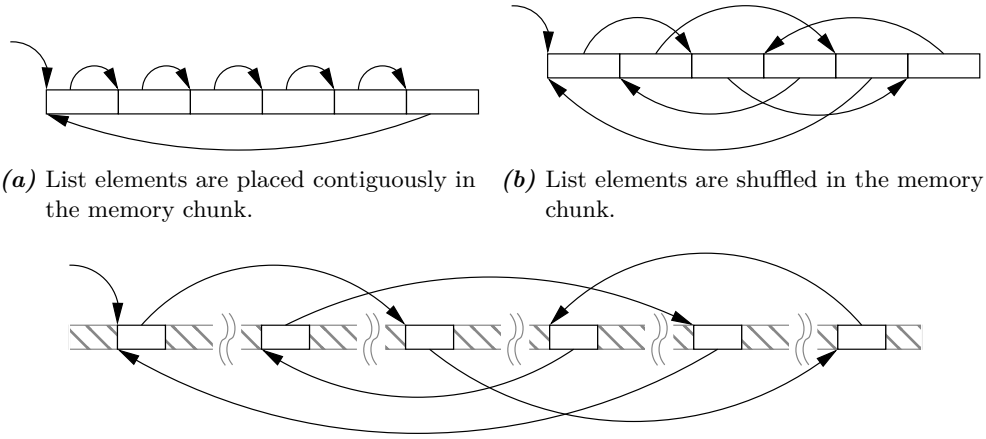
Additionally, the list structure can be made as much sensitive to CPMD as possible by a proper design of the list elements. Such a design allows afterwards conclusions about the worst case costs.

The possibility to modify data is important to study the costs of CPMD in case of modified cache lines. This property is achievable by extending the list elements by a writable data element.

Fig. 3.3(a) shows a possible design of the structure. However, it has one major drawback: The hardware prefetcher of a system can load data from main memory before the data is really needed because the distance of two consecutive reads is always constant, except if the last pointer to the first element is reached. Therefore, this structure is not sufficient to measure realistic access times to the different cache levels and main memory.

Shuffling the list elements as shown in Fig. 3.3(b) prevents the hardware prefetcher to work because the distance of two consecutive reads is no longer constant. Therefore, realistic access times to the caches and main memory are measurable.

An additional improvement is to align the list elements to cache line boundaries (Fig. 3.3(c)). Recall that data between the caches, and between the last level cache and



(a) List elements are placed contiguously in the memory chunk.

(b) List elements are shuffled in the memory chunk.

(c) List elements are shuffled and aligned within a cache line.

Figure 3.3: Linked list as a data access structure.

main memory is changed cache line wise. Thus, placing only one list element within a cache line leads to a maximum sensitivity to CPMD.

The constructed list structure of Fig. 3.3(c) has got all needed properties and is used in the experiments to simulate the working set of the measurement thread.

The used phrase "access the working set" means in case of a linked list that starting at an arbitrary list element all other list elements are accessed step by step until the initial one is reached again. Depending on the number of list elements, at least parts of working set are loaded into the cache.

3.3 Background load

The goal of this section is to describe the necessity of a well defined background load to study the influences of CPMD in loaded systems. Furthermore, two different approaches to create background load are introduced.

The term background load is defined as the set of all user threads in the system excluding the measurement thread. In this sense, for example the former defined cache polluter threads are elements of the background load.

The background load is necessary to achieve effects of cache interference between the measurement thread and threads of the background load. If there is none of the latter one, the cache will not be polluted when the measurement thread is preempted. Accordingly, the working set of the measurement thread resides in the cache and CPD will be around zero[1].

[1] In practice the costs of CPD might be also available in an idle system because the scheduler of the system is invoked and it might also change some contents of the cache during its execution. Furthermore, interrupts handled by the operating system can occur, possibly evicting useful cache lines.

A lack of background load will not remove the costs of CMD if modified cache lines are in one of the private caches of the initial core: The cache lines have to be transferred to the target core and, thus, the costs of a migration are not zero.

If a thread does not modify the memory, CMD can be zero because the working set of the thread might be already contained in the private cache of the target core due to a previous execution, but this is unlikely.

A typical system comprises of more than one running thread. Hence, from the perspective of a single thread there is always background load existent and, thus, CPMD is always present. Therefore, in order to use the measurement thread to detect CPMD a background load has to be modeled.

Two approaches to create usable background load are introduced in the next subsections.

3.3.1 Constructed background load

User defined threads only responsible for polluting the cache are called *constructed background load*. These threads have several benefits:

- no constraints in reference to the used working set
- reproducible results

If there are no constraints concerning the working set and its access, arbitrary background loads can be created and, thus, the pollution of the caches can be controlled at a fine-grained level.

Another advantage is the reproducibility of measurements: Experiments can be repeated under the same settings arbitrarily and, thus, the influence of different system properties can be studied.

The major drawback of a constructed background load and its associated access to the working set is that the background load is not realistic: A regular application executes instructions to solve a specific problem. Therefore, the access to the working set varies highly with the type of an application and it might be the case, that not all factors influencing CPMD are detectable with the constructed background load.

To reduce the implementation effort, the measurement thread's data structure of a shuffled linked list (Fig. 3.3(c)) is reused for the constructed background load. Hence, the size of the working set and the rate used to modify cache lines can be easily controlled. The cache polluters can be started and stopped on demand on arbitrary cores in the system.

3.3.2 Realistic background load

To study the influence of a more realistic background load the functionality of NOVA and the services provided by NUL are used to setup virtual machines (VMs). Each VM is started before the measurement and runs permanently on exactly one core. The VMs compile the Linux kernel in an endless loop. The compilation of the kernel needs a great amount of computing power and has a permanently changing working set. Because the

needed source files of the Linux kernel came from a disk, interrupts occur frequently making the background load even more realistic. The number of VMs running in parallel can be changed during startup to observe the influences on CPMD.

3.4 Experimental design

This section provides an overview about the experiments that were carried out to detect the costs of CPMD.

3.4.1 Memory access times

The goal of this experiment is to determine the access times to the different cache hierarchies and main memory in an otherwise idle system. The experiments also show the penalties of accessing remote memory through QPI in the Dell system.

The measurement uses the list structure defined in Sec. 3.2. The experiment is run under three different circumstances depending on the state of the cache lines *before* the measurement is started:

1. valid cache lines
2. invalid cache lines
3. modified cache lines

All three experiments are carried out using different working set sizes.

The first experiment enables to determine the access time to the different cache levels. The working set – or at least parts of it – are loaded into the cache before the measurement is started. Thereafter, the access times to the cache or main memory are measurable by accessing the working set. If its size grows above the size of a cache level, the cache cannot hold all the needed data and, thus, the access time will increase because the next cache level or main memory has to handle the request. This experiment allows to determine the *cache warm* access times that are near the best case possible.

The *cache cold* access times are studied with the other remaining experiments: If the cache contains only invalid cache lines (second experiment) beforehand, each list element has to be fetched from main memory to get the address of the next list element. Thus, this setup measures main memory access times. The worst case access times of an otherwise idle system will be determined, if the cache contains only modified cache lines (third experiment) before the working set is accessed: In order to load the working set, modified cache lines must be evicted.

3.4.2 Accuracy of preemption length

As described in Sec. 3.1 the length of a preemption is controlled by a timer service implemented in NUL. The accuracy of the length of a preemption has to be verified, otherwise no statements about CPMD in relation to the preemption length are possible.

In an otherwise idle system a thread measures the point in time t_1 before it is preempted for a period of time l. Immediately after the preemption the point in time t_2 is measured. The period $t_2 - t_1$ and l should be approximately equal.

The results of the experiment in an idle system will be the same in a loaded system if all user threads are running with a lower priority than the SC of the EC that is responsible for handling the timer interrupt and the priority of the measurement thread. That is the case in the used design.

3.4.3 Preemption

The costs of a preemption are determined with this experiment. The measurement thread is permanently bound with exactly one cache polluter to a single core. The following parameters are changed to study the costs of a preemption:

- working set size of measurement thread
- preemption length d

Changing the working set of the measurement thread allows to determine its sensitivity to CPD. Additionally, different preemption lengths influence the time the cache polluter thread is able to run, so the relation of the preemption length and CPD can be studied.

Furthermore, the experiment is repeated using the realistic background load instead of the constructed one. Additionally, CPD is studied in a heavily loaded system where each core runs a VM.

3.4.4 Migration

Different kinds of migrations are explored within this experiment on the Dell and Phenom system to gain knowledge about CMD: between cores sharing a cache level, and between cores with no caches in common[2]. Thereby, the measurement thread is periodically migrated between the involved cores determining the necessary time to access its working set after each migration.

The experiment is repeated using different system states: idle, loaded with constructed background load, loaded with realistic background load.

The working set size of the measurement thread is varied to study its influence on CMD.

[2] The Phenom system does only allow migrations between cores that share the L3 cache due to the limitation of one physical processor.

4 Implementation

This chapter describes important implementation details. The first section gives an overview about the implementation of a thread migration. The second section covers the collection of statistical data, followed by details about the mechanisms that were used to control the polluter threads.

The last section is about the implementation of an additional system call in the kernel that is needed by one of the experiments.

4.1 Migration

NOVA contains no possibility to create a thread like object that can be migrated directly between different cores of a system. Therefore, such a mechanism has to be implemented.

To explain the details, this section is divided into two additional subsections: The first one discusses the necessary setup, the second one shows what happens during runtime when migrating a thread.

4.1.1 Setup phase

To achieve the ability to migrate a thread between different cores of the system, an EC for the thread has to be created on every core that should be able to execute it afterwards. To limit the overhead of a migration, it is necessary to share as much resources as possible per EC (e.g. stack).

Additionally, two portals for exception handling are created per EC. The *startup exception handler* is called the first time when an EC starts its execution. It sets up the instruction pointer properly to ensure that the EC begins at the correct instruction after the handler returns. The other handler is called *recall exception handler*. It is invoked when the *recall* system-call is used.

Fig. 4.1 shows an exemplary setup of a thread that should be migratable between core 0 and core 1 afterwards. The main thread creates an EC with the associated handlers on the first core. Additionally, a SC is created and bound to the EC by the main thread that blocks on a semaphore, waiting for the setup of the EC to complete.

The newly created SC is scheduled eventually and produces a startup exception because it does not know which instruction to execute yet. Therefore, the startup exception handler is called and sets up the instruction pointer properly. After it returns, the EC starts its execution and uses the recall system-call to trap into the kernel, ending up with the invocation of the recall exception handler. Its implementation ensures that the main thread is unblocked again, so it can proceed to setup ECs on other cores as well. The handler itself blocks on a semaphore used to control its execution because

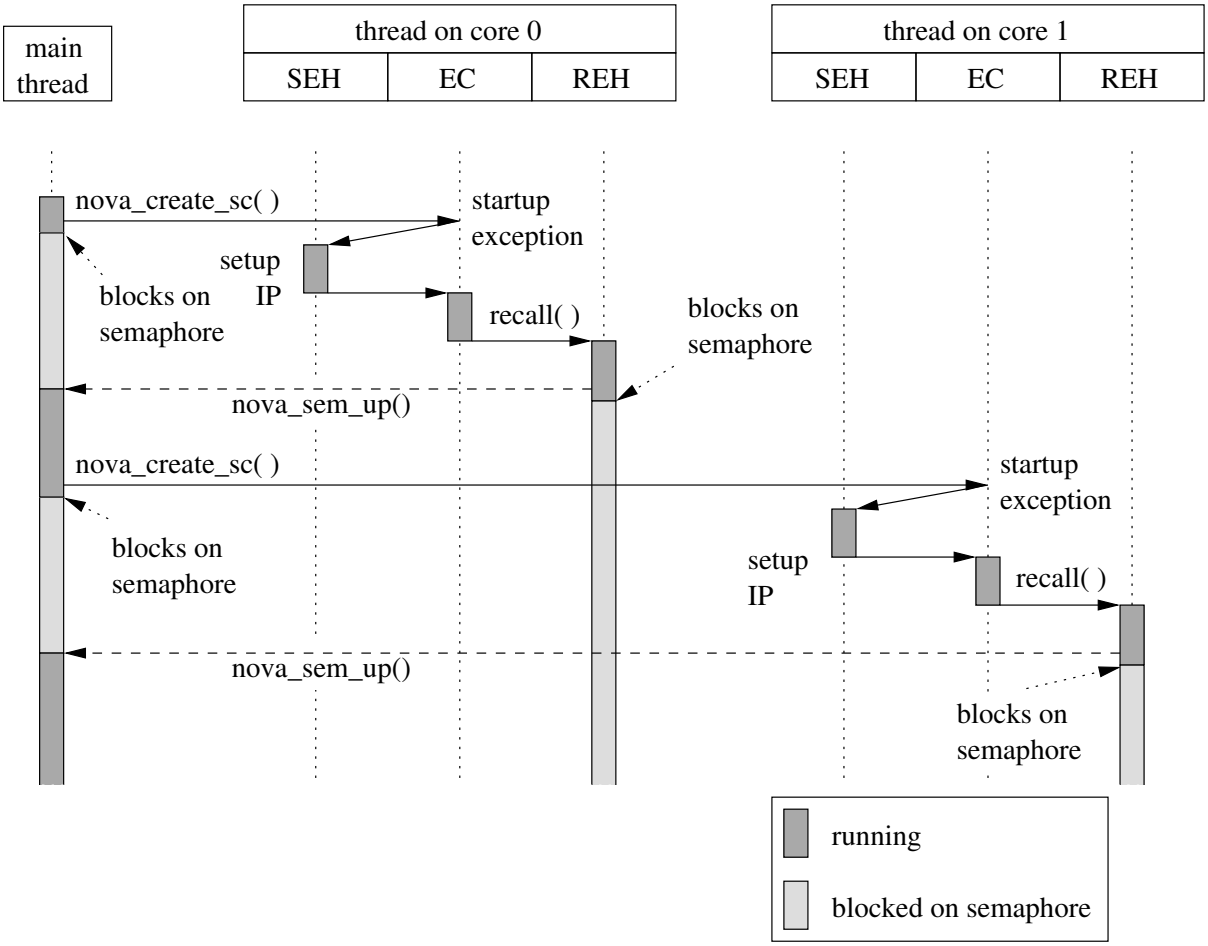


Figure 4.1: Setup of a migratable thread. On each core there is the startup exception handler (SEH), the execution context (EC), and the recall exception handler (REH).

there is no other work to do yet. The corresponding EC will not be rescheduled before its execution handler returns.

After the last handler wakes up the main thread again, the following situation occurs: The main thread is ready to run and every created EC is blocked on a dedicated semaphore in its associated recall handler.

It would be possible to avoid the sequential setup if every EC uses its own stack at least during the setup phase. This stack can be small in size just to handle the execution of the recall system-call. Nevertheless, the parallel setup is not used because the overall speedup is insignificant due to its one-time initialization at the beginning.

4.1.2 Migrations during execution time

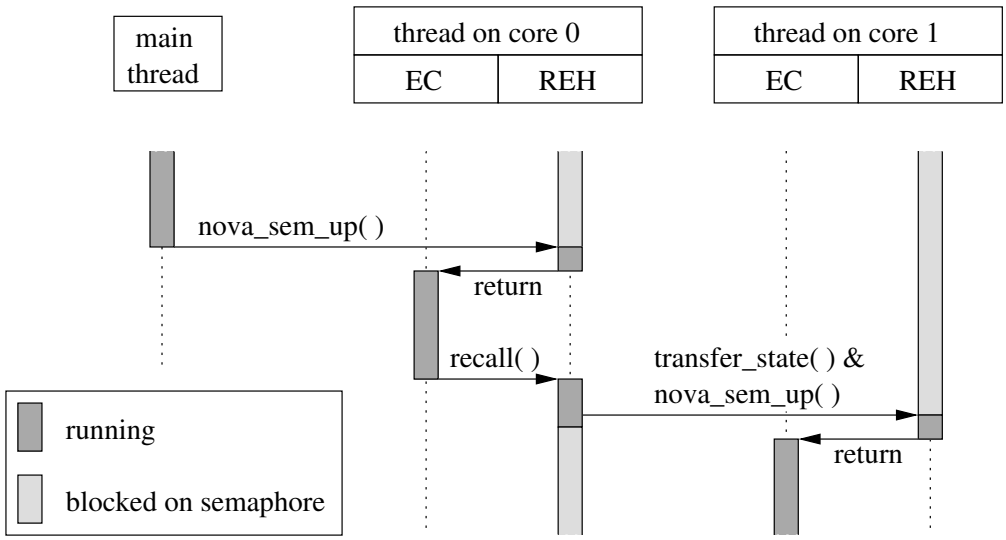


Figure 4.2: Example of a migration from core 0 to core 1. The recall exception handler implements the actual migration. It copies the necessary registers between the involved UTCBs.

Fig. 4.2 shows an exemplary execution of the formerly set up thread. The main thread starts one of the ECs by calling the up-operation on the semaphore that blocks the recall exception handler on the target core. Thus, it continues with its execution and jumps into the kernel after its return. Depending on the definition of the handler, parts of its UTCB are copied into the CPU registers before the EC proceeds running.

Let us recall the last executed function of all ECs: They executed the recall system-call to trigger an exception and entered the recall exception handler. Only one EC continues with the next instructions now; the other ECs stay blocked.

At some point in time the running thread triggers a migration. This is done by writing the id of the target core into the *EBX*-register and by calling the recall function, which enters the recall handler. It determines the target core of the migration by reading its UTCB that contains the value of EBX. Now the recall exception handler transfers the content of its own UTCB, which refers to the state of the interrupted EC, into the

UTCB of the exception handler on the target core. Thereafter, the EC is unblocked. The handler of the origin EC blocks on its own semaphore, waiting for a future release.

The next instruction of the recall exception handler of the new core is a return; the values in its UTCB are already updated and, thus, the values of the UTCB are copied into the CPU registers by the kernel before the EC resumes on the new core afterwards.

This implementation allows to migrate threads between prepared cores. The thread can trigger migrations on its own purpose, but in theory it might also be possible to start a migration from the outside. Caution is recommended in the latter case, because objects can be combined to CPU local resources like portals. If such resources are in use during a migration, critical errors stopping the execution will be likely.

4.2 Capturing of statistical data

The measured data has to be processed. On the one hand, the data can be saved somewhere locally. On the other hand, the data can be transmitted to another computer for additional analysis. However, one major goal of the setup is to avoid as much sources of unpredictability as possible. Therefore, the usage of both hard disks to save the measured results and network drivers to send the data over the network using ethernet during the measurements is not acceptable. Otherwise, services of NUL have to handle frequently occurring interrupts during the measurements. The handling of this interrupts would increase the variance, but this is undesired.

Because of the absence of other persistent storage spaces, the data is transmitted to another computer for additional analysis by the serial port. It is already supported by NOVA and does not add large overhead. The limited bandwidth require the online calculation of meaningful numbers directly after the experiment's execution to reduce the amount of data.

Because NUL does not support the printing of floating point numbers, CPU cycles are used and stored as 64 bit numbers when necessary.

Two approaches are available to calculate the results online: Values can be collected and saved in memory during the measurements or the results can be calculated on the fly during two consecutive measurements. Unless all values can be stored in registers, the latter approach has no additional advantages, because both procedures store otherwise values in the memory manipulating the cache. Because the number of registers capable of holding 64 bit values is limited, values are collected and saved in the memory for later evaluation.

4.3 Setup and control of cache polluters

When the cache is polluted by a constructed background load, some mechanism to control the execution of the cache polluter threads is needed.

At the beginning one EC is created and bound to a SC on every core. Additionally, one semaphore per core is added to control the execution of the cache polluter. When the threads start to run, they block on this semaphore until the corresponding up-operation is called. Thereafter, they begin to pollute the cache by accessing their working set.

A possibility has to be offered to stop specific threads without preventing them from a later restart at a defined point in time. Again, the recall exception is used to interrupt a polluter thread to update its instruction pointer. After the handler returns, the EC starts at a well defined function and blocks again on the former mentioned semaphore until the up-operation is called.

Nevertheless, it is not enough to set up the instruction pointer in the recall handler. It is also necessary to prepare the stack and to reset the stack pointer. Otherwise, the called function would not find the right parameters or the stack would overflow eventually.

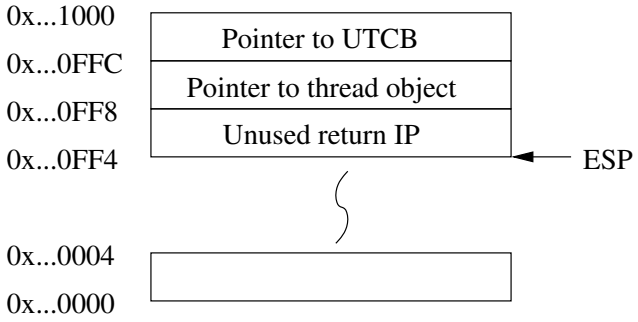


Figure 4.3: Layout of the stack.

Fig. 4.3 shows the layout of the stack. Whenever the recall handler is called, it resets the stack pointer to the shown position and sets the instruction pointer to a wrapper function:

```
void function_wrapper(CachePolluter *ptr, Utcb *utcb);
```

Various services of NUL expects the pointer to the UTCB to be at the beginning of the stack. The other parameter is used as shown for a pointer to some object whose methods can be called from the wrapper function afterwards. The pointer named *unused return IP* is necessary, because the GCC compiler expects the return address of the function there. If it is omitted, the mapping of the function parameters will be wrong.

4.4 Additionally implemented system call

One additional system call to enable the measurement of the worst case memory access time is added. It is necessary in the first experiments as described in Sec. 3.4.1. The implementation has to be in the kernel because a privileged instruction is executed that cannot be used in the user land.

The system call executes the *WBINVL* instruction [int11c, page 1347]. Therefore, all modified cache lines of every CPU cache are written back to main memory before they become invalid. Hence, the cache affinity of all jobs in the system is completely lost after this instruction.

5 Evaluation

This chapter presents the results of the experiments that were described in Sec. 3.4. Precise statements demand on additional analysis. Therefore, the first section of this chapter classifies the used background loads. Furthermore, the accuracy of the preemption length is estimated beforehand.

5.1 Classification of background load

This section is divided into three parts: First of all, the realistic background load is analyzed, before the constructed background load is examined. Thereafter, both are compared to each other.

5.1.1 Realistic background load

To evaluate the background load produced by GNU's tool chain when compiling the Linux kernel, the performance counters of recent CPUs are used. Thereby, one can conclude rough statements about the characteristics of the background load.

#CPUs	1	2	3	4	5	6
#loads in thousand/s	6,105	12,832	18,896	25,532	31,892	38,045
#load-misses in thousand/s	471	1,116	2,031	3,009	4,097	5,324
load hit rate in percent	92.29	91.3	89.25	88.2	87.3	86.01
#stores in thousand/s	5,997	12,309	1,8510	24,526	30,543	36,169
#store-misses in thousand/s	829	1,762	2,823	3,761	4,864	5,813
total #misses in thousand/s	1,300	2,878	4,854	6,770	8,961	11,137
refill time (L3) in ms	151.2	68.3	40.5	29.0	21.9	17.7
main memory throughput in MB/s	79.3	175.7	296.3	413.21	546.9	679.7
main memory throughput per core in MB/s	79.3	87.85	98.8	103.3	109.4	113.4

Table 5.1: Measurements on Xeon X5650 to classify the realistic background load.

Table 5.1 shows the number of L3 loads, load-misses, stores, and store-misses per second, depending on the number of cores that were used for compilation. The numbers were extracted on the Dell system using Linux and *perf*, a tool to read performance counters of the CPU.

One single kernel build was started on n cores. Thereby, *taskset* was used to pin the compilation to static cores. The number of processes spawned by *make* was limited to $n + 1$. Each core was traced for 120 s.

The number of loads, load-misses, stores, and store-misses increases with the number of used cores as shown in the table. The total number of loads from the L3 cache becomes larger when more cores compile the kernel due to a higher parallelism. In addition, the number of cache misses also grows because all cores share one L3 cache. Therefore, the effective amount of storage per core in the cache decreases and, hence, the overall cache hit rate decreases, too.

The total number of misses per second is equal to the number of main memory accesses per second[1]. Every access stores a new cache line, possibly evicting a valid or modified one. Because the total number of cache lines is known, it is possible to estimate the time needed to "reload" all cache lines of the L3 cache. Thus, for example in case of the execution on a single core, after 151.2 ms the number of cache lines loaded from main memory into the cache is larger than the total number of cache lines that can be saved. Therefore, most cache lines are replaced after this amount of time.

Preemption lengths larger than L3's refill time result in CPD near the worst case. Anyway, caution is recommended when using this numbers: A virtual machine (VM) in NOVA only has access to one CPU of the system. Several VMs compiling the Linux kernel in parallel all have their own working set including the necessary tools to compile the kernel. Due to so absence of any shared memory between the VMs, the total working set is larger and, therefore, the load hit rate becomes lower.

5.1.2 Constructed background load

The working set size of one thread of the constructed background load was set to 4 MB.

#CPUs	1	2	3	4	5	6
total #misses in thousand/s	0.69	3,267	13,826	23,356	41,338	49,608
load hit rate in percent	99.999	96.4	84.1	71.1	7.5	3.2
refill time (L3) in ms	—	60.2	14.2	8.1	4.8	4.0
main memory throughput in MB/s	0,04	199	844	1.487	2.523	3.028
main memory throughput per core in MB/s	0.04	99.7	281.3	371.6	504.6	504.6

Table 5.2: Measurements on Xeon X5650 to classify the constructed background load.

Table 5.2 shows the characteristics of the background load. The values were measured on the Dell system using Linux and perf to determine the number of cache misses. Thereby, the number of used cores was varied from 1 to 6. Each core executed exactly one cache polluter.

The cache load hit rate becomes worse for more than 4 cache polluters because their combined working set does not fit into the L3 cache. Furthermore, the needed memory bandwidth grows with the number of cores.

[1] This statements assumes, that a modified cache line is written back to main memory in parallel to the load using some kind of store buffer. Otherwise, the number of memory accesses would be higher than the sum of all cache misses.

The memory bandwidth used by one core will be constant if the cache cannot save the combined working set of all running threads (504.6 MB/s in case of 5 or 6 cores). In this case, most of the data requests must be handled by main memory. The bandwidth to the memory is not exhausted because the memory throughput per core would decrease otherwise.

The time needed to refill the L3 cache can be calculated from the memory bandwidth. Because the working set size fits into the L3 cache when using one core, it is impossible to reload the whole cache. Surprisingly, this might be possible with 2 or 3 polluter threads, even if their combined working set would fit into the cache. If more than one core is utilized, Intel's Smart Cache Technology [sma] will partition the cache into parts that are shared among the cores. Therefore, the usable part of the L3 cache per core might not be large enough to hold its working set.

5.1.3 Comparison of the used background loads

Utilizing all cores of the system, both kinds of background load can be used to evict all cache lines from the L3 cache. In comparison to the realistic background load, the constructed one produces up to 4.5 times more cache misses, which results in more accesses to main memory and more evicted cache lines.

Thus, the expected costs of CPMD are higher for the constructed background load: During a preemption more cache lines will be evicted in the same period of time resulting in longer CPD. Furthermore, the probability that a requested cache line will be present in the cache after a migration is lower, so the costs of CMD will increase, too.

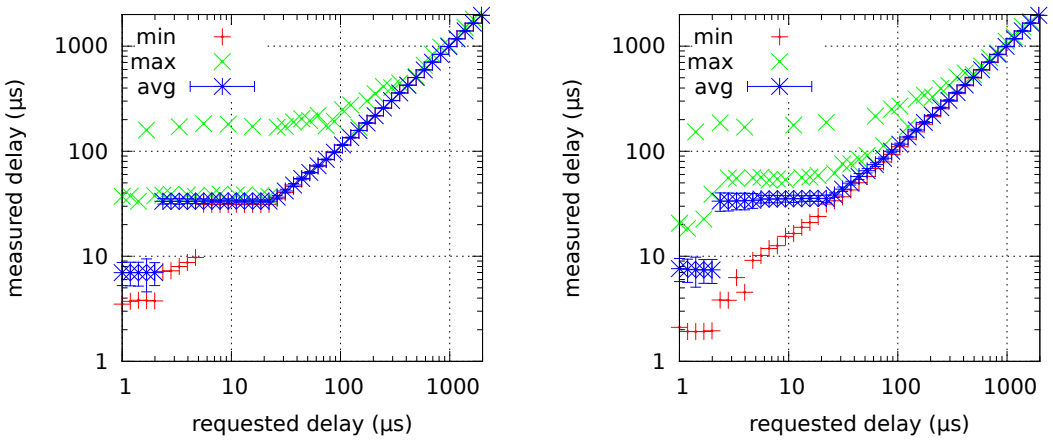
One difference between the two loads is a higher variance of the measured data when using the realistic background load due to its changing workload during compilation. This property can be seen when comparing Fig. 5.4(a) and Fig. 5.6(a). Nevertheless, both kinds of load show approximately the same behavior.

5.2 Accuracy of preemption length

This section evaluates the accuracy of the preemption delay that can be freely chosen by the measurement thread. The evaluation is necessary to reason about the correlation of preemption length and resulting costs.

The graphs in Fig 5.1 show the requested preemption length on the x-axis and the measured one on the y-axis in both an idle and loaded system. The duration of every depicted preemption length was measured 500 times using the time stamp counter (TSC). The blue curve shows the average and its standard deviation. Ideally, the curve should be a straight line with a slope of 1. In practice this behavior is achieved on average for preemption lengths larger than 22 μs.

The red and green points show the minimum and maximum measured delays. Outliers are available even for preemption lengths longer than 22 μs. However, they can be disregarded because the variance of the measured data is negligible especially for delays larger than 1 ms, the minimum used preemption length in all experiments.



(a) idle *(b)* loaded

Figure 5.1: Measured accuracy of the preemption delay.

The measured preemption length does never fall below the requested preemption length in all tests. Additionally, the behavior of the implementation of the preemption shows no significant differences in a loaded system (Fig 5.1(b)).

5.3 Experiments

5.3.1 Cache/memory access times

This section evaluates the results of the first experiment: The time necessary to access the different cache levels and main memory is presented for both measurement systems. Several characteristics of the systems – especially the caches – are included to analyze the results.

Intel Xeon X5650

The red curve in Fig. 5.2 shows the average access times to the different cache levels and main memory measured on a core with direct memory access in an otherwise idle system. Ten million list elements per working set size were accessed to determine the number of needed CPU cycles per data access.

The number of cycles grows depending on where the data originate from. A core loads working set sizes smaller than 32 KB completely from its private L1 data cache. Larger working set sizes up to 256 KB are handled by the L2 cache before the L3 cache must be accessed.

The curve gives the misleading impression that the number of needed CPU cycles per data request changes fluently when the working set exceeds the size of the L3 cache: Some of the accessed list elements are still available in the L3 cache whereas other list elements must be loaded from main memory. Therefore, cache hits and cache misses are

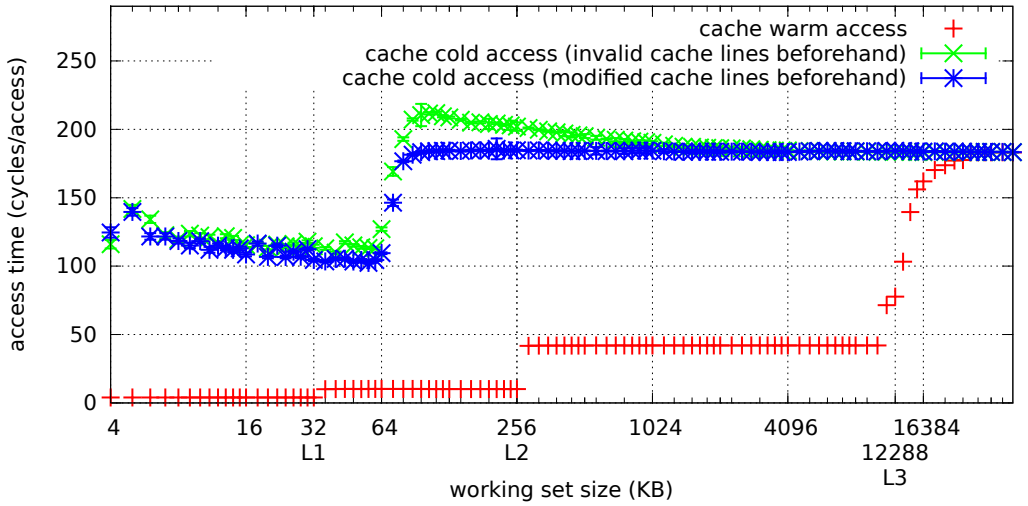


Figure 5.2: Measured access times to the cache and local main memory in cycles per access (Xeon X5650). The green and blue curve include the standard deviation. The working set is only read.

intermingled resulting in an increasing curve. However, every data request of working set sizes larger than approximately 22 MB is a cache miss accessing main memory.

Surprisingly, the average access time already grows when the working set approaches the size of the L3 cache, even though no other tasks pollute the cache. As discussed in Sec. 2.2.1 Intel has to use some kind of hash function to map the physical address to a specific set of cache lines because the size of the L3 cache (12 MB) is not a power of 2. It is likely that this mapping chooses some sets with a higher probability for cache line insertion. Therefore, cache lines being useful later will be evicted even if there is unused space left in other sets.

Access type	CPU cycles
L1 cache	4
L2 cache	10
L3 cache	42
Main memory	183 (312)

Table 5.3: Access times to the different cache levels and main memory (Xeon X5650).

Table 5.3 shows the determined memory access times to the different cache levels and main memory. The number of needed CPU cycles to access the L1 and L2 cache corresponds to the values announced by Intel in [int11a]. Intel specifies a minimum access time of 35 cycles to access the L3 cache when the frequency of the core and uncore is equal. This property holds because Intel Turbo Boost Technology [tur] was disabled during the measurements. However, the minimum measured access time to the L3 cache was 42 cycles.

The number of needed CPU cycles in case of a memory access depends on whether the memory is attached locally or remotely to the processor. In the first case, 183 cycles are necessary. Otherwise, the CPU has to wait 312 cycles. The additional 129 cycles are needed for QPI to request and transmit a cache line.

The green curve in Fig. 5.2 presents the average access time to main memory and its standard deviation. The number of CPU cycles is determined by going through the list structure for 1000 times per working set size. Between two runs the WBINVL instruction enforces that there were no valid cache lines left. Thus, every data request results in an access to main memory.

The blue curve shows the number of cycles to access main memory when a modified cache line has to be evicted to bring in a new one. This curve was determined by measuring 250 values per working set size. The reduced number of rounds was used because the effort to modify the cache between two measurements is time consuming.

Both the green and blue curve show approximately the same behavior, even though one might expect that the access time grows in case of the modified cache lines. Since *Intel Core Microarchitecture* the eviction of a modified cache line is happening in parallel to the load of a new one [int11a, Sec. 2.2.5.1]. Hence, no additional latency is required.

Interestingly, the access times of the green and blue curve are explicitly lower for working set sizes smaller than 64 KB. Because each data request is a cache miss, the shape of the curve must correspond to the properties of main memory. Today's main memory cells are organized in rows and columns [Dre07]. Cells accessible in one row compose a *DRAM page*. Consecutive memory accesses to the same DRAM page are faster due to the lack of row changes. It is likely, that working sets larger than 64 KB incur frequently row conflicts, because there are 8 independent banks within a memory chip and each of them has a DRAM page size of 8 KB. However, details about the mapping of a physical address to DRAM columns, rows, and banks are not available[2].

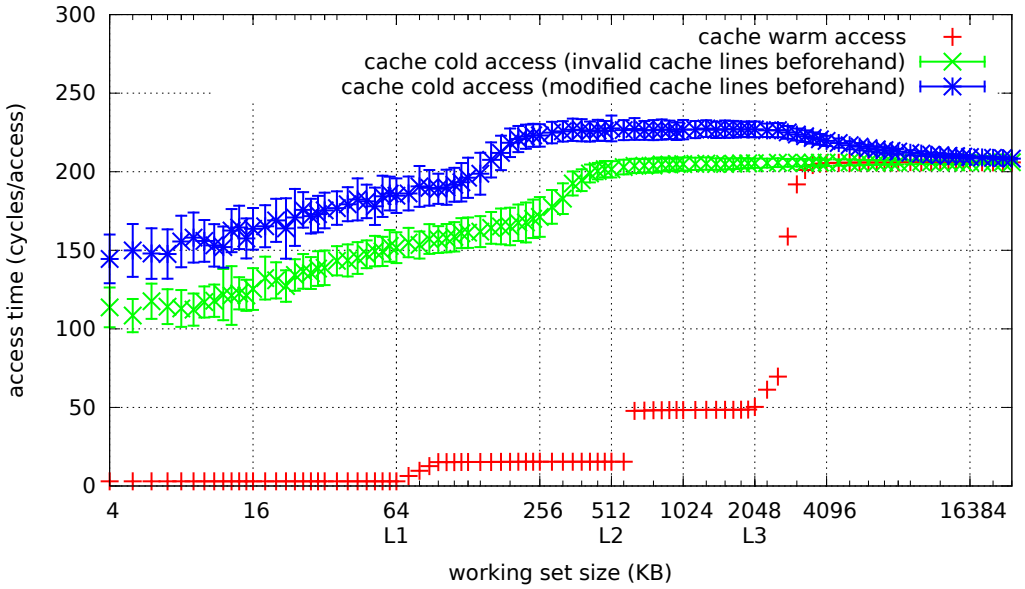
Surprisingly, the access time to main memory for working set sizes between 96 KB and 1 MB will be higher if no cache lines have to be evicted.

Phenom 9550

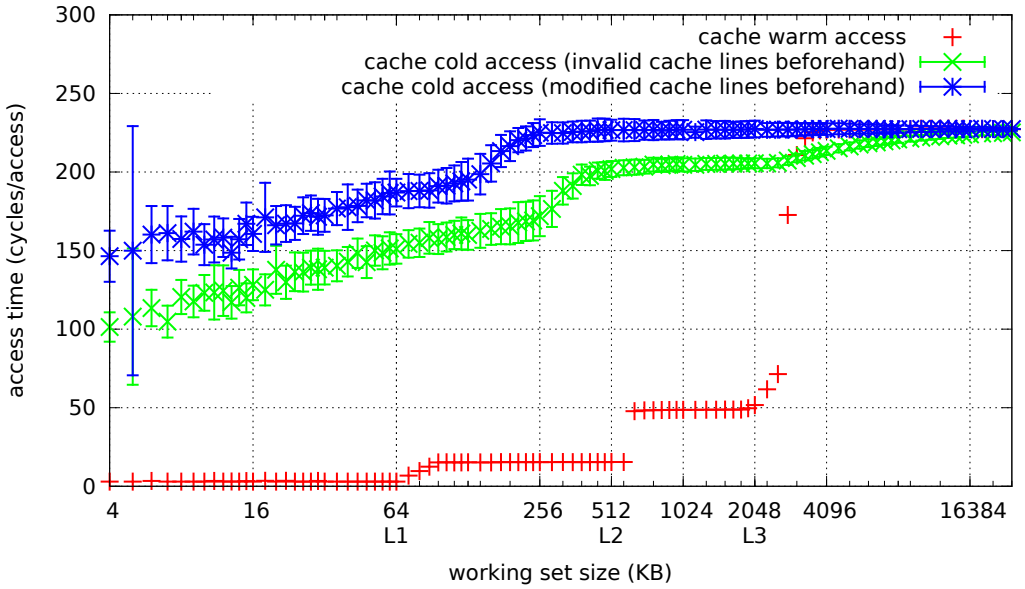
Fig. 5.3(a) presents the same measurements on the Phenom 9550 manufactured by AMD. The execution details of the experiments were the same as described before.

The red curve shows the access times to the different cache levels and main memory. The increasing number of cycles indicates when the next cache level has to be accessed. The number of CPU cycles per access changes fluently when going from the L1 cache to L2 cache because cache hits are mixed up with cache misses: In an n-way set associative cache a newly loaded cache line evicts an old one in a specific set. Assume a working set that comprises of exactly one additional cache line, such that it cannot be hold by the cache due to its capacity. Then cache misses occur only on cache lines in one set: When the core requests data that is placed in the former evicted cache line, its index determines the same set again and, thus, another cache line from this set has to be

[2] The Dell system consists of DD3-SDRAM memory modules with the part number M393B5773CH0-CH9 factored by Samsung (`http://www.compuram.de/documents/datasheet/984633ds_ddr3_2gb_c-die_based_rdimm_rev101.pdf`).



(a) The working set is only read.



(b) The working set is read and also modified.

Figure 5.3: Measured access times to the cache and main memory in cycles per access (Phenom 9550). The green and blue curve include the standard deviation.

evicted and so on. This behavior is easily extendable to two or more effected cache sets. AMD's L1 cache possesses 512 sets and, hence, the curve's *increasing interval* has a width of 32 KB. Therefore, all data requests of working set sizes larger than 96 KB are handled by the L2 cache[3].

An additional vertical line at 512 KB outlines the size of the L2 cache. One can see that the access time does not grow directly after the working set size exceeds the size of the L2 cache. Basically, this behavior depends on the exclusive design of AMD's caches: Working set sizes up to 576 KB, the sum of the size of the L1 and L2 cache, can be handled without any invocation of the L3 cache. Data requests of working set sizes smaller than 2 MB are handled by the L3 cache before main memory has to be accessed.

For working set sizes larger than approximately 4 MB every data request is a cache miss.

Access type	CPU cycles
L1 cache	3
L2 cache	15
L3 cache	49
Main memory	206

Table 5.4: Access times to the different cache levels and main memory (Phenom 9550).

Table 5.4 shows the determined access times to the different cache levels and main memory.

In contrast to Intel, AMD cannot hide the latency that is necessary to evict a modified cache line to save a new one. The green and blue curve show this detail: For every examined working set size the blue curve was above the green one.

Additionally, the variance of the average access times of working set sizes smaller than the last level cache is up to two orders of magnitude larger in comparison to the Dell system.

Both curves show a growing behavior up to a working set size of 512 KB (green curve) respectively 256 KB (blue curve). Assuming only invalid or modified cache lines filled with other content beforehand, no caches can be used for any data request and the behavior of the curves corresponds to the characteristics of main memory.

The blue curve in Fig 5.3(a) converges to the green curve when accessing main memory. Obviously, it is necessary to write back the cache polluter's modified cache lines to main memory when reading the first part of the working set. As the measurement thread does not modify its own working set, no additional cache lines must be evicted in order to read the remaining working set. Hence, the access time decreases until the green curve is reached.

For completeness Fig. 5.3(b) shows the same experiments. In addition, the measurement thread modifies its working set. The behavior is different when working sets larger than the size of the L3 cache are accessed: The green and red curve converge to the blue one because modified cache lines must be written back to the memory all the time.

[3] The increasing interval in case of the Dell system has a width of only 4 KB because there are only 64 sets. It is too small and can therefore not be seen in Fig. 5.2.

5.3.2 Preemption

This section presents the measured delays caused by a preemption. The first part uses the constructed background load to pollute the cache systematically. In the second part of this section, the same results are presented using the described realistic workload.

Constructed background load

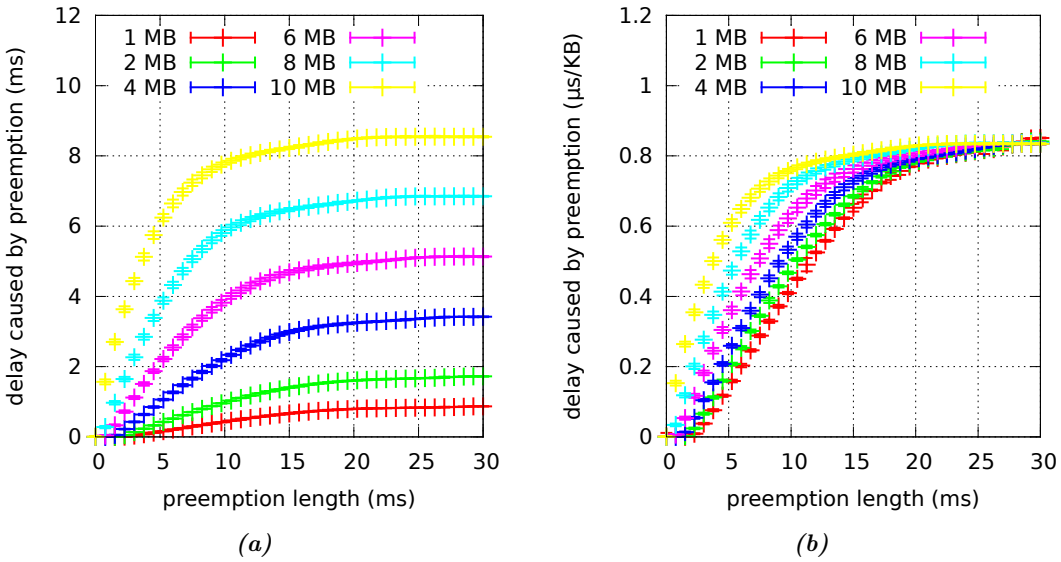


Figure 5.4: Delay caused by a preemption using the constructed background load (Xeon X5650). Each curve presents a specific working set size. Fig 5.4(a) shows the absolute delay caused by a preemption, whereas Fig 5.4(b) shows the experienced delay in $\mu s/KB$ of the working set. The curves of both figures include the standard deviation.

Fig. 5.4 shows the influence of the preemption length on the measurement thread. One cache polluter with a working set size of 24 MB was started on the same core as the measurement thread. The working set size was chosen based on the results of Fig. 5.2 to ensure that most cache lines of the L3 cache were evicted when the polluter thread runs sufficiently long. During a preemption of the measurement thread the cache polluter was scheduled and accessed, depending on the length of the preemption, parts of its working set. The experiment run 200 times per preemption length for each depicted working set size.

The polluter thread's influence on the measurement thread depends obviously on the length of the preemption and on the working set size of the measurement thread (Fig. 5.4(a)). Short preemption length up to 2 ms affect only working set sizes larger than 4 MB because the time is too short to evict a large number of cache lines. Nevertheless, when the working set size is in the dimension of the last level cache, also short running times of the cache polluter evict enough cache lines to see increasing costs of CPD.

Most of the cache polluter's data requests are handled by main memory due to its large working set. The number of cycles to access main memory estimated in Sec. 5.3.1 and the size of the working set allows to conclude that at the latest after 27 ms the working set is completely accessed and the whole cache is polluted. Therefore, the maximum CPD is reached after this time. The worst case costs of CPD can also be calculated by using the number of cycles to access the L3 cache and main memory. The following example shows the calculation for a working set size of 10 MB, that corresponds to the yellow curve in Fig 5.4:

$$\frac{(10 * 1024 * 1024)\,\text{Bytes}}{64\,\text{Bytes/access}} * \frac{(183 - 46)\,\text{cycles/access}}{2,660,000,000\,\text{cycles/s}} \approx 8.44\,\text{ms}$$

The variance of the measured delay is larger when the curves in Fig. 5.4(a) increase: Starting with an empty cache, the polluter thread has to load all of its data from main memory on its first invocation. Thereafter, the measurement thread runs again and measures the time needed to access the working set. After it has preempted itself, the polluter thread runs again. Parts of its working set are still present in the cache. Therefore, it can load more parts of its own working set within the same time evicting other's cache lines. Hence, the conditions before the measurement thread starts again change slightly on every invocation, resulting in an increased variance of the measured data.

Fig 5.4(b) is just another view. Instead of showing the absolute delay that is caused by a preemption, the y-axis shows the costs one KB of the working set incurs depending on the preemption length. When the cache polluter can evict all useful cache lines, all working set sizes incur the same costs per loaded KB.

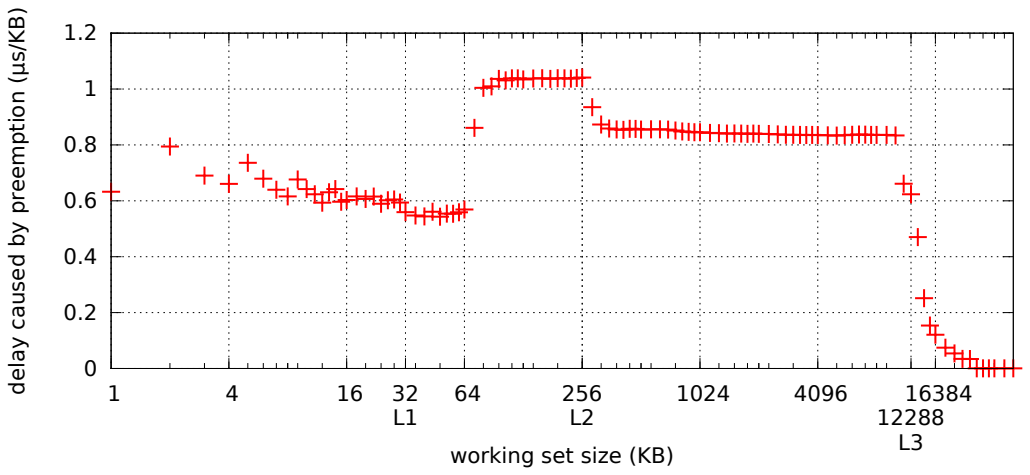


Figure 5.5: Maximum costs of a preemption determined by the cache warm and cache cold access times.

The costs caused by a preemption will converge to zero for working set sizes that are considerably larger than the L3 cache because even if the measurement thread runs continuously without a preemption, it will permanently incur cache capacity misses.

Whenever a preemption occurs that completely pollutes the cache, it will not add additional costs as shown in Fig. 5.5. The data of the graph is calculated by subtracting the cache warm access times from cache cold access times. The results are the additional costs that occur whenever data must be loaded from main memory instead of the caches. The lower costs up to 64 KB depend on the properties of main memory as already described. Furthermore, higher costs occur in the L1 cache, since one can see a slight decrease of the costs when going from the L1 cache into the L2 cache at 32 KB.

Realistic background load

The costs of a preemption are examined when using the former defined realistic background load. In the first part of this section, only one VM runs on a single core compiling the Linux kernel, whereas in the second part all cores in the Dell system are used to produce a heavy background load.

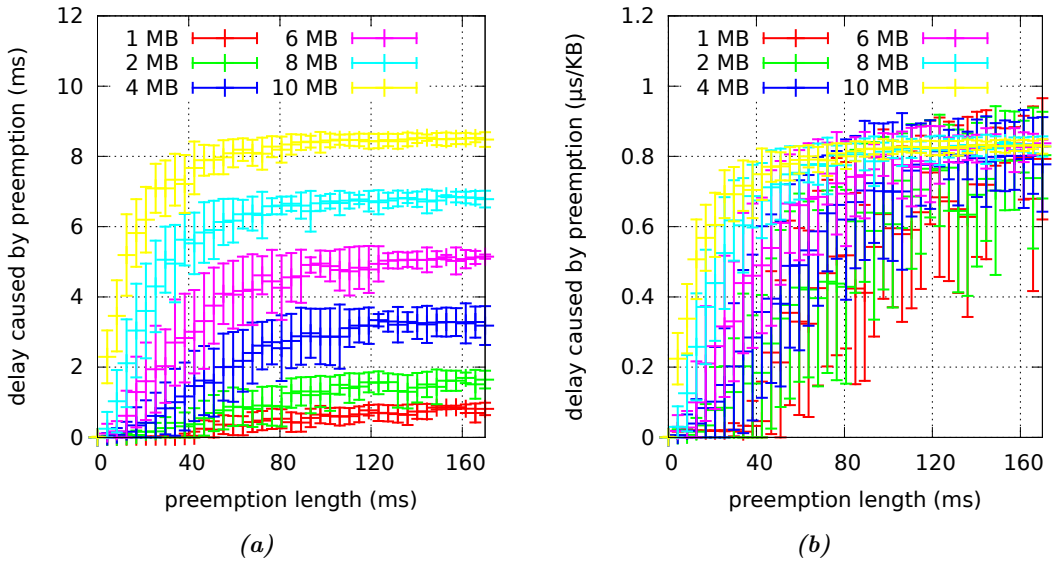


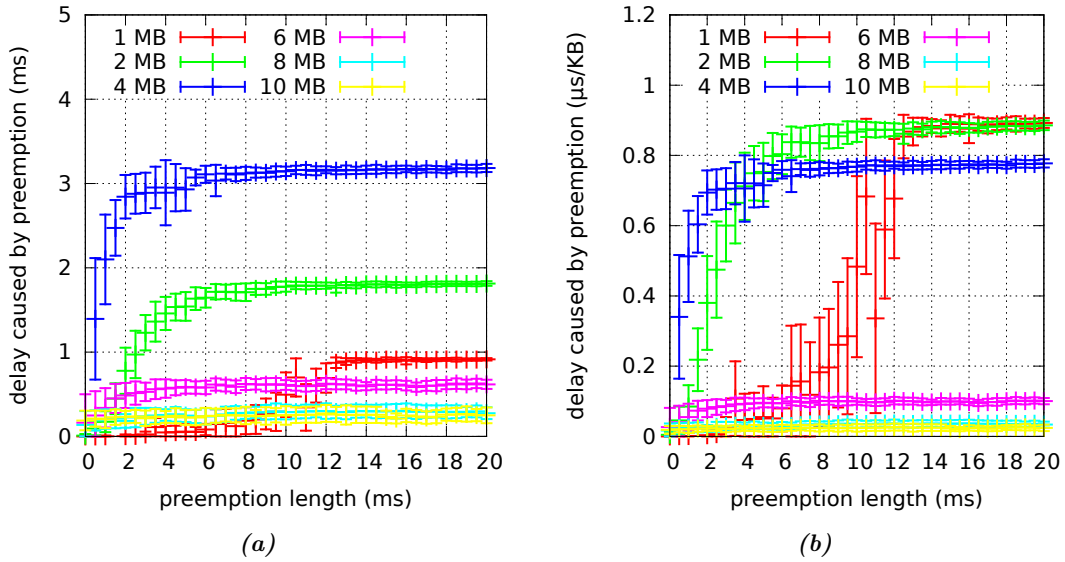
Figure 5.6: Delay caused by a preemption using the realistic background load on one core (Xeon X5650). Each curve presents a specific working set size. Fig 5.6(a) shows the absolute delay caused by a preemption, whereas Fig 5.6(b) shows the experienced delay in $\mu s/KB$ of the working set. The curves of both figures include the standard deviation.

Fig. 5.6 shows the influence of the preemption length on the measurement thread when running the VM only on a single core. During the preemption the VM compiling the Linux kernel was scheduled and, thus, polluted the cache. Each experiment was run 200 times per preemption length for the depicted working set sizes.

In comparison to Fig. 5.4 one can see a larger variance of the measured preemption costs. This fact depends on the realistic background load that has a changing working set. If for example tests are performed to check the availability of tools necessary for compilation, the access patterns will differ from the execution of the compiler itself.

The variance of the measured data of Fig. 5.6(b) is also larger in comparison to Fig. 5.4(b), but both graphs basically show the same characteristics.

After a preemption length of 160 ms, most cache lines are evicted from the cache, resulting in the worst case costs. This number corresponds to the estimated time to pollute the whole cache by one VM, calculated in 5.1.1.



(a) *(b)*

Figure 5.7: Delay caused by a preemption using the realistic background load on all cores (Xeon X5650). Each curve presents a specific working set size. Fig 5.7(a) shows the absolute delay caused by a preemption, whereas Fig 5.7(b) shows the experienced delay in $\mu s/KB$ of the working set. The curves of both figures include the standard deviation.

Another situation is presented in Fig. 5.7: The VMs were started on all cores during setup.

The absolute costs of a preemption (Fig. 5.7(a)) increase for the depicted working set sizes up to 4 MB. Larger working sets experience reduces costs that converge to zero. The situation is comparable to the execution in Fig. 3.2. The measurement thread loads its working set into the shared L3 cache, but the parallel execution of the VMs already starts to evict these cache lines again. When the measurement thread starts to access its working set for a second time to determine the "cache warm" access time e_1, it has to reload most parts of its working set already from main memory. The same happens in case of a preemption, regardless of its length: All cache lines must be reloaded from main memory. Hence, the costs of a preemption converge to zero even if the working set of the measurement thread fits into the L3 cache. With other words one can say that the effective amount of the shared L3 cache usable by the measurement thread is shrunk by the parallel execution of the VMs.

Intel's Smart Cache Technology partitions the cache into blocks of 2 MB that are assigned to the cores in the Nehalem Architecture [PH09, p. 539] in loaded systems.

Therefore, as one can see in Fig. 5.7(b) working set sizes up to 2 MB experience high costs per KB for long preemption lengths. In this case the working set can be handled by the L3 cache.

The red curve in Fig. 5.7(b) shows the costs one KB of a working set size of 1 MB experiences. Some time is needed before the working set is evicted from the cache. Therefore, the curve increases not immediately, but between around 3 ms and 13 ms. After this time all data must be reloaded from main memory. The incurred costs are comparable to those of the constructed background load (Fig. 5.4(b)).

5.3.3 Migration

This section presents the results of different migration kinds. Both the costs of migrations on cores of the same processor and the costs of migrations between different processors are evaluated. The latter case can only be studied on the Dell system because the AMD system possesses only one physical core.

The first part of this section shows the results in an otherwise idle system, the second part presents migrations in a loaded system using the constructed and realistic background load.

Appendix A contains the measured data of all graphs included in this section.

Idle system

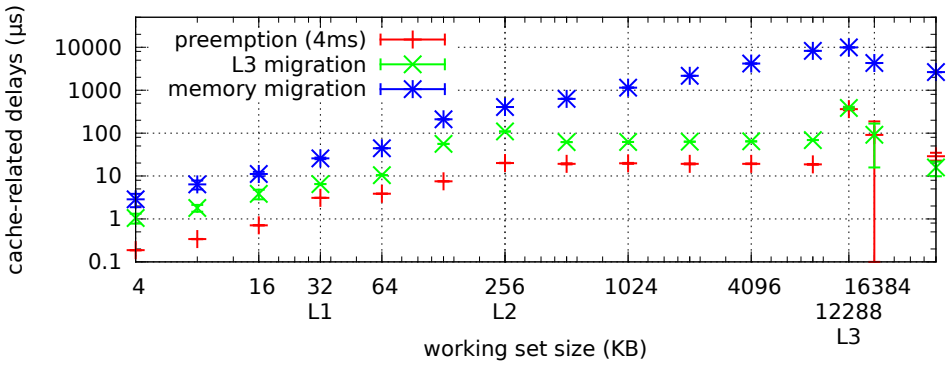
A migration in an otherwise idle system allows to determine the time requirements that are necessary to move the working set between different caches. Depending on which cores are involved it is possible to study the influences of private and shared caches.

Fig. 5.8 shows the results of different migration-kinds measured on the Dell system. The average costs and their standard deviation are included. Fig 5.8(b) is another view of the data presented in Fig 5.8(a): Costs are shown in μs per KB of the working set. This view allows to determine higher costs of a migration or preemption in reference to the cache sizes.

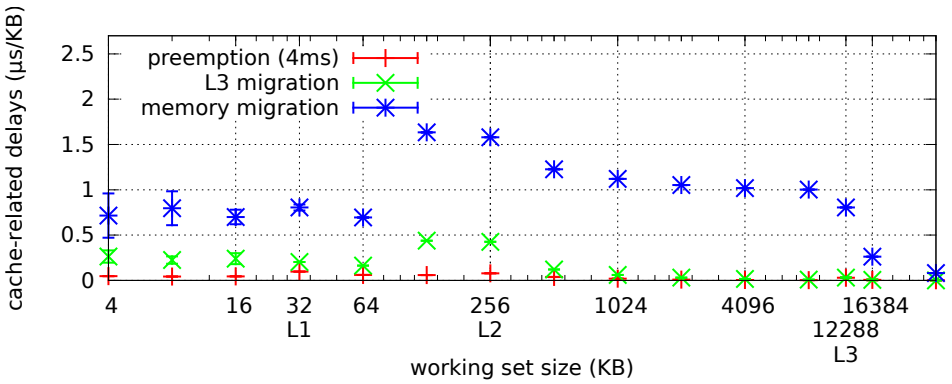
The measurement thread accessed and updated all cache lines within the working set. Each migration was executed 512 times for every working set size; the included preemption curve was determined by measuring 128 values per working set size. In contrast to the experiments of Sec. 5.3.2, the preemption length is fixed now and the size of the working set varies.

As one can see from the blue curve, the costs of a memory migration grow as long as the working set fits into the processor's L3 cache. Working set sizes larger than the 12 MB L3 cache incur reduced costs because the benefits of the cache vanish even in case of a nonexistent migration.

If the working set fits into the L3 cache, a memory migration will add around 1.1 $\mu s/KB$ additional costs which corresponds to the necessary main memory access time to write back a cache line (Tab. 5.3: 1 KB $\equiv$ 16 memory accesses). The experiment of the design leaves all cache lines in a modified state. To transfer the data from the L3 cache to the target processor the MESIF protocol [HG05], an extension of the MESI



(a) Average costs and their standard deviation.



(b) Average costs per KB and their standard deviation.

Figure 5.8: Migration costs in an otherwise idle system (Xeon X5650). The blue curve presents the costs of a memory migration using QPI, the green one the costs of a migration through a shared L3 cache. The red curve shows the costs of a preemption with a length of 4 ms.

protocol, also updates main memory[4]. The target core receives the data through QPI packets.

The highest costs per KB will be experienced if the whole working set size fits into the L2 cache (Fig. 5.8(b)). In this case the data of the inclusive L3 cache is not up to date and must be loaded from the L2 cache before it can be written back to main memory. The reduced costs for working set sizes smaller than 64 KB are based on the properties of main memory chips again.

The green line shows the migration costs through a shared L3 cache. As expected, the costs are always below the costs of a memory migration and grow with the working set size (Fig. 5.8(a)). Higher costs per KB are present for working set sizes up to 256 KB,

[4] Before updating a data value, the target core always reads a cache line to get the pointer to the list element located in the next cache line. Therefore, the cache line is shared between 2 physical processors. The MESIF protocol specifies that the content of a modified cache line has to be written back to main memory before its state changes to *shared* on the source core and to *forwarded* on the target core.

the size of the L2 cache. After migrating to the new core, it is necessary to bring the needed data into the L2 cache of the target core by loading its content from the private L2 cache of the initial core. The additional time required by this operation can be studied in Fig. 5.8(b). It is likely that the step of the graph at 64 KB originates from the properties of main memory again. An L3 migration will have negligible costs in an idle system if the working set size is larger than the size of the L2 cache (256 KB): Besides the last modified cache lines that still reside in the L2 cache of the initial core, the target core will find most parts of the actual data in the L3 cache. Even in case of an execution on one core the data would be loaded from there. Thus, the costs are negligible. The standard deviation in case of an L3 migration is negligible for all working set sizes, except for a working set size of 16 MB. In this case, data requests are handled either by the L3 cache or by main memory. For example, saving the results of the previous measurement already changes the number of cache hits and cache misses, resulting in an increased standard deviation.

A preemption in an idle system results in very low costs. Working set sizes smaller than 1 MB incur costs lower than 0.1 $\mu s/KB$ and are therefore negligible. The costs are not zero because even the setup of the timer and the measurement of the required time itself pollute some contents of the cache. Fig. 5.8(a) gives the misleading impression of a significantly increased standard deviation of a working set size of 16 MB, but it is caused by the logarithmic scale. The reasons are the same as described in case of an L3 migration.

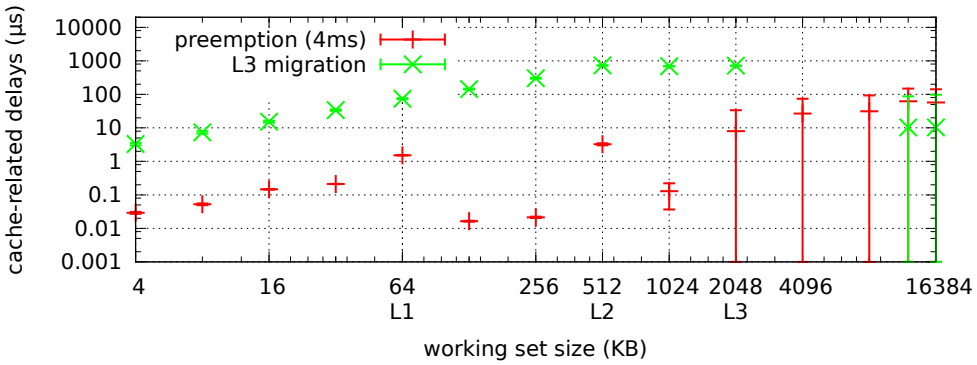
The fixed preemption length of 4 ms shows the same properties as other preemption lengths.

Fig. 5.9 includes the same measurements on the Phenom 9550, except for memory migrations that are not measurable in the system. The costs of an L3 migration grow with the size of the working set up to 0.72 ms. The highest costs of around 1.42 $\mu s/KB$ occur when the working set fits into the L2 cache (Fig. 5.9(b)). Due to the nature of AMD's exclusive caches, a migration in this case enforces to move a cache line from the L2 (or even from the L1) cache of the original core into the private L1 cache of the target, possibly evicting a modified cache line from there. This operation takes place for large parts of the working set resulting in high costs.

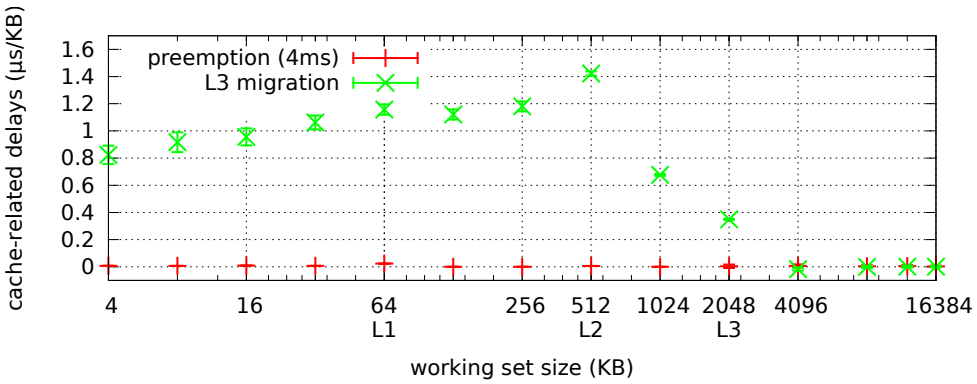
Surprisingly, a working set of 4 MB or 8 MB size incurs negative migration costs (on average 70μs in case of a working set size of 4 MB). The access times determined in the first experiment (Fig. 5.3(b)) are used to calculate the migration costs. Working sets of these size are completely loaded from main memory. After a migration, parts of the working set are available in the private caches of the initial core. When this data is requested, it can be moved into the core's L1 cache without invoking main memory, so the total number of main memory accesses is reduced. Therefore, the costs are negative.

This effect cannot occur in systems with inclusive caches: If the target core replaces the whole content of the L3 cache by loading the working set from main memory, the contents of the private caches of the other cores will be invalidated. Otherwise, a private cache would contain cache lines that are not placed in the L3 cache, which is by definition of an inclusive cache not allowed.

The costs of a preemption with 4 ms length are negligible for all working set sizes in an otherwise idle system as shown in Fig. 5.9(b). Additionally, the variance of the



(a) Average costs and their standard deviation.



(b) Average costs per KB and their standard deviation.

Figure 5.9: Migration costs in an otherwise idle system (Phenom 9550). The green curve shows the costs of a migration through a shared L3 cache, whereas the red one presents the costs of a preemption with a length of 4 ms.

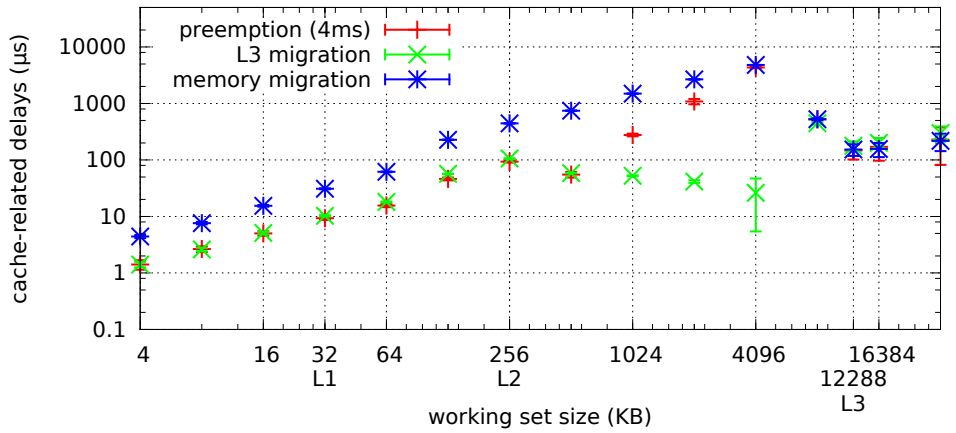
measured data is quite small, even if Fig. 5.9(a) gives again the misleading impression of an increased variance especially for working set sizes exceeding the L3 cache of 2 MB.

Loaded system - Constructed background load

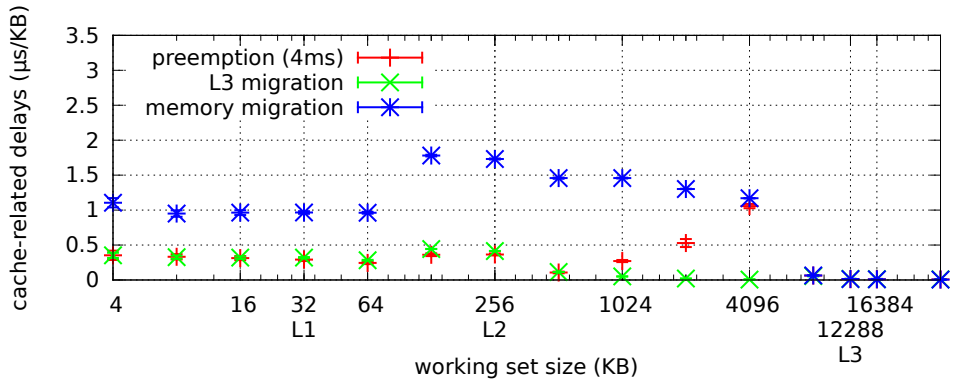
In contrast to the previous section, the system was started with one cache polluter thread per core, each of them accessing a working set of 4 MB. The number of measurements is not changed to achieve comparable results to the previous experiment.

Fig. 5.10 includes the measured data of the Dell system. The maximum costs are present when executing a memory migration. Especially for working set sizes smaller or equal 4 MB the costs of a memory migration are overwhelming in comparison to a preemption or L3 migration. For example, a memory migration of a working set size of 4 MB incurs up to 195 times more costs than an L3 migration (Fig. 5.11).

If the working set exceeds a size of 4 MB, the polluter threads will always evict all cache lines of the measurement thread from the L3 cache. Hence, the costs of a preemption,



(a) Average costs and their standard deviation.



(b) Average costs per KB and their standard deviation.

Figure 5.10: Migration costs in a loaded system (Xeon X5650).

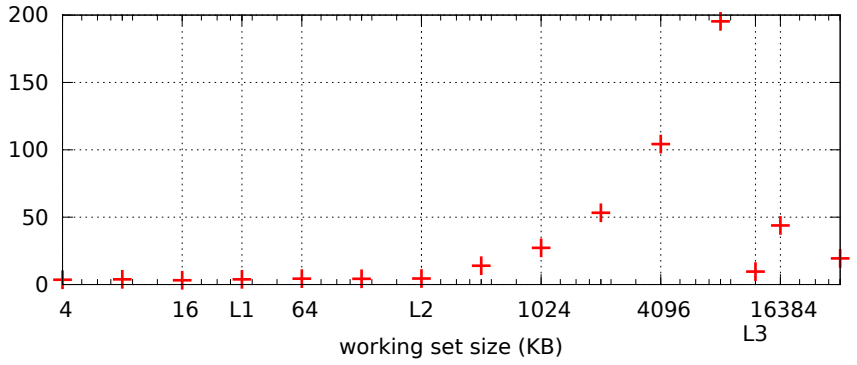
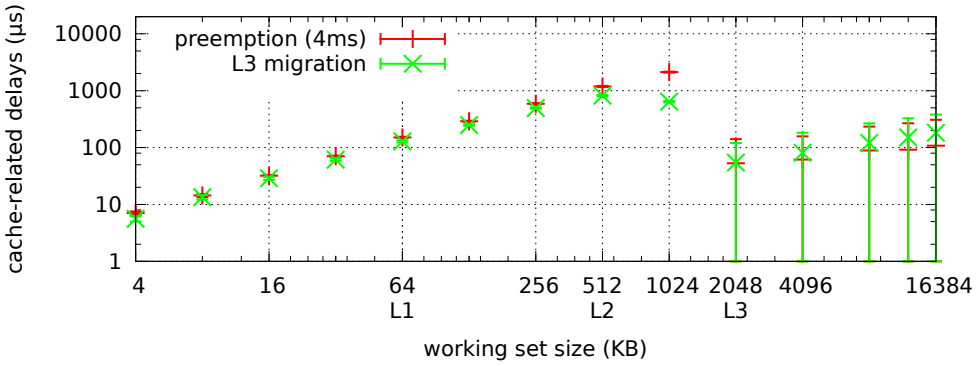


Figure 5.11: Ratio of the costs of a memory and L3 migration.

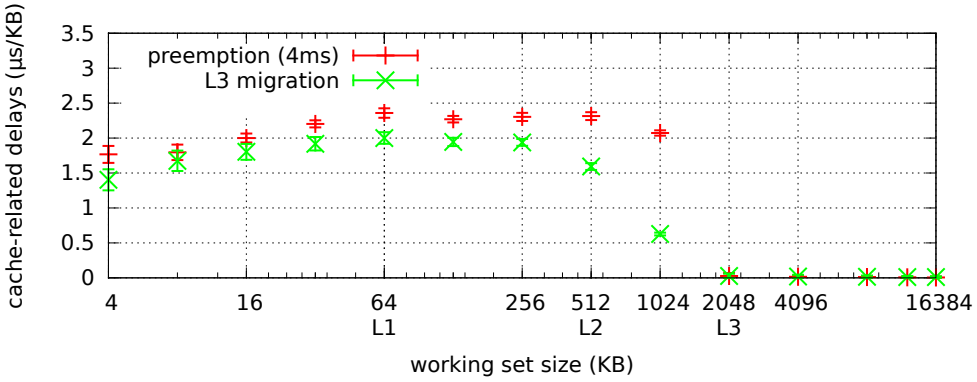
L3, and memory migration become equal. All needed data has to be loaded from main memory.

Jobs with larger working set sizes than the L2 cache do not experience additional costs when migrating between cores that share the L3 cache (Fig. 5.10(b)). If the working set fits into the L2 cache, additional costs up to 0.44 $\mu s/KB$ necessary to move the data between the caches will be experienced.

The experiment was also carried out using varied preemption lengths: If the length of a preemption grows above 5 ms, the costs of a preemption will always be higher than the costs of an L3 migration. This number heavily depends on the load of the system.



(a) Average costs and their standard deviation.



(b) Average costs per KB and their standard deviation.

Figure 5.12: Migration costs in a loaded system (Phenom 9550).

Fig. 5.12 shows the costs of a 4 ms preemption and L3 migration in the AMD system. The maximum costs per KB of a preemption are around 2.3 $\mu s/KB$ for working set sizes that fit into the L2 cache. Working sets larger than 1 MB experience no additional costs. Even longer preemption lengths do not increase the maximum costs and, therefore, one can be sure that 4 ms are enough for the cache polluter thread to evict all useful cache lines of the L2 cache. Varying the preemption length changes only the first part of the curve and a preemption length above 4 ms results in costs that are always above the

costs of an L3 migration. The preemption costs decrease for working set sizes larger than 512 KB and are negligible for working sets that exceed the size of the L3 cache.

The measurement thread experiences the highest L3 migration costs when its working can be hold by the L1 and L2 cache. A working set size of 64 KB incurs on average 2.0 $\mu s/KB$ delay during an L3 migration. The costs become negligible for working set sizes about 2 MB.

Loaded system - Realistic background load

The load of the system was replaced with one VM per core in this experiment. Each VM compiled the Linux kernel in an endless loop, polluting the caches. The number of rounds per experiments was not changed, so the results are comparable to the previous measurements.

Fig. 5.13 shows the measured values of different working set sizes. As before, memory migrations incur the highest costs because the working set has to be transferred between the physical cores using QPI.

In contrast to the previous experiments, a preemption and L3 migration show higher variances. This property is based on the kernel compilation that is not as regular as the constructed background load. However, the characteristics of an L3 and memory migration are the same as before.

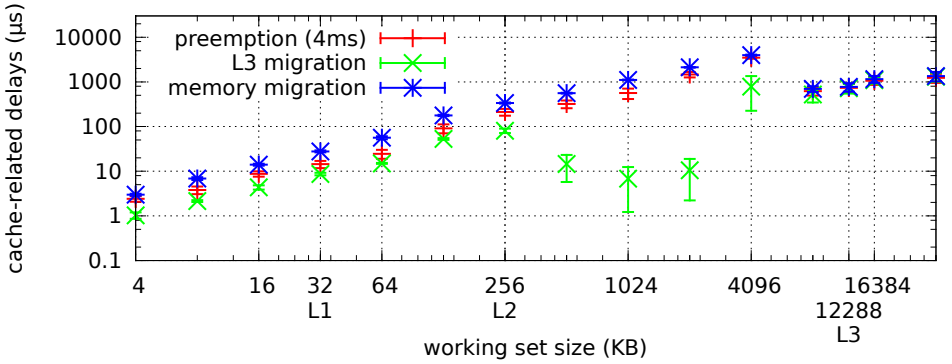
The costs of a preemption per KB grow for working set sizes up to 4 MB. If the working set is larger than 4 MB, the VMs will always evict all cache lines of the measurement thread from the L3 cache. Hence, the costs of a preemption, L3, and memory migration becomes equal (Fig. 5.7(b)).

A preemption length longer than 3 ms results in preemption costs that are always above the costs of an L3 migration in the system.

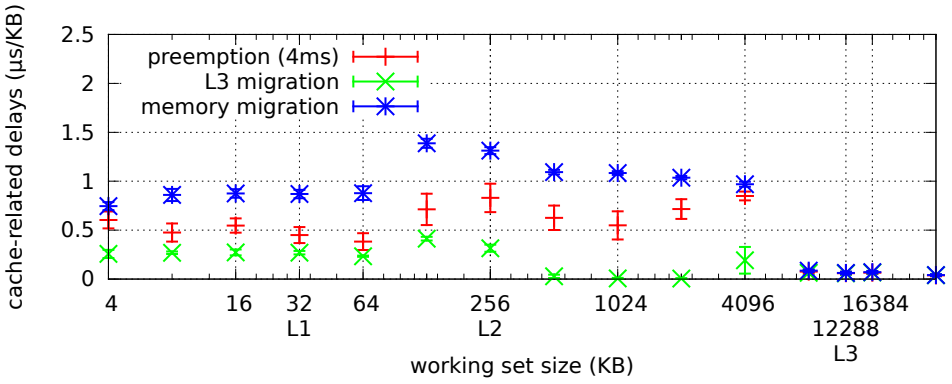
Bastoni at al. did not use a fixed preemption delay in their measurements when they compared preemption and migration costs. To evaluate their observations on the Dell system, Fig. 5.13(c) is included: The red curve was constructed by sampling 1024 measurements per working set size. It shows the costs of a preemption and its standard deviation with uniformly distributed preemption lengths having a mean of 25 ms. Obviously, the variance of the measured costs is higher. Depending on the randomly chosen length of the preemption, more or less valid cache lines are evicted from the cache. However, the costs of a preemption for all working set sizes that fit into the L3 cache are lower than the costs of a memory migration.

The maximum occurred costs per KB of a memory migration are always around 0.2 $\mu s/KB$ above the maximum costs of a preemption for working set sizes smaller than 4 MB. For larger working set sizes the difference of the worst case costs of an L3, memory migration, and a preemption are negligible.

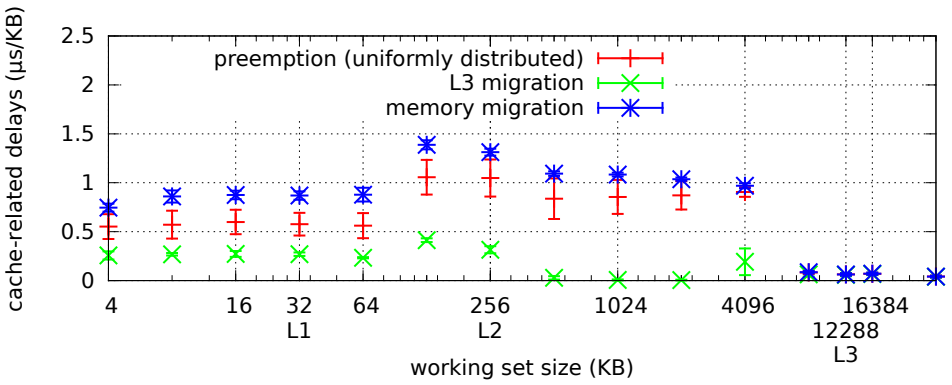
The basic characteristics of the AMD system (Fig. 5.14) are analogous to the measured values using the constructed background load (Fig. 5.12). However, due to the changing load of the VMs, the variance of the costs of both a preemption and migration is increased.



(a) Average costs and their standard deviation.

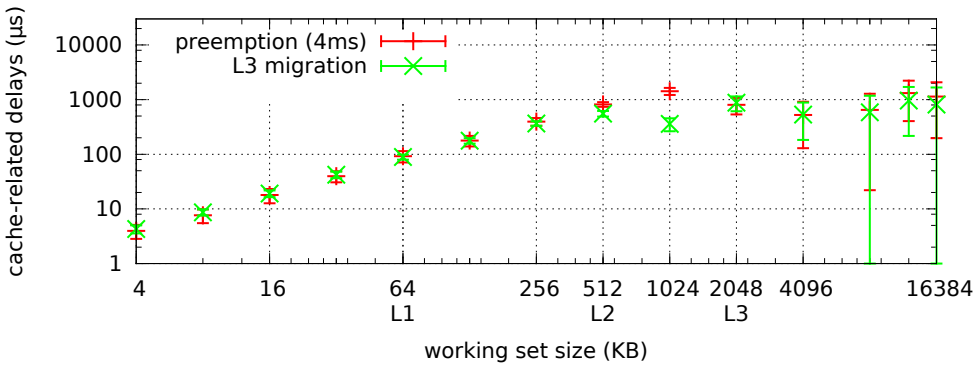


(b) Average costs per KB and their standard deviation.

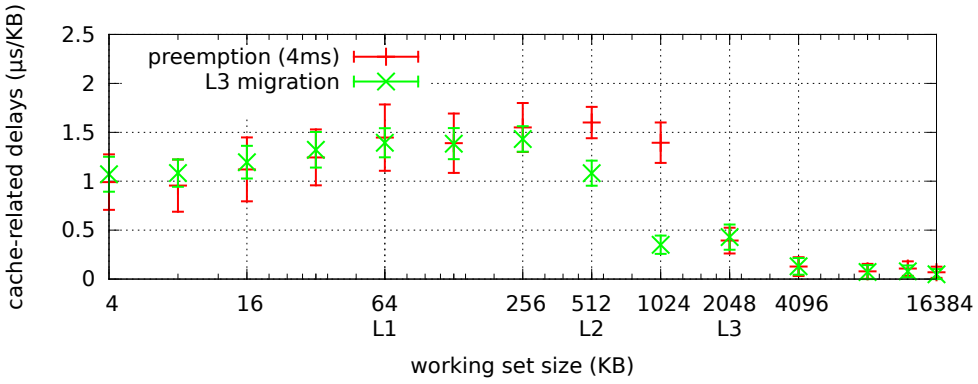


(c) Average costs per KB and their standard deviation. Uniformly distributed preemption length (mean 25 ms).

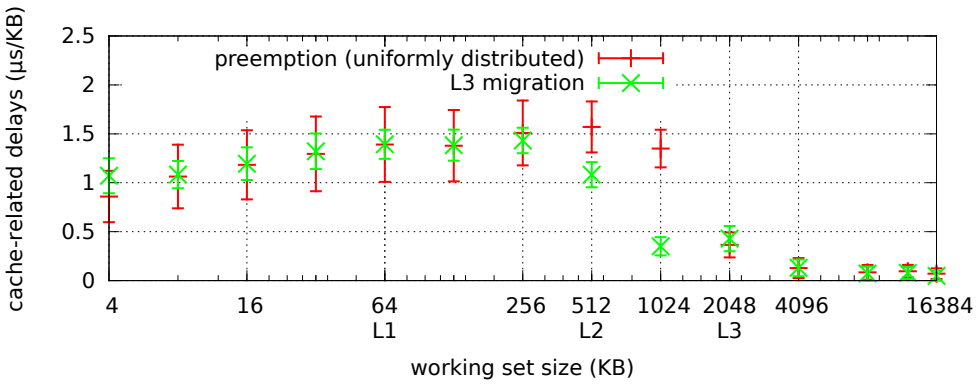
Figure 5.13: Migration costs in a realistically loaded system (Xeon X5650).



(a) Average costs and their standard deviation.



(b) Average costs per KB and their standard deviation.



(c) Average costs per KB and their standard deviation. Uniformly distributed preemption length (mean 25 ms).

Figure 5.14: Migration costs in a realistically loaded system (Phenom 9550).

On average a preemption of 4 ms length produces comparable costs to an L3 migration in the AMD system. Only shorter preemption length reduce the costs of a preemption in a way that an L3 migration becomes more expensive on average.

6 Conclusion

The first section of this chapter compares the observations made by Bastoni et al. in [BBA10a] with the results of the evaluation in Chapter 5. Thereafter, a summary of factors influencing the costs of CPMD is provided. The third section gives a summary about the predictability of CPMD. Furthermore, the fourth section announces sound scheduling decisions minimizing the costs of CPMD derived from the results of this work. Finally, additional observations are presented in the fifth section followed by a short summery that completes the work.

6.1 Comparison to observations of Bastoni et al.

Sec. 2.5 gives a short overview about the five observations made by Bastoni et al. in [BBA10a]. Their observations, given again in *emphasized letters* at the beginning of the following paragraphs, are compared to the measurements on the Dell and AMD system.

1. *The predictability of CPMD depends heavily on the size of the different cache levels.*

   The variance of the data measured by Bastoni et al. increases significantly when the working set approaches the size of the L2 cache that is shared between two cores in their system. They conclude that overhead estimations are imprecise and unpredictable for these working sets.

   The costs of CPMD measured on the Dell and AMD system are shown in Fig. 5.13 and Fig. 5.14. The results show differences between the variances of the Dell and AMD system that are contrary to the observation made by Bastoni et al.

   In the loaded Dell system the variance of the measured data in case of a memory migration stays negligible for all measured working set sizes. This effect is introduced by the MESIF cache coherency protocol used in NUMA systems.

   An L3 migration shows an increased standard deviation for some working set sizes especially if cache hits are intermingled with cache misses. In contrast to [BBA10a], the standard deviation is smaller, such that overhead estimations are still predictable. Bastoni et al. argue about the memory bus contention that produces the increased variance, so it is likely that these are architecture dependent properties.

   An increased variance in case of a preemption cannot be seen in the loaded Dell system, so the costs introduces by a preemption are predictable.

   The AMD system shows an increased variance for both preemption and migration in case of working set sizes that exceed the size of the L3 cache, but it is not as large as announced in [BBA10a].

Next to the system architecture, the different properties might be based on the used operating systems. Bastoni et al. used an adapted version of Linux called LITMUS^{RT}. The size of NOVA including the services running in user land are orders of magnitude smaller and may not influence the caches in such a way as Linux does.

2. *There are no substantial differences between the average and maximum costs of a preemption and migration in a loaded system.*

   Bastoni et al. used a uniformly distributed preemption length with a mean of 25 ms. The corresponding measurements on the Dell and AMD system are shown in Fig. 5.13(c) and Fig. 5.14(c), the average and maximum measured delays can be found in Appendix A.

   The observation does not match on the loaded Dell system: The average and maximum costs of an L3 migration are fundamentally lower than those of a memory migration. The costs of a preemption are in between.

   The measurement results of the AMD system are comparable to the observation for most of the working set sizes in the average case. The maximum measured costs of a preemption are higher than those of an L3 migration.

3. *Preemptions cause always less delay than migrations in an idle system. The costs of L3 and memory migrations are comparable.*

   The measurements on the Dell and AMD system are shown in Fig. 5.8(b) and Fig. 5.9(b).

   In the Dell system the costs of a preemption are as expected near zero, but the costs of an L3 and memory migration are significantly different. The Xeon processors are connected to each other via QPI. Therefore, a memory migration is much more expensive than a migration on cores sharing the L3 cache.

   Corresponding to the observation, the preemption costs are negligible in the idle AMD system, but it is possible to achieve negative L3 migration costs under very specific conditions (Sec. 5.3.3). So, contrary to their observation, there are rare situations where an L3 migration produces fewer costs than a preemption in an idle system.

4. *The costs of CPMD are strongly related to the preemption length*[1] *unless cache affinity is completely lost.*

   Sec. 5.3.2 shows the same results.

5. *The number of tasks in a system does not significantly influence CMPD.*

   The schedule-sensitive method (Sec. 3.1) is needed for this analysis, which was not implemented in this work. Therefore, the lack of measurements allows no statements.

[1] Bastoni et al. also studied the migration costs after an initial preemption, therefore they speak about the preemption length related to CPMD and not only CPD.

6.2 Factors influencing the costs of CPMD

This section summarizes the different factors that influence the costs of CPMD. Some of these factors depend on the background load that is at least in interactive systems unknown beforehand. An approach to get rid of the influences on CPMD introduced by the background load is presented.

The previous section shows that there are essential differences in the measured costs of CPMD between the used architectures. Therefore it is obvious that the characteristics of a system architecture are one of the main factors influencing the costs of CPMD. This includes among others:

- size of the caches
- kind of caches: inclusive vs. exclusive; shared vs. private
- way the memory is accessed: UMA vs. NUMA
- cache coherency protocol
- ratio of the access times (caches, main memory)

Besides the system-architecture's characteristics, the running application influences its sensibility to CPMD by its own

- working set size,
- memory access pattern and
- rate of memory accesses that modify cache lines.

Further, factors influencing CPMD include:

- length of a preemption
- cores involved in a migration
- properties of the operating system

Finally, the background load also changes the costs of CPMD an application experiences:

- cores that run background load
- combined working set size, memory access pattern and rate of memory accesses that modify cache lines

The listed factors are not complete. Especially in interactive systems, the background load is typically unknown beforehand and not well defined. This additionally complicates the estimation of the costs for a particular application. One existing approach exists that allows to separate the background load from an application: *cache coloring* [LHH97]. This technique partitions the caches into distinct sets of cache lines. Running the background load that is only known during runtime in a separate cache partition prevents the eviction of cache lines that would otherwise influence the costs of CPMD for a specific application. This mechanism reduces the total number of factors that influence CPMD and it might be possible by a generic function to predict the costs of CPMD in advance.

6.3 Prediction of CPMD

The prediction of CPMD costs could help the scheduler of a system to optimize its scheduling decisions. This section shows an approach to predict the worst case costs of a preemption. Further, the prediction of the migration costs is discussed in the last part of this section.

As shown in the previous section, the costs of CPMD are influenced by many factors. Therefore, a generic function that calculates the costs of CPMD would take a large number of factors as its input that are possibly unknown in advance, so the calculability of such a generic function is problematic.

However, it will be possible to calculate the worst case costs of a preemption, if the access times to the different cache levels and to main memory are known: In the best case scenario, all cache levels would be filled with distinct parts of the working set. The total time to access these parts can be calculated approximately. Additionally, the number of cycles per main memory access allows to calculate the worst case time to load the working set. Subtracting the best case access time from worst case one results in worst case CPD costs. For more details see Sec. 5.3.2.

Reasoning about the costs of CMD might be impossible without measurements. The costs depend on how the working set can be transferred between the caches that might be private or shared between cores. Additionally, the cache coherency protocol also influences the costs because it specifies when to update main memory. The background load must be taken into account because it changes the time a valid cache lines will stay stored in the cache, etc.

6.4 Derived scheduling decisions for the Dell and AMD system

The experiments that were carried out in this work estimated the costs of CPMD for the Dell and AMD system. For most of the existing workloads, the determined costs are the worst case costs in this systems even if there might be some very rare cases that result in higher costs of CPMD. The measured costs allow to draw conclusions about sound scheduling decisions that will minimize the costs. These decisions are presented in this section for both systems.

Fig. 5.13(c) allows to reason about scheduling decisions that minimize CPMD costs in the loaded Dell system. The figure shows that working set sizes about 8 MB experience the same costs in case of a preemption, L3 migration, and memory migration. If the scheduler of the system detects a core that is overloaded, it can migrate an application with a working set size of about 8 MB to any other core without introducing additional costs of CMD.

On average an L3 migration causes less costs than a preemption with an uniformly distributed preemption length of 25 ms. Therefore, whenever the scheduler has to decide between a (long) preemption and an L3 migration, it should migrate the job to another free core on the same processor to prevent a preemption.

If the working set size of an application is smaller than 256 KB, the preemption length known beforehand and shorter than 4 ms, the application should be preempted. The boundary of 4 ms depends on the background load. In another scenario where the working set of the background load is very small, a preemption might always be the best choice.

Fig. 5.14(c) is used to extract sound scheduling decisions for the AMD system. Except for working set sizes between 256 KB and 2048 KB, the costs of a preemption and L3 migration are similar, but the standard deviation of the average costs of a preemption is two times larger than in case of an L3 migration. Because the costs are similar, the scheduler can migrate the job to another core, or it can also preempt the job.

For working set sizes fitting into the L3 cache a migration should be preferred because of the lower costs.

6.5 Observations not presented in [BBA10a]

This section presents two additional observations that were not presented in [BBA10a].

1. *Memory migrations should be avoided in NUMA systems using the MESIF cache coherency protocol.*

   Regardless of the job's working set size, a memory migration should be avoided in a NUMA system that uses the MESIF cache coherency protocol. A memory migration copies the data of modified cache lines back to main memory introducing high costs. As a common rule, migrations between physical processors should be avoided whenever possible.

2. *Migrations can produce negative costs of CPMD in idle systems with exclusive caches.*

   An application can experience negative costs of CPMD as described in Sec. 5.3.3, if it meets some very specific conditions: otherwise idle system, working set of an application exceeds total size of the caches that are usable by a core, exclusive caches, randomly accessed working set. Basically, the migration enlarges the total cache size by using the private caches of the core the job was migrated to.
   Unfortunately, the number of necessary constraints and the overall speedup of such a migration is not relevant for practical usage (Fig. 5.9), but it shows that the costs of a migration can also be negative.

6.6 Summary and future work

Within this work, a mechanism to determine CPMD with a special measurement task using NOVA was introduced and the resulting costs were estimated on two systems. Two different approaches to simulate background load were used to get realistic costs.

Furthermore, several observations made by Bastoni et al. were verified by this work. Nevertheless, differences related to the characteristics of migrations were founds that are probably based on the different architectures of the used systems.

The costs of CPMD estimated in the measurements on the Dell and AMD system were used to conclude about scheduling decisions that would help to minimize the costs of CPMD. Additionally, two observations were generalized from the measurements: At first, memory migrations in a NUMA system should be avoided. In addition, a migrated application might experience also negative costs of CMD.

In future work, the migration costs on other NUMA based systems should be studied. Thereby, it is important the measure memory migration costs on systems using different cache coherency protocols. Additionally, the lack of multiple processors in the used AMD system does not allow the analysis of memory migrations. This measurements would have to be repeated with a suitable system to get more information about the resulting costs.

Finally, after examining the Dell and AMD system it is possible to draw conclusions about the costs of preemption and migration delays that are mainly based on the working set size of an application. However, a generic prediction – if such a prediction is possible at all – of CPMD for arbitrary tasks requires further studies.

Appendix A

Measured CPMD costs

A.1 Xeon X5650

WSS (KB)	Pre. (4 ms)	Pre. (uniformly distr.)	L3 mig.	Mem mig.
4	0,19	0,21	4,59	8,08
8	0,36	0,36	5,16	11,24
16	0,79	0,78	8,22	16,18
32	3,17	3,15	6,57	27,67
64	3,88	3,83	10,76	47,74
128	7,60	7,61	56,08	210,51
256	22,45	22,14	109,74	406,11
512	20,31	22,63	62,44	638,13
1024	20,98	23,17	63,06	1.163,21
2048	20,75	22,74	63,88	2.159,40
4096	20,24	22,50	65,03	4.180,75
8192	19,97	22,48	69,71	8.222,11
12288	463,54	484,04	625,56	9.933,27
16384	446,31	460,10	453,42	4.319,74
32768	45,08	48,09	32,62	2.653,43

Table A.1: Maximum delay in μs in idle system.

WSS (KB)	Pre. (4 ms)	Pre. (uniformly distr.)	L3 mig.	Mem mig.
4	2,10	5,31	1,88	5,23
8	3,54	8,54	3,18	8,96
16	7,06	16,82	5,84	17,20
32	12,75	32,50	11,24	32,74
64	21,34	58,08	19,12	64,36
128	65,01	192,32	57,11	232,87
256	117,31	380,46	107,71	450,14
512	94,73	654,68	60,12	760,34
1024	306,99	1.303,49	55,90	1.511,99
2048	1.175,63	2.443,58	63,23	2.687,75
4096	4.388,15	4.816,63	324,84	4.835,19
8192	612,57	749,96	609,36	634,10
12288	301,07	408,19	334,59	318,68
16384	401,00	504,18	410,61	420,40
32768	781,74	694,62	685,62	642,01

Table A.2: Maximum delay in μs in system with constructed background load.

WSS (KB)	Pre. (4 ms)	Pre. (uniformly distr.)	L3 mig.	Mem mig.
4	3,33	3,39	2,34	3,59
8	5,37	6,33	2,27	8,31
16	10,57	12,24	6,03	16,04
32	19,88	23,67	9,23	29,95
64	39,43	47,22	18,74	64,95
128	134,11	166,07	57,01	194,63
256	294,16	319,05	104,93	378,65
512	454,02	564,41	43,20	612,53
1024	913,57	1.051,34	34,30	1.180,15
2048	1.918,42	2.075,19	85,04	2.274,71
4096	3.948,62	4.048,75	3.484,04	4.306,89
8192	984,45	1.263,38	1.123,38	1.162,41
12288	1.309,33	1.545,21	1.176,16	1.335,90
16384	1.635,19	1.824,97	1.827,57	1.846,09
32768	1.909,28	2.608,37	2.203,13	2.498,84

Table A.3: Maximum delay in μs in system with realistic background load.

WSS (KB)	Pre. (4 ms)		Pre. (uniformly distr.)		L3 mig.		Mem mig.	
	Avg.	Std. dev.	Avg.	Std. dev.	Avg.	Std. dev.	Avg.	Std. dev.
4	0,19	0,00	0,19	0,00	1,05	0,27	2,86	0,98
8	0,34	0,00	0,34	0,00	1,80	0,32	6,37	1,49
16	0,71	0,01	0,73	0,01	3,84	0,98	11,19	1,29
32	3,10	0,01	3,13	0,01	6,50	0,02	25,76	1,06
64	3,88	0,00	3,82	0,00	10,47	0,01	44,45	0,25
128	7,54	0,01	7,57	0,00	56,06	0,08	209,09	0,49
256	20,13	0,22	20,50	0,12	109,09	0,23	404,43	0,59
512	19,27	0,53	19,31	0,52	61,67	0,03	628,16	1,86
1024	19,74	0,51	19,69	0,55	62,30	0,05	1.147,36	1,98
2048	19,20	0,50	19,55	0,56	63,10	0,06	2.154,51	1,64
4096	19,27	0,31	19,01	0,92	64,24	0,08	4.173,12	2,31
8192	18,69	0,46	18,75	0,55	68,95	0,12	8.213,65	3,26
12288	362,07	20,48	373,12	11,75	386,12	24,63	9.899,60	4,39
16384	90,99	95,77	91,94	78,95	91,16	75,37	4.293,41	13,79
32768	28,84	6,07	30,54	5,87	15,82	5,95	2.632,98	6,86

Table A.4: Average delay in μs in idle system.

WSS (KB)	Pre. (4 ms)		Pre. (uniformly distr.)		L3 mig.		Mem mig.	
	Avg.	Std. dev.	Avg.	Std. dev.	Avg.	Std. dev.	Avg.	Std. dev.
4	1,41	0,27	3,79	1,00	1,41	0,15	4,42	0,27
8	2,65	0,31	6,63	1,55	2,61	0,18	7,60	0,37
16	5,01	0,45	13,15	3,19	5,12	0,26	15,41	0,55
32	9,32	0,66	27,20	6,41	10,25	0,28	30,90	0,60
64	15,60	1,00	48,68	11,36	18,03	0,36	61,61	0,86
128	46,23	2,60	167,38	44,37	56,75	0,11	227,84	1,56
256	93,45	3,80	335,49	85,51	106,13	0,31	443,18	2,09
512	55,10	6,21	574,78	164,67	58,53	0,47	746,11	4,53
1024	277,38	15,32	1.147,24	323,54	52,52	0,87	1.493,41	5,35
2048	1.082,21	117,26	2.265,42	483,75	41,57	2,17	2.666,97	5,79
4096	4.328,51	125,67	4.642,69	468,85	26,20	20,75	4.793,98	9,30
8192	522,81	19,26	550,04	59,20	447,44	22,54	526,67	17,97
12288	146,94	45,31	180,43	43,01	178,01	35,30	153,00	34,97
16384	170,33	73,62	183,99	57,24	200,64	43,40	156,30	43,88
32768	229,31	147,39	249,05	96,52	300,50	91,14	217,71	74,56

Table A.5: Average delay in μs in system with constructed background load.

WSS (KB)	Pre. (4 ms)		Pre. (uniformly distr.)		L3 mig.		Mem mig.	
	Avg.	Std. dev.	Avg.	Std. dev.	Avg.	Std. dev.	Avg.	Std. dev.
4	2,42	0,35	2,21	0,51	1,03	0,17	2,98	0,16
8	3,81	0,74	4,57	1,14	2,15	0,10	6,88	0,50
16	8,75	1,17	9,58	1,99	4,37	0,48	14,00	0,71
32	14,40	2,62	18,46	3,72	8,65	0,56	27,82	1,38
64	24,51	5,51	35,91	8,20	14,96	0,38	56,22	4,51
128	91,22	20,49	135,30	22,68	52,83	2,47	177,67	5,92
256	212,36	37,10	268,38	48,38	80,75	9,20	336,02	9,08
512	320,47	63,77	428,51	105,97	14,49	8,75	559,95	11,84
1024	561,97	147,05	875,75	178,17	6,83	5,61	1.109,31	19,52
2048	1.465,56	206,88	1.784,84	296,34	10,52	8,31	2.122,56	34,53
4096	3.479,08	185,12	3.706,57	201,28	786,08	559,83	3.969,66	103,71
8192	622,52	127,07	677,40	113,59	523,54	178,64	699,61	103,19
12288	739,26	166,51	793,46	147,30	722,66	174,59	771,13	166,40
16384	1.063,45	205,09	1.141,28	179,46	1.093,77	180,88	1.153,35	203,55
32768	1.232,74	285,29	1.368,02	310,85	1.308,37	295,34	1.349,21	316,24

Table A.6: Average delay in μs in system with realistic background load.

A.2 Phenom 9500

WSS (KB)	Pre. (4 ms)	Pre. (uniformly distr.)	L3 mig.
4	0,03	0,06	4,17
8	0,06	9,50	9,01
16	0,15	0,18	19,28
32	0,22	0,24	42,13
64	1,54	1,58	91,18
128	0,02	-0,01	172,15
256	0,02	0,53	385,32
512	3,36	3,50	794,16
1024	0,77	5,47	752,59
2048	206,41	246,88	771,27
4096	298,78	296,16	201,83
8192	304,28	306,87	261,54
12288	325,66	331,96	287,37
16384	312,15	341,43	275,64

Table A.7: Maximum delay in μs in idle system.

WSS (KB)	Pre. (4 ms)	Pre. (uniformly distr.)	L3 mig.
4	8,57	9,47	7,31
8	17,62	18,67	17,36
16	36,44	41,41	35,56
32	79,84	82,09	76,95
64	175,55	179,80	156,50
128	339,08	340,88	299,59
256	681,27	687,10	591,57
512	1.370,03	1.372,77	1.110,94
1024	2.492,85	2.502,89	810,13
2048	569,17	565,29	572,54
4096	569,46	580,28	597,12
8192	589,96	611,68	639,71
12288	580,64	594,78	640,16
16384	586,87	604,66	659,09

Table A.8: Maximum delay in μs in system with constructed background load.

WSS (KB)	Pre. (4 ms)	Pre. (uniformly distr.)	L3 mig.
4	9,59	12,41	8,45
8	15,41	23,63	13,15
16	57,10	64,32	28,13
32	80,17	105,04	67,57
64	164,44	186,63	132,33
128	293,08	398,06	266,88
256	855,04	788,05	533,26
512	1.108,43	1.353,85	788,47
1024	2.359,10	2.448,70	719,94
2048	1.822,96	2.091,86	2.110,67
4096	1.897,90	2.334,47	2.753,51
8192	3.931,90	4.488,40	3.604,41
12288	4.314,37	5.085,33	3.784,79
16384	5.262,35	4.852,59	4.200,05

Table A.9: Maximum delay in μs in system with realistic background load.

WSS (KB)	Pre. (4 ms)		Pre. (uniformly distr.)		L3 mig.	
	Avg.	Std. dev.	Avg.	Std. dev.	Avg.	Std. dev.
4	0,03	0,00	0,03	0,00	3,28	0,27
8	0,05	0,00	0,07	0,29	7,29	0,58
16	0,15	0,00	0,15	0,00	15,21	1,00
32	0,21	0,00	0,20	0,00	33,80	1,61
64	1,51	0,01	1,52	0,02	73,67	2,46
128	0,02	0,00	-0,03	0,00	142,73	4,74
256	0,02	0,00	0,03	0,02	300,59	9,36
512	3,22	0,22	3,06	0,09	724,17	8,00
1024	0,13	0,09	0,41	0,25	688,94	7,06
2048	7,95	25,48	14,97	11,20	711,32	7,36
4096	26,50	47,12	24,50	42,20	-70,07	41,81
8192	30,92	61,16	34,20	64,15	-11,08	61,73
12288	61,68	86,21	56,70	79,58	10,22	76,46
16384	56,71	84,81	57,97	87,09	10,22	86,86

Table A.10: Average delay in μs in idle system.

WSS (KB)	Pre. (4 ms)		Pre. (uniformly distr.)		L3 mig.	
	Avg.	Std. dev.	Avg.	Std. dev.	Avg.	Std. dev.
4	7,03	0,48	6,87	1,52	5,58	0,60
8	14,29	0,88	14,10	2,70	13,32	1,16
16	31,84	1,02	32,10	6,85	28,69	1,81
32	70,12	1,63	64,85	14,03	61,06	3,07
64	150,13	4,31	145,06	30,14	127,35	5,30
128	289,00	6,01	278,49	52,43	247,55	7,47
256	586,31	14,97	566,09	98,57	493,37	12,24
512	1.178,70	28,77	1.143,56	172,82	811,41	24,77
1024	2.111,12	39,50	2.091,09	167,48	639,26	20,33
2048	52,64	87,13	49,00	71,76	54,05	65,39
4096	61,32	95,68	64,91	106,29	80,44	100,90
8192	88,40	144,30	78,64	138,54	120,55	143,72
12288	91,12	175,36	90,92	172,24	150,24	174,03
16384	107,51	198,63	102,35	193,65	180,81	197,69

Table A.11: Average delay in μs in system with constructed background load.

WSS (KB)	Pre. (4 ms)		Pre. (uniformly distr.)		L3 mig.	
	Avg.	Std. dev.	Avg.	Std. dev.	Avg.	Std. dev.
4	3,94	1,13	3,42	1,04	4,26	0,71
8	7,60	2,12	8,47	2,59	8,62	1,11
16	17,84	5,20	18,83	5,62	19,01	2,66
32	39,59	9,08	41,21	12,13	42,05	5,80
64	92,03	21,56	88,55	24,31	88,67	9,46
128	176,79	38,65	175,52	46,39	176,17	20,20
256	394,49	63,59	384,15	84,27	364,55	33,12
512	814,67	81,40	799,45	132,50	551,12	64,96
1024	1.419,04	210,23	1.374,53	195,35	357,44	94,57
2048	801,09	267,48	742,81	260,15	873,45	263,11
4096	520,13	391,65	523,08	417,33	530,35	348,31
8192	646,45	624,52	686,71	610,14	584,36	585,05
12288	1.308,13	905,67	1.174,43	794,52	958,18	742,00
16384	1.129,34	933,83	1.153,77	897,38	807,03	856,48

Table A.12: Average delay in μs in system with realistic background load.

Glossar

APIC advanced programmable interrupt controller

ccNUMA cache coherent non uniform memory access

CMD Cache related migration delay

CPD Cache related preemption delay

CPMD Cache related preemption and migration delay

CPU Central processing unit

EC Execution context

L1 Level 1

L2 Level 2

L3 Level 3

LRU Least recently used

MESIF Modified exclusive shared invalid forwarded

MESI Modified exclusive shared invalid

NOVA NOVA OS Virtualization Architecture

NUMA Non uniform memory access

NUL NOVA UserLand

PD Protection domain

QPI Intel QuickPath Interconnect

SC Scheduling context

SM Semaphore

SMM System management mode

SMT Simultaneous Multithreading

TCB Trusted computing base

TSC Time stamp counter

UMA uniform memory access

UTCB user thread control block

VM Virtual machine

WBINVL Write back invalidate

Bibliography

[amd11] *AMD64 Architecture Programmer's Manual, Vol 2: System Programming*, September 2011. 6

[BBA10a] Andrea Bastoni, Björn. B. Brandenburg, and James. H. Anderson. Cache-related preemption and migration delays: Empirical approximation and impact on schedulability. In *Proceedings of the Sixth International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT 2010)*, pages 33–40, 2010. VI, 3, 8, 17, 20, 57, 61

[BBA10b] Andrea Bastoni, Björn. B. Brandenburg, and James. H. Anderson. An empirical comparison of global, partitioned, and clustered multiprocessor edf schedulers. In *Real-Time Systems Symposium (RTSS), 2010 IEEE 31st*, pages 14 –24, 30 2010-dec. 3 2010. 1

[Dre07] Ulrich Drepper. *What Every Programmer Should Know About Memory*, November 2007. 40

[HG05] Herbert H. J. Hum and James R. Goodman. *Forward state for use in cache coherency in a multiprocessor system (United States Patent)*. Intel, July 2005. 47

[int11a] *Intel®64 and IA-32 Architectures, Optimization Reference Manual*, June 2011. 13, 39, 40

[int11b] *Intel®64 and IA-32 Architectures, Software Developer's Manual, Volume 1: Basic Architecture*, May 2011. 12, 16

[int11c] *Intel®64 and IA-32 Architectures, Software Developer's Manual, Volume 2: Instruction Set Reference, A-Z*, October 2011. 33

[int11d] *Intel®Server Boards S5520HC, S5500HCV, and S5520HCT Technical Product Specification, rev 1.9*, June 2011. 13

[LHH97] Jochen Liedtke, Hermann Härtig, and Michael Hohmuth. Os-controlled cache predictability for real-time systems. In *IEEE Real Time Technology and Applications Symposium*, pages 213–. IEEE Computer Society, 1997. 60

[Liu00] Jane W. S Liu. *Real-time systems*. Prentice Hall, 2000. 1

[PH09] David A. Patterson and John L. Hennessy. *Computer organization and design: the hardware/software interface*. Morgan Kaufmann, 2009. 46

[qpi09] *An Introduction to the Intel®QuickPath Interconnect*, January 2009. 13

[SK10] Udo Steinberg and Bernhard Kauer. Nova: A microhypervisor-based secure virtualization architecture. *Proceedings of EuroSys 2010*, April 2010. 14

[sma] Intel smart cache technology. `www.intel.com/content/www/us/en/architecture-and-technology/intel-smart-cache.html`. 37

[Ste11] Udo Steinberg. *NOVA Microhypervisor Interface Specification*, Januar 2011. 15

[TEL98] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: maximizing on-chip parallelism. In *25 years of the international symposia on Computer architecture (selected papers)*, ISCA '98, pages 533–544, New York, NY, USA, 1998. ACM. 12

[tur] Intel turbo boost technology. `http://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html`. 39

[ZPL01] Yuanyuan Zhou, James F. Philbin, and Kai Li. The multi-queue replacement algorithm for second level buffer caches. In *In Proceedings of the 2001 USENIX Annual Technical Conference*, pages 91–104, 2001. 8