Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS

Vlastimil Babka

Influence of Resource Sharing on Performance

Department of Software Engineering
Supervisor: Doc. Ing. Petr Tůma, Dr.
Study Program: Computer Science, Software Systems

I would like to thank my supervisor, Doc. Ing. Petr Tůma, Dr., for his valuable support and advice.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

I hereby declare that I have elaborated this master thesis on my own and listed all used references. I agree with making this thesis publicly available.

Prague, August 9, 2007 Vlastimil Babka

Contents

Název práce: *Vliv sdílení prostředků na výkon*
Autor: *Vlastimil Babka*
Katedra: *Katedra softwarového inženýrství*
Vedoucí diplomové práce: *Doc. Ing. Petr Tůma, Dr.*
e-mail vedoucího: *petr.tuma@mff.cuni.cz*
Abstrakt: *Sdílení prostředků nastává v případech, kdy několik současně aktivních procesů či softwarových komponent využívá stejné systémové prostředky, což ovlivňuje výkon v porovnání s individuálním během. Izolované měření dob trvání klíčových operací pro řešení modelů predikce výkonu tudíž může přinášet nepřesné výsledky. Sdílení prostředků také nastává mezi měřeným kódem a měřící infrastrukturou, která sbírá a ukládá výsledky, což nepřímo zvyšuje její režii.*

Tato práce kvantifikuje vlivy sdílení na výkon pro několik často sdílených prostředků, jmenovitě procesorových caches a souborových systémů. Horní odhad možného ovlivnění výkonu sdílením caches je stanoven pomocí syntetických testů. Účinky na praktický kód a jejich závislosti na různých faktorech, jako frekvence a intenzita trashování cache, jsou poté změřeny pomocí experimentů s existujícími implementacemi algoritmů FFT a LZW a aplikací pro zpracování videa. Efekty sdílení souborového systému na rychlost jsou změřeny pomocí experimentů provádějících hromadný zápis a čtení z několika souborů. Za určitých okolností lze pozorovat významné dopady sdílení u každého z uvažovaných prostředků.

Na základě výsledků těchto měření je nadále navrženo několik rad pro řešení problému režie měřící infrastruktury. Také je zde diskutována použitelnost provedených experimentů a jejich výsledků pro účely modelování výkonu.
Klíčová slova: *sdílení prostředků, měření výkonu, modelování výkonu, procesorové cache, souborové systémy*

Title: *Influence of Resource Sharing on Performance*
Author: *Vlastimil Babka*
Department: *Department of Software Engineering*
Supervisor: *Doc. Ing. Petr Tůma, Dr.*
Supervisor's e-mail address: *petr.tuma@mff.cuni.cz*
Abstract: *Resource sharing occurs when multiple active processes or software components compete for system resources, which influences the observed performance compared to an individual execution. Isolated benchmarking of durations of key operations for solving of performance prediction models may therefore yield imprecise results. Resource sharing also occurs between the measured code and the benchmark infrastructure for obtaining and storing samples, imposing an indirect overhead.*

This thesis quantifies the effects of sharing on performance for several resources that are often shared, namely the processor caches and the file systems. The highest possible performance impact of cache sharing is determined by synthetic benchmarks. Impact on practical code and its dependency on a number of factors such as cache trashing frequency and intensity are then determined by experiments with existing implementations of FFT and LZW algorithms and a video stream processing application. Effects of file system sharing are measured by experiments that read and write multiple files simultaneously. For both resources, situations with significant performance impact of sharing have been observed.

Based on the results of the experiments, several suggestions for dealing with the overhead of performance monitoring infrastructure are proposed, and applicability of the experiments and their results for performance modeling is discussed.
Keywords: *resource sharing, performance evaluation, performance modeling, processor caches, file systems*

1 Introduction

Performance is one of the key properties of software systems. Not only does the code have to be correct, it should also execute reasonably fast to be of practical use. The observed performance is determined by complexity of the individual algorithms, efficiency of the code implementing them, the architecture of the whole system, and the performance of the platform the system is being deployed on.

While complexity of the algorithms can be theoretically analyzed and performance of the system architecture predicted by modeling, the usual method of determining performance of the resulting code on a given hardware is empirical, relying on benchmarking experiments. Possible goals of benchmarking include comparing speed of hardware or competing software implementations, detecting performance regressions between versions during development, or determining durations of atomic operations for performance modeling of the system being designed.

Obtaining results applicable for these purposes becomes tricky when conducted on modern hardware and operating systems. The performance observed in such a complex environment is influenced by various factors. One of them is resource sharing, which occurs when multiple processes or parts of the same process compete for the same hardware or operating system resource. This can prolong their execution. The aim of this thesis is to examine the impact of resource sharing on code performance and the influence this has on performance modeling and monitoring.

1.1 Benchmarking and Resource Sharing

The simplest way to evaluate the overall performance of an application (or the system that is executing it) is to measure the time to complete the whole task. This is suitable for applications performing relatively long tasks that require no interaction, such as file compression. In order to obtain reproducible and comparable values, the measurement is often performed on a freshly installed and booted system, and in a dedicated configuration, with practically no other interfering processes running. The experiment is repeated a number of times, discarding values obtained during the warm-up phase. The typical duration is then calculated from the remaining samples as an average, median, or using more complex approach [15].

The important question is whether the results obtained during such isolated execution really represent the performance in the target environment of the application. During the benchmark, the measured task is the only significant workload in the system, virtually no other processes can be scheduled during its work and influence its performance by resource sharing. The task can therefore take advantage of all available memory and processor caches, and has exclusive access to files or network interfaces. During the warm-up phase of the benchmark, caches and buffers are pre-filled with the task's data, and various heuristic algorithms present in modern systems can adapt to the task's resource usage patterns. This might naturally cause the task to perform better than when executed in a real environment.

This difference between the individually measured and the real performance can for example result in a wrong deployment decision. Consider two processors of the same microarchitecture where the first has a higher frequency and the second has larger caches. A task with working set that fits just in the caches of the first processor

will perform faster on that processor when measured individually, but sharing the caches with other tasks when in real use can result in faster performance on the processor with larger caches.

In many cases, only durations of relatively short operations are being measured instead of the whole application's performance. These operations are usually part of a complex software system, which cannot be measured in its entirety for various reasons. Some systems inherently cannot have a simple global performance indicator such as throughput or service round-trip, and a set of representative operations has to be chosen, for example duration of a remote method invocation or marshalling for CORBA implementations benchmarking [31], or duration of TCP Ping or HTTP Ping for MONO regression benchmarking [16].

In the case of performance modeling, the entire system cannot be measured because it is obviously not fully implemented, as there would be not much use for performance modeling otherwise. For models based on interaction of components through atomic actions, benchmarking is used to measure durations of these atomic actions in prototype component implementations in order to solve the model [17][20][21][32].

Benchmark experiments with these short operations are often performed in isolation [21], with no parallelization [32] or with a fixed degree of parallelization [20]. However, in the target environment, these operations would compete for resources not only with other processes, but also with operations performed by the same process. These operations could be executed either in different parallel threads, or in the same thread, which may influence the results. For example, if a component's function invocation is interleaved with different function invocations, its code and data may be evicted from the caches upon invocation and thus its duration would significantly differ from the duration when measured alone. Results from the performance model populated with durations obtained with none or different parallelization, or omitting code that would be interleaved with the measured durations inside a same thread, would therefore become inaccurate.

To improve the precision of performance models, the durations of the function invocations used to solve the model would have to be measured under the same conditions as in the fully implemented and running system, which is not yet available for this purpose. Another approach would be to include resource sharing into the performance model. This would however increase the complexity of the model, and could be infeasible for resources that have no precise performance models, or have a model based on a different formalism than the performance model of a software system. Because of these obstacles, resource sharing is often simply omitted [20][21][32], or mentioned as high cost [17]. However, intuition suggests that the effect of resource sharing on model precision could be significant, and this thesis should prove that.

In benchmarks that resemble whole software systems, such as transaction processing applications in TPC [29], or an auction site prototype in RUBiS [28], performance sharing should be captured well, because everything that would use resources in the real system is executed as part of the benchmark. However, we think that care should be taken when preparing environment for these benchmarks, e.g. populating the database and creating images of products in the auction site. Doing

this at once may result in less scattered and fragmented files compared to a real scenario where the data is inserted, deleted and modified over a longer period of time.

Another situation where resource sharing may influence the results of benchmarking comes from the fact that the benchmark infrastructure for obtaining and storing the results of measurements also shares resources with the measured code. The code that captures and stores timestamps in the memory may trash the code and data caches, writing large results in files may interfere with usage of files by the measured system. This problem is important especially in the case of detailed performance monitoring and code profiling, where the measured code is interrupted and results collected relatively very frequently. Due to resource sharing, the overhead of the infrastructure code is not additive – we cannot simply measure the whole task duration with and without detailed monitoring, and determine the overhead by subtracting the results.

1.2 Goals

This thesis aims to address some of the problems with benchmarking due to resource sharing described in the previous section. The first goal is to analyze what resources are most frequently shared and how this sharing can influence performance. We will consider a scale of resources from the processor caches to the file systems.

The second and most important goal is to quantify the effects of sharing empirically, using specially designed benchmark experiments. Our aim is not to determine absolute values like durations of a function invocation with and without shared cache – these values would likely be too specific to the benchmark scenario and the hardware platform executing it to be of direct use in e.g. performance models. Rather than the absolute values, we want to determine resources and situations where sharing influences performance significantly and thus should not be ignored.

Our third goal is to analyze possibilities of applying the results obtained from the experiments to remedy the problems with resource sharing in benchmarking described earlier. Namely, we will discuss possible ways to reduce or deal with the observed resource sharing effects in the overhead of performance monitoring infrastructure, and to incorporate resource sharing into performance models.

1.3 Structure of the Thesis

With respect to the outlined goals, the structure is as follows.

Chapter 2 gives an overview of resources that are often used and therefore shared in software systems. We analyze how these resources work, the expected forms of sharing, and propose methods to quantify the effects of such sharing empirically. We proceed by choosing several of the resources for further experiments.

In Chapter 3, we introduce the framework designed and implemented to measure effects of resource sharing. Then, for each benchmarked resource, we describe the design of each benchmarking experiment in detail, with both expected and obtained results and their evaluation.

Chapter 4 evaluates the impact of the resource sharing observed in the experiments and proposes several advices for reducing the benchmarking

infrastructure overhead. Related work is discussed in Chapter 5, with Chapter 6 concluding the thesis, proposing possible tasks for future work.

2 Shared Resources

Computers consist of various hardware subsystems, each providing some needed functionality, also called resource. Code executed on the computer is using these resources to perform its work, some of them inherently (processor, memory), some of them usually explicitly (disk, network). One of the roles of operating systems is to abstract these hardware resources, and allow them to be used by several processes simultaneously. To facilitate this, an operating system creates and maintains own virtual resources – file systems, network sockets, virtual memory tables and so on. Similar kind of resources can be provided by runtime libraries to the running process – heap management and garbage collection, remote procedure calls and other middleware services etc.

System resources are always limited in some way, which can cause performance impact when they are being shared. One of the limitations is the amount of the resource, e.g. memory or cache size, network bandwidth. Multiple processes can use such resources simultaneously, but once the limit is saturated, processes are swapped out of memory, get their cached data evicted, or have to share bandwidth.

Another kind of a resource limitation is the need for exclusive usage. The best example of this would be a single-core processor without hyper-threading. Multiple tasks have to be scheduled for execution. The total time to execute those tasks could be naively calculated as sum of durations of each task when executed alone. But there will be certain overhead caused by the scheduling algorithm, sharing of processor caches and other reasons.

On modern hardware and operating systems, high performance is achieved not only by high clock rates, but also via sophisticated optimizations such as caching and branch prediction in the processor, buffer cache for files, often with read-ahead prefetching and delayed writing, and other heuristic algorithms. Because these optimizations dynamically adapt to the execution patterns of the running task, frequent switching between multiple tasks can render them to perform less optimally.

The next sections discuss several selected resources with details on their usage, expected behavior under sharing, and basic methods that could be used to measure effects of sharing. The resources are roughly sorted by their position on an imaginary hierarchy building from the physical resources of hardware up to the virtual resources of the operating system and applications.

2.1 Processor Memory Caches

A CPU Cache is a memory buffer located between the CPU and the system memory, used to achieve smaller average latency or memory accesses. This is increasingly important in modern systems, where memory access is often a performance bottleneck – many processor cycles are wasted (stalled) when waiting for memory transfers. Caches are much faster and closer to the CPU (often on-die) than the system memory and therefore have significantly lower latency. The downside is that caches are expensive, which limits their possible size. Because of this, a cache can store only small portion of system memory, divided to cache entries (also called lines) tagged with the memory addresses of the stored data. A memory access that requests data present in the cache is called a cache hit. A request for data not

currently present in the cache is called a cache miss. In this case a whole block of data the size of cache line, which contains the requested address, is fetched from the system memory and stored in a selected cache line, replacing (also called evicting or trashing) the victim data currently stored there. A common method is to keep least recently used entries, assuming the executed code will be accessing these cache lines again relatively soon.

To determine if the requested data is in the cache, all entries that can contain the requested address have to have their tag compared to the address. Since comparing many tags would need too expensive hardware to be done at once, or would be too slow when done sequentially, there may be only limited amount of cache entries that each memory address can map to, using some form of hash function applied to the virtual or physical memory address. This is called associativity [12]. Fully associative caches can map each memory address to any cache entry, while direct mapped caches can store each memory address in exactly one cache entry. Set associative caches have their entries grouped to sets with a fixed number of entries (called ways) per one set. Each memory address maps to exactly one set, but the data can be stored in any of the ways inside the set. Current processor caches are mostly 2-way, 4-way, sometimes 16-way set associative. Cache replacement algorithms, often approximation of LRU, are used to choose the way inside a set that will be replaced with the newly fetched data.

There are usually more caches than one located between the CPU core and the memory, organized in hierarchy of 2-3 levels, with smaller and faster caches closer to the CPU and larger but slower caches closer to the main memory. The first level usually consists of separate code cache and data cache. The code cache is sometimes implemented as a trace cache [27] that stores decoded instructions rather than unmodified copy of the fetched memory. The level 2 (and higher) cache is unified and can therefore contain both code and data. In the case of miss in the L1 code or data cache, code or data is fetched from the L2 cache (or from the system memory if it misses also in the L2 cache) and stored in L1, which thus duplicates subset of L2 cache the same way L2 cache duplicates subset of the main memory (inclusive design). Recent AMD processors use different, exclusive approach, where the code or data can be stored only in at most one cache. When a cache line is being fetched from the L2 cache or the system memory, the entry that is replaced in the L1 cache is transferred back to the L2 cache. In both cases, the transfer from the L2 cache back to the system memory needs to be done when the copy of the data in the caches has been modified.

As stated above, caches improve performance by lowering average latency of memory access. The efficiency of caches depends on the amount of cache hits and misses in each level of the hierarchy – best performance is gained when all memory accesses result in the L1 cache hit, worst performance occurs when all data has to be fetched from main memory as a result of the L2 cache miss. For a code that is executed alone, cache misses can be divided into three groups (paraphrasing [12]):

- compulsory misses – occur when a cache line is accessed for the first time and therefore cannot be cached yet
- capacity misses – caused by cache size being too small for the memory footprint, even if it was direct mapped

- conflict misses – caused by the limited associativity which maps more addresses to one set than fit the number of ways in the set

When multiple tasks are executed simultaneously on the same processor and thus sharing its caches, each task can evict cache entries of other tasks by replacing them with its own code or data, resulting in increased number of capacity or conflict misses. This effect, called cache trashing, can obviously occur when the processor has hyper-threading or multiple cores sharing the same caches and a number of threads is thus executed truly in parallel. The effect of cache trashing on hyper-threading has been studied for example in [13].

With respect to the goals of this thesis, we will focus on a different scenario, without true parallelism, where trashing is caused by task switching due to process scheduling or switching of different operations inside one process or more cooperating processes. To measure effects of such trashing, we need to design benchmarking experiments that would interleave execution of some measured workload with a procedure that will artificially trash given amount of data or code from the caches. By varying the size and frequency of trashing and comparing the time needed to execute the measured code, we can quantify performance impact of cache sharing on the particular code.

2.2 Processor Translation Caches

A page-translation cache (also called translation-lookaside buffer, or TLB for short) is an important unit in processors using virtual memory addressing. On such architectures, the address of each instruction or data reference needs to be translated to physical addresses in order to be accessed in the system memory. A TLB is an associative cache, storing the virtual to physical memory mapping for a fixed number of (usually least recently used) memory pages.

Successful queries to the TLB (called hits) are considerably faster than actions that need to be performed if the address is not found there (a TLB miss). The missed mapping is then either looked up in the page table hierarchy automatically by the hardware (the faster case, used for example by processors of the Intel IA-32 architecture), or need to be handled by the operating system (used for example in the MIPS family of processors). The result of the lookup is then stored in the TLB and the instruction that referenced the memory address can continue to be executed, but with significant delay.

The effect of TLB sharing should be roughly similar to the memory caches, because the size of TLB is also limited and the executed tasks compete for them. Additionally, on some architectures including IA-32, each context (address space) switch results in TLB being completely emptied, except for entries set as global by (and practically reserved only for) the operating system. This needs to be done, because different processes have different mapping and the TLB on these architectures is not designed to keep any address space identification with the mappings.

The experiments for quantifying the effect of TLB sharing will be also similar to the memory caches, interleaving the measured workload with code that will artificially replace the TLB entries by accessing different memory pages.

2.3 System Memory

The code and data of all running tasks, including the operating system, share the system memory. If the memory gets filled up, the system uses secondary storage (usually hard disk drive) to swap out data that was not recently used, and fetches it back in when it is needed again. This process is very slow compared to memory access – much slower than system memory latency compared to the processor caches. If swapping occurs frequently because the working set of the executed code does not fit, performance is reduced drastically, which is called thrashing. Benchmarking this scenario thoroughly is not so interesting, because practically it is something highly undesirable and avoided as much as possible.

Thus, there are not practically only two possible states. The first state is when everything fit in the memory and runs optimally, and the second with frequent trashing. Performance prediction can thus simply compute sum of memory occupation of all components and compare it with the available memory size.

Since the system memory is also used to buffer data from files, the amount of available memory can also affect the performance of file operations. This will be discussed in the section about file systems.

2.4 Heap Management

Most applications that allocate objects of varying sizes dynamically in memory use heap for managing the allocations and deallocations. Sharing the heap increases its size and therefore possibly increases the duration of the heap operations. Dynamic allocation also causes some memory fragmentation, which could potentially increase through sharing because the multiple components sharing the heap can increase the diversity in size of the allocated objects. This may result in the combined memory usage to be higher than the sum of usages of individual components. There are two types of heap which we will discuss separately. The first type requires explicit object deallocation, the second type uses garbage collection to detect and free the unused objects automatically.

Because contemporary allocators with explicit deallocation have close to O(1) complexity and low fragmentation [19], the performance of the heap operations should be significantly affected neither by the heap size nor by the number of components. We thus probably do not need to focus on this type of allocation.

For allocators with garbage collection, memory allocation costs nothing if the collector is compacting, which also eliminates fragmentation. Because deallocation is not done explicitly, performance is affected only by the execution of garbage collection. This is performed asynchronously, usually when the heap size limit is reached, and the total performance impact depends on the frequency and duration of the collector runs. Note that the impact can be both direct – running the garbage collections consumes CPU cycles, and indirect – due to the cache sharing affecting performance of the interrupted code.

The performance impact of garbage collection has been studied in [18]. The indirect impact has been found to be negligible. On the other hand, the direct impact can be significant and cause over 100% slowdown in the worst case. The authors have also observed that the impact depends mostly on the frequency of garbage collection runs and less on the heap size. The duration of individual garbage collections grows roughly linearly with heap size and with the number of objects, but

the decreased frequency thanks to larger heap size reduces the total cost garbage collection significantly.

We can therefore conclude that it is important to include the garbage collection overhead during benchmarks. Individual benchmarks of components using large enough heap size may complete without any collection performed, tampering with the precision of such results for performance prediction. A possible way to include the overhead would be to limit the heap size to reflect the portion of memory available to the component when running in the target environment, which would result in the frequency to be similar. After the overhead for individual components is known, the combined overhead could be determined easily because of the linear dependency of the duration of the garbage collection on the heap size and the number of objects.

2.5 File systems

Files, organized in file systems, are a frequently used resource for storing permanent data used by the applications. In contrast to the compact resources described earlier, files are a complex abstract resource provided by the operating system, which is backed by many individual but related resources. These resources are described below together with expected effects of sharing.

Directory structure. File systems are traditionally organized into trees of directories. Each directory may contain additional subdirectories and files, together called directory entries. A large number of entries in a directory may affect file system performance when working with files in that directory, which therefore can be considered a shared resource. This effect depends on the data structures that the particular file system uses to represent the directory, which may for large directories influence the performance by a factor of 50 - 100 [6].

System memory. Another resource occupied by files is the system memory. Each opened directory or file is represented by a kernel structure allocated in memory in order to hold its metadata. Free memory (not occupied by applications) is also used for *buffer cache* that stores the file data that was recently read or written in order to take advantage of data locality. The operating system may also detect sequential reading and perform a read ahead to prefetch data that is likely to be used in the near future. Similarly, writes can be delayed, the data accumulated in the buffer cache and later flushed to the disk in larger blocks.

Naturally, more memory available to buffer the data of a file can result in better performance, and working with several files simultaneously can reduce the amount of memory available for each file. This amount is also limited by the memory occupied by running applications, which makes this kind of resource sharing even more complex. Many buffer cache replacement algorithms exist to decide what pages to evict from the memory when a free page is needed. One of the most known is the WSCLOCK [7], an approximation of LRU.

Note that too aggressive buffer cache may even cause the application code and data to be swapped out and result in thrashing [23]. Another interaction exists between the prefetching and buffer cache replacement efficiency [5].

Hard drives. Files are typically stored on a hard disk, which is a very slow resource compared to the processor and main memory. It has its own hardware

buffer, which works similarly to the buffers in main memory (performing read-ahead and delayed write), but has much smaller size (typically 8 – 16 MB). Hard drives have limited read and write speeds, typically tens of megabytes per second, thus simultaneous read or write requests may get queued and the processes performing them are blocked unless they can perform other tasks in the meantime. The main bottleneck of hard drives is the latency of seek, which is the time needed for the disk to position its head to the cylinder containing the requested sector (few milliseconds for contemporary drives). This time depends on the distance between the current and the desired head position. Because file systems try to organize file data in a minimal number of continuous fragments, sequential reading or writing of one file is not affected much by seek latency. But random accesses to a large file which spans many cylinders, or simultaneous accesses to a number of files result in significant seek latencies, affecting performance greatly.

Creating continuous files is easiest when the file is being written alone and the disk has enough space. However, writing more files simultaneously can result in the fragmentation of the data. The amount of fragmentation also depends on the particular file system implementation and amount of free space. File fragmentation can have residual effect on performance – future reading of fragmented files can result in more seeking, even if the reading is sequential and one file at a time.

2.6 Summary

We have discussed some of the most shared resources and the expected causes of performance impact due to sharing. We will now select some of the resources for benchmarking experiments that will quantify the expected sharing effects empirically:

- Processor caches – Both translation and memory caches promise significant sharing effects and their sharing is inevitable because all code uses them for their operations. To quantify the effects of sharing, we will interleave the execution of measured workload with code that will artificially trash the caches by accessing or executing data or code located in a different memory region that the working set of the measured code.
- File systems – Working with multiple files simultaneously should yield significant slowdown compared to working with one file, because it increases the seek rate and the latency of a seek is relatively high. To determine the slowdown, we will interleave the reads or writes of a file with the same operations on different files and compare the measured duration with duration of operations performed individually. Files used for reading will be also created either by separated or interleaved writes, to measure the residual effect of file fragmentation.

The resources we will not experiment with and the reasons for this decision are listed below:

- System memory – The effects of exceeding the available memory with working sets of active operations are already known to degrade performance drastically and are avoided at all costs. Prediction of memory usage of the whole system

from the memory occupation of individual components is simple, even if pessimistic.

- Heap management – Heap operations in heaps without garbage collection should not exhibit resource sharing effects. The overhead of garbage collection has been already studied and found to be significant for some workloads, but we believe that predicting overhead of the whole system from the overhead of individual components should be simple.

3 Benchmarking experiments

In this chapter, we will design and execute experiments to quantify effects of sharing on the resources we have selected in the previous section – processor caches and files. To achieve this, most of our experiments will be interleaving the execution of two separate operations – the first is the measured workload, the second is artificially trashing the given resource. Both will have parameters, affecting their use of the resource. Benchmarking with different combinations of these parameters would show their effects in detail.

3.1 The benchmarking framework

For the purpose of our experiments, a simple benchmarking framework called RIB has been designed and implemented in C++ and Linux environment. Individual benchmarks incorporated to this framework have to provide unified description of their parameters and interface to prepare and invoke their operation. Some of the benchmarks were implemented specifically for the purpose of the experiments, and a number of existing implementations of standard algorithms was incorporated via wrappers providing the needed interfaces.

3.1.1 Mode of operation

The operation of the framework is controlled by a experiment description file which specifies two workloads to be run (the second can be specified as none) as well as values for their parameters, which can be a single value, set of values, or a range of values with linear or exponential step. There are also two general parameters, determining the number of repeat in a single run, and number of values to discard during the warm-up phase.

After the framework is executed, it will perform the measurements with all possible combinations of the parameters' values. This means that for each combination of the values chosen from supplied sets or ranges, the measurement is performed repeatedly as many times as specified, storing the results in a pre-allocated memory array. Then the next combination of parameters is chosen and executed. When all combinations are exhausted, the framework flushes out the obtained samples to the results file and exits. Further repeated executions of the framework append their new samples.

Because performance is affected by the initial random state upon execution [15], we will obtain relatively low number of samples per run, repeatedly executing the benchmark many times to reduce the initial state impact. This is especially important for the cache experiments, where the virtual to physical address mapping, performed by the operating system in a non-deterministic way, can affect the number of cache misses greatly.

3.1.2 Time measurement

All experiments we will perform need to measure time to complete some operation. Because this operation can be very short (only few microseconds), we need a precise timing method. Although it is usually possible to measure with less precise timing by

executing the operation multiple times, and calculating the average, we cannot use this approach here. Because we interleave the measured workload with trashing which has no fixed duration (the influence of cache sharing is mutual), we would not be able to separate the duration of trashing from the whole duration precisely. Also, average timing involves a loss of potentially interesting information.

The precision offered by the operating system calls is not enough. Linux increments its internal clock by the value of a so-called jiffy. On the IA-32 platform, this value is determined by the frequency of timer interrupts, which defaults to 100 and can be raised to 1000, which corresponds with a milisecond precision. This determines the precision of the *times(2)* syscall, which has the advantage of accounting time per process and distinguishing user and kernel mode. The *gettimeofday(2)* syscall provides only real-time timing, with theoretical microsecond precision. However, the real precision is also based on jiffies, and may be extended by interpolation. The actual implementation varies with Linux kernel versions and the presence of optional hardware such as HPET timers.

Fortunately, modern IA-32 compatible processors come with an elegant solution to this problem, providing own timestamp counter (TSC). This is a special register incremented with each CPU clock cycle, thus giving the ultimate precision. The measured values can be divided by the processor frequency to obtain the real time duration. Only care must be taken so that the BIOS or operating system is not configured to dynamically adjust the frequency on demand. The TSC is easily accessible by a special instruction RDTSC, which can be executed in user mode and stores the result into general registers. Thus, in our experiments we will use RDTSC timing wherever possible.

3.1.3 Performance counters

Aside from measuring time, we can also take advantage of several more performance counters implemented in modern processors. These counters can be used to collect various events inside the logical units of the CPU, such as floating-point operations, registers usage, memory access details, branch prediction efficiency etc. The set of available counters is different for each processor family and even revision. Obviously, the most interesting for our experiments are the counters related to the TLB and the caches, in particular number of hits and misses in the TLB, the L1 data cache, the L1 code cache and the L2 cache.

Unfortunately, using these counters is not as simple as executing the RDTSC instruction to obtain the value of timestamp counter. The processor does not collect all of the supported events automatically, but there is a limit on the number of events that can be collected simultaneously, and these events have to be selected beforehand. Additionally, some combinations of events can be impossible. The instructions for setting the events and reading of the results can only be executed in kernel mode and thus needs special syscalls to be used from user mode. Although the Linux kernel does not have such support natively, it can be added via the Perfctr [26] patch. The PAPI library [25] provides a portable user interface to performance counters, using platform-specific backends (e.g. Perftctr on Linux and Intel compatible processors).

PAPI can work with two kinds of performance events, native and preset. Native events are directly provided by the processor, and there is a utility to detect which

events are available. Preset events is a set of predefined events that should be supported on most of processors, either mapped directly to some native event, or calculated using two or more different native events. For example, the number of the L1 data cache misses events can be calculated by subtracting the number of the L2 data cache misses from the number of total data cache misses. This means the limit of events counted simultaneously can be lower when using preset events.

Additionally, during first experiments with PAPI we have found that some preset events give clearly wrong results on the Athlon 64 processor - PAPI was not mapping them correctly to native events. The actual mapping also is not documented. Thus, we will use only native events. Care must be taken to read the events description in the processor documentation properly, to interpret the results correctly. There can be subtle details, like hardware prefetching and speculative accesses resulting in extra cache hits or misses that can distort the results.

Because the calls to the PAPI functions are not as trivial as the RDTSC instruction, they cost more CPU cycles and could potentially trash some code or memory cache. We will measure the overhead experimentally.

3.1.4 Hardware platforms and experiment template

The rest of this chapter presents in detail each of the performed benchmark experiments for given shared resource. The platforms used to run the experiments are described below.

Platform 1: AMD Athlon 64 3000+ Venice DH7-CG 1.8 Ghz with exclusive 64 KB data L1, 64 KB code L1, 512 KB unified L2 caches; 2×512 MB dual-channel DDR RAM; Hitachi Deskstar T7K250 250 GB, 7200 rpm, 8 MB buffer, SATA-2 without NCQ; Gentoo Linux with 2.6 kernel, gcc 3.4, ext3 file system

Platform 2: Intel Pentium 4 Northwood 2.2 Ghz wich inclusive 8 KB data L1, 12 Kops L1 trace cache, 512 KB unified L2 cache; 512 MB RAM; Hitachi Deskstar T7K250 250 GB, 7200 rpm, 8 MB buffer, ATA UDMA5; Fedora Core 6 with 2.6 Linux kernel, gcc 4.1, ext3 file system

Platform 3: Dual CPU Intel Pentium 4 Xeon 2.2 GHz HyperThreading[1]; Maxtor DiamondMax Plus D740X 80GB; 7200 rpm, 2 MB buffer, ATA UDMA5; Fedora Core 5; ext3 file system

[1] Only one CPU was enabled and hyper-threading disabled for the experiments; the platform was used in file system benchmarks, the processor cache sizes are thus not important.

For each experiment, we first explain why is the benchmark (or set of benchmarks) performed, what is the measured workload and what parameters it has, and what we expect to learn from the results. Then we use the following template to present the experiment details and results:

Experiment: An identifier of the experiment for reference.
Purpose: The reason for the experiment in short.
Platforms: Where is the benchmark performed.
Measured: Short description of the code which duration is measured.
Parameters: Used values of parameters that the measured code has.
Expected results: What should the results generally show, and why, based on the theoretical knowledge.
Actual results: An analysis of the observed results. If different from the expected results, possible explanation of why that happened and how to prove or refute it with another experiment.

3.2 Processor caches

The benchmarking experiments for processor caches will first examine the cache sizes and latencies of the processors used in the tested platforms, and then measure the effects of cache sharing on several types of workloads. For this, many of the experiments use a special procedure described below, to artificially trash the caches. This operation is interleaved with the measured code. Such experiments include the following line in the experiment template described in Section 3.1.4:

Trashing: Used values for the parameters of trashing procedure.

3.2.1 Trashing procedure

The code used for trashing the caches works by accessing addresses in a previously allocated area of memory, which fetches them to the caches, potentially trashing cached data or code of the measured workload. The amount of evicted cache lines depends obviously on the cache size, the number of lines in the cache used by measured code and amount of memory accessed by the trashing procedure. Due to the limited cache associativity described in Section 2.1, the actual memory addresses of the cache lines of both measured code and trashing procedure also play an important role, affecting the number of addresses accessed by trashing that actually conflict with the addresses used by the measured code. If we wanted to control the exact number of these conflicts and thus the exact number of lines evicted by trashing, we would need the trashing code to access memory addresses in a specific patterns, which would be hard for following reasons:

- We would have to know the exact memory access pattern of the measured code for each set of its parameters.
- Because the code works with virtual addresses and some caches use physical addresses for mapping to index, we would need to know mapping between physical and virtual addresses created by the kernel.
- We would need to know the implementation details of the cache precisely, namely the hash function used to map memory addresses to indexes and the replacement policy.

This information could be infeasible to obtain and it would be only useful for the given instance of measured code and the CPU running the experiment. Another option is to give up on the exact knowledge of lines evicted by trashing, and design the trashing procedure to be as much independent on the benchmarked code and the processor cache implementation details, as possible. This way, it could be easily used anywhere. We assume that the memory addresses that would trash the caches in a real scenario are independent of the addresses used by the benchmarked code. Under these circumstances, trashing random addresses in random order should yield the best results. We do not know how many lines occupied by the measured code are actually evicted by the given number of lines we access during trashing, but the randomness should result in low distortion. To achieve this, the trashing function has three parameters:

- Mode of trashing. Code or data, depending on which caches we want to trash.

- Memory range allocated for trashing. We cannot simply access arbitrary memory addresses, the memory needs to be allocated first, as a continuous virtual memory area. Should be large enough to cover all cache entries (through virtual to physical memory mapping and memory address to cache index mapping) so that accessing all addresses in the area trashes the caches completely. The very minimum is thus the cache size. It will be determined experimentally.
- Memory access granularity. Because an access of one address will trash the whole cache line, there is no reason to access more addresses inside the line – it would only make the trashing unnecessary slower. Thus, the granularity is best set to the cache line size, which can be obtained from the processor specification, or determined experimentally.
- The amount of accessed memory. Must not exceed the allocated memory range, and must be a multiple of access granularity (this value divided by the granularity yields the number of memory addresses the trashing function will access).

The data trashing procedure then works as follows. Before the actual measurements, the memory range for trashing is allocated. Also created is a temporary array of pointers to this area, initialized linearly with the granularity setting determining the distance between the target address of two adjacent pointers. The array is then randomly shuffled. Number of pointers (corresponding with the amount of memory we want to access) from the beginning of the array is then copied to the allocated memory range, so that to the *n+1*-th pointer in the array is copied to the address given by *n*-th pointer. The address of the first pointer is stored separately and the resulting linked list is terminated by a *NULL* pointer. The trashing procedure thus simply walks through this list. The accessed addresses are randomly distributed inside the allocated memory range, and each address is read at most once. Code trashing is almost the same. Instead of the linked list, a chain of instructions that just load and jump to the address of the next instruction, is created and executed as a procedure – the last instruction in the chain is a *RET* instruction. *NOP* instructions are used for padding.

3.2.2 Benchmarking framework overhead

When we study the effect of resource sharing, we should keep in mind that the benchmarking framework is also using some resources - processor, memory, files to store the results, thus potentially affecting the performance of measured code. Because in the RIB framework, we flush the results to file after all samples are obtained, here we are concerned only about the use of processor and memory, which can have impact on the processor caches, adding some extra trashing influencing the measured code indirectly. This effect can be minimized by repeating the measured code multiple times and computing average duration, but this is not possible if we want to interleave the code with trashing. We will therefore assume that our code, which calls *RDTSC* before and after the measured code and stores the difference in a memory array, should not cause significant impact on the measured code.

In addition to the abovementioned indirect impact, a direct overhead of the time or performance counter measurement may obviously exist. This overhead at first seems to be relatively easy to determine by measuring an empty operation – if the observed duration or performance counter value is non-zero, it can be subtracted from all future measurements to eliminate this overhead. However since this “overhead code”

itself uses resources, it might not just affect the measured code, but also be mutually affected by the measured code itself, and subtracting the results obtained individually may not be precise. We will quantify these effects by experiments for both RDTSC and performance counters, comparing the values obtained with no cache trashing and full trashing of code, data, or both caches. Note that we have to perform the cache trashing outside of the time or performance counter measurement so that we can measure an empty operation, but the real overhead caused by the code executed inside the measurement might be slightly higher.

Experiment: Overhead.1
Purpose: Determine the overhead of the RDTSC time measurement and its dependency on code and data trashing.
Platforms: 1, 2
Measured: The difference in values obtained from two immediately following RDTSC calls.
Trashing: Mode: code and data; allocated: 2 MB; accessed: none, 2 MB; granularity: 64 B.
Expected results: Because the code between the two RDTSC calls just moves the result from two 32-bit registers into the memory by two instructions, it is accessing just one or two cache lines of stack, thus it should be affected by data cache trashing only when this line is evicted. Chance of hitting this line with trashing grows with the amount of trashing. Similarly for instruction cache - because the whole code fits into one line, at worst two cache lines, it should be affected only if this line or lines get evicted.
Actual results: The difference in the number of clocks observed with no cache trashing on Platform 1 is 5. Code trashing prolongs the time to 8 clocks, while data trashing has no effect. On Platform 2, the results are always 84 clocks, not affected by any cache trashing. Possible explanation is that RDTSC on this processor is serializing, thus the second RDTSC instruction waits for the first one to retire before it is executed. This waiting probably masks any effect of cache trashing on the following instructions.

Experiment: Overhead.2
Purpose: Determine the overhead of performance counters, duration of the calls, which start and stop the counters, and their dependency on the code and data cache trashing as well as on the number of events counted simultaneously.
Platforms: 1, 2
Measured: The duration of *PAPI_start()* and *PAPI_stop()* calls, which are used to start the counting of previously selected performance events, and to stop the counters and return their results, respectively. Also measured are values of various available cache and TLB related events, obtained by starting and immediately stopping the counters.
Parameters: Number of simultaneously counted events: 1 – 4.
Trashing: Mode: code and data; allocated: 2 MB; accessed: none, 2 MB; granularity: 64 B.
Expected results: Unlike the simple RDTSC instruction, PAPI calls are much more

complex – they need to switch into the kernel mode in order to program or read the performance counters, accessing more cache lines and TLB entries in the process. Thus, we expect the duration of the *PAPI_start()* and *PAPI_stop()* calls to be much more than just few clocks, and more greatly affected by the cache trashing. Number of events counted simultaneously should also affect the duration. The results obtained from the counters might be non-zero and also affected by the trashing.

Actual results: The results are presented in Table 3.1 for Platform 1 and Table 3.2 for Platform 2. Values obtained with no trashing are in the upper part of a row and values obtained with full data and code trashing in the lower part of a row. We can see a slight increase in the values of the counters and the duration of PAPI operations as the number of simultaneously counted events increases. The trashing has significant impact especially on code misses in both the L1 code cache and the L2 cache, and on the L2 cache accesses (caused probably by the L1 data cache misses) on Platform 2. We should therefore consider the overhead when using these counters on a relatively short code. The duration of PAPI operations does not concern us, as we query the timestamp counter inside the *PAPI_start()* and *PAPI_stop()* calls.

	number of simultaneously counted events			
	1	**2**	**3**	**4**
PAPI_start() [cycles]	3551 ± 175	3598 ± 88	3625 ± 73	3730 ± 1817
	19355 ± 972	19670 ± 862	19694 ± 960	19619 ± 1961
PAPI_stop() [cycles]	2368 ± 174	2427 ± 112	2584 ± 130	2666 ± 230
	5814 ± 516	6192 ± 355	6408 ± 503	6562 ± 344
L1 data misses	1.1 ± 0.5	1.0 ± 0.6	1.5 ± 0.9	2.3 ± 0.9
	6.1 ± 1.8	6.1 ± 1.9	6.8 ± 2.1	8.1 ± 1.8
L1 code misses	0.0 ± 0.2	0.0 ± 0.2	0.0 ± 0.2	0.0 ± 0.1
	24.5 ± 1.0	24.5 ± 1.7	25.0 ± 1.4	25.3 ± 0.8
L2 data misses	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
	3.6 ± 1.5	3.6 ± 0.9	3.6 ± 1.4	3.6 ± 1.3
L2 code misses	0.0 ± 0.1	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
	22.7 ± 0.9	22.7 ± 1.3	23.1 ± 0.9	22.9 ± 1.0
L1 DTLB misses	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
	0.0 ± 0.1	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
L1 ITLB misses	0.0 ± 0.3	0.0 ± 0.2	0.0 ± 0.3	0.0 ± 0.2
	0.0 ± 0.2	0.0 ± 0.3	0.0 ± 0.2	0.0 ± 0.2
L2 DTLB misses	0.0 ± 0.1	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
	1.0 ± 0.1	1.0 ± 0.1	1.0 ± 0.1	1.0 ± 0.1
L2 ITLB misses	0.0 ± 0.2	0.0 ± 0.0	0.0 ± 0.2	0.0 ± 0.3
	2.0 ± 0.0	2.0 ± 0.0	2.0 ± 0.0	2.0 ± 0.0

Table 3.1: *The overhead of performance counters on Platform 1 with no trashing (upper values) and full code and data trashing (lower values)*

	number of simultaneously counted events			
	1	**2**	**3**	**4**
PAPI_start() [cycles]	9637 ± 2688	10209 ± 2348	11241 ± 6212	11619 ± 1381
	36281 ± 3167	36976 ± 3573	38106 ± 4151	38589 ± 5251
PAPI_stop() [cycles]	5868 ± 627	6138 ± 5222	6233 ± 2187	6428 ± 964
	12187 ± 2181	12619 ± 1907	13137 ± 1944	13069 ± 2839
L1 code misses	1.3 ± 1.0	1.5 ± 1.0	1.2 ± 1.2	1.1 ± 0.9
	5.6 ± 1.3	5.7 ± 1.7	6.7 ± 1.6	6.9 ± 1.5
L2 accesses	20.6 ± 5.2	21.8 ± 6.3	23.3 ± 10.4	23.9 ± 10.2
	98.7 ± 9.4	97.7 ± 11.3	99.8 ± 13.4	99.5 ± 12.4
L2 misses	0.0 ± 0.4	0.0 ± 0.3	0.0 ± 1.1	0.0 ± 0.6
	23.5 ± 3.5	23.5 ± 3.8	23.4 ± 3.8	23.4 ± 4.4
DTLB misses	0.0 ± 0.2	0.0 ± 0.2	0.0 ± 0.4	0.0 ± 0.2
	1.0 ± 0.6	1.0 ± 0.7	1.0 ± 0.7	1.0 ± 0.8
ITLB misses	0.0 ± 0.8	0.1 ± 1.2	0.0 ± 1.0	0.0 ± 0.8
	2.4 ± 0.6	2.4 ± 0.7	2.5 ± 0.9	2.3 ± 0.7

Table 3.2: *The overhead of performance counters on Platform 2 with no trashing (upper values) and full code and data trashing (lower values)*

Note that while we measured the duration of whole PAPI calls, the counters are activated only at some point during the *PAPI_start()* call, not immediately upon the invocation. Similarly in *PAPI_stop()*, the counters are stopped possibly long before the call returns, because it stores the results. Thus, we will not be able to see all the events generated during the calls. While this reduces the number of events recorded but not generated by the measured code, it is still possible that these hidden events (for example cache misses) affect the performance of the measured code. That is why we will primarily measure only time and employ performance counters only when needed, checking whether the duration obtained with the counters does not significantly

3.2.3 Artificial benchmarks

Before measuring the effect of cache trashing on practical code, we will perform several benchmarks with artificial code. Because these algorithms are designed to simply access memory addresses in certain parameterized patterns, the results should be as we expect them, based on theoretical knowledge about caches. These experiments should also prove that the framework works, and help to find out or confirm the declared parameters (capacity, line size) of the caches, and to determine the static parameters for trashing operation, as described in the previous section.

In the first experiment, we will measure the effects of sharing the TLB by accessing a number of memory pages by reading one address inside each page. To avoid using extra memory that would interfere with the TLB, the data we read contains a pointer to the next page, forming a circular list. We walk through this list hundred times and calculate average duration per one iteration to reduce overhead of the measurement. In order to avoid or minimize the number of data cache misses, we determine the offsets of cache lines inside a page (from the cache line size), and assign one of the offsets to each page, distributing them evenly. This minimizes the number of accesses to cache lines that have the same index and thus the number of

conflict misses. We will also use the results of performance counters to verify the number of misses in TLB and caches.

Experiment: TLB.1
Purpose: Determine the size of the TLB and the latency of a TLB miss.
Platforms: 1, 2
Measured: Time to access number of memory addresses from different memory pages (4 KB large), as described above.
Parameters: Pages allocated: 1024; pages accessed: 0 – 1024
Expected results: The number of CPU cycles per one access should be constant as long as the number of addresses fits in the TLB, then raise to a higher value and remain constant again The length of the transition will depend on how close to the ideal LRU the TLB replacement algorithm is. The Athlon 64 processor used in Platform 1 has two levels of TLB, so we should see two such raises.

Actual results: The results from Platform 1 are shown in Figure 3.1 as a graph of CPU cycles per page access. There is a clear jump from 3 to 8 cycles per access when the number of accessed pages reaches 33, which matches the number of declared L1 DTLB entries (32) for the Athlon 64 used. Performance counter for the *L1 DTLB Miss and L2 DTLB hit* events confirms that the slowdown is indeed caused by these events. We can therefore conclude that the impact of L1 DTLB miss is 5 clocks, and the replacement algorithm does perfect LRU for a FIFO access pattern that our algorithm produces.

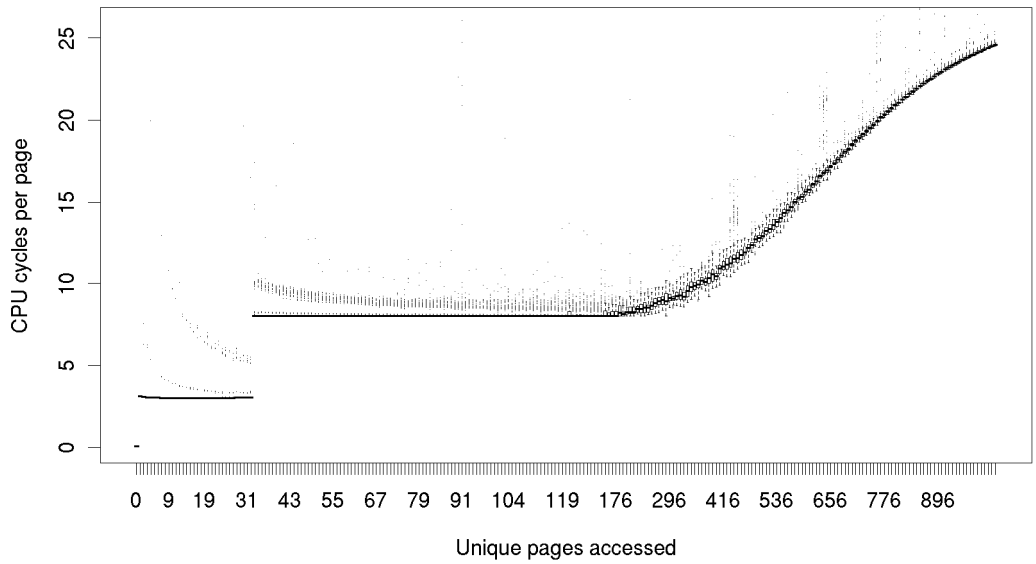

Figure 3.1: *The effect of TLB trashing on Platform 1*

For accesses that miss both in L1 and L2 DTLB, the situation is a bit more complex. We can see a gradual increase in duration after the number of accessed pages reaches approximately 168, while the declared number of entries is 512. Performance counter for the *L1 DTLB Miss and L2 DTLB miss* event shows similar

behavior, increasing from zero after the same number of accessed pages. The L1 cache miss counter remains zero, which means that the cache misses are not influencing the results. We can thus compute the impact of L2 DTLB miss by dividing the number of cycles added by the misses (which means subtracting 8 from the total number of cycles per access) with the L2 DTLB miss value where it is non-zero. The resulting average latency is 20.5 ± 2.1 cycles in addition to the L1 DTLB latency. Note that the counters have also shown that the processor is caching the translation tables in the L2 cache, but not in the L1 cache – the number of the TLB reload requests in the L2 cache is exactly the same as the number of the L2 DTLB misses (it would be lower were the tables cached in L1) and the counter for victim transfer from L1 to L2 is always zero, which means nothing is replacing entries in the L1 caches.

The results from Platform 2, which has only one level of DTLB, are shown in Figure 3.2 and are similar to the L2 DTLB results from Platform 1 in the aspect of gradual increase which occurs after the number of accessed pages reaches 27 (the declared size is 64 entries). The performance counter for the TLB miss event confirms this increase, and the performance counter for the L2 cache access event shows that the increase is not caused by the L1 data cache misses (the Pentium 4 processor in Platform 2 provides no separate counters for the L1 data cache). The computed average number of cycles per access with no miss is 2.5 and the calculated TLB miss latency is approximately 60 cycles.

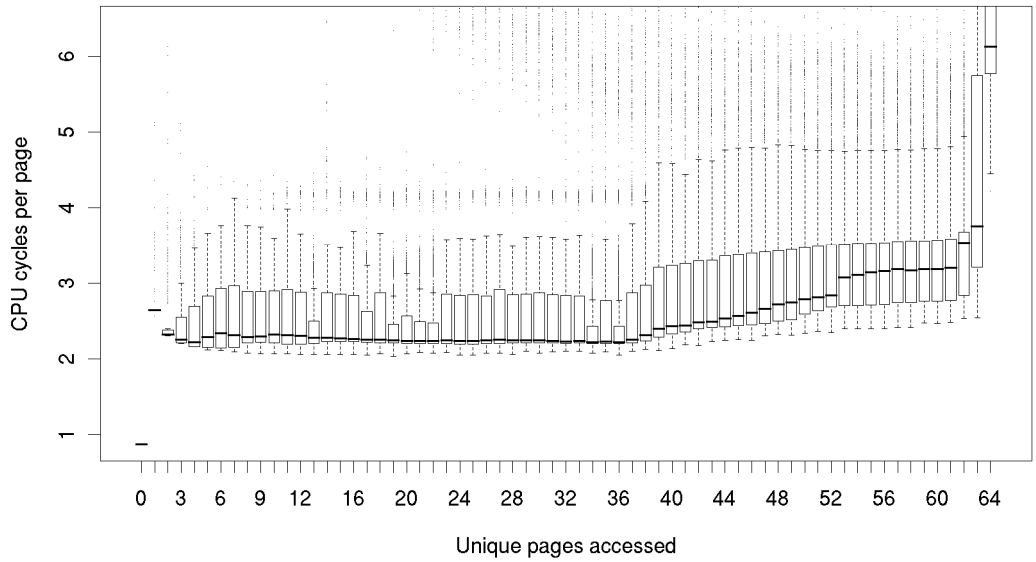

Figure 3.2: *The effect of TLB trashing on Platform 2*

Although the impact of TLB miss on a single memory access may be significant, especially on Platform 2, we believe that it is not so important to determine its impact separately from the cache trashing. With small working sets and high locality, TLB miss would occur only on the first access to a page and its relative impact would get lower with each subsequent access to the same page. With large working set and low locality, the L1 data cache and the L2 cache are likely to be under similar pressure, and this is very unlikely to cause more TLB misses than cache misses. This

is why we will not measure the effect of TLB trashing separately in the later experiments with real-world code examples. The TLB effect will be a part of the effect caused by our cache trashing.

The following experiments will determine the cache line used in the processor, the code L1, data L1 and L2 sizes and latencies and the memory range we need to allocate for the trashing so that it can trash the caches completely when the whole range is accessed.

Experiment: Caches.1
Purpose: Determine or confirm the cache line size used in the processor.
Platforms: 1, 2
Measured: The same workload as the data trashing procedure performs, but in a different memory area.
Parameters: Mode: data; allocated: 512 KB; accessed: 512 KB; granularity: 4 bytes
Trashing: Mode: data; allocated: 512 KB; accessed: 512 KB; granularity: 4 B – 512 B increasing exponentially.
Expected results: The effect of trashing and thus the measured duration should be the same for all granularity values up to cache line size, because the whole line is evicted on the first access and subsequent accesses to other words in the line do not make a difference. Increasing granularity over the cache line size should result in decreasing the number of evicted lines, thus decreasing the duration of the measured code.
Actual results: On Platform 1, the results were as expected – the measured duration is roughly constant for trashing granularity up to 64 bytes, which is the declared cache line size, and decreases with larger granularity. However, results on Platform 2 indicate cache line size of 128 bytes, although declared size is 64 bytes. This can be attributed to the fact that Pentium 4 fetches two lines into the L2 cache at once [14]. We will thus use the granularity of 128 bytes for the following artificial experiments. However, we will use 64 bytes granularity in order to trash the L1 cache properly. Although this might result in more memory trashed from the L2 cache than expected in general, there will be no difference when the whole allocated range for trashing is accessed.

Experiment: Caches.2
Purpose: Determine the sizes of the L1 data cache and the L2 unified cache and latencies of data cache hits and misses.
Platforms: 1, 2
Measured: The duration of reading a memory area of given size and with given read granularity either linearly or randomly.
Parameters: Access pattern: random, linear; area size: 0 – 1024 KB with 4 KB step; granularity: 64 B (Platform 1), 128 B (Platform 2)
Expected results: Although there is no trashing performed in the benchmark, increasing the memory area should eventually exceed the size of the L1 data cache and later the L2 unified cache, which should increase the average duration per one access by the latency of the L1 cache miss and the L2 cache miss respectively.

Actual results: The results of random access on Platform 1 are shown in Figure 3.3. We can see a gradual increase caused by the L1 data cache misses, roughly centered around the declared L1 size of 64 KB, and another increase caused by the L2 cache misses, centered roughly around the sum of L1 and L2 sizes (576 KB). The performance counters confirm that the causes are the L1 data cache and the L2 cache misses, and also indicate that each access causes a L1 TLB miss in the whole range, because of the random access in area of more memory pages that fit in the L1 TLB. Accesses that cause both L1 data cache and L1 TLB miss therefore cost approximately 18 cycles, which is six times the cost of access that hits (3 cycles). Accesses causing L2 data caches miss and L1 TLB miss cost about 146 cycles.

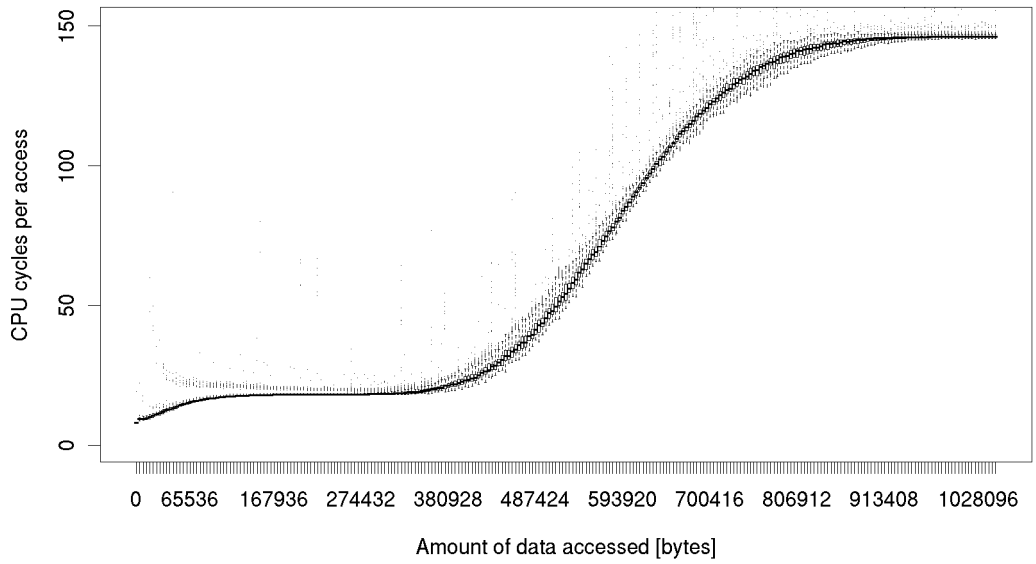

Figure 3.3: *The L1 and L2 data cache miss latencies (including the L1 TLB miss latency) on Platform 1 with random access*

The costs of cache misses are not so large with linear access, as shown in Figure 3.4. The number of TLB misses per access diminishes, as each page is accessed 64 times in a row, reusing the TLB entry. Linear access also triggers hardware prefetch of the next cache line before its address is decoded from the previous line. As a result, accesses causing L1 miss cost approximately 15 cycles compared to 18 cycles with random access. The L2 cache miss latency, which reflects the system memory latency, is much lower here – only 29 cycles compared to 146 with random access. This is because DRAM memory also yields better performance with linear access patterns [33].

On Platform 2, the results of random access show similar behavior as on the Platform 1. Average cost per access is 3.5 CPU cycles with no cache nor TLB misses, 60 cycles with both L1 cache miss and TLB miss, and almost 400 cycles with L2 and TLB misses. The experiment was also performed using only 16 KB memory range and with step of 512 bytes in order to eliminate the TLB misses and see the effect on the L1 data cache more precisely. Figure 3.5 shows how the increase of cycles per access centered around the L1 data cache size (8 KB). The cost of access causing pure L1 miss without TLB miss is therefore approximately 33 cycles.

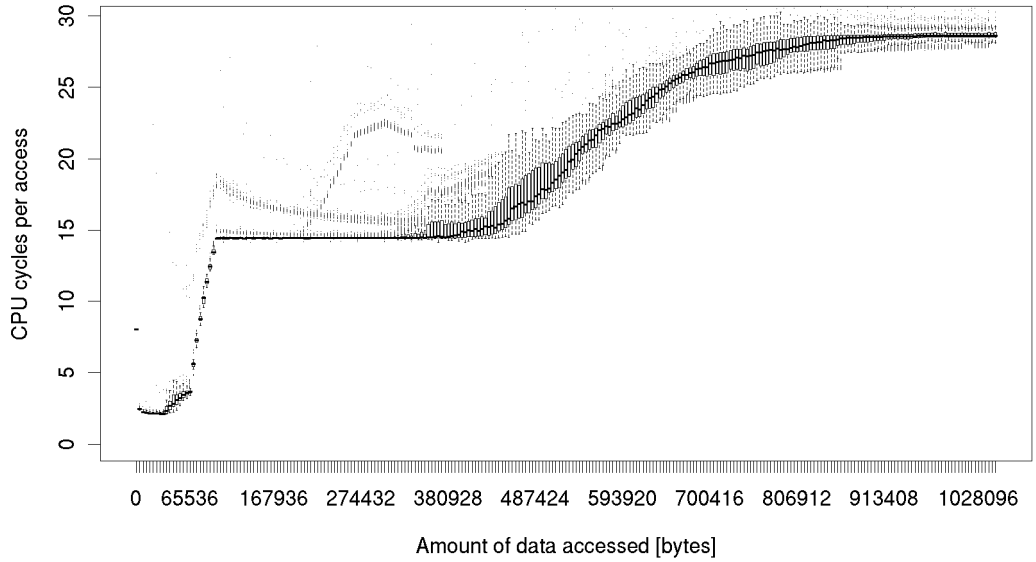

Figure 3.4: *The L1 and L2 data cache miss latencies on Platform 1 with linear access*

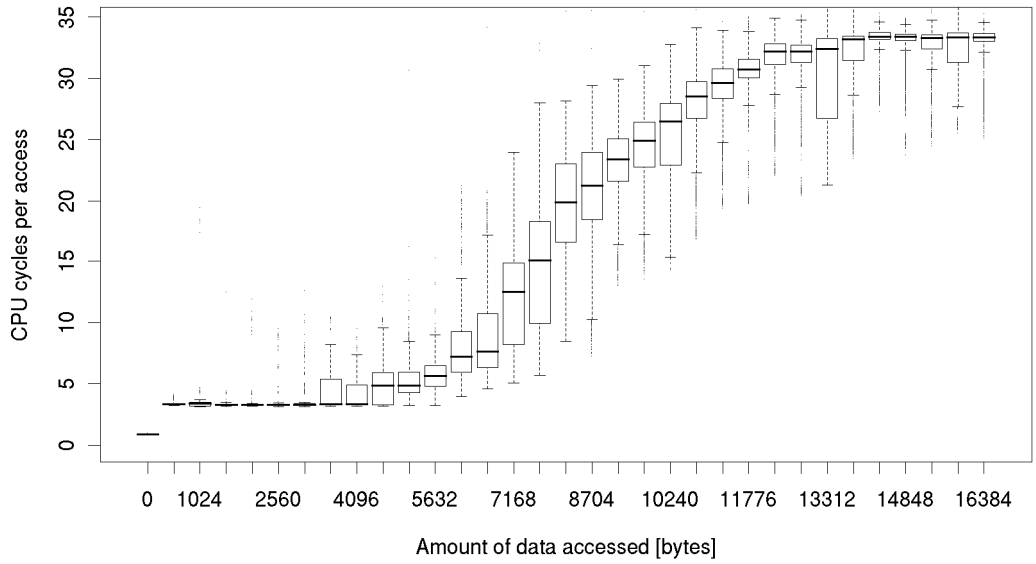

Figure 3.5: *The L1 data cache miss latency on Platform 2 with random access*

The results of linear access on Platform 2 showed surprisingly low cost of access with L1 miss – just about 5 cycles, compared to 15 on Platform 1. The cost of access with L2 miss was approximately 106 clocks.

Experiment: Caches.3

Purpose: Determine the sizes of the L1 code cache and the L2 unified cache and latencies of code cache hits and misses.

Platforms: 1, 2

Measured: The duration of executing a code consisting of jump instructions in a memory area of given size and with given jump granularity. The addresses of the jumps are either linear or random.

Parameters: Access pattern: random, linear; area size: 0 – 1024 KB with 4 KB step; granularity: 64 B (Platform 1), 128 B (Platform 2)

Expected results: The results should be similar to the previous experiment with data caches with respect to possibly different L1 code cache size. The miss latencies should not differ significantly from the data access latencies.

Actual results: On Platform 1, the results with random code access show similar behavior to the previous experiment with data access, except that the increase of duration related to L2 misses starts and ends roughly 150 KB earlier. The performance counter for the L1 code cache misses shows that each access causes almost two misses, although our jump instructions fit into one line. This could be caused by the hardware prefetch which probably fetches always the next cache line and does not try to adapt to the access pattern, because code is usually executed linearly. The duration of one access is approximately 5 CPU cycles without misses, 35 cycles with the L1 code cache misses and the L1 ITLB miss, and 142 cycles with the L2 cache miss and the L1 ITLB miss.

The results of linear code access support the hypothesis of prefetch of the following cache line – the number of L1 cache misses is at most one per access, and the increase of duration related to L2 misses is no longer shifted. The duration per one access is approximately 26 cycles with the L1 code cache miss, and 66 cycles with the L2 cache miss.

The results of random code access on Platform 2 show the expected behavior for the L2 cache miss effect. The L1 code cache miss effect is more interesting as shown in Figure 3.6 for the experiment repeated with a smaller memory area to eliminate the TLB misses. Because the Pentium 4 processor uses a trace cache as the L1 code cache, only the decoded micro-operations of the instructions that have been executed are stored instead of the whole cache lines. Because of this, it needs more accesses with the 128 bytes granularity to fully saturate the cache, which has a capacity of 12 K micro-operations.

The duration per access is approximately 10 cycles with no misses, 47 cycles with the L1 code cache miss, 63 cycles with both the L1 code cache miss and ITLB miss, and 400 cycles with L2 cache miss and ITLB miss. The results of linear code access on Platform 2 yield also 47 cycles with L1 code cache miss (which indicates there is no code prefetch), and 138 cycles with the L2 cache miss.

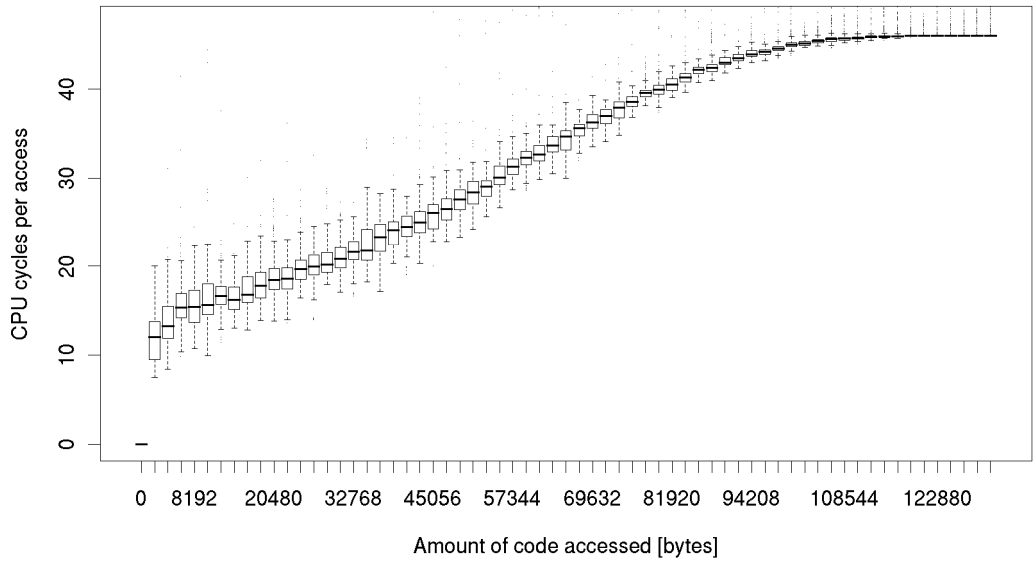

Figure 3.6: *The L1 code cache miss latency on Platform 2 with random access*

Experiment: Caches.4
Purpose: Determine the memory area size needed for trashing all caches
Platforms: 1, 2
Measured: The same workload as the data trashing procedure performs, but in a different memory area.
Parameters: Mode: data; allocated: 512 KB; accessed: 128 KB, 512 KB, 1024 KB; granularity: 64 B
Trashing: Mode: data; allocated: 256 KB – 8 MB with 256 KB step; accessed: all that is allocated; granularity: 64 B
Expected results: The measured duration should increase as the memory range allocated for trashing increases, to a point where the whole buffer of the measured workload is evicted from both the L1 data cache and the L2 cache, and the duration should stay constant from that point on.
Actual results: On both platforms, the point where more trashing did not prolong the measured duration anymore, was below 2 MB for all buffer sizes, and therefore 2 MB will be used in further experiments as the memory range allocated for trashing and maximum amount of memory accessed during trashing.

3.2.4 FFT

The artificial benchmarks have shown that cache trashing can have very significant performance impact on code due to large miss latencies. However, the artificial code is purposely memory intensive and does not perform anything else. Thus, the results of artificial benchmark should be perceived as an extreme limit of the trashing effect. For practical purposes, we would like to know how much affected are some practical algorithms. The rest of the experiments with caches will therefore use existing implementation of commonly used algorithms. The first algorithm is the Fast Fourier Transform, FFT for short.

A usual FFT implementation takes a memory buffer with input data (array of complex numbers, alternatively split into two separate arrays for real and imaginary parts) and transforms the data inside the buffer – the input is overwritten with output. Alternatively, it uses two separate buffers for input and output data, in which case the input is only read. During the computation, no extra buffers are generally needed. The FFT duration depends on the size of input data, and is relatively short on modern processors and input sizes comparable to cache sizes and thus should not be interrupted and rescheduled during its invocation. This means that the cache sharing can affect performance only by evicting the input buffer or the FFT implementation code from the caches before the FFT is invoked.

Based on this assumption, our experiments will perform the cache trashing between input buffer fill and the FFT invocation. We will use two different FFT implementations for the experiments. The first one is based on the simple FFT benchmark implementation [22] and can perform in-place transformation only. The second is the FFTW 3.1.1 library [10], which supports both in-place operation and distinct input and output buffers - we will test both modes. The benchmark therefore has two parameters – the implementation including the mode of operation to use, and the size of input buffer to allocate.

Experiment: FFT.1
Purpose: Determine the impact of data cache sharing on performance of the simple FFT implementation.
Platforms: 1, 2
Measured: Duration of a FFT transformation with varying input buffer size.
Parameters: FFT method: simple; FFT buffer size: 4 KB – 512 KB increasing exponentially
Trashing: Mode: data; allocated: 2 MB; accessed: 0, 8 KB – 2 MB increasing exponentially; granularity: 64 B
Expected results: In all cases, accessing more data during trashing should prolong the FFT duration. The relative slowdown should decrease as the buffer sizes increase – our random trashing should ensure the same fraction of the buffer is evicted with any size, but since larger buffers result in more cache misses even without trashing, the number of misses our trashing can add gets relatively lower. Additionally, the complexity of FFT is $O(n \log n)$, while number of cache misses grows at most linearly with the buffer size, further decreasing the relative impact of cache misses on the FFT duration.
Actual results: The results from Platform 1 are shown in Figure 3.7. As

expected, the most significant slowdown of more than 10% is observed with the smallest FFT buffer size used. We can clearly see how increasing the amount of trashing up to 128 KB evicts the buffer from the L1 data cache, and between 256 KB and 1 MB, the data is gradually evicted also from the L2 cache. Generally, larger FFT buffers indeed yield less impact of trashing, with the exception of 32 KB and 64 KB buffers and trashing up to 256 KB. The impact on FFT with buffer of size 256 KB and larger is 1% at most.

The results from Platform 2 were too unstable to give meaningful results, even after obtaining 100 000 samples. The only exception were the results obtained with a 8 KB large FFT buffer, which were similar to the results with a 4 KB buffer on Platform 1 and also yielded more than 10% slowdown.

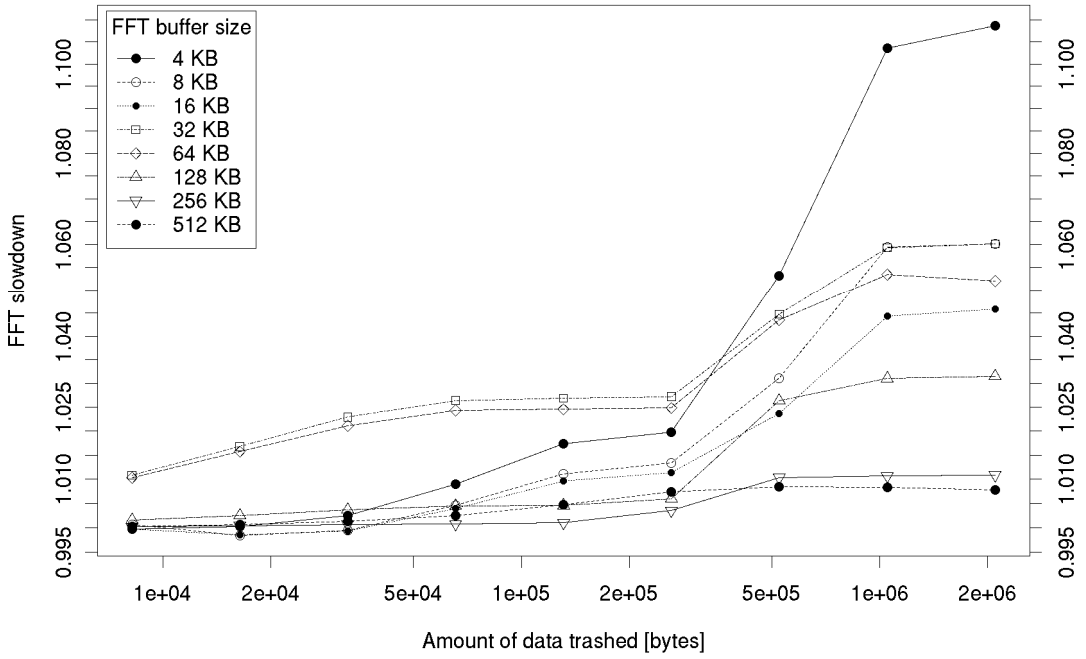

Figure 3.7: *Effect of data cache trashing on the simple FFT implementation performance on Platform 1*

Experiment: FFT.2
Purpose: Determine the impact of code cache sharing on performance of the simple FFT implementation.
Platforms: 1, 2
Measured: Duration of a FFT transformation with varying input buffer size.
Parameters: FFT method: simple; FFT buffer size: 4 KB – 512 KB increasing exponentially
Trashing: Mode: code; allocated: 2 MB; accessed: 0, 8 KB – 2 MB increasing exponentially; granularity: 64 B
Expected results: The impact of code cache trashing should be low, because the simple FFT implementation consists of just a few functions with nested loops. Thus, the code evicted from the caches has to be fetched only on the first iteration. In addition, the compiled object file of the simple FFT implementation is smaller than 4 KB,

which is well below L1 code cache sizes on both tested platforms. Increasing the FFT buffer size up to the L1 data cache size should decrease the relative slowdown, because the absolute slowdown in clocks stays constant (code trashing does not affect L1 data cache) and FFT duration increases with data buffer size. FFT buffer sizes that do not fit in the L1 data cache will eventually be evicted from the unified L2 cache by the code trashing similarly to the data trashing. However, as we saw in the previous experiment, the slowdown with large buffers is insignificant, and should be even lower here, because a portion of the buffer is still kept in the L1 data cache.

Actual results: The results from Platform 1, presented in Figure 3.8, have fulfilled the expectations – the performance impact was lower than the impact of data cache trashing. The highest observed slowdown was 6% with a 4 KB buffer and under 3% with an 8 KB buffer, both well below the impact of an equivalent amount of data trashing. All larger buffer sizes yielded less than 2% slowdown.

The results from Platform 2 were again too unstable, with the exception of the results obtained with a 4 KB buffer which yielded performance impact similar to the data trashing with 8 KB buffer in the previous experiment, also with more than 10% slowdown.

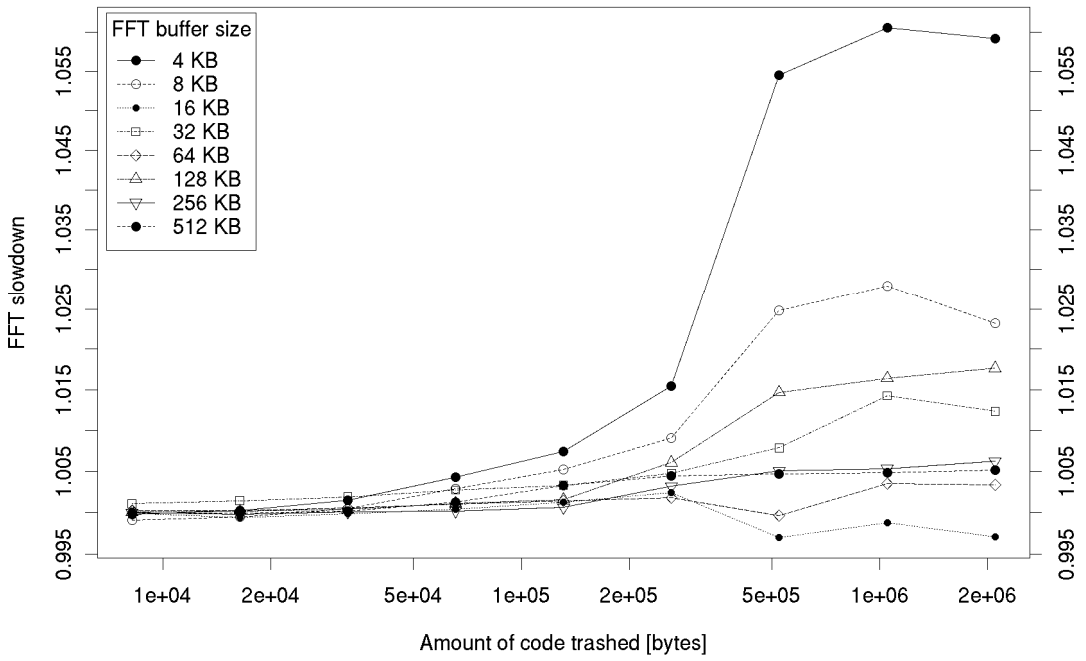

Figure 3.8: *Effect of code cache trashing on the simple FFT implementation performance on Platform 1*

Experiment: FFT.3
Purpose: Determine the impact of data cache sharing on performance of the FFTW in-place transformation.
Platforms: 1, 2
Measured: Duration of a FFT transformation with varying input buffer size.

Parameters: FFT method: in-place FFTW; FFT buffer size: 4 KB – 512 KB increasing exponentially

Trashing: Mode: data; allocated: 2 MB; accessed: 0, 8 KB – 2 MB increasing exponentially; granularity: 64 B

Expected results: We expect the FFTW implementation to be affected relatively more than the simple implementation, because it is much more optimized for speed, including efficient cache usage. FFTW empirically chooses the fastest algorithm for given buffer, which should minimize the number of cache misses without trashing. Thus, the amount of misses added by our trashing should be relatively larger.

Actual results: As expected, data trashing causes much more significant slowdown of the FFTW than of the simple implementation. Results from Platform 1 are presented in Figure 3.9. With the exception of a 4 KB buffer, increasing the buffer size yields less significant slowdown. The most significant observed slowdown is over 80% with 8 KB and 16 KB buffer. For 16 KB and smaller buffers, trashing amount of 64 KB is enough to cause 10% or larger impact. Buffers of 128 KB or larger sizes yield less than 5% slowdown.

Results from Platform 2, shown in Figure 3.10 show even more significant impact than Platform 1. With a 4 KB buffer, the FFT duration increases by a factor of 3 with 512 KB or more trashing. Larger buffers generally yield less significant slowdown with the exception of 32 KB and 64 KB buffers and trashing amount of 256 KB or less. Only a 512 KB buffer yields less than 5% slowdown.

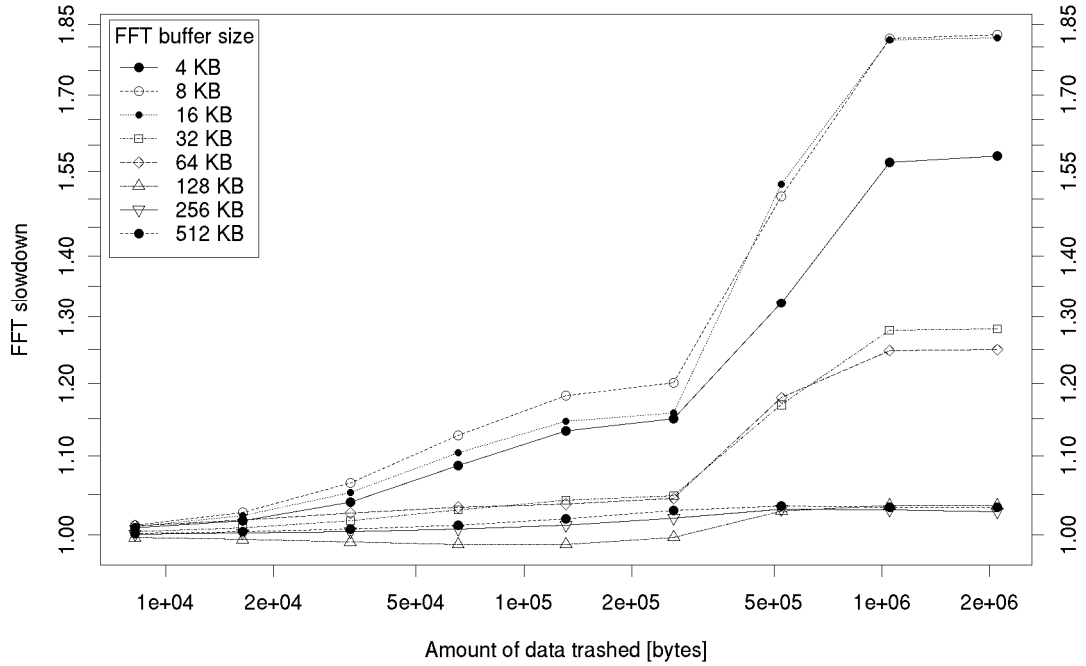

Figure 3.9: *Effect of data cache trashing on the FFTW in-place transformation performance on Platform 1*

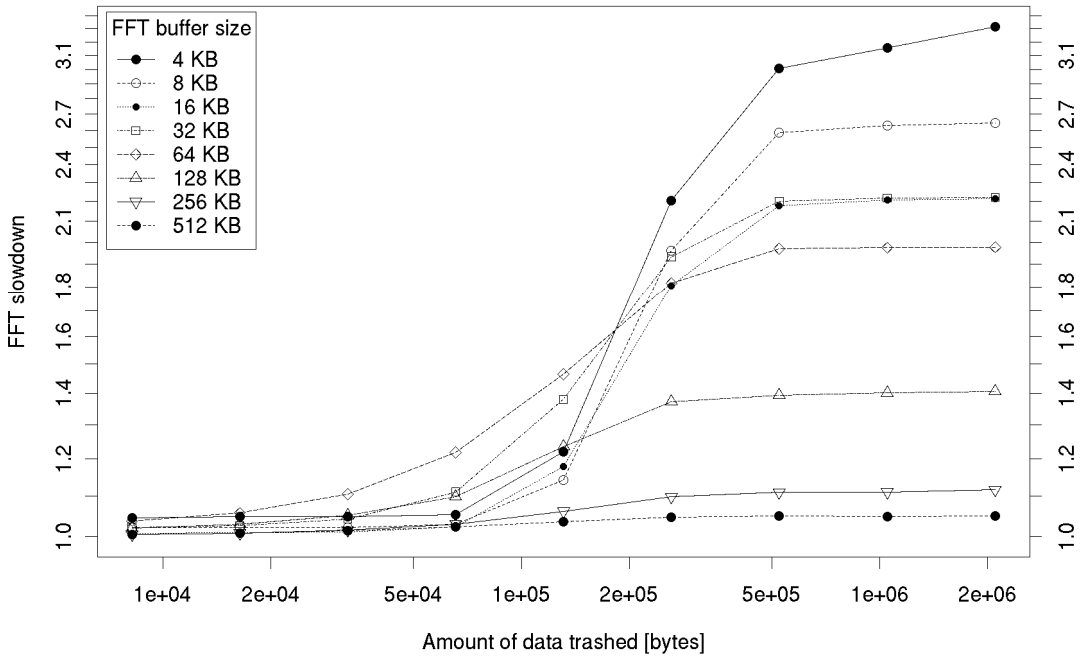

Figure 3.10: *Effect of data cache trashing on the FFTW in-place transformation performance on Platform 2*

Experiment: FFT.4
Purpose: Determine the impact of code cache sharing on performance of the FFTW in-place transformation.
Platforms: 1, 2
Measured: Duration of a FFT transformation with varying input buffer size.
Parameters: FFT method: in-place FFTW; FFT buffer size: 4 KB – 512 KB increasing exponentially
Trashing: Mode: code; allocated: 2 MB; accessed: 0, 8 KB – 2 MB increasing exponentially; granularity: 64 B
Expected results: Based on the results obtained with the simple FFT implementation (Experiment FFT.2), we expect the code trashing to yield less significant impact than the data trashing. The difference could be however smaller because the FFTW implementation consists of much larger and complex code than the simple implementation.

Actual results: Figure 3.11 shows the results from Platform 1. For an amount of trashing up to 128 KB, the impact is equal or lower than the impact of equivalent data trashing, as expected. However, code trashing of 256 KB or more result in more significant slowdown (more than 2.1 with 4 KB buffers) than the data trashing yields. This is most probably caused by both the FFTW code being evicted from the L2 cache, which outweighs the benefit of the data being kept in the L1 data cache.

The results from Platform 2 in Figure 3.12 also show that the code trashing can have more impact than the data trashing, even with lower amount of trashing (64 KB). This can be attributed to the significant difference between the L1 code

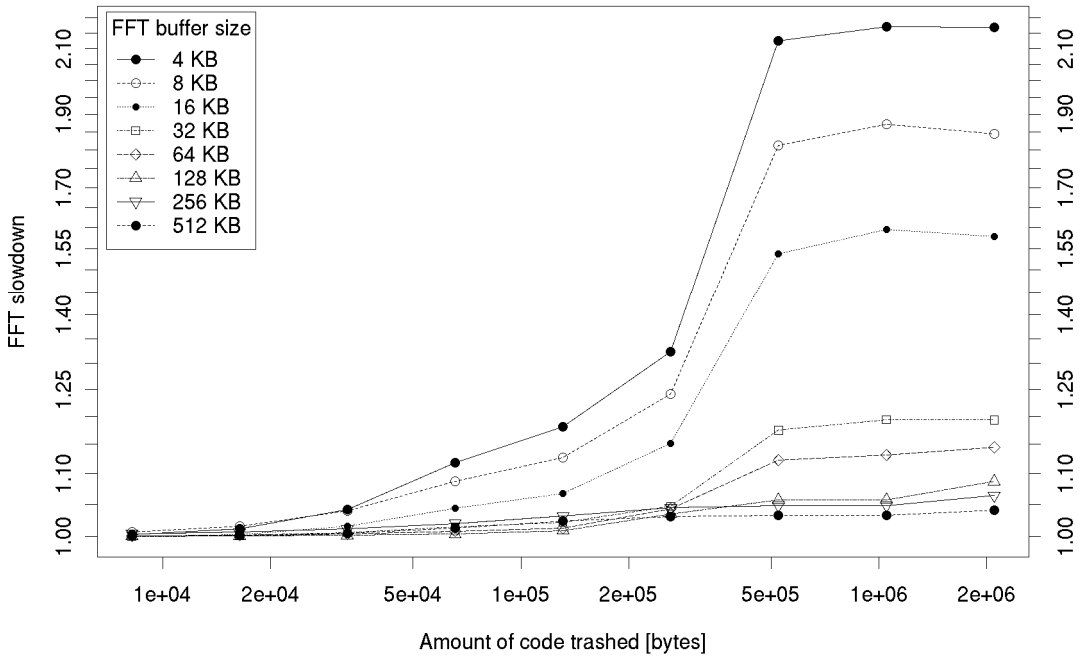

Figure 3.11: *Effect of code cache trashing on the FFTW in-place transformation performance on Platform 1*

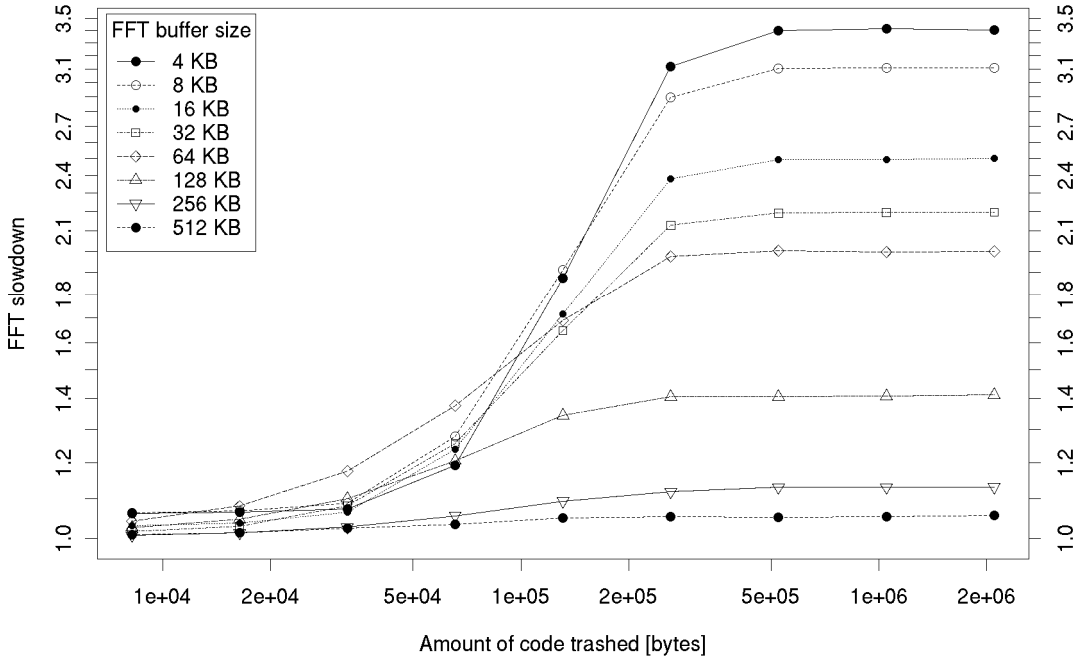

Figure 3.12: *Effect of code cache trashing on the FFTW in-place transformation performance on Platform 2*

cache miss latency (even with linear access) and the L1 data cache miss latency we have observed in the artificial benchmarks (the experiments Cache.1 and Cache.2) on this platform.

The results presented so far have always obeyed one rule – increasing the amount of code or data trashing impacts the performance of the measured operation negatively, or at least not positively. This is obviously expectable, as accessing more memory outside of the code or data area used by the measured operation can only evict more of this code or data from the caches, and cannot fetch it back. However, there are situations where the opposite happens, and more trashing actually improves performance.

When incorporating the simple FFT implementation [22] into the RIB framework, it was first executed almost exactly as the authors designed it – the FFT transformation was followed immediately by an inverse transformation of the results (duration of these two invocation was measured together). The results were then scaled to resemble the original input data again. The original benchmark does this to determine the precision by computing the standard deviation of the difference between these results and the original input. In our framework, only the duration of the forward transformation is measured, but the inverse FFT and scaling was initially performed between the measured transformation and trashing (instead of initializing the buffer with new data).

First runs of this benchmark on Platform 1 had unexpected results. For smaller FFT buffer sizes, increasing trashing increased also the slowdown, as in the Experiment FFT.1. However, with larger buffers, the FFT duration was first increased and then decreased, as Figure 3.13 shows for a 256 KB buffer. On Platform 2, FFT duration for all tested buffer sizes increased with trashing as expected. After replacing the inverse FFT and scaling with a simple input buffer reinitialization, the weird effect disappeared. Further experiments revealed that the scaling was actually performed not just on the FFT buffer, but a memory area twice the buffer size – correcting this also removed the strange behavior. Thus, we arrived at a scenario, where our intentional random trashing between FFT invocations was preceded with another, linear trashing – addresses outside of the buffer were being accessed. Instead of the effects these two kinds of trashing adding up, increasing the random trashing to a certain point started to improve the performance of the FFT invocation, but only on one of the two tested platforms.

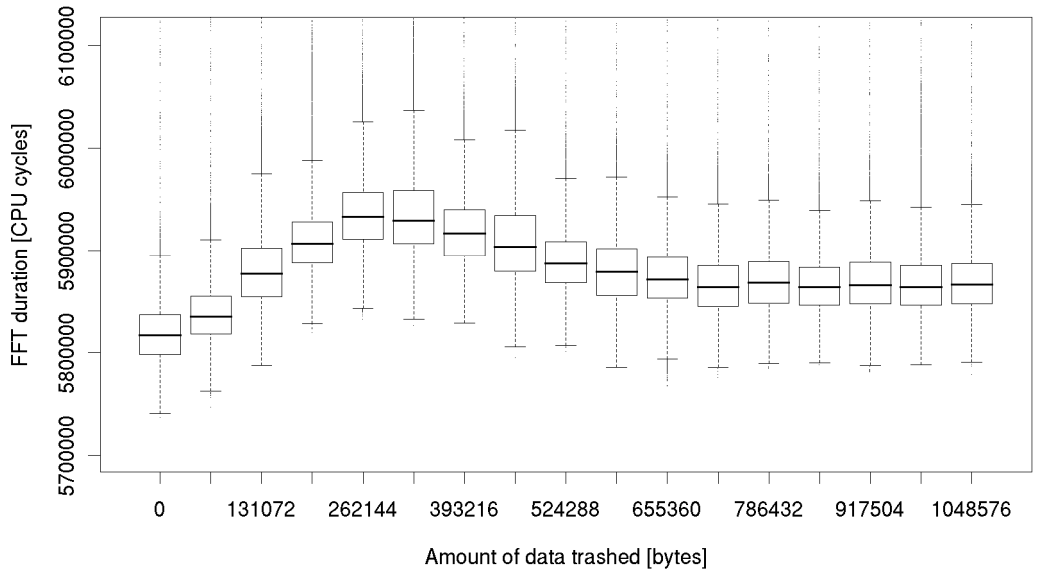

Figure 3.13: *Unusual effect of data cache trashing improving the simple FFT performance with a 256 KB buffer*

To find the cause of this behavior, we have repeated the experiment with a 256 KB buffer, collecting the cache miss events of all caches. The number of L1 data cache misses was roughly constant, because the buffer itself was already four times larger than the L1 cache size. The number of L2 cache misses increased with trashing up to 512 KB and then remained constant. This proved that the trashing was working correctly, and the weird effect was not caused by some hypothetical possibility of the trashing putting the cache replacement algorithms in a state that fits better the FFT memory access pattern.

Because values of the cache miss counters suggested that the behavior was caused by something else, we had no other choice but to capture and analyze values of all performance counters provided by the processor. We were looking for a counter that either collects events with negative performance impact and shows a decreasing behavior, or collects events with positive performance impact (such as cache hits) and shows an increasing behavior. The hypothesis was that the change in events collected by such counter would eventually overcome the negative effect of L2 cache misses.

Of the counters we measured, some had roughly constant values (e.g. number of retired instructions or mispredicted branches), some showed increasing behavior (L2 cache misses and corresponding refills from the system memory), some just roughly followed the changes of the RDTSC counter, meaning the ratio of events per clock was fixed (e.g. number of clocks that the CPU did not spend in a halted state, number of dispatch stalls etc.). Because Athlon 64 has an integrated memory controller, its event counters were also available. One of these counters – the memory controller page miss event counter – showed a behavior that could be responsible for the performance improvement. Figure 3.14 shows the number of events collected by this counter during the FFT invocation depending on the amount of trashing.

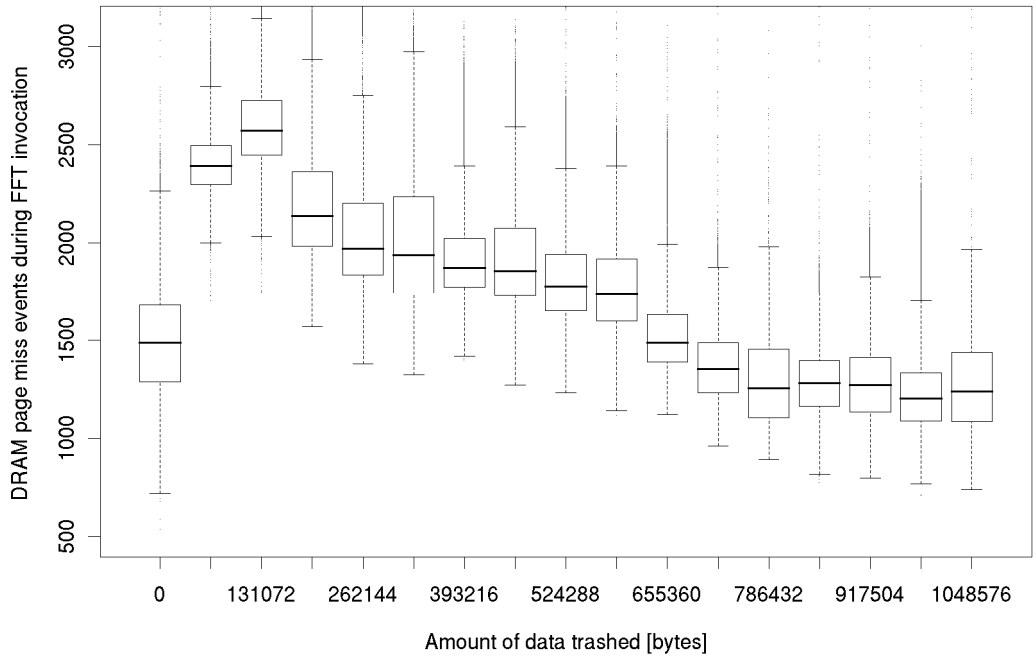

Figure 3.14: *Values of the DRAM page miss counter which could explain the unusual effect of trashing on FFT duration*

In the memory controller, a *page miss* occurs when a memory access has to perform both page (also called *row*) access and *column* access, which increases latency compared to subsequent column accesses to the same row, because pages stay *opened* for a while [33]. We would expect the number of page misses to be roughly proportional the number of memory accesses caused by the L2 cache misses. However, we can see that while there is quite a large difference between none and 64 KB trashing, further increasing the trashing reduces the number of page misses, even below the value observed with no trashing.

Possible explanation of this behavior is the fact that the Athlon 64 processor automatically adapts the number of clocks (*idle cycle limit*) the page stays open [4]. A higher limit increases the chance of another access to the same page (*page hit*) within the limit, but also the chance of access to a different page – a *page conflict*. Page conflicts have higher latency than page misses, because the currently opened page has to be closed first, before the requested page can be open. Thus, the idle cycle limit is dynamically incremented and decremented based on the number of recently occurred page misses and conflicts. It is possible that the linear trashing (caused by scaling data outside of the FFT buffer) adapts the idle cycle limit to a value that is not optimal for the execution of the FFT, while our random trashing gradually shifts the limit back towards the ideal value.

However, the AMD documentation states [4] that “penalties [due to a page miss or conflict] may be overlapped by DRAM accesses for other requests and don't necessarily represent lost DRAM bandwidth”, which means that the FFT duration could actually be unaffected by the number of page misses we observed. To verify this, we disabled the dynamical adaptation of the idle cycle limit and set the limit manually by programming one of the configuration registers as described in [4]. We

repeated the FFT measurement with 256 KB buffer with the idle cycle limit set to 0, 16 and 256 clocks. The behavior of page miss counter with 256 clocks limit was similar to the auto-adjusted limit – also decreasing, just lower in absolute values. With a zero clock limit, the behavior changed drastically, as Figure 3.15 shows. However, the FFT duration was not affected by this change, still showing the same behavior as on Figure 3.13. This means that the number of page misses was not responsible for the behavior after all, and we are still uncertain about the reason.

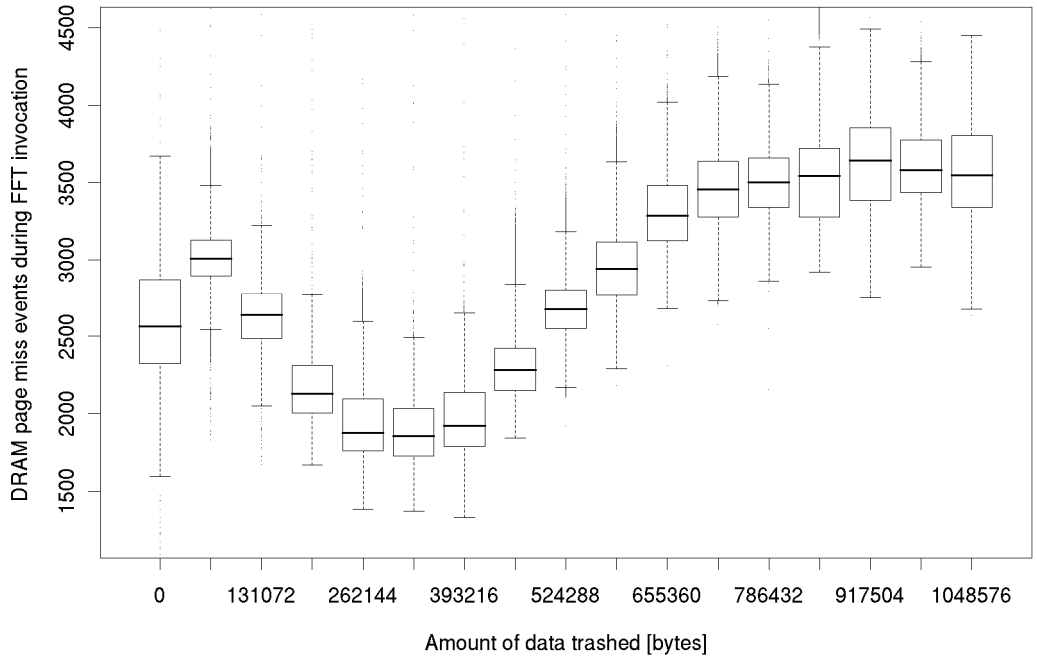

Figure 3.15: *Values of the DRAM page miss counter with the idle cycle limit set to zero and disabled dynamical adjustment*

While the scenario discussed above was discovered accidentally, could be reproduced only on one tested platform and may seem too artificial, the weird behavior of more trashing improving performance was later confirmed also in a regular experiment with FFTW using separate input and output buffers.

Experiment: FFT.5
Purpose: Determine the impact of data cache sharing on performance of the FFTW transformation with separate input and output buffers.
Platforms: 1, 2
Measured: Duration of a FFT transformation with varying buffer size.
Parameters: FFT method: FFTW with sep. buffers; FFT buffer size: 4 KB – 512 KB increasing exponentially
Trashing: Mode: data; allocated: 2 MB; accessed: 8 KB-1 MB (exponentially); granularity: 64 B
Expected results: We expect the results to be similar to in-place FFTW, but with generally lower slowdown, because using separate input and output buffers should result in double the memory being accessed compared to in-place transformation, and the slowdown of in-

place transformation was generally lower with higher buffer sizes.

Actual results: Contrary to the expectations, the results show that the slowdown is roughly the same compared to the in-place transformation, and even higher for lowest buffer sizes, on both tested platforms. The real surprise, however, are the results with 256 KB and especially 512 KB buffer sizes, where we can observe more than 3% performance improvement with increased amount of trashing. Unlike the abovementioned special scenario with simple FFT, this impact is reproducible on both tested platforms. Figure 3.16 shows this effect on Platform 2, where the speedup effect is even larger than on Platform 1.

Note that similar effect can be observed also with code cache trashing, because it also results in the L2 cache trashing. Note that this behavior should not be caused by the FFTW choosing a more efficient implementation based on the performance under heavy trashing, because the measurement and method selection is done only once for a given buffer before the actual FFT and trashing invocations, and then used for all subsequent FFT invocations.

We did not further investigate the reasons of this effect like with the simple FFT scenario, due to time constraints. Nevertheless, these results demonstrate that cache trashing can have very unexpected results, depending on many specific details that would be very complex to model.

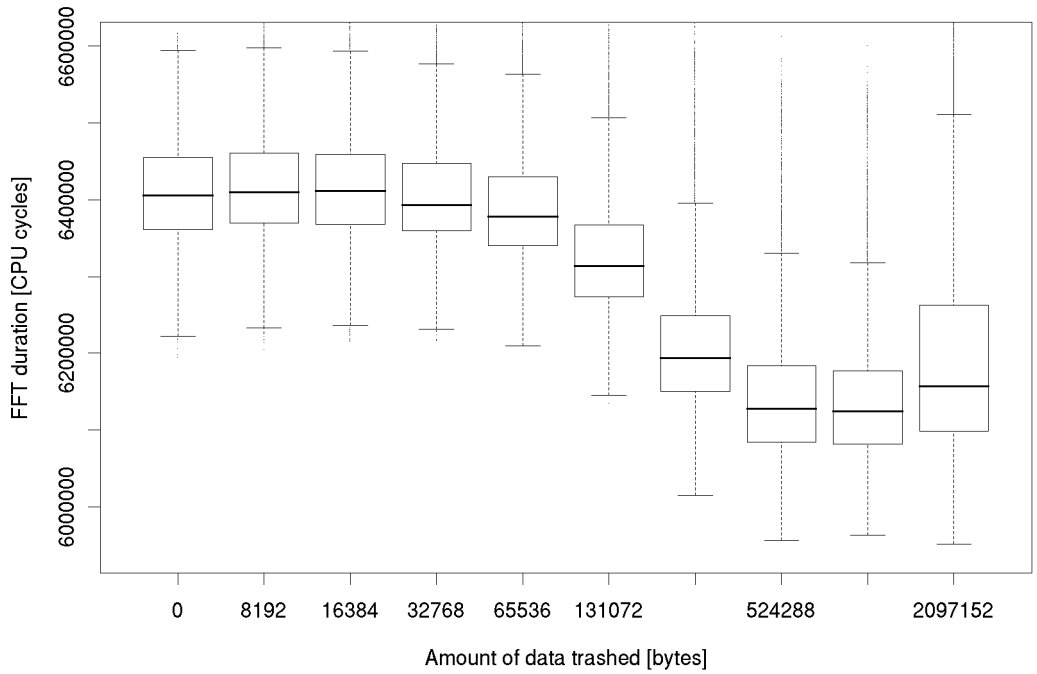

Figure 3.16: *Unusual effect of data cache trashing improving performance of FFTW with two separate 512KB buffers on Platform 2*

3.2.5 LZW

The next code we will test to determine effect of cache trashing is an implementation of the Lempel-Ziv-Welch data compression algorithm. During its operation, this algorithm is accessing not just the data to compress (which is typically significantly larger than the processor caches), but also relatively large (comparable to the cache sizes) internal memory structures – the translation table (dictionary) used for the compression. This dictionary is initialized to a small default size and then grows and changes based on the compressed data. When compressing large files, preemptive scheduling and waiting for I/O operations can result in both the pending input data and dictionary to be evicted from the caches by other processes.

To model this scenario in our experiments, we will have to perform the cache trashing repeatedly throughout the LZW operation to affect its performance – trashing just between the whole operations like with FFT would have no effect on the dictionary.

For the experiments, the source code of the standard UNIX utility *compress* [9] was incorporated into the RIB framework. This utility reads the given input file and writes the LZW compressed output into another file, using 8 KB input and output buffers. The read and write syscalls are natural places where the process would block to perform the I/O operation, and potentially be rescheduled, which is why we replaced the syscalls with callbacks to the framework. During the callback, the time measurement is paused so that the file system and disk performance does not distort the results. Then, the actual read or write operation is performed. Before resuming the time measurement and returning from the callback, the cache trashing function may be called, based on the benchmark's parameters. These parameters specify that the trashing is to be performed after each ***N*** reads and/or after each ***M*** writes. A value of zero results in no trashing.

Another parameter of the benchmark reflects the *maxbits* parameter of the compress utility, which limits the maximum size the dictionary can grow to. It is specified as the maximum number of bits for the output codes. Possible values are from the range of 9 to 16 and correspond with the dictionary size of approximately 8 KB up to 800 KB, growing exponentially. The experiments will determine how the dictionary size limit, the frequency and amount of trashing affect the LZW performance.

Experiment: LZW.1
Purpose: Determine the effect of data cache sharing on performance of the LZW compression.
Platforms: 1, 2
Measured: Time to compress a 75 MB large tar file consisting of source code, excluding the duration of I/O operations.
Parameters: Dictionary size limit: 9 - 16; trashing frequency: none, each 1-32 (increasing exponentially) read calls.
Trashing: Mode: data; Allocated: 2 MB; accessed: 4 KB – 2MB increasing exponentially; granularity: 64B.
Expected results: Increasing the amount of trashed memory should obviously make the LZW performance worse. The most frequent trashing should yield the strongest impact, while low trashing frequency should

make the effect negligible at some point, as the time the code spends warming up the caches after trashing relatively decreases. Based on the results of FFT benchmarks, we can expect the slowdown to decrease as the maxbits setting and thus dictionary size increases.

Actual results: Generally, more data accessed during trashing results in longer LZW duration, as expected. Figure 3.17 shows the slowdown on Platform 1 with the most frequent trashing rate and various maxbits settings. Contrary to the expectations, increasing maxbits between 10 and 15 results in more significant impact of trashing rather than less significant impact. This can be attributed to the fact that while the dictionary size (and thus the number of cache misses we inflict by trashing it) grows exponentially, the observed duration of LZW with no trashing grows linearly. The relative impact of trashing thus increases. With maxbits setting of 16 the impact decreases for trashing amounts of 512 KB and more, probably because the number of cache misses with no trashing increases as the dictionary size gets close to or exceeds the L2 cache size. The results with no trashing confirm nonlinear increase between the maxbits settings of 14 and 16. The most significant observed slowdown on Platform 1 is 54%.

Results from Platform 2, presented in Figure 3.18 are similar. The difference between maxbits settings between 9 and 15 is less significant and the phase where increased maxbits result in less significant slowdown begins earlier, with maxbits setting of 14. The most significant observed slowdown on Platform 2 is almost 60%.

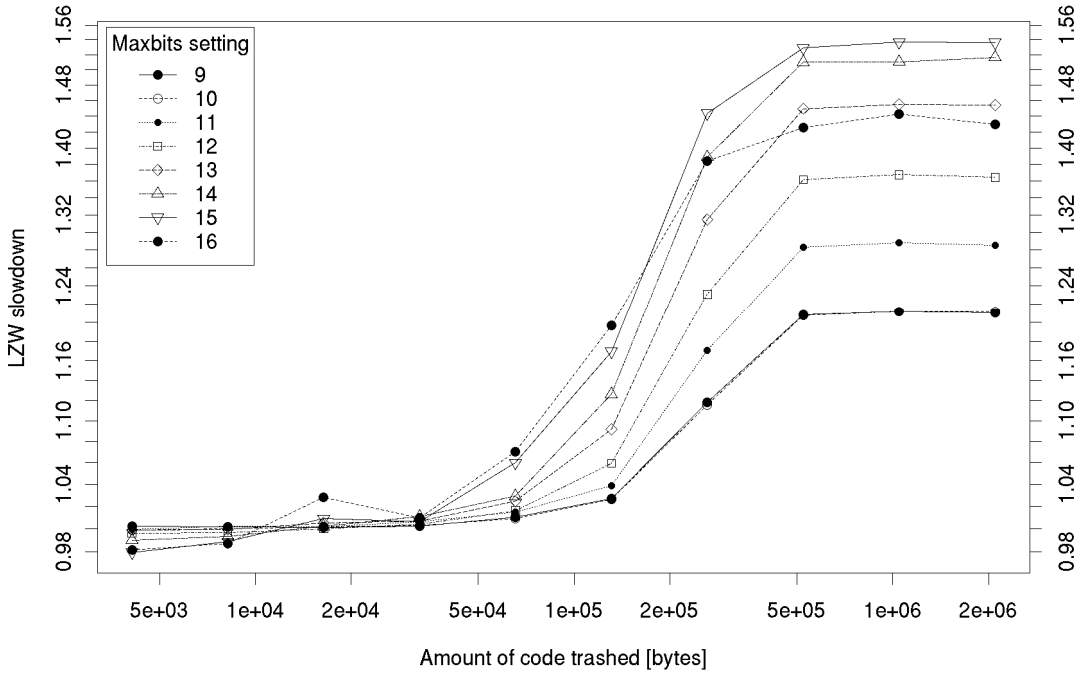

Figure 3.17: *Effect of data cache trashing on the LZW performance on Platform 1, various dictionary sizes, trashing after each 8 KB of input data*

Figure 3.19 shows the LZW slowdown on Platform 2 with maxbits set to 9 and various trashing frequencies. We can see that the frequency influences the performance impact notably. Trashing with the most frequent rate (after each 8 KB

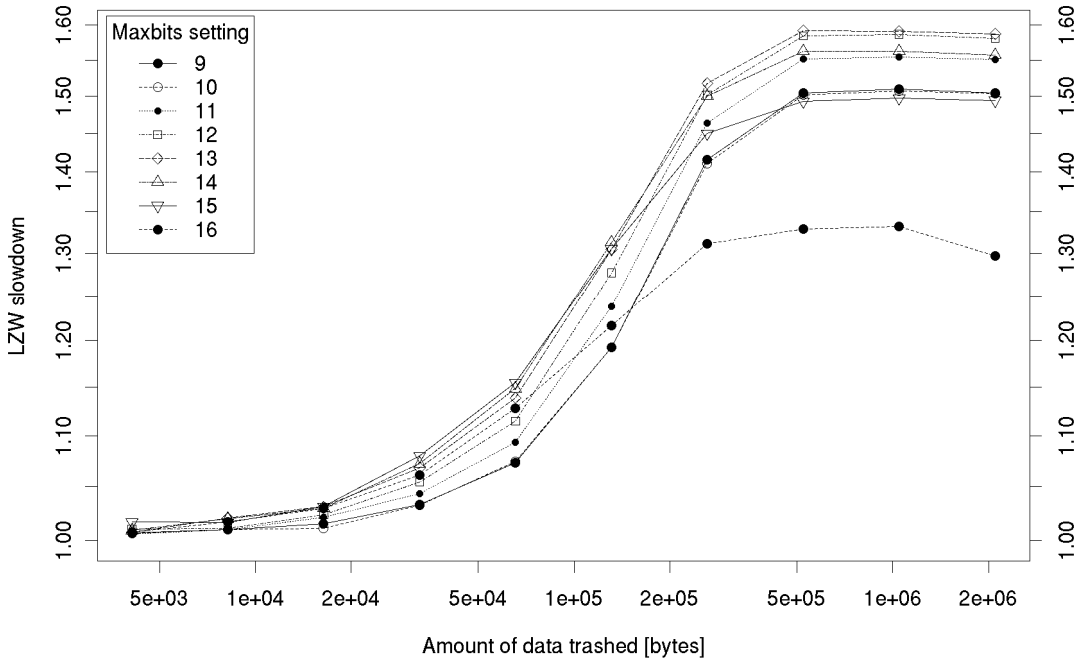

Figure 3.18: *Effect of data cache trashing on the LZW performance on Platform 2, various dictionary sizes, trashing after each 8 KB of input data*

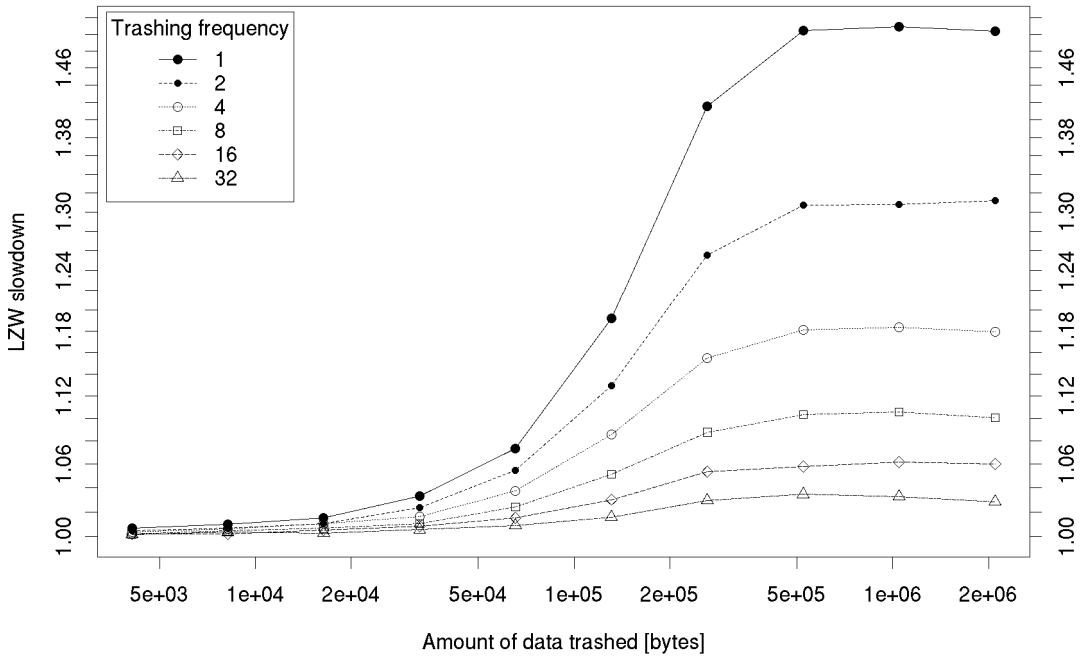

Figure 3.19: *Effect of data cache trashing on the LZW performance on Platform 2, various trashing frequency, maxbits set to 9*

of input data) yields at most 50% slowdown, while the rate of 32, which means trashing occurs after each 256 KB of input data, the slowdown does not exceed 4%. Results from Platform 1 show very similar effect.

Experiment: LZW.2
Purpose: Determine the effect of data cache sharing on performance of the LZW compression.
Platforms: 1, 2
Measured: Time to compress the same file as in the experiment LZW.1.
Parameters: Dictionary size limit: 9 - 16; trashing frequency: none, each 1-32 (increasing exponentially) read calls.
Trashing: Mode: code; Allocated: 2 MB; accessed: 4 KB – 2MB increasing exponentially; granularity: 64B.
Expected results: Based on the results of the experiments with FFTW, we can expect the impact of code trashing to be slightly more significant than the impact of data trashing on both platforms, especially Platform 2 due to the high latency of the L1 code cache miss.
Actual results: The results on Platform 2 show that compared to the data trashing, the code trashing affects the LZW performance slightly more with lower amounts of trashing (where the L1 code cache is trashed), but the slowdown is similar with 512 KB and higher amounts of trashing. Figure 3.20 shows this with maxbits set to 9 and various trashing frequency and thus can be directly compared with Figure 3.19 which shows the data trashing effect using the same parameters.

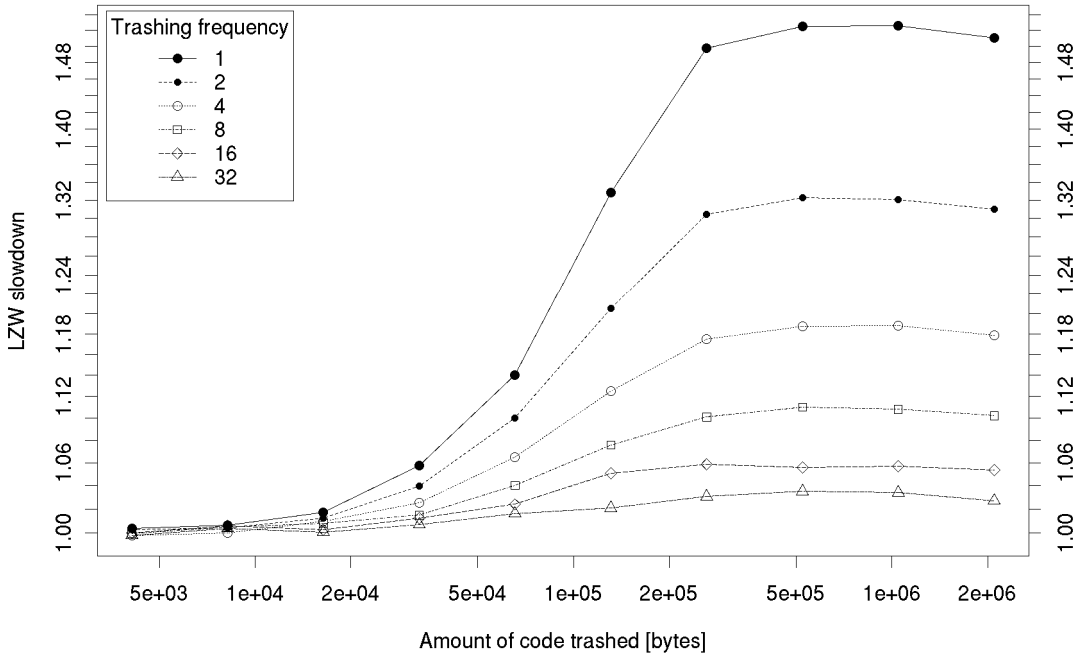

***Figure 3.20:** Effect of code cache trashing on the LZW performance on Platform 2, various trashing frequency, maxbits set to 9*

Even on Platform 2, the impact of code trashing is more significant than the impact of data trashing for lower amounts of trashing, most notably with 256 KB.

However, with 512 KB and more trashing, data trashing yields more significant impact. Figure 3.21 shows the results of code trashing with highest rate and various maxbits settings, and can be compared with Figure 3.17.

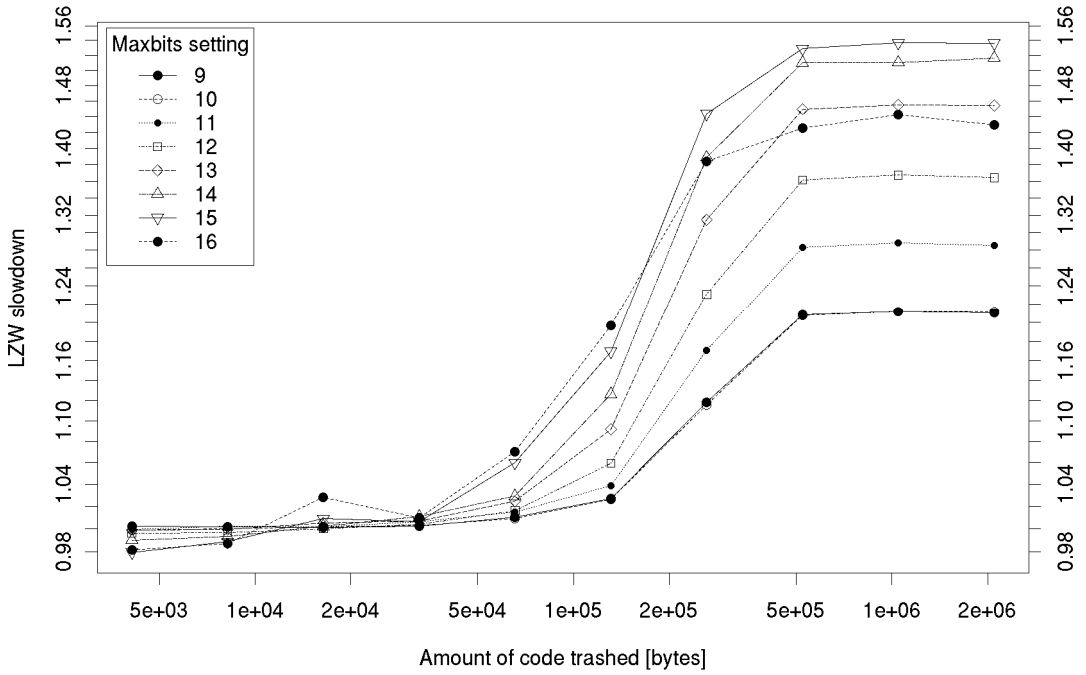

Figure 3.21: *Effect of code cache trashing on the LZW performance on Platform 1, various dictionary sizes, trashing after each 8 KB of input data*

The conclusion is that code that keeps and works with some internal state is likely to be affected by cache trashing, if the state data fits into the L2 cache. We saw that the impact depends mostly on the frequency of trashing, and that infrequent trashing has negligible effect. Calculated from the LZW results, the average time to compress 32 read buffers was 11 ms, which is below the default quantum of most operating systems. This means that trashing the cache by rescheduling the process has insignificant performance impact and we should focus on scenario where a number of relatively quick operations (preferably with internal state) are interleaved within one process.

3.2.6 Transcode

The experiments with LZW compression showed that the impact of cache trashing depends heavily on the trashing frequency, and task switching done by the operating system is too infrequent for the impact to be significant. Thus, in the following experiments we will focus on a scenario where frequent switching of relatively short operations occurs inside one process. This behavior should occur in audio-video processing applications, such as Transcode [30].

Transcode is a tool for converting and filtering audio-video files, frame by frame[2]. Its architecture is threaded and uses *n* pre-allocated buffers (configurable, 10 by default) for storing the video frames. There is always one thread that decodes the input video, storing the decoded frames into available free buffers, blocking if there are none. Then there are *m* threads (also configurable and defaults to one) performing requested transformations on the frames, such as resize, crop, and other specified filters like smooth, sharpen etc. After applying all filters on a frame, it is passed to encoding. This is done by another thread, which encodes the frame into the specified audio and video format, and stores it in the output file. The buffer is then marked as free and can be filled by another frame by the decoder thread.

The operations performed on frames by the threads are relatively short (depending on the video resolution and the processor speed) and memory-intensive (working with frames stored in buffers) and thus could be significantly affected by cache sharing. The code sharing should be obvious, as each operation is performed by a different code, unless all the code fits into L1 code cache simultaneously.

The impact of the data sharing could be trickier. The performance of an operation should be faster if the processed buffer is cached. This is likely to happen if the buffer was just decoded or processed by the previous filter, and there was no operation performed on a different buffer meanwhile. Thus, the least data cache trashing should occur when each frame is processed by all filters in one go. The most data trashing should occur when each thread processes all buffers, and then the next thread starts processing the first buffer, which is already evicted from the cache. On the other hand, this scenario is better for the code cache, as the code of an operation is executed several times without being trashed by the code of another operation. It could be also better for filters that need to remember information from past frames for processing (such as 3D denoise filters).

The resulting performance should therefore depend on the combined effect of both outlined types of sharing, and the number of buffers and the thread scheduling details should make a difference. We will measure the resulting impact of sharing via benchmarking experiments with Transcode, version 1.0.2.

This experiment is different from the previous ones. We do not incorporate Transcode into the RIB framework and interleave its execution with artificial trashing. Instead, we try to observe impact of cache sharing in a real scenario, by comparing performance of a filter in Transcode when used alone with performance of multiple filters chained together. We prepare a video file[3], and measure the duration of decoding and filtering of this file. We use the null encoder, which just

[2] We will focus on the video frames only.

[3] The video has 1498 frames with a 608×240 resolution with 24bpp colors, encoded in MPEG-4 with 1102 kbps rate.

discards the frames, because a real encoder's performance could be affected by the video changes after filtering. We run Transcode through the UNIX *time* command, measuring the *user* time, to obtain these values:

d .....the time to decode the file with no filtering
f_x.....the time to decode and process the file with the filter x
$c_{x,y}$ .....the time to decode and process the file with the filters x and y, in that order

With these values, we can calculate the difference in duration of the two filters used together and the durations when each filter is used alone:

$$r_{x,y} = \frac{c_{x,y} - d}{f_x + f_y - 2d} - 1$$

Positive values mean that the combination of the two filters performed slower, negative values would mean that the combination of the filters performed actually faster. We will present these values in percents.

For the experiments, we used four of the filters provided by Transcode. The *smooth* and *xsharpen* filters work only with one frame and thus do not need to keep internal frames. The *denoise3d* and *hqdn3d* filters need, according to the documentation, at least two frames to work, and thus should be keeping internal state. Due to its limitations, the *xsharpen* filter cannot have multiple instances.

The results from Platform 1 are shown in Table 3.3. We can see that for all filter combinations, the duration of processing with two filters simultaneously was at least slightly longer than each filter alone. In addition, using the same filter twice yields less significant slowdown than combining two different filters. We can attribute this to the code cache being shared by two different filters.

$r_{x,y}$ [%]	**denoise3d**	**hqdn3d**	**smooth**	**xsharpen**
denoise3d	1.0 ± 0.3	2.1 ± 1.3	1.4 ± 1.0	2.3 ± 1.4
hqdn3d	2.6 ± 1.3	0.6 ± 0.5	1.4 ± 1.0	2.0 ± 1.5
smooth	1.5 ± 1.0	1.5 ± 1.1	0.1 ± 0.0	1.4 ± 1.0
xsharpen	4.6 ± 1.4	2.8 ± 1.4	1.4 ± 1.0	N/A

***Table 3.3:** Slowdown of Transcode with combinations of two filters on Platform 1*

The slowdown of using same filter twice should be caused purely by data sharing. We can see it is relatively low, even negligible in case of the smooth filter. This might be because both filters are applied to a frame in one thread and not separate threads, and thus are unlikely to be interrupted in between. Thus, the second filter instance works with data that is present in the cache. Still, some data cache sharing might occur due to internal states of each instance of the filter. Indeed, the filters that keep information from the previous frames, seem to be affected more than the smooth filter.

It is strange that while the order of combined filters does not seem to cause much difference in slowdown in most cases (which is expectable), there are exceptions. The biggest difference occurs in the combination of Xsharpen and Denoise3d where

one filter order yields twice slowdown of the other one. Based on the assumption that the slowdown is caused mostly by cache sharing, the order should not matter – with one filtering thread, the invocations of the two filters should be strictly interleaved. The only difference is the decoding thread, which is mostly invoked before the first filter. However, the order of invocations of the decoder and the first filter should not affect how much of the code of the second filter is evicted from cache.

The results from Platform 2 are presented in Table 3.4. Surprisingly, we can see even measurable speedup for some combinations where the xsharpen filter is invoked as the second filter in the chain. However, this does not happen when the filters is invoked as the first in the chain, which is strange. The most significant slowdown is observed with the hqdnd3d-denoise3d combination of filters, but there is no measurable slowdown when their order is reversed.

$r_{x,y}$ [%]	**denoise3d**	**hqdn3d**	**Smooth**	**xsharpen**
denoise3d	0.1 ± 1.2	-0.1 ± 1.9	0.0 ± 0.1	-4.8 ± 1.7
hqdn3d	4.8 ± 2.0	-0.4 ± 1.1	-0.1 ± 0.2	-3.8 ± 1.1
smooth	0.1 ± 0.1	-0.2 ± 0.2	-0.2 ± 0.1	-0.8 ± 0.2
xsharpen	1.6 ± 2.2	0.6 ± 1.1	-0.8 ± 0.2	N/A

Table 3.4: *Slowdown/speedup of Transcode with combinations of two filters on Platform 2*

In the second experiment, we examine the hypothesis about data cache sharing dependency on the age of the buffer being processed. We decode and apply one filter on our video file. By varying the number of buffers used for processing, we change the limit of the number of frames that can be decoded between the decoding and the filtering of a single frame, evicting the buffer storing this frame from the caches. We repeat this measurement for each of the four filters used in previous experiment, setting the number of buffers in the range of 1 to 10.

Figure 3.22 shows the results on Platform 1. We can see that using only one buffer yields the best performance, and the difference between two and more buffers is relatively little. Results from Platform 2 are similar. This could be explained by the fact that the frame size of our video file is approximately 430 KB, which is close to the L2 cache size. Thus, one frame fits in the cache, using two or more buffers result in the older frames being almost completely evicted before the filter processes them.

To verify this, the benchmark was repeated with the video resized so that each frame occupied only 30 KB. The results were strange – with all filters except smooth, transcoding supposedly took only 10ms with one buffer, which was not possible. After taking a closer look at the experiment, we noticed that Transcode with one buffer used only a fraction of the CPU, and the rest was spent idling, which somehow tampered with the utime measurement. An investigation of the source code of Transcode revealed the cause. Instead of proper producer-consumer synchronization, the decoding thread performs 10ms sleep when it does not have a free buffer for decoding. If the filtering thread processes all the decoded buffers before the decoder wakes up, the whole Transcode process waits.

Average processing duration per frame of the bigger video was 6-8 ms on Platform 1 for the denoise3d, hqdn3d and xsharpen filters. This means that if the

decoder enters the sleep before the filtering thread is scheduled, the filtering finishes before the decoder wakes up, and whole process waits 2-4 ms. Since filtering two or more buffers takes more than 10ms, the decoder wakes up before the filtering thread is done, and does not sleep again because some frame is already filtered and thus free. This could explain the results we have obtained, if the sleeps were somehow affecting utime measuring precision, making it record less time. While we could not think of a clear reason for this, we tried to repeat the experiments using the *perfex* tool from Perfctr [26], which also offers the same time measurement as *times*, but with a CPU cycle precision. Unfortunately, the results obtained this way were clearly wrong – the measured time was always around 150 ms even in cases where Transcode took more than 10 seconds fully using the CPU. This can be attributed to issues between the perfex tool and the multithreaded workload of Transcode.

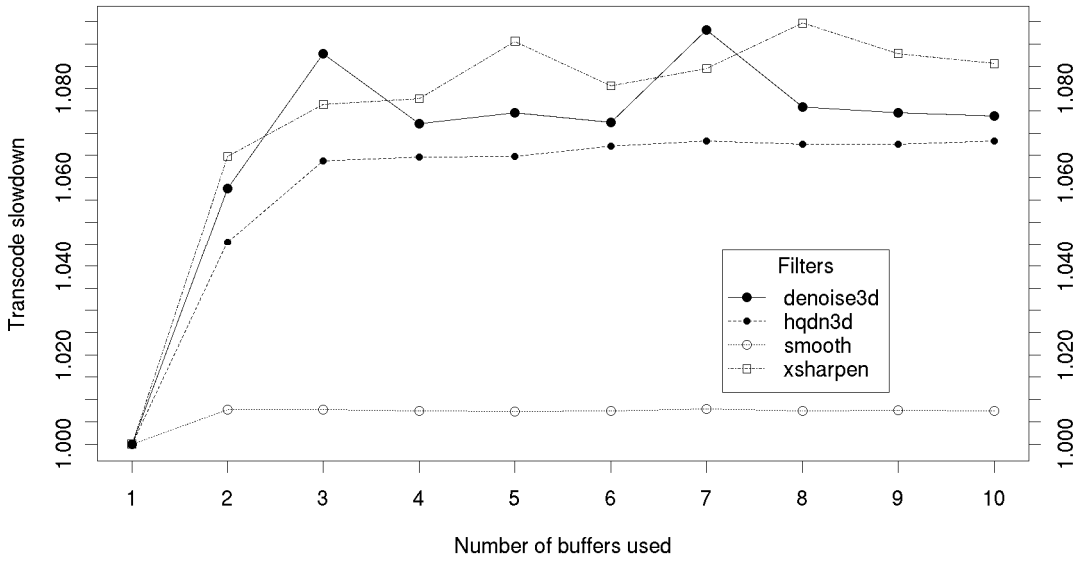

Figure 3.22: *Effect of the number of buffers used in Transcode on its performance with one filter on Platform 1*

Another possibility was to try to fix the synchronization in Transcode, using condition variables instead of sleep. This was eventually done, even though we have realized that changing as complex an application as Transcode is will be tricky and our results might be questionable. After the change, Transcode CPU usage was no longer low even for a small number of buffers. Then, the experiment measuring effect of number of buffers was executed again with the fixed version, using utime and real time, which yielded similar results. As Figure 3.23 shows for Platform 1, the number of buffers has no longer any significant effect on Transcode duration. This suggests that the data cache sharing effect on Transcode performance is insignificant, and the differences were due to imprecise measurement of utime because of the sleeps.

With the fixed Transcode, we have also repeated the first experiment on Platform 1, measuring the effects of cache sharing on filter combinations. We did not expect any significant difference, because the experiment uses the default of 10

buffers which uses the CPU well even without fixed synchronization. However, as Table 3.5 shows, the slowdown of filter combinations became negligible in most cases, and even negative in few cases. Still, we believe that the previous measurement with unmodified Transcode could not be affected by imprecise measurement. It is possible that the synchronization fix changed the thread scheduling order to one that does not exploit cache sharing as much.

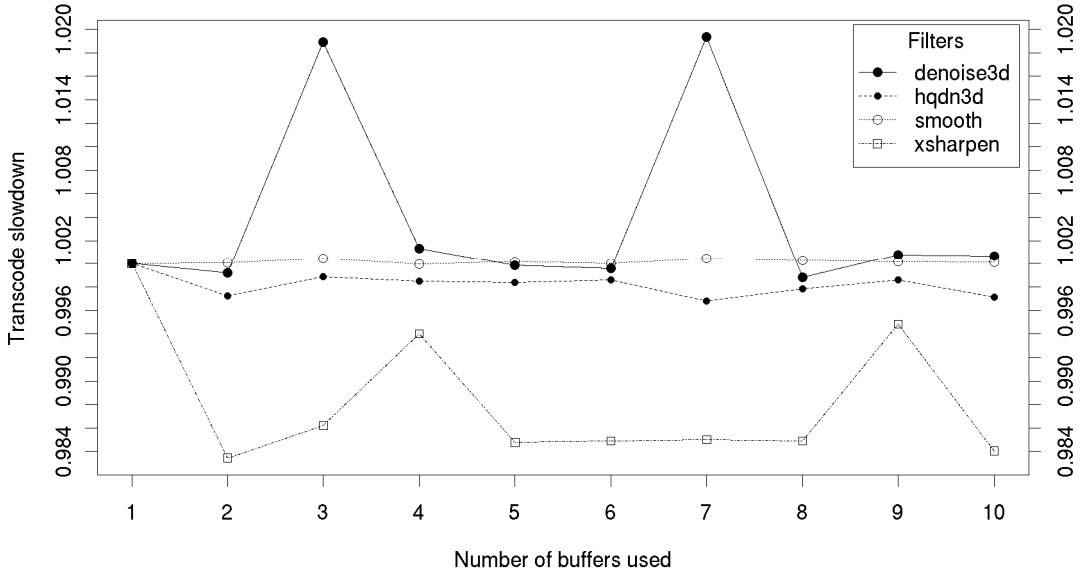

Figure 3.23: *Effect of the number of buffers used in Transcode with fixed synchronization on its performance with one filter on Platform 1*

$r_{x,y}$ [%]	**denoise3d**	**hqdn3d**	**smooth**	**xsharpen**
denoise3d	-0.8 ± 0.3	-1.1 ± 0.4	0.0 ± 0.0	-0.2 ± 0.2
hqdn3d	-0.3 ± 0.3	-0.4 ± 0.5	-0.1 ± 0.1	-0.4 ± 0.2
smooth	0.0 ± 0.0	-0.1 ± 0.1	0.1 ± 0.0	0.0 ± 0.0
xsharpen	1.3 ± 0.3	0.1 ± 0.5	0.1 ± 0.0	N/A

Table 3.5: *Slowdown/speedup of Transcode with combinations of two filters, using Transcode with fixed synchronization on Platform 1*

To reduce the randomness in thread scheduling, we also further modified the synchronization in Transcode so that the decoder, filter and encoder threads always process all available buffers before signaling the next thread in the pipe. Then we repeated the experiments with this modified Transcode.

The results of the experiment with filter combinations on Platform 2 are shown in Table 3.6. We can see that the results are roughly similar to the results with unmodified Transcode on the same platform (Table 3.4). The results from Platform 1 are presented in Table 3.7, still very different from the results with unmodified Transcode on the same platform. The speedup of xsharpen in combination with the denoise3d and hqdn3d filters is similar to the speedup on Platform 1.

$r_{x,y}$ [%]	denoise3d	hqdn3d	smooth	xsharpen
denoise3d	0.3 ± 0.8	-1.5 ± 0.8	0.0 ± 0.3	-5.7 ± 0.7
hqdn3d	4.3 ± 0.6	-0.3 ± 0.8	-0.6 ± 0.2	-4.4 ± 0.5
smooth	-0.1 ± 0.2	-0.6 ± 0.2	0.0 ± 0.2	-1.2 ± 0.2
xsharpen	0.6 ± 0.8	-0.2 ± 0.5	-1.2 ± 0.2	N/A

Table 3.6: *Slowdown/speedup of Transcode with combinations of two filters, using Transcode with modified synchronization on Platform 2*

$r_{x,y}$ [%]	denoise3d	hqdn3d	Smooth	xsharpen
denoise3d	-0.3 ± 0.2	-0.7 ± 0.8	0.0 ± 0.0	-3.6 ± 0.1
hqdn3d	0.1 ± 0.6	-0.4 ± 0.8	0.0 ± 0.1	-3.2 ± 0.4
smooth	0.0 ± 0.0	0.1 ± 0.1	0.0 ± 0.0	-0.6 ± 0.0
xsharpen	0.1 ± 0.1	-0.1 ± 0.4	-0.6 ± 0.0	N/A

Table 3.7: *Slowdown/speedup of Transcode with combinations of two filters, using Transcode with modified synchronization on Platform 1*

As the last experiment, the effect of number of buffers on Transcode with one filter was repeated with the modified synchronization. The results from Platform 2 are shown in Figure 3.24, results from Platform 1 are similar. We can see some slight performance improvement as the number of buffers increases, especially with the denoise3d filter. However, this might be attributed to the less overhead of thread synchronization rather than cache sharing.

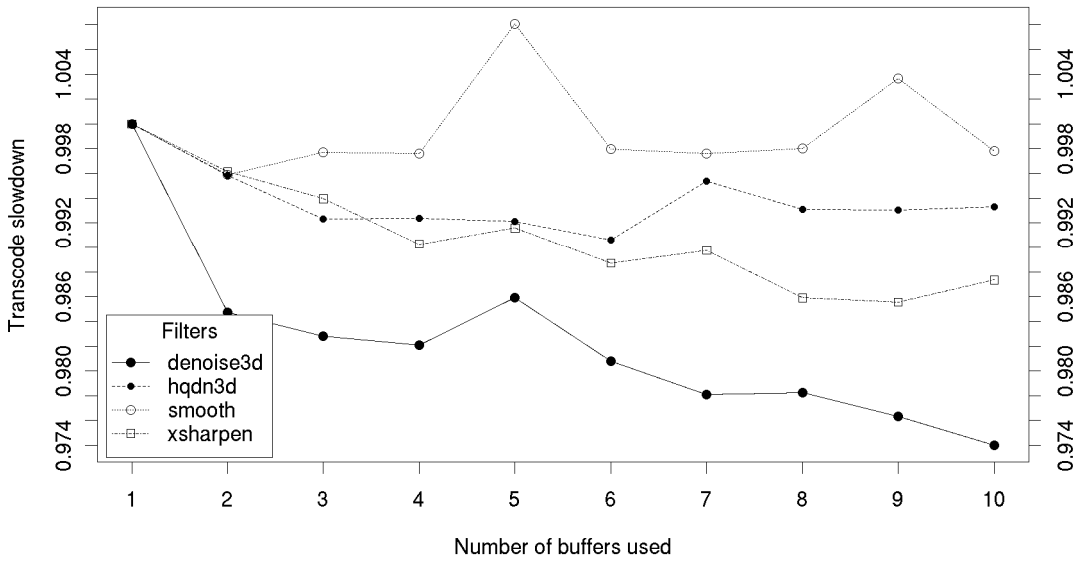

Figure 3.24: *Effect of the number of buffers used in Transcode with modified synchronization on its performance with one filter on Platform 2*

3.3 File systems

In this chapter, we will perform benchmarks that measure the file read and write durations. We will test several scenarios working with a varying number of files simultaneously, which we expect to have resource sharing effects, and compare the results with durations measured when working with each file alone, to determine the slowdown. The experiments are implemented in the RIB framework, which is using standard UNIX functions (*read*, *write*) to perform the file operations with a 32 KB buffer, filled with random data when writing.

In each experiment, first, a given number of files of given size (two parameters) is written, measuring the write duration. Also given is the write method - the files are written either one after another (we will refer to this method as *separate* writing), or simultaneously. In the second case, the files are opened all together and writes are interleaved using a fixed order – a block of data is written to the first file, then to the second file, after the last file another block is appended to the first file and so on. The block size for this interleaved writing is given as a parameter. We will call this method as *interleaved* writing. Note that interleaved writing where the block size equals to the file size would be identical to the separate writing.

After writing, the *sync* function is called to flush all delayed writes to the disk, and the file buffers are dropped from the memory by writing to the *drop_caches* pseudo-file in the *proc* kernel interface. This ensures that subsequent reads are not already cached. Then, the duration of the read operation on the files is measured. Similarly to writing, reading can also be either *separate* or *interleaved*, using the read block size as a parameter.

Experiment: Files.1
Purpose: Determine the slowdown of interleaved reading of separately written files.
Platforms: 1, 2, 3
Measured: Interleaved and separate read duration after separate writes.
Parameters: files written and read: 2 – 4; file size: 64 MB, 128 MB, 256 MB; writing mode: separate; read block size: 32 KB, 1 MB, 16 MB
Expected results: Interleaved reading should be slower than separate reading, because the buffer cache in the system memory and the disk cache are shared, and the disk needs to seek between the files. Read-ahead could reduce this seeking, because files are read sequentially. Lower block sizes should result in more rapid seeking and thus yield more significant slowdown. Increasing the number and size of files extends the area occupied on the disk, which can prolong the latency of seeks. In addition, buffers of more and larger files occupy more memory and therefore can make read-ahead perform worse.
Actual results: The observed slowdown of interleaved reading for all combinations of parameters is presented in Table 3.8 for Platform 1 and Table 3.9 for Platform 3. We can see that the most significant slowdown (up to 172% on Platform 3) occurs on both systems with 32 KB block size, where seeking should be most frequent. Interleaving megabyte blocks yields 35-40% slowdown on Platform 1, which is still significant, while 16 MB blocks are large enough to make the impact negligible. We can see that the results with the 32 KB block size are roughly similar

on all systems, but there is some difference with 1 MB block size (the omitted results from Platform 2 are roughly fit between the two shown platforms). The difference between 32 KB and 1 MB block sizes suggests that read-ahead buffer size is less than 1 MB. The number and size of files seems to matter only with the 32 KB block size. Doubling the number of files from two to four can add up to 39% to the slowdown (with 256 MB files). File size has less effect, increasing the size from 64 MB to 256 MB adds 14% with four files.

files	file size	read block size		
		32 KB	1 MB	16 MB
2	64 MB	1.73 ± 0.09	1.36 ± 0.02	1.01 ± 0.01
	128 MB	1.90 ± 0.06	1.36 ± 0.03	1.00 ± 0.02
	256 MB	1.88 ± 0.04	1.35 ± 0.02	1.02 ± 0.02
3	64 MB	2.06 ± 0.05	1.36 ± 0.03	1.00 ± 0.02
	128 MB	2.22 ± 0.05	1.41 ± 0.06	1.03 ± 0.05
	256 MB	2.42 ± 0.05	1.36 ± 0.04	1.03 ± 0.01
4	64 MB	2.28 ± 0.13	1.40 ± 0.03	1.01 ± 0.01
	128 MB	2.39 ± 0.06	1.36 ± 0.02	1.02 ± 0.02
	256 MB	2.59 ± 0.08	1.39 ± 0.03	1.03 ± 0.00

Table 3.8: *Slowdown of interleaved reading compared to separate reading; separately written files, Platform 1*

files	file size	read block size		
		32 KB	1 MB	16 MB
2	64 MB	1.58 ± 0.09	1.22 ± 0.04	1.00 ± 0.04
	128 MB	1.64 ± 0.11	1.22 ± 0.09	1.02 ± 0.08
	256 MB	1.76 ± 0.04	1.27 ± 0.03	1.02 ± 0.02
3	64 MB	1.93 ± 0.10	1.22 ± 0.04	1.01 ± 0.03
	128 MB	1.97 ± 0.06	1.23 ± 0.03	1.01 ± 0.04
	256 MB	2.33 ± 0.05	1.31 ± 0.03	1.02 ± 0.03
4	64 MB	2.17 ± 0.18	1.21 ± 0.11	1.01 ± 0.08
	128 MB	2.42 ± 0.09	1.25 ± 0.04	1.01 ± 0.03
	256 MB	2.72 ± 0.12	1.30 ± 0.06	1.01 ± 0.05

Table 3.9: *Slowdown of interleaved reading compared to separate reading; separately written files, Platform 3*

Experiment: Files.2
Purpose: Determine the slowdown of interleaved writing.
Platforms: 1, 2, 3
Measured: Duration of interleaved and separate writing.
Parameters: files written: 2 – 4; file size: 64 MB, 128 MB, 256 MB; writing write block size: 32 KB, 1 MB, 16 MB
Expected results: The slowdown is expected to be less significant than for interleaved reading, because the files can be just written interleaved on the disk, without need to seek between them. Additionally, writes are accumulated in the memory buffers and flushed in larger sequential blocks.
Actual results: Practically no slowdown was observed on Platforms 1 and 2, even with 32 KB block size and four 256 MB files. Results from Platform 3, presented in Table 3.10 even show notable speedup with more than two 256 MB files and all block sizes, which is strange. A possible explanation could be suboptimal delayed write performance with separated writing.

files	file size	write block size		
		32 KB	1 MB	16 MB
2	64 MB	0.96 ± 0.21	1.00 ± 0.06	1.02 ± 0.18
	128 MB	0.99 ± 0.09	1.00 ± 0.10	0.99 ± 0.08
	256 MB	1.02 ± 0.17	1.01 ± 0.15	1.00 ± 0.07
3	64 MB	0.98 ± 0.15	1.01 ± 0.08	1.00 ± 0.11
	128 MB	0.99 ± 0.12	0.97 ± 0.13	0.98 ± 0.06
	256 MB	0.87 ± 0.08	0.94 ± 1.18	0.85 ± 0.15
4	64 MB	1.01 ± 0.18	1.00 ± 0.20	0.97 ± 0.19
	128 MB	0.98 ± 0.09	0.98 ± 0.14	1.00 ± 0.13
	256 MB	0.89 ± 0.08	0.87 ± 0.05	0.88 ± 0.09

Table 3.10: *Strange speedup of interleaved writing compared to separate writing on Platform 3*

To determine the impact of delayed write, the experiment was also repeated using synchronous writing[4], which disables delayed writing. The results for Platform 1 are in Table 3.11 (results from Platform 2 are similar). It shows that even with 1 MB block size, there is some slowdown, which means that the delayed write is probably flushing the buffers in larger blocks than 1 MB. Note that the slowdown is much less significant than the slowdown of interleaved reading, which means there is indeed less disk seeking thanks to the physical data interleaving. Because the slowdown with 32 KB blocks is still significant (about 20% with four files), we can assume that the physical interleaving is using larger blocks.

The results of synchronous writing on Platform 3 in Table 3.12 have questionable stability, but indicate that the speedup of interleaved asynchronous writes could have been caused by delayed writes, as there is no notable speedup of synchronous writes.

[4] Passing the *O_SYNC* flag to the *write* syscall.

files	file size	write block size		
		32 KB	1 MB	16 MB
2	64 MB	1.06 ± 0.02	1.03 ± 0.02	1.00 ± 0.02
	128 MB	1.06 ± 0.01	1.04 ± 0.01	1.00 ± 0.02
	256 MB	1.07 ± 0.02	1.03 ± 0.02	1.01 ± 0.01
3	64 MB	1.12 ± 0.02	1.03 ± 0.03	1.00 ± 0.01
	128 MB	1.13 ± 0.02	1.03 ± 0.02	0.99 ± 0.02
	256 MB	1.15 ± 0.02	1.03 ± 0.02	1.00 ± 0.02
4	64 MB	1.18 ± 0.02	1.02 ± 0.02	1.01 ± 0.01
	128 MB	1.20 ± 0.02	1.03 ± 0.01	1.00 ± 0.01
	256 MB	1.22 ± 0.02	1.04 ± 0.01	1.00 ± 0.01

Table 3.11: *Slowdown of interleaved synchronous writing compared to separate synchronous writing on Platform 1*

files	file size	write block size		
		32 KB	1 MB	16 MB
2	64 MB	1.06 ± 0.80	1.02 ± 0.75	1.00 ± 0.73
	128 MB	1.06 ± 0.80	1.02 ± 0.76	1.01 ± 0.74
	256 MB	1.05 ± 0.86	1.01 ± 0.81	1.00 ± 0.79
3	64 MB	1.10 ± 0.83	1.03 ± 0.74	1.01 ± 0.73
	128 MB	1.08 ± 0.87	1.01 ± 0.78	1.01 ± 0.79
	256 MB	1.03 ± 0.78	0.94 ± 0.69	0.95 ± 0.70
4	64 MB	1.13 ± 0.88	1.02 ± 0.76	1.01 ± 0.75
	128 MB	1.11 ± 0.92	1.01 ± 0.81	1.01 ± 0.80
	256 MB	1.07 ± 0.81	0.97 ± 0.70	0.95 ± 0.69

Table 3.12: *Slowdown of interleaved synchronous writing compared to separate synchronous writing on Platform 3*

Experiment: Files.3

Purpose: Determine the residual effect of interleaved written files on separate reading.

Platforms: 1, 2, 3

Measured: Separate reading after interleaved and separate writing.

Parameters: files written/read: 2 – 4; file size: 64 MB, 128 MB, 256 MB; writing mode: separate, interleaved; write block size: 32 KB, 1 MB, 16 MB; reading mode: separate

Expected results: Because interleaved writing results in physically interleaved (fragmented) files on the disk, the fragmentation should prolong the successive read duration due to seeking between the fragments. The slowdown should not be too significant because the seeking should occur in only one direction and skip relatively small areas depending on number of files and block size.

Actual results: The strongest impact was observed on Platform 1 with results shown in Table 3.13. As expected, the slowdown is relatively small (not more than 8%). We can see that there is almost no difference between 32 KB and 1 MB block size, which supports the hypothesis from the previous experiment, that physical interleaving is larger than the megabyte write block size. Results from Platform 2 were similar to the results from Platform 1, while the observed impact on Platform 3 was less significant – 2% on average.

files	file size	write block size		
		32 KB	1 MB	16 MB
2	64 MB	1.05 ± 0.03	1.04 ± 0.01	1.03 ± 0.02
	128 MB	1.06 ± 0.02	1.03 ± 0.02	0.96 ± 0.04
	256 MB	1.08 ± 0.02	1.09 ± 0.03	0.99 ± 0.02
3	64 MB	1.07 ± 0.02	1.04 ± 0.03	1.01 ± 0.02
	128 MB	1.05 ± 0.02	1.05 ± 0.01	1.01 ± 0.03
	256 MB	1.07 ± 0.01	1.06 ± 0.01	1.02 ± 0.01
4	64 MB	1.08 ± 0.04	1.08 ± 0.03	1.03 ± 0.02
	128 MB	1.06 ± 0.02	1.04 ± 0.02	1.01 ± 0.02
	256 MB	1.04 ± 0.02	1.03 ± 0.01	1.00 ± 0.00

***Table 3.13:** Slowdown of separate reading caused by interleaved writing, compared to separate writing; Platform 1*

Experiment: Files.4
Purpose: Determine the residual effect of interleaved written files on interleaved reading.
Platforms: 1, 2, 3
Measured: Interleaved reading after interleaved and separate writing, using the same block size for both interleaved reading and writing.
Parameters: files written/read: 2 – 4; file size: 64 MB, 128 MB, 256 MB; writing mode: separated, interleaved; reading mode: interleaved; read/write block size: 32 KB, 1 MB, 16 MB
Expected results: As we already saw, interleaved writing results in physically interleaved files. Were the physical fragments exactly as large and ordered as the writes were issued, interleaved reading of the same block size would be as fast as separated reading of separately written files. We have seen that this is not the case because physical fragments are usually larger. Still, interleaved files should result in faster interleaved read, because the fragments are smaller than the whole files, reducing the seek latency.

Actual results: The results from Platform 3 in Table 3.14 show that interleaved writing can really improve interleaved reading performance quite notably, although still far from the performance of separated reading of separately written files. Results from Platform 2 are similar. However, the results with 1 MB block size on Platform 3 shown in Table 3.15 are unexpected – the performance is slightly worse. It may be due to the fact that more fragmented files need more – possibly not continuous –

sectors for metadata, increasing the number of seeks, which outweighs the benefit of smaller seeks.

files	file size	read/write block size		
		32 KB	1 MB	16 MB
2	64 MB	0.92 ± 0.03	0.99 ± 0.03	1.01 ± 0.05
	128 MB	0.91 ± 0.05	1.01 ± 0.07	1.01 ± 0.07
	256 MB	0.82 ± 0.02	0.94 ± 0.03	1.02 ± 0.03
3	64 MB	0.90 ± 0.03	1.00 ± 0.03	1.02 ± 0.03
	128 MB	0.85 ± 0.03	0.97 ± 0.03	1.03 ± 0.04
	256 MB	0.73 ± 0.02	0.92 ± 0.02	1.00 ± 0.02
4	64 MB	0.87 ± 0.06	1.02 ± 0.09	1.06 ± 0.08
	128 MB	0.75 ± 0.03	0.96 ± 0.03	0.99 ± 0.03
	256 MB	0.71 ± 0.03	0.94 ± 0.04	1.00 ± 0.05

Table 3.14: *Speedup of interleaved reading of interleaved files compared to separately written files on Platform 3*

files	file size	read/write block size		
		32 KB	1 MB	16 MB
2	64 MB	1.02 ± 0.05	1.05 ± 0.04	1.02 ± 0.00
	128 MB	0.99 ± 0.06	1.03 ± 0.02	1.00 ± 0.01
	256 MB	1.03 ± 0.02	1.08 ± 0.03	1.01 ± 0.02
3	64 MB	0.99 ± 0.03	1.01 ± 0.03	1.02 ± 0.01
	128 MB	0.98 ± 0.03	1.02 ± 0.05	0.97 ± 0.05
	256 MB	0.89 ± 0.03	1.03 ± 0.03	1.02 ± 0.01
4	64 MB	0.98 ± 0.05	1.03 ± 0.02	1.02 ± 0.01
	128 MB	0.97 ± 0.02	1.04 ± 0.02	1.02 ± 0.01
	256 MB	0.86 ± 0.02	1.01 ± 0.02	0.99 ± 0.00

Table 3.15: *Effect of interleaved reading of interleaved files compared to separately written files on Platform 1*

The previous experiments have determined the effect of file system sharing on files that were written and read sequentially, which is a very common scenario, used for example by http or ftp servers with HTML pages and other transferred files, or for writing log files and processing them in reporting. The following benchmark experiments will read the files randomly – such scenario is common when accessing indexed data, e.g. in databases. Blocks of given size will be randomly chosen and read, which means that some blocks may be read more than once and others not at all. As in the previous experiments, files will be written and read either separated or interleaved by blocks.

Experiment: Files.5

Purpose: Determine the slowdown of interleaved random reading of separately written files.

Platforms: 2, 3

Measured: Separated random reading and interleaved random reading after separated writing.

Parameters: files written/read: 2 – 4; file size: 64 MB, 128 MB, 256 MB; writing mode: separated; read mode: random separated, random interleaved; read block size: 32 KB, 1 MB, 16 MB

Expected results: Random interleaved reading should not affect performance compared to random separated reading as much as in the case of interleaved and separated sequential reading, because random reading of a file already causes disk seeking and does not benefit from read-ahead prefetching like the sequential reading. Interleaved reading should just increase the length of seeks and cause more buffer cache sharing, which is not a big problem without read-ahead.

Actual results: As expected, the slowdown was much smaller than with sequential reading – compare results from Platform 3 in Table 3.16 with Table 3.9. However, the impact is still notable with 32 KB block size and 256 MB files. Results from Platform 2 are very similar.

files	file size	read block size		
		32 KB	1 MB	16 MB
2	64 MB	1.05 ± 0.05	0.98 ± 0.11	1.02 ± 0.17
	128 MB	1.05 ± 0.03	0.97 ± 0.08	1.02 ± 0.17
	256 MB	1.16 ± 0.02	1.05 ± 0.04	1.02 ± 0.12
3	64 MB	1.07 ± 0.02	1.02 ± 0.05	0.98 ± 0.11
	128 MB	1.12 ± 0.02	1.03 ± 0.06	1.03 ± 0.14
	256 MB	1.26 ± 0.01	1.09 ± 0.04	1.02 ± 0.09
4	64 MB	1.11 ± 0.10	1.02 ± 0.08	0.98 ± 0.15
	128 MB	1.26 ± 0.02	1.04 ± 0.04	1.00 ± 0.12
	256 MB	1.36 ± 0.07	1.17 ± 0.03	1.08 ± 0.09

Table 3.16: *Slowdown of interleaved random reading compared to separate random reading of separately written files on Platform 3*

Experiment: Files.6

Purpose: Determine the slowdown of random separated reading of interleaved written files.

Platforms: 2, 3

Measured: Random separated reading of separately and interleaved written files.

Parameters: files written/read: 2 – 4; file size: 64 MB, 128 MB, 256 MB; writing mode: separated/interleaved; write block size: 32 KB; read mode: random separated; read block size: 32 KB, 1 MB, 16 MB

Expected results: As in previous experiment, the slowdown should be smaller when compared to the analogous experiment with sequential reading (Files.3), because seeking already occurs due to random reading, and fragmented files should increase it only a little.

Actual results: Results from Platform 3 are shown in Table 3.17, results from Platform 2 are similar. Surprisingly, speedup is observed with 32 KB read block size, without good explanation. It could be somehow related to the interleaved write performance, which was also better than sequential on Platform 3 (see Table 3.10). However, this was not observed on Platform 2.

files	file size	read block size		
		32 KB	1 MB	16 MB
2	64 MB	0.98 ± 0.02	1.02 ± 0.08	1.09 ± 0.20
	128 MB	0.99 ± 0.04	1.07 ± 0.09	1.07 ± 0.20
	256 MB	0.96 ± 0.02	1.00 ± 0.04	1.00 ± 0.11
3	64 MB	0.99 ± 0.02	1.05 ± 0.05	1.05 ± 0.14
	128 MB	0.97 ± 0.01	1.04 ± 0.05	1.09 ± 0.13
	256 MB	0.94 ± 0.01	0.99 ± 0.03	1.08 ± 0.10
4	64 MB	0.97 ± 0.08	1.01 ± 0.10	1.05 ± 0.15
	128 MB	0.94 ± 0.02	1.01 ± 0.04	1.11 ± 0.14
	256 MB	0.90 ± 0.04	0.92 ± 0.03	1.00 ± 0.08

Table 3.17: *Slowdown of random separate reading caused by interleaved writing, compared to separate writing; Platform 3*

4 Evaluation and applicability of the results

To summarize the results of our benchmarks, we will now list the major observations that have emerged during the experiments. The list will be roughly divided into three parts based on the observed performance impact - from the most serious down to negligible impact. Then, we will discuss applicability of our findings for improving benchmarking and performance modeling precision

The results with most serious performance impact (more than 10% performance change) are:

- Data and code cache trashing impact on the duration of FFT transformations with FFTW library and buffer size up to 128 KB – the duration can be prolonged by a factor of 3 with data trashing, and by a factor of 3.5 with code trashing on a Pentium 4 processor. Peak slowdown on Athlon 64 is around 2 for both types of trashing. Transforming larger buffers generally results in smaller performance impact.
  For the purpose of performance prediction, this situation could be generalized as employing software components, which perform optimized memory intensive computations in a memory buffers of size up to the quarter of the L2 unified cache. If more of such components are deployed on the same processor, and their execution frequently switched, the performance could be significantly impacted.
- Cache trashing impact on the performance of LZW implementation – trashing prolongs the compression at most by 60%. In contrast to buffer size effect in FFTW, working with larger dictionaries results in larger performance impact on LZW, as long as the dictionary fits in the L2 cache. The most important factor is frequency of trashing.
  We can therefore expect similar impact on components that randomly access some internal memory structure, such as hash table, occupying memory roughly the size of the L2 cache, or less. The impact will be most significant when bursts of operations on this structure, short enough not to access most of it more than once, are interleaved with operations of other components.
- Interleaved sequential reading of multiple separately written files is slower by a factor of 2.6 compared to separate reading with four 256 MB large files and 32 KB read block size. Even with 1 MB read block size, the slowdown is more than 30%.
  This naturally concerns multiple active components that perform sequential reading of files, to e.g. send them over network or process the data in the files.
- Random sequential reading of multiple separately written files can yield more than 30% slowdown with 32 KB block size. This could be generalized as multiple components retrieving small indexed data from the files.

The situations with less serious but still measurable impact were:

- Cache trashing impact on duration of a simple FFT implementation, which yielded slowdown of at most 1.10 for data trashing and 1.06 with code trashing on the Athlon 64 processor. This is pretty low compared to impact on

FFTW, which performs the same operation but more efficiently. The possible lesson we can learn is that using simple component prototypes of processor and memory intensive components for performance prediction may yield imprecise results, if the component implementation is to be later optimized, or more code is added to it, increasing the code cache impact.

- The effect of interleaved written files on reading. It is common knowledge, that file fragmentation should be kept low for better performance. This is true when the files are read separately and sequentially, where fragmented files cause up to 8% slowdown. However, interleaved sequential reading can benefit from fragmented files in some cases, and surprisingly even separated random reading, where we observed up to 10% improvement.
- Strange effects of more intensive trashing causing performance improvement of FFTW performance with large separate buffers for input and output. The reason is still not clear.
- Unexpected speedup of interleaved writes on one of the platforms.

Finally, the situations where negligible impact was observed.

- FFTW performance with buffers equal to or larger than the L2 cache size. A component intensively working with such amount of memory will itself cause so many cache misses, that the effect of cache misses due to sharing becomes negligible.
- Slowdown of LZW where trashing occurred infrequently. Because task scheduling in the operating system is even less frequent, we can assume that negligible cache sharing due to the scheduling should occur between multiple active tasks, provided they can spend their whole quantum working and not frequently blocking and yielding the CPU.
- The experiments with Transcode. After first promising results, bad threading was found that tampered with the following experiment. After this was fixed, no significant results were obtained. Still, the other experiments with cache sharing indicate that applications with workload similar to the one of video processing applications should significantly exploit effects of cache sharing, and a better implementation for experiments could be eventually found.

To conclude, the results of our experiments have shown situations where resource sharing can affect performance significantly both in the case of processor caches and file systems, and thus should be taken into account when benchmarking or modeling performance. The rest of this chapter will discuss possible ways to deal with resource sharing.

4.1 Reducing benchmarking infrastructure overhead

When performing a benchmark or monitoring performance of a software system, it is desirable not to influence the measured performance with the infrastructure code that captures and stores the results of measurements. Our experiments suggest how this code may affect the measured durations:

- Measuring very short operations might be relatively significantly influenced by the overhead of acquiring timestamps or other performance counters.

- Capturing and storing results of performance indicators during performance monitoring trashes the processor caches – the influence depends on the frequency of taking samples, amount of the code executed for that and on the amount and locality of data stored.
- Keeping all results in memory until the benchmark or monitoring is finished occupies memory that could be otherwise used by the measured system. When all memory is occupied, swapping may occur. If the system works with files, it has less memory for file buffers, resulting in worse performance.
- Storing the results of monitoring in files influences performance of system that is also working with files – the effect depends on the frequency and block size of writes performed by the monitoring infrastructure, with respect to the frequency and block size of file operations of the monitored system.

To minimize some of these effects, we propose few, quite simple advices. They are mostly intuitive, and probably already followed in many experiments:

- Use a pre-allocated fixed size memory buffer to store the results, allocate whole memory pages with *mmap()* and not *malloc()* or *new*. This will ensure the buffer is aligned on cache line size and memory page size. Place all results linearly in this single buffer if possible, instead of more buffers simultaneously. The goal is to reduce number of TLB and cache entries accessed and thus evicted when storing results.
- If the measured system is able to use all available memory, either directly or through file buffers, the test platform should have more memory available than the target platform, with the extra memory reserved for the infrastructure code and the buffer for storing the results.
- Flushing the buffer to a file should be done at once when the buffer is full, to minimize its frequency. The writes should be unbuffered and synchronous to avoid using extra memory and delayed writes. The results file should be best located on a separate disk, file system or at least pre-allocated if on the same file system to avoid fragmentation. Samples obtained shortly after the flush until the system stabilizes again should be discarded as a wamp-up phase.

There are, however, cases where the effects of resource sharing cannot be minimized enough, such as detailed code profiling through instrumentation, which results in frequent trashing of caches. In these cases, computing the overhead from the difference in overall performance of untouched and instrumented code is imprecise – the overhead is not additive. Experiments similar to those in this thesis that would measure the effect of cache trashing on parts of the system could yield better results.

4.2 Performance models

Dealing with resource sharing in performance models is studied in detail in our paper [3]. The following summarizes the paper and points out how the experiments performed in this thesis contribute to the method proposed there.

The paper considers performance models describing component interaction through atomic actions, represented either by requests in Queuing Networks, transitions in Petri Nets, actions in Stochastic Algebras, or other formalisms. The

atomic actions in the model represent function invocations on software components, and duration of these invocations needed to solve the model are typically determined by benchmarking. The problem is that resource sharing which influences the duration of function invocations is mostly captured neither in the software model nor during the benchmarks. However, results from this thesis show that resource sharing may have significant effect and thus may affect the precision of such model considerably.

An obvious solution would be to use incorporate models of shared resources into the performance model. This would however result in very complex and hard-to-solve models, or would be infeasible because the model for a resource either does not exist, is based on a different formalism, or focuses only on some special feature of the resource.

Another option is to measure the durations of function invocations under resource sharing, which means trashing the resource similarly to the experiments in this thesis. However, to approximate the trashing that would occur in the modeled system, one would need to know how much the resource is used (this is called degree of resource usage in the paper), to adjust the intensity of trashing accordingly. The problem is that this degree may not be known until the model is solved - for example, the number of parallel requests being served at a time depends on how quickly the system is able to finish processing of the requests.

The paper proposes using a combined performance model, which consists of separate performance and resource models. Any of the common performance models omitting resource sharing may be used without modifications. The resource model is used to determine duration of function invocation based on the degree of resource usage, and may be replaced by benchmarking if no such model is available.

The combined model is solved by iterating between the performance and resource models – the performance model is first populated with durations of actions corresponding with sensible initial degree of resource usage, and the results are used to update this degree for next iteration, which is repeated until the results eventually stabilize. Separating the two models ensures that the complexity of performance model does not increase, and iterating solves the problem of mutual dependency between the results of performance and resource model (or benchmarking, as outlined above).

The resource model, which combines performance models of shared resources, is an important part of this approach. It gives a relation between degree of resource usage and the duration of a measured function invocation, which is exactly what has been obtained empirically in most of the experiments in this thesis. The number of files opened for concurrent reading or writing is one example of degree of resource usage influencing the read or write duration.

The results of the experiments can also tell us which resources we should include in the model and which are not so important. For example, we should not need to include CPU caches in cases where the degree of their usage is determined only by the number of running threads (i.e. concurrently serviced requests), because the results of LZW benchmark suggest that cache trashing with frequency comparable to the frequency of task switching has minimal effect on performance.

Ideally, we would like to have such performance model of a resource that would need only few key values to be obtained empirically by benchmarking and simply

compute the duration from given degree of resource usage. The results of our and similar experiments may be used to construct these models, or to show that we do not understand the details of a resource well enough to model it properly, like in the case of cache trashing improving performance of FFT transformation. In these cases, we can still use the same benchmarking experiments to obtain durations for each degree of resource usage needed during the iteration, if we accept the higher cost of benchmark experiments compared to solving a model.

The iterative combined performance model approach has been demonstrated on a proof-of-concept example in the paper [3]. The Common Component Modeling Example (CoCoME) [8], which resembles an enterprise information system that keeps tracks of products sold by a chain of stores, has been chosen as a reasonably complex test platform, implemented using contemporary middleware technologies.

The performance model in this example is based on activities that make up or interfere with a sale, to answer the question of how many concurrent sales can the system handle. LQN has been chosen as the formalism, and thus solving the model provides also queue lengths and processor utilization values used to determine the degree of resource usage for the resource model.

The resource model provides the average durations of two atomic actions – the stock item query and the sale booking update. Benchmarking experiments have suggested that these durations are mostly influenced by two resources that are thus described in the model, namely the Derby database cache and the system memory. Durations of the two atomic actions are measured both when the data is cached and fetched, together with the additive unit overhead of swapping. Further experiments with Hibernate and Derby yield the details about the Derby cache and system memory usage dependencies on the number of concurrent transactions, products and stores. Equations for computing the probabilities of cached and swapped operation are thus based on this knowledge and have the same parameters. The number of concurrent queries is computed from the queue length, which is output of the performance model during the iteration.

On average, the combined model has converged in three steps, giving results in the form of average throughput in stock items per second and average sale length. Real benchmark of the modeled scenario was also performed, and the results compared with those from the model, which showed the model, although simple, was reasonably precise in predicting the effects of resource sharing.

5 Related work

This thesis deals with benchmarking experiments, hardware and virtual resources, performance modeling and prediction. Large amount of related work exists on each of these topics. Instead of a comprehensive list, we will divide the related work into several categories along with few examples of each.

Performance modeling. Performance of the software systems is often predicted by formal models employing Queuing Networks or similar formalisms, with values for the parameters of these models determined by benchmarking. We can divide the existing approaches into those that deal with resource sharing, and those that omit resource sharing.

An example of a work that incorporates resources into the performance model is [17], which models a whole multiprocessor system with its cache hierarchy, PCI subsystem and DMA transfers for the purpose of predicting performance of static content serving via HTTP. While the precision of this model is excellent, the cost of solving is high, because many parts of the model have to resort to simulation.

Because of the high complexity of models incorporating resources, resource sharing is often omitted. [21] models the performance of J2EE messaging components in isolation from the application implementation, omitting the possible resource sharing between them. [32] models the performance of an EJB-based system, and the model is calibrated with data from only a single-client workload, omitting resource sharing that would occur between multiple clients running in parallel. In [20], the benchmark obtaining parameters to solve the model has a fixed number of client threads, which captures only the resource sharing that occurs under this degree of parallelization.

Resources. The performance of hardware and system resources such as the processor caches is evaluated by benchmarks [11], simulation [12], or via analytical models [1]. The latter two are often used to evaluate proposed performance enhancements, such as trace caches [27] or optimal replacement policies [2].

Evidence however shows that due to the relations between some resources, the performance enhancements of a resource should not be evaluated in isolation. [24] demonstrates the impact of increasing cache sizes on the efficiency of the DRAM page miss rate.

In the area of file systems, similar kind of interaction between kernel prefetching and buffer cache replacement is studied in [5]. The authors show that the prefetching can significantly affect the relative performance of the algorithms in terms of hit ratio and actual disk I/O. The results suggest that prefetching and cache replacement should be studied together and not separately as previously done.

Perhaps closest to the experiments performed in this thesis is work that evaluates performance on SMT processors where resource sharing is obvious and known problem. The paper [13] evaluates the performance of Java applications executed simultaneously in pairs on the Hyper-Threading Pentium 4 processor through benchmarks and performance counters. The results show that the impact of the trace cache sharing is so high that pairs of trace cache intensive codes can perform significantly slower than when scheduled serially.

6 Conclusion

The first goal of this thesis was to analyze the possible effects of sharing of several components on code performance. For a selected set of typically shared hardware and software resources, the details of the resource usage and possible effects of resource sharing were discussed.

The second goal was to determine the effects of resources sharing empirically. For this purpose, a number of benchmarking experiments which focused on processor caches and file system was designed, implemented and executed on several hardware platforms as a native code in the Linux operating system.

The artificial experiments with caches promised large potential for possible performance impact due to cache sharing. Experiments with the FFT and LZW implementations and artificial cache trashing determined the effects of sharing on a practical code and its dependency on the code's memory intensiveness and frequency and intensity of trashing. Based on these dependencies, the slowdown of the measured code duration ranged from very significant to negligible. In some cases, even slightly positive effect of trashing was observed, for reasons that remained unclear. One important observation is that cache trashing of frequency comparable to the typical preemptive task switching frequency has insignificant effect.

Further experiments focused on scenarios with frequently switching short operations such as audio-video processing. The experiments were performed on one such implementation – Transcode. Unfortunately, initially promising results were rendered questionable due to poor implementation of synchronization and not reproducible after an attempt to fix it. Further results with fixed Transcode showed only very little effects which cannot even be attributed to cache sharing for sure.

The experiments with file system sharing determined how working with multiple files at once influences read and write performance, and how this impact depends on the number and the size of the files and the interleaving block size. The slowdown again ranged from significant to negligible, and there were situations where speedup was observed. Residual effects on performance were also demonstrated.

The third goal was to analyze applicability of the results for improving the precision of performance monitoring and modeling. Several guidelines for reducing the overhead of performance monitoring were proposed, based on the dependencies of resource sharing impact observed in the experiments. The approach of incorporating performance sharing into performance modeling proposed in [3] was then discussed, pointing out the important role of the type of experiments performed in this thesis for the approach.

6.1 Open issues and future work

There are several open issues regarding the experiments that have been performed. The reasons of positive performance impact of cache trashing in some of the experiments with FFT have not been clearly explained. The experiments with Transcode did not yield sufficient evidence of the effect of cache sharing. One of the tasks for future would be to try the experiments with a different implementation of audio/video processing or a different kind of workload that consists of frequent switching of short operations.

Another kind of open issues is related to the experiments that have not been performed. After an agreement with the thesis supervisor, experiments on the Windows platforms and JVM and CLI processes, mentioned in the official thesis assignment, have been omitted – especially due to the volume of the results of the experiments that have already been performed on other platforms. The second possible task for future could therefore be to repeat the experiments on the omitted platforms for completeness, and to experiment with resources that were not included in this thesis.

Probably the most promising future work could focus on further advancements in the area of resource sharing performance modeling. Possible tasks include developing methods and tools to aid identifying situations with heavy resource sharing from the behavior specification, or improving the precision of the performance prediction. The focus on component systems also suggests prospective resources for further experiments in the previously proposed task of further benchmarking experiments, such as middleware technologies and databases.

7 References

[1] Agarwal, A., Hennessy, J., and Horowitz, M. *An analytical cache model*, ACM Trans. Comput. Syst. 7, 2, pp. 184-215, ACM 1989

[2] Al-Zoubi, H., Milenkovic, A., and Milenkovic, M.: *Performance evaluation of cache replacement policies for the SPEC CPU2000 benchmark suite*, in Proceedings of the 42nd Annual Southeast Regional Conference, pp. 267-272, ACM 2004

[3] Babka, V., Decky, M., Tuma, P.: *Resource Sharing in Performance Models*, to appear in Proceedings of the 4th European Performance Engineering Workshop (EPEW 2007), Springer 2007

[4] BIOS and Kernel Developer's Guide for AMD Athlon™ 64 and AMD Opteron™ Processors, Publication #26094, AMD 2006

[5] Butt, A. R., Gniady, C., Hu, Y. C.: *The Performance Impact of Kernel Prefetching on Buffer Cache Replacement Algorithms*, IEEE Transactions on Computers, vol. 56, no. 7, pp. 889-908, IEEE 2007

[6] Cao, M., Ts'o T. Y., Pulavarty, P., Bhattacharya, S., Dilger, A., Tomas, A.: *State of the Art: Where we are with the Ext3 filesystem*, in Proceedings of the Linux Symposium, Ottawa, ON, Canada, 2005

[7] Carr, R. W., Hennessy, J. L.: *WSCLOCK – a simple and effective algorithm for virtual memory management*, in Proceedings of the Eighth ACM Symposium on Operating Systems Principles (SOSP 1981), pp. 87-95, ACM 1981

[8] CoCoME, http://agrausch.informatik.uni-kl.de/CoCoME

[9] Compress, http://ncompress.sourceforge.net/

[10] Frigo M., Johnson S.G.: *FFTW*, http://www.fftw.org

[11] Gee, J., Hill, M., Pnevmatikatos, D., Smith, A. J.: *Cache Performance of the SPEC92 Benchmark Suite*, IEEE Micro 13, 4, pp. 17-27, IEEE 1993

[12] Hill, M. D., Smith, A. J.: *Evaluating Associativity in CPU Caches*, IEEE Transactions on Computers, Volume 38, Issue 12, pp. 1612-1630, IEEE 1989

[13] Huang, W., Lin, J., Zhang, Z., and Chang, J. M.: *Towards Pairing Java Applications on SMT Processors*. in Proceedings of the 13th IEEE international Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2005), pp. 7-14, IEEE 2005

[14] IA-32 Intel Architecture Optimization Reference Manual (Order Number 248966), Intel

[15] Kalibera, T., Bulej, L., Tuma, P.: *Benchmark Precision and Random Initial State*, in Proceedings of the 2005 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS 2005), pp. 853-862, SCS 2005

[16] Kalibera, T., Bulej, L., Tuma, P.: *Quality Assurance in Performance: Evaluating Mono Benchmark Results*, in Proceedings of the Second International Workshop on Software Quality (SOQUA 2005), pp. 271-288, Springer 2005

[17] Kant K., Sundaram C. R. M.: *A Server Performance Model for Static Web Workloads*, ISPASS'00, IEEE 2000

[18] Kim, J. and Hsu, Y.: *Memory system behavior of Java programs: methodology and analysis*, in Proceedings of the 2000 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2000), pp. 264-274, ACM 2000

[19] Lea, D.: *A memory allocator.* Unix/Mail, Hanser Verlag 1996

[20] Liu Y., Fekete A., Gorton I.: *Predicting the Performance of Middleware-Based Applications at the Design Level*, in Proceedings of the 4th International Workshop on Software and performance (WOSP), ACM 2004

[21] Liu Y., Gorton I.: *Performance Prediction of J2EE Applications Using Messaging Protocols*, CBSE'05, Springer 2005

[22] Mayer, R., Buneman, O: *FFT Benchmark*, ftp://ftp.nosc.mil/pub/aburto/fft

[23] McDougall, R., Mauro, J.: *Solaris Internals, (2nd edition)*, Prentice Hall PTR 2006

[24] Mekhiel, N.M.: *The Effect of an Intercepting Cache on Performance of Fast Page and Cache DRAM*, in Proceedings of the ISCA 13th International Conference on Computers and Their Applications (CATA 1998), pp. 360-363, ISCA 1998

[25] PAPI: A Portable Interface to Hardware Performance Counters, http://icl.cs.utk.edu/papi/

[26] Pettersson, M.: *Perfctr*, http://user.it.uu.se/~mikpe/linux/perfctr/

[27] Rotenberg, E., Bennett, S., and Smith, J. E.: *Trace cache: a low latency approach to high bandwidth instruction fetching*, in Proceedings of the 29th Annual ACM/IEEE international Symposium on Microarchitecture, pp. 24-35, IEEE 1996

[28] RUBiS, http://rubis.objectweb.org

[29] TPC Benchmarks, http://www.tpc.org/information/benchmarks.asp

[30] Transcode, http://www.transcoding.org/

[31] Tuma, P., Buble, A.: *Open CORBA Benchmarking*, in Proceedings of the 2001 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS 2001), SCS 2001

[32] Xu J., Oufimtsev A., Woodside C. M., Murphy L.: *Performance Modeling and Prediction of Enterprise JavaBeans with Layered Queuing Network Templates*, SIGSOFT SEN 31(2), ACM 2006

[33] Yu, H., Kedem, G.: *DRAM-Page Based Prediction and Prefetching*, 2000 IEEE International Conference on Computer Design (ICCD'00), IEEE 2000