OS-Controlled Cache Predictability for Real-Time Systems

Jochen Liedtke* Hermann Härtig** Michael Hohmuth**

Abstract

Cache-partitioning techniques have been invented to make modern processors with an extensive cache structure useful in real-time systems where task switches disrupt cache working sets and hence make execution times unpredictable. This paper describes an OS-controlled application-transparent cache-partitioning technique. The resulting partitions can be transparently assigned to tasks for their exclusive use. The major drawbacks found in other cache-partitioning techniques, namely waste of memory and additions on the critical performance path within CPUs, are avoided using memory coloring techniques that do not require changes within the chips of modern CPUs or on the critical path for performance. A simple filter algorithm commonly used in real-time systems, a matrix-multiplication algorithm and the interaction of both are analysed with regard to cache-induced worst case penalties. Worst-case penalties are determined for different widely-used cache architectures. Some insights regarding the impact of cache architectures on worst-case execution are described.

* IBM T. J. Watson Research Center, 30 Saw Mill River Road, Hawthorne, NY 10532, and GMD — German National Research Center for Information Technology, SET-RS, 53754 Sankt Augustin, Germany, email: jochen@watson.ibm.com

** TU Dresden, Dept. of Computer Science, 01062 Dresden, Germany, email: hermann.haertig@inf.tu-dresden.de

1 Introduction

The primary distinguishing requirement for real-time systems is that deadlines must be met. To meet deadlines, worst-case execution times need to be known.

Access times in modern memory hierarchies vary greatly. Pentium's first-level cache allows 2 accesses per cycle. Cache misses in first-level and second-level caches may require 50 or more cycles if the cache line to be replaced needs to be written back before it can be filled. These large differences make the approach of using disabled caches in deriving worst-case estimates completely obsolete.

Realistic worst-case execution times in systems with extensive caches are increasingly hard to determine since they depend on reference locality which is strongly influenced by thread switches and address-space switches. In particular, the impact of externally-triggered context switches (interrupts) is nearly unpredictable.

For these reasons, caching architectures are currently not commonly used in real-time systems. However, if increasingly real-time applications find their way into standard workstations, realistic worst-case behaviour needs to be asserted.

This observation has attracted a number of researchers to invent cache-partitioning techniques that allow to dedicate partitions of caches to applications. However, the currently proposed techniques either require changes in the critical path of a CPU's implementation or waste large amounts of memory.

This paper describes a cache-partitioning technique that isolates tasks from cache disruptions by other tasks and the operating system. The technique enables partitioning caches and assignment of partitions to tasks. We describe an experimental environment to measure the worst-case effect of cache disruptions due to task switches in an asynchronous system. Section 5 discusses experimental results of applying the partitioning technique to some simple algorithms commonly used in real-time systems.

2 Rationale

2.1 Predictability

In real-time systems, the optimization criterion is not the average but the *worst-case* execution time. Since a real-time task has to meet its deadline under all circumstances, always enough resources for the worst-case must be scheduled. The real-time load is thus limited by the sum of worst-case execution times, worst-case memory consumption etc.

The closer the worst-case cost $c_{\max}$ of a task gets to its best-case cost $c_{\min}$, the better predictable the system is. We use $c_{\max}/c_{\min}$ to describe the *unpredictability* of a task. Since we always discuss time costs, the unpredictability can be regarded as the worst-case slow-down factor.

A reduced unpredictability (increased predictability) of cached memory systems is required for better real-time properties of such systems, in particular if time-sharing and real-time applications should coexist in the system. We concentrate on the unpredictability that arises from concurrent execution of multiple tasks, not on the influence of of different input data to a single task.

The reader is assumed to be familiar with currently used cache architectures. Detailed descriptions can be found in [13; 2; 16; 14; 4].

2.2 The operating system's view

It is an *application-specific* problem to make a single task run-alone predictable. In contrast, it is an *operating-system* problem to preserve this predictability for a given set of multiple run-alone-predictable tasks. Conventional scheduling techniques can be applied if the tasks behave sufficiently deterministic in time. However, all methods rely on the assumption that the tasks' execution times are independent of each other: given two tasks α and β, it is assumed that an interleaved execution with $2k$ context switches between them will execute in $T_{\alpha|\beta} = T_\alpha + T_\beta + 2kT_s$, where T_α and T_β are the run-alone execution times of the tasks and T_s is the context-switch time.

Introducing caches violates this assumption, since in an interleaved execution, α and β compete for cache lines. In one activation interval, α might use cache lines which have been originally used by β. In its next activation interval, β has to refill the lines and perhaps even to save them to memory prior to refill. Accordingly, β imposes cache-miss overhead on α's next activation interval. Although all is done by hardware, "invisible" to the programs, it needs a substantial amount of time.

The additional overhead, the interference cost I, depends on the dynamic cache usage of α and β and the points in time $(t_1, t_2, \ldots t_{2k})$ when context switching occurs: $T_{\alpha|\beta} = T_\alpha + T_\beta + 2k\,T_s + I\,(\alpha, \beta, t_1, t_2, \ldots t_{2k})$. In practice, more than two tasks have to be coordinated by the operating system. Accordingly, the complexity of the I-function increases.

In an open real-time system, even run-alone unpredictable and unforeseen interactive (non real-time) tasks have to be included. For example, a workstation being part of a video conference should be able to combine audio and video tasks with interactive spreadsheet handling. Here, the real-time components must execute predictably, independently of the non-real-time tasks. Traditionally, a priority scheme ensures that the real-time tasks get resources first. The other tasks get only time slices and memory frames which are not required for the real-time tasks. Unfortunately, the priority strategy relies also on the same assumption of non-interference; respectively it requires to bound the interference cost reasonably.

Concluding, a basic problem is to manage the entire system in such a way that I is limited by a fixed and safe upper bound being reasonably small.

Since we aim at general-purpose operating systems, closed as well as open real-time systems, any solution has to be transparent to user programs and must accept any given binary. Furthermore, it must work for dynamically changing sets of tasks with partially unknown behaviour.

This suggests reducing or at least limiting the interference. Ideally, the additional cost for task α is at most $I'(\alpha)$ per context switch towards α and *does not depend on the other tasks.* Even if β was unknown and the complete interference cost $I\,(\alpha, \beta, t_1, t_2, \ldots t_k) \leq k\,I'\,(\alpha) + k\,I'\,(\beta)$ therefore would be unpredictable, α could be scheduled in such a way that it terminates in time $2\,T_\alpha \;+\; k\,I'\,(\alpha) \;+\; 2kT_s$ or earlier. Simply select a time slice length of $T_\alpha/k + I'(\alpha) + T_s$. Then allocate the odd slices to α and the even slices to β where the last time slice may be arbitrary long to cover all remaining execution time of β.

3 Cache partitioning

The basic idea to reduce interference is *cache partitioning:* ensure that tasks α and β use always different cache lines. If the intersection of both cache working sets is empty, there is no interference at all, $I'(\alpha) = 0$. Since the associated cache line (respectively its set) of an object is determined by some bits of its memory address, this goal can be achieved by placing code and data of both tasks accordingly in memory.

3.1 Cache partitioning by main-memory management

An operating system can to a certain extent determine the *physical* address of code and data by mapping virtual to physical memory. The nice features: it is completely transparent to user programs, i.e. can be applied to any program, and can be done dynamically, i.e. can be changed at any time. Its disadvantages: since it is based on mapping virtual pages to physical page frames, it can only be applied to physically-indexed caches and offers only page granularity.

We describe the technique for second-level caches which usually are physically indexed. A page size of 2^p divides a direct-mapped cache in cache banks of also 2^p bytes each. To index an element within a bank, the least significant p bits are used. If the cache has a size of 2^c, the next $c - p$ bits in the address select the cache bank.[1] The remaining most-significant part of the address is used for tag comparison.

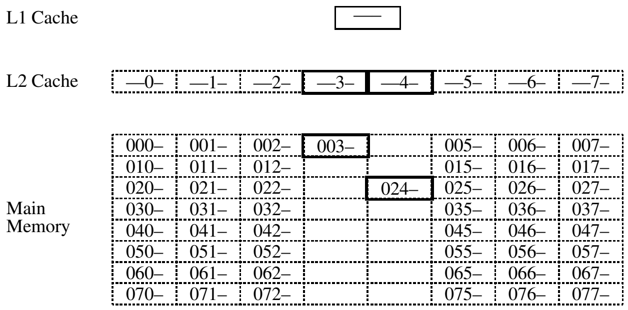

Figure 1: *Cache Partitioning.*

The direct-mapped second-level cache in figure 1 consists of 8 banks. Only the second least-significant digit of the physical addresses (octal numbers denote the physical page frame number in the figure) controls the bank selection. Thus bank 2 can hold only cache lines of physical page frames 002, 012, 022... *This divides main memory in a set of classes, called* colors*, whose members are all physical page frames which select the same cache bank.*

Hence, cache conflicts can only occur between page frames of the same color. To avoid cache conflicts between any two tasks, it is sufficient to ensure that no color is used by both tasks. In figure 1, all page frames of the colors 3 and 4 except frame 003 and 024 are unused. Cache banks 3 and 4 therefore work exclusively for these two page frames.

A cache bank can be assigned to a task, a group of tasks, a region of an address space, or any other unit by assigning colors of main memory to that unit. By assigning one or more colors exclusively to one unit, this unit can be protected from second-level cache conflicts with other units.

A μ-kernel like L4 [8] permits even physical memory management at user level. Thus the above-mentioned strategy could easily be implemented by a memory server which gives frames of special colors only to dedicated tasks and all other frames to the pagers which implement usual timesharing. To avoid interference with the μ-kernel, the colors corresponding to its code (12K) have not been used for special allocation.

However, the main drawback as already observed by Wolfe [15] remains: If a certain percentage of the complete cache is to be assigned to a task, the same percentage of the main memory has to be reserved for that task. If a time-critical algorithm needs 50% of the available cache to meet its deadlines, also 50% of the main memory must be reserved for it.

3.2 Free coloring to separate cache and main memory management

Wasting main memory by cache partitioning can be avoided if the operating system can assign any color to any physical main-memory frame.

Free coloring [7] is based on the idea to provide multiple physical addresses per frame. The same memory frame number a is addressed by physical addresses $a00_{xxx}$, $a01_{xxx}$, $\ldots a77_{xxx}$, where $_{xxx}$ denotes the offset part of the address. Hence, the physical frame can be used under any color from 0 to 77.

The simple and intuitive implementation is shown in the following figure, where *cpu bus* denotes the processor's physical address bus and *memory bus* denotes the physical address bus of the memory system. This requires only proper wiring on a motherboard. It does neither affect cache addressing nor virtual-to-physical address translation. Performance is not reduced since the described mechanism is static and lies outside the critical path of a memory architecture.

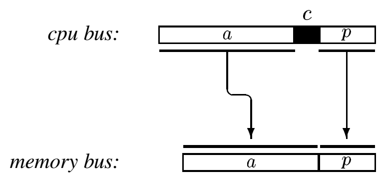

Figure 2: *Free Coloring.*

Instead of mapping a virtual address v_{xxx} to the physical address a_{xxx}, the operating system maps v_{xxx} to $a03_{xxx}$ when it decides use the memory frame of number a with color 03.

Note that the mentioned mechanism can be implemented without changing the secondary cache architecture and with standard memory chips. However, c-bits wider second-level tags are required for accessing the same amount of memory as in a conventional system. For the same reason, main memory is limited to 2^{n-c} bytes, where n is the physical-address width.

[1] For an n-way set-associative cache, use $n2^c$ as cache size. Correspondingly, the bank size is $n2^p$.

3.3 Applications of the technique

There are several ways to apply the partitioning techniques (whether combined with free coloring or not). The first and obvious application is to isolate one task or a set of cooperating tasks. Thus the worst-case execution time of the task can be determined without respect to cache usage of other tasks and the operating system.

Another type of application is a long running task that has to meet a deadline and is preempted by other tasks for a known number of times. The effect of the interrupt tasks' cache accesses can be isolated by assigning a separate cache partition to the interrupt handler tasks. (Of course, this does not apply for the very first interrupt or the startup phase of the application. Then, the assigned cache partition is still cold. From the predictability point of view, these effects are either negligible or can be eliminated by preexecuting the interrupt handlers respectively the application when they are installed.)

The experiments described in section 5 deal with both scenarios and their interaction.

Partitioning caches effectively reserves partitions of a cache for a unit that needs certain worst-case assurances. The reserved partition is no longer available for the remainder of the system and may considerably slow it down. However, the technique can be used to determine the amount of caches needed that sufficiently bound worst-case execution times. Such results then can be used to determine the size of caches for a given real-time application.

4 Expectations

There is an obvious counter-argument against the effectiveness of the described method: it cannot eliminate the unpredictability of the first-level cache. Further topics to be discussed include memory consumption and cache locking.

4.1 Cache penalties

It is a widespread misconception that programs in the worst case execute as slow as if the caches were disabled. This is not true. Most programs' worst-case execution times are much better with caches than without, even if neither data nor instructions are accessed twice.

Caches increase memory bandwidth significantly even in no-hit situations. First, cache lines are larger than one word, typically 8 words. Second, due to multi-word-wide buses and burst memory accesses, loading an 8-word cache line requires much less time than 8 single-word loads; factors of 2 to 4 are realistic. Third, caches often use bypass techniques so that the processor can resume execution as soon as the first word (which induced the cache miss) arrives. The remaining cache fill can execute in parallel to instruction execution.

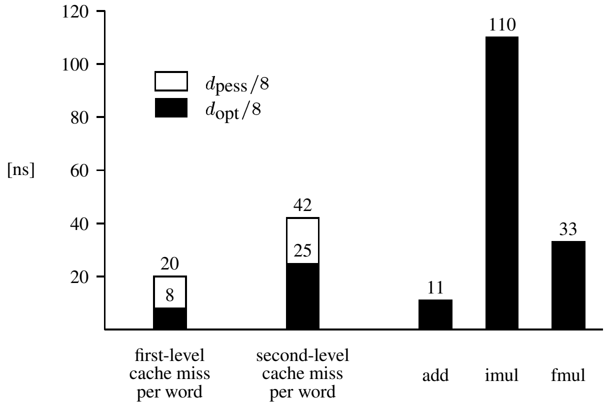

Figure 3: *Expected Delays on Pentium, 90 MHz.*

Figure 3 illustrates cache miss penalties on a 90-MHz Pentium with write-through caches. The optimistic penalty d_{opt} is the delay for a cache miss if the memory system was not busy, i.e. the time between detecting the miss and delivering the first word. The pessimistic penalty d_{pess} is the maximum delay if the memory system is busy on a refill while the miss occurs. For both cases, we assume that the bus is currently not used for writes and that no other bus masters are active.

We have to take into consideration that a program usually does not access only one word per cache line. For the addressed real-time applications, it seems reasonable that every word of an accessed cache line will be read during program execution. Thus the average delay per data word will be between $d_{opt}/8$ and $d_{pess}/8$ for 8-word caches. Figure 3 relates these average delays to the time required for integer addition integer multiplication and floating-point multiplication.

Unprecise estimations

A typical real-time task receives a signal, then does some calculations and finally falls asleep, waiting for the next signal. In the startup phase, directly after receiving the signal, the cache-miss cost will be dominated by instruction-cache misses. Once all required code is loaded, the working phase begins.

In many cases, the working phase will be a simple loop. For a first estimation, we assume a short loop needing two words from memory and 150 ns execution time per iteration, i.e. a multiplication, some additions, compares and jumps. If only first-level cache misses occur, we can optimistically expect a maximum delay of $2 \times 8/150 \approx 11\%$ and pessimistically a delay of $2 \times 20/150 \approx 26\%$.

Note that for a given program, the real maximum delay is completely determined by its own cache access pattern.

In particular, it can be determined by the method described in section 5, and it will not float between the optimistic and the pessimistic value. If the loop is constructed in such a way that at most one cache miss per iteration can occur, a cache-line fill ($8 \times 20 = 160$ ns) can nearly be completed before the next cache miss. So we can expect a worst-case delay of about 12%.

In the presence of second-level cache misses, an optimistic estimation for the worst-case delay gives $2 \times 25/150 \approx 33\%$. However, since the according $d_{\text{pess}} = 8 \times 42 = 336$ ns exceeds the 150 ns of one iteration, additional time might be required every second cache miss. If cache misses occur in two adjacent iterations (while the next 6 iterations hit in the cache), we could expect additional $(336 - 150)/8/150 \approx 15\%$.

All these calculations are unprecise. Nevertheless, rough estimations are very important to understand the interaction in the entire system and to get an impression about the expectable order of magnitude. Furthermore, the experimental results can then be used to validate or invalidate our understanding. The presented rough estimations show:

1. First-level cache penalties will be significantly lower than second-level penalties. This prediction contrasts to the well-known fact that first-level cache misses are more important for *average* performance than second-level misses. The relative cost (miss:hit) is lower for a secondary cache (typically 3:1) than for a primary cache (typically 10:1). However, the *worst-case* performance is determined by the absolute cost which is three times higher for a second-level miss than for a first-level miss.

2. Worst-case delays on Pentium will be surprisingly low, 30–40% during the working phase. As we will see later, much higher penalties must be expected for second-level write-back caches.

Loading a 100-word program can lead to penalties from 2 μs (first-level misses only) up to 4 μs. (For comparison, interrupt latency of the L4 μ-kernel is about 2 μs.) This startup overhead is responsible for the unpredictability for runs of few loop iterations, although the worst-case absolute time delay remains small. The startup overhead can be ignored for larger numbers of iterations.

4.2 Further penalty sources

Cache partitioning can improve predictability but cannot eliminate all uncertainties. For example, the number of interrupts in a real-time system needs to be known for penalty estimates. Occurrences can be controlled by techniques to disable interrupts and by scheduling techniques. In modern μ-kernel-based systems, interrupts are transformed to messages and task-scheduling techniques (priorities, deadline scheduling) are used to control them.

A single TLB miss is approximately twice as expensive as a single cache miss. Nevertheless, worst-case TLB cost is limited by the number of pages which belong to the working set. Many processors use their data caches for parsing the page tables in the case of a TLB miss. Accordingly, they profit from partitioning the second-level cache.

Worst-case memory penalties are caused by unpredictable memory refresh cycles, DMA and in shared-memory multiprocessor systems even bus locks by other processors. Refresh typically degrades secondary-cache fill and write back. The maximum overhead depends on the memory architecture. 20% will be a safe upper bound for almost all architectures, 10% for most architectures. This value has to be multiplied by the worst-case miss rate. Cache partitioning makes miss rates highly predictable and thus limits refresh penalties as well. The same holds for delays caused by simultaneous activity of other processors or DMA. In particular, a program which fits completely into its secondary-cache partition will never suffer from the mentioned penalties.

5 Experiments

The first experiments consider the scenario of a high-priority real-time task running for frequent short intervals. In between these active intervals, other tasks execute unpredictably. The objectives of the experiment are (a) to determine the worst-case delay imposed on the response time of the real-time task that is caused by *prior* cache usage of other tasks and the operating system and (b) to demonstrate the effectiveness of partitioning to limit this worst-case penalty in comparison to minimal-interference execution. First the minimal-interference execution time $c_{\min}$, then worst-case execution times $c_{\max}$ in partitioned and unpartitioned execution are determined

The second class of experiments considers a lower-priority real-time task being interrupted by a high-priority task at a constant rate. Again, we measure $c_{\min}$ and $c_{\max}$ for partitioned and unpartitioned execution. Finally, both experiments are combined.

5.1 Experimental environment

The experiments are based on the L4 μ-kernel. By means of a user-level memory server ("pager"), a real-time specific memory management policy has been implemented to partition the cache as explained in section 3.1. The mentioned pager offers an OS interface and an application interface:

		size	cache line	associativity	write strategy	colors
486	L1	8 K	16 B	4-way	through	1
33 MHz	L2	256 K	16 B	direct mapped	through	64
Pent	L1	8+8 K	32 B	2-way	through	1
90 MHz	L2	256 K	32 B	direct-mapped	through	64
Pent	L1	8+8 K	32 B	2-way	back	1
133 MHz	L2	256 K	32 B	direct-mapped	back	64
PPro	L1	8+8 K	32 B	4-way + 2-way	back	1
133 MHz	L2	256 K	32 B	4-way	back	16

Table 1: Cache Systems.

- The system operator or an automatic scheduler can allocate colors to tasks or group of tasks for exclusive use.
- Any application can determine the color of any of its pages whithin the set of colors that have been granted to it by the operating system for exclusive use.

Using the second interface is optional. Although partitioning will be done transparently in most cases, coloring of different objects inside the application can be customized for specifically cache-optimized alogrithms like the matrix multiplication described on section 5.4.

The experiments are done on machines based on Intel 486, Pentium and Pentium Pro CPUs (trademarks), all with first-level and second-level caches. Table 1 shows the different cache architectures. The chipset of the older (90 MHz) Pentium machine supports only the write-through cache policy whereas the newer Pentium machine (133 MHz) supports also write-back. Execution times are measured by a 1-μs-timer (486) respectively by the processor's timestamp register (Pentium, Pentium Pro) which counts processor cycles. Both timer and timestamp register can be read in user mode without OS intervention.

For the first set of experiments, a high-priority real-time task is activated with a constant rate. To eliminate interrupts as a source for the variation of the response time, interrupts are disabled during the active periods of the real-time task.

To determine $c_{\min}$ of the real-time task, all other applications are stopped and interrupts remain disabled. To determine $c_{\max}$, a *cache flooder* runs as a low-priority job: It accesses memory in a systematic pattern such that all available cache lines are accessed and modified. To ensure that the flooder in any case (even on the relatively slow 486) floods the complete cache between subsequent activations of the real-time task, we chosed an activation frequency of only 12.5 Hz. In this extreme situation, we get always worst-case behaviour. Higher frequencies up to 100 kHz show the same worst-case behaviour, but worst cases occur less frequently.[2]

The measurements were done without and with partitioning. In the first case, the flooder had access to all lines of the secondary cache, in the second case only to those that were not assigned to the real-time task. The real-time task had as many cache lines as it needed. The maximum times were taken as worst-case penalty.

5.2 Filter

The algorithm used in the real-time task is a digital filter. Digital filters are used to inhibit certain frequency components in a digitized signal. The used filter algorithm works as follows: At a discrete time step t, a sample x_t of a signal is fed into the filter. The filter uses an array of $n+1$ coefficients c_i and a buffer holding the n previous inputs. The filter output is computed as $\sum c_i x_{t-i}$. The number n and the coefficients c_i determine the characteristics of the filter. To vary the problem size and hence the cache load, an increasing n is used.

The diagrams in figure 4 show the unpredictability $c_{\max}/c_{\min}$. For better intuitive understanding, unpredictability of 1.5 is shown as "+50%". The measurements are made for various problem sizes (number of coefficients) from $n = 4$ up to $n = 8192$. Solid lines show the unpredictability (worst-case penalty) of unpartitioned execution, dotted lines show unpredictability of partitioned execution. Basically, dotted lines quantify the first-level cache influence.

Figure 4 shows the penalties of the filter running on 486, Pentium and Pentium Pro processors.

1. As expected, small problem sizes suffer under significantly larger penalties than larger sizes. Accordingly, the partitioning effect is much higher for small sizes.
2. The measured *unpartitioned* unpredictability is consistent with our expectations. For larger n, it is about +35% on Pentium.
3. The measured *partitioned* unpredictability is consistent with our expectations, e.g. +10% for $n = 256$. After reading 2×1024 data words, the cache is full and further reads suffer *predictably* from first-level misses: The current run always removes the data from the cache that is required for the next run.
4. Unpartitioned execution on Pentium Pro has substantially larger penalties than on Pentium and 486, even for large problems. First- and second-level cache on

[2] Only when the time left for flooding is no longer sufficient to access as much memory as the real-time task accesses per activation interval, the absolute worst-case costs can decrease. However note that flooding 1 K costs only 10 μs on the 90-MHz Pentium.

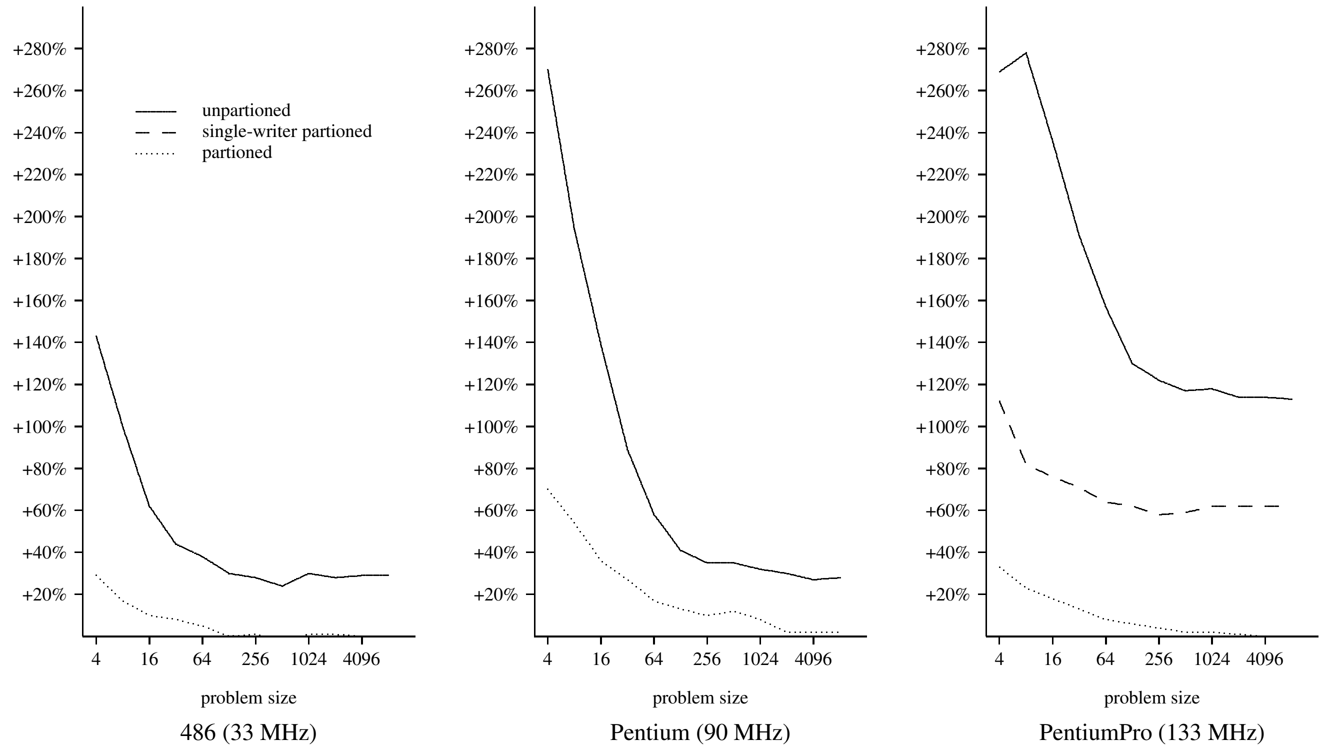

Figure 4: *Unpredictability* $c_{\max}/c_{\min}$ *(worst-case penalties) for filter.*

Pentium Pro use a write-back strategy so that in the worst case a cache line has to be written back to memory. Although write buffers permit to fill the cache line prior to write the old content, this feature has limited effect in burst situations. Furthermore, the memory supports burst-mode reads but no burst-mode writes. Writing back a cache line needs therefore 3 times longer than filling it.

5.3 Single-writer partitioning

We observed that write-back cache architectures are definitely more sensitive to worst-case penalties than write-through architectures. Therefore, it seemed reasonable also to experiment with a somehow weaker strategy.

Assume that color x is assigned to a real-time task. Different to strict partitioning, all page frames of color x which are not used by the real-time task can be used by other tasks, but *only read-only mapped*. Then other tasks can use the critical cache bank x, but they will never make it dirty. Pentium Pro permits to enforce a write-through strategy on a per-page-basis. Thus mapping pages read/write, but with write-through cache strategy, gives the same effect. The dashed line in figure 4 shows the worst-case penalty for that case. The worst-case penalty can be limited to about 60% for larger problem sizes.

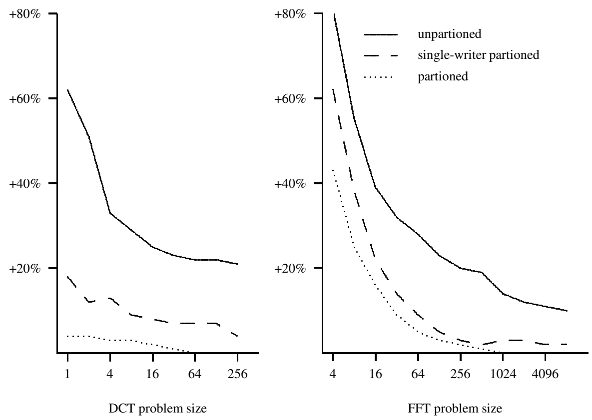

Figure 5: *DCT and FFT on Pentium Pro*

Two more compute-bound programs, discrete cosine transformation (DCT) and fast Fourier transformation (FFT), show similar behaviour (figure 5). However, the quantitative effects are not as high, since more computational cycles are required per memory access than in the filter.

5.4 Matrix multiplication

Now we look at a longer-running real-time task which is frequently preempted by one or more high-priority real-time tasks. As long-running task we use a highly-optimized matrix multiplication, $c := ab$, where a and b are 64×64 matrices of 64-bit floating-point values.

The Pentium's floating point unit has a 3-cycle latency per multiplication or addition. However, due to pipelining, the unit accepts one operation per cycle if no result is used earlier than 3 cycles after the corresponding operation started. The achieved performance for the innermost loop is about 4.5 cycles for $c_{ij} := c_{ij} + a_{ik}b_{kj}$ as long as not even first-level cache misses occur.

We furthermore optimized the memory/cache performance of the algorithm: First, the second matrix b is stored columnwise to enable stride-1 accesses. Second, we compute the matrix product in steps of 8×8-submatrices like shown in figure 6.

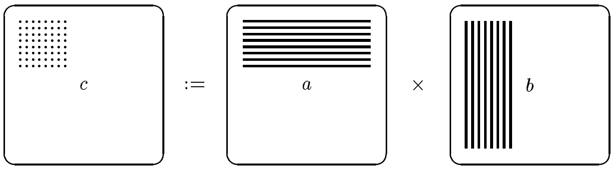

Figure 6: *Matrix multiplication.*

For computing submatrix $c_{1\ldots8,1\ldots8}$ rows $a_{1,} \ldots a_{8,}$ and columns $b_{,1} \ldots b_{,8}$ are needed. With proper alignment of a and b, the 8 rows respectively 8 columns occupy in each case exactly one page. Together, they make optimal use of the 8K first-level cache: In the run-alone case, any row or column is loaded only once into the cache so that the number of cache misses is definitely minimal. The resulting algorithm needs 5.51 cycles per basic step, i.e. $64^3 \times 5.51 \approx 1.44$ million cycles (10.9 ms) for the entire matrix multiplication on a 133-MHz Pentium.

Larger matrices profit in the same way when not only c but also a and b are subdivided into submatrices. With proper row/column alignment, this is consistent with row- respectively columnwise allocation, since blocking allocation is needed only in the cache, not in main memory.

A simple and effective method to avoid any cache interference would be to enable preemption only when switching to the next submatrix [12]: disable interrupts while a complete submatrix is computed. Unfortunately, this takes at least $64^2 \times 5.51$ cycles, approximately 170 μs. Since, usually, such long interrupt-disabled periods cannot be tolerated, we again have to use partitioning. We compare three different schemes:

1. *Unpartitioned:*
   The matrices are allocated sequentially in physical memory. For both source matrices together (32 K each), we use thus 16 colors. Since there is no partitioning, these colors are subject to flooding.

2. *Partitioned:*
   We are interested to minimize the number of exclusively allocated colors. Therefore, on Pentium, we request only 3 colors (out of 64) for the matrix-multiplication task. We associate the first color to matrix a, the second to b and the third color to the code. (The color of the result matrix is irrelevant: The cache does not support write allocation, and one write-through operation per 128 reads is negligible.) Due to its 4-way second-level cache, 1 color (out of 16) is sufficient on the Pentium Pro.

3. *Single-writer partitioned:*
   For the matrix-multiplication task, memory is allocated like in the partitioned case. However, the remaining page frames of the used colors are available read-only respectively write-through for flooding.

Figure 7 shows the measured unpredictability for (uninterrupted) timeslices from 25 μs to 1000 μs. For the experiment, first- and second-level caches are flooded after each timeslice expires. $c_{\max}$ is the matrix multiplication's cpu time under presence of these interruptions; $c_{\min}$ is the cpu time without any interruption. Each experiment is presented in two diagrams that use different scales and ranges for better illustration.

The results illustrate that partitioning can have substantial effects even when the allocated second-level cache is much smaller than the memory working set. Furthermore, they corroborate that write-back caches (which are in general more efficient) combined with single-writer partitioning behave nearly as well as write-through caches (which are in general less efficient).

5.5 Combining short and long running real-time tasks

In a further experiment, the long-running matrix multiplication is combined with an optimized floating-point version of the short-running filter. We measure the real time required for one matrix multiplication which is interrupted by the filter at rates from 2 kHz up to 20 kHz.

The filter uses 512 coefficients so that it accesses 2 pages per activation. In the *partitioned* case, the filter gets two colors different from the colors allocated to the matrix task.

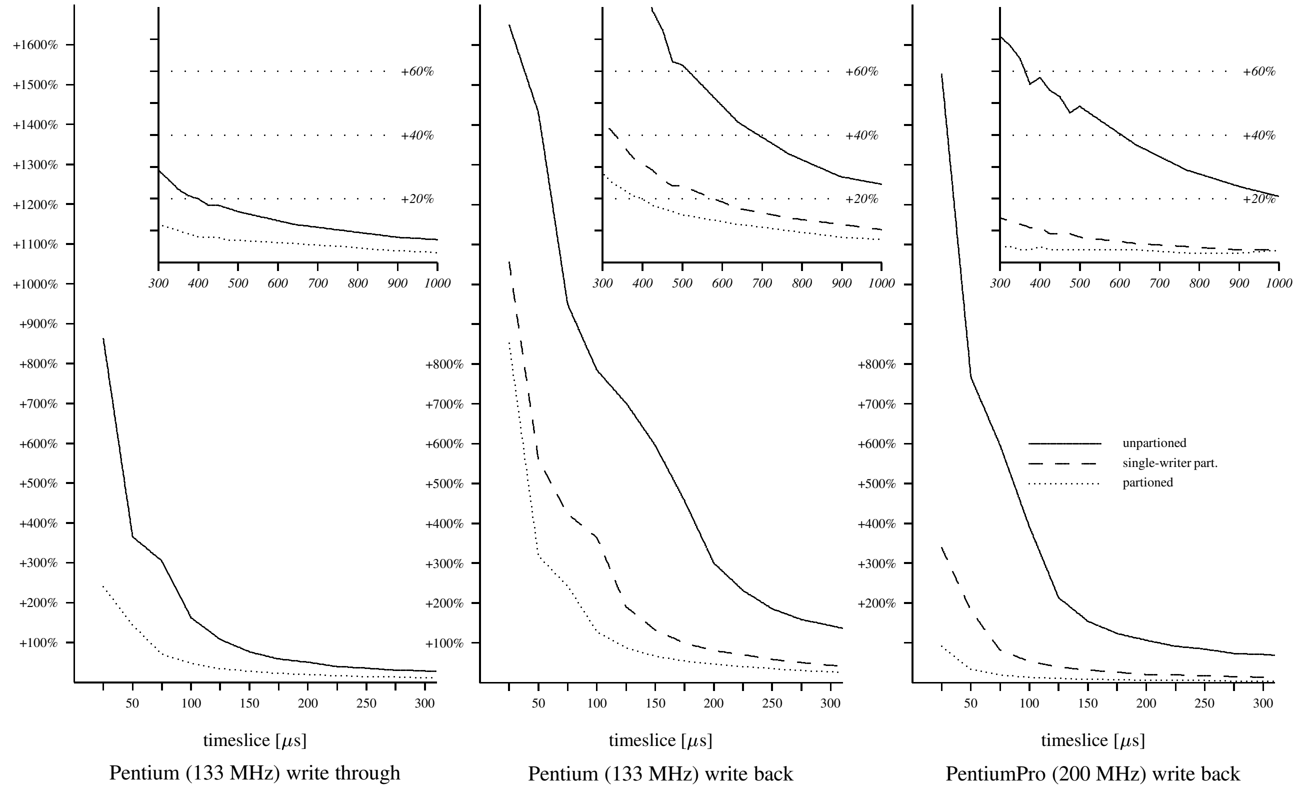

Figure 7: *Unpredictability* $c_{\max}/c_{\min}$ *for* 64×64 *matrix multiplication.* Different scales on lower and upper diagrams!

To simulate a reasonably bad *unpartitioned* case, the two source matrices are allocated like in the partitioned case, but the filter uses exactly the same colors so that any filter activation invalidates all matrix data in both caches. Since both algorithms basically read data and do not modify it, this is a bad case but by far not the worst case.

Nevertheless, figure 8 shows that partitioning can have significant effects. The shaded bars in the small diagram show the ratio of the times required for unpartitioned and partitioned allocation. The maximum effect, a slowdown factor of nearly 3, occurs at 5 kHz. Partioning becomes less effective when the filter-activation rate is decreased: The cache used by the matrix multiplication is flushed less frequently so that the matrix algorithm's average cache-hit rate increases. However, partitioning becomes also less effective when the filter-activation rate increases beyond 5 kHz: Interrupt handling, task setup and first-level cache misses (which take the same time in both schemes) need more and more time relative to the decreasing timeslice length.

Note that the ratios which are shown as shaded bars are *not* unpredictabilities: Here we compare a concrete good case and and a concrete bad case, not the theoretical best and worst case. On a first glance, it might therefore surprise that the measured maximum ratio of 2.8 (+180%) is *greater* than the matrix multiplication's unpredictability in the write-through experiment. The reason: partitioning improves not only the matrix multiplication but also the filter algorithm. Recall that the filter is activated with a constant rate. In any timeslice, first the filter is served completely, then the matrix multiplication gets the remaining time. Now partitioning has two effects: (a) the matrix multiplication needs less time in total and (b) it gets more time per timeslice since the filter also runs faster.

Partitioning might have surprisingly high effects when multiple real-time tasks are combined.

6 Related work

Mogul and Borg [9] study the *average* impact of context switches in time sharing systems for several cache architectures. They report an increase of the CPI (cycles per instruction) of about 0.1 to 0.5 for the first 50,000 cycles after a context switch for one of the simulated architectures (64-K-instruction and 64-K-data first-level cache). From then on, the impact is below an increase of 0.05, from about 150,000 cycles it is negligible. In their summary of experiments, they report an overhead ratio due to context switches under 5% for most of studied architectures and benchmarks. The estimated penalty per switch ranges

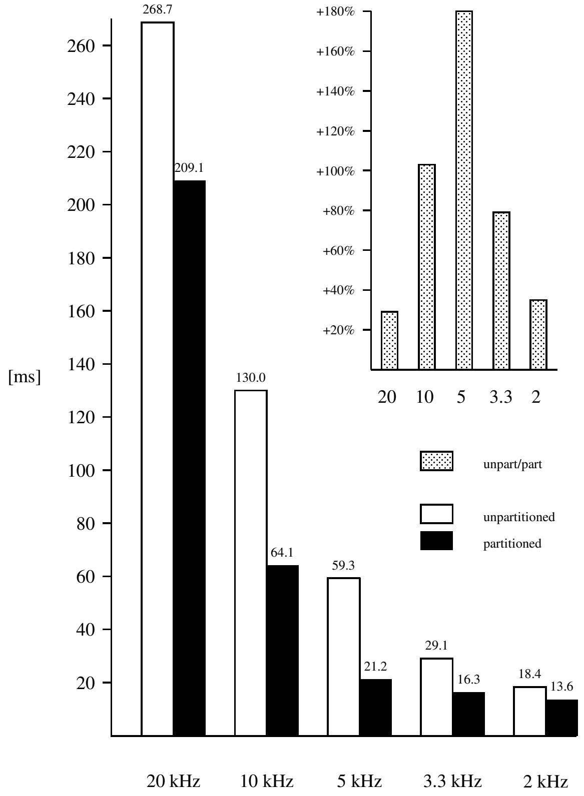

Figure 8: *MM time on Pentium, 133 MHz.*

from 10 μs to 400 μs on DEC 5000 which is roughly comparable to the switching overhead of about 70 μs in other Unix-based systems. However, they consider their results as very optimistic. Our work also studies the impact of context switches on execution time. However, we look at the worst-case impact on execution times. Their and our results together could be used to relate average and worst-case penalties. However, the addressed applications, the underlying hardware and operating systems differ significantly. In particular, we address very high context-switch (interrupt) frequency, and we use an operating system with small context switching overhead in contrast to the high switching overhead of Unix-inherited systems.

The question of how to increase the predictability of cache based systems in real-time environments has attracted some attention:

Li et al. [6] analyze the cache interference within a single task. A genuine approach to estimate worst-case penalty in the presence of multiple tasks is to preschedule all context switches and consider the cache to be in a worst-case state after a switch. Niehaus et al. [11] propose this approach for the Spring real-time system. Our work addresses a more open environment where not all context switches can be prescheduled.

Kirk [5] describes SMART, *Strategic Memory Allocation for Real Time*. Essentially, he proposes to use extra information for the cache line mapping function, e.g. user id or high significant address bits. The drawback of the proposed technique is the need for extra mapping hardware that potentially adds extra cycles in the critical path. Our technique does *not need extra cycles* for recoloring and allows cache partitioning to be based on software.

Pioneering work on software-based cache-partitioning techniques has been done by Wolfe [15] and Müller [10]. Both propose to place code and data such that cache conflicts are avoided. Wolfe exploited this idea to hand-coded programs; Müller applied it to automatic code generation and data placement of compiler systems. Wolfe already hinted at a possible usage of his ideas in virtual memory management systems. These techniques have two drawbacks that are not acceptable in our view: First, they have the side effect as also mentioned by Wolfe that the reservation of a large chunk of cache requires the reservation of an equally large part of the main memory. The author's statement that large cache requirements in general relate to large requirements of main memory is questionable. The filter example studied in this paper is of the opposite type. Second, the technique requires (a) that the set of tasks is static and (b) that all programs are compiled by the same compiler prior to the system start or can be recompiled by the OS. Both preconditions are not fulfilled for many closed real-time systems and cannot be fulfilled for open real-time systems. Although the technique can be used in dedicated and closed systems, it is not applicable to real time systems in general. In contrast, our technique relies on the memory management component of an operating system to isolate a task with arbitrary address access pattern. It is completely transparent for the application programmer and is not restricted to static environments. In addition, it allows to clearly separate cache and main-memory assignments, hence avoids waste of memory. Similar to Wolfe's and Müller's work, our approach does not require processor-chip changes to partition second-level caches.

The technique described in our paper requires knowledge about worst-case execution times in the absence of external interference. Several publications deal with that problem. Arnold et al. [1] use a so-called static-simulation technique to determine worst-case bounds for cached programs. It relies on estimating the worst-case instruction-cache performance for each loop and procedure.

Bershad et. al. [3] describe *dynamic recoloring* as a technique to improve cache performance. When large numbers of cache misses are detected by a specific hardware device, they dynamically remap pages to page frames with better colors. This could be combined with our technique of recoloring physical page frames without copying.

7 Summary

This paper describes an experimental environment to determine the worst-case impact of asynchronous cache activities on tasks whose *ideal*, i.e. minimal interference execution times are known and a partitioning technique to effectively limit worst-case penalties of second-level cache interference. It demonstrates the achievable effects for a digital filter program, discrete cosine transformation, fast Fourier transformation and matrix multiplication.

More and more complex applications will need to be examined to finally determine the usefulness of the cache-partitioning approach in general.

Acknowledgements

The authors thank Peter Hochgemuth for triggering this work and Robert Baumgartl for providing benchmark programs. This work has been partially supported by Deutsche Forschungsgemeinschaft in the Sonderforschungsbereich 358.

References

[1] R. Arnold, F. Müller, and D. Whalley. Bounding worst-case instruction cache performance. In *IEEE Real-Time Systems Symposium*, pages 172–181, 1994.

[2] S. Bederman. Cache management system using virtual and real tags. *IBM Technical Disclosure Bulletin*, 21(11):4541, April 1979.

[3] B. N. Bershad, D. Lee, T. Romer, and B. Chen. Avoiding conflict misses dynamically in large direct-mapped caches. In *6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 158–170, San Jose, CA, October 1994.

[4] Intel Corp. *Pentium Processor User's Manual, Volume 3: Architecture and Programming Manual*, 1993.

[5] D. B. Kirk. SMART (strategic memory allocation for real-time) cache design. In *IEEE Real-Time Systems Symposium*, pages 229–237, December 1989.

[6] Y. S. Li, S. Malik, and A. Wolfe. Cache modeling for real-time software. In *IEEE Real-Time Systems Symposium*, January 1997.

[7] J. Liedtke. *Eine Speicherarchitektur zur Unterstützung farbtreuer Allokation für verbesserte Cache- und TLB-Ausnutzung*. Deutsches Patentamt, München, October 1995. Patent application 195 38 961.1, P 44 37 866.1, decisioned to grant (Sep 1996), PCT/EP95/04103 (European Patent Office, October 1995).

[8] J. Liedtke. On μ-kernel construction. In *15th ACM Symposium on Operating System Principles (SOSP)*, pages 237–250, Copper Mountain Resort, CO, December 1995.

[9] J. C. Mogul and A. Borg. The effect of context switches on cache performance. In *4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 75–84, Santa Clara, CA, April 1991.

[10] F. Müller. Compiler support for software-based cache partitioning. In *ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems*, La Jolla, CA, June 1995.

[11] D. Niehaus, E. Nahum, and J. A. Stnakovic. Predictable real-time caching in the Spring system. In *IFAC Systems and Software Workshop*, Atlanta, 1991.

[12] J. Simonson and J. H. Patel. Use of preferred peemption points in cache-based real-time systems. In *IEEE International Computer Performance and Dependability Symposium*, pages 316–325, April 1995.

[13] A. J. Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.

[14] W. H. Wang, J. L. Baer, and H. Levy. Organization and performance of a two-level virtual-real cache hierarchy. In *16th Annual International Symposium on Computer Architecture (ISCA)*, pages 140–148, Jerusalem, May 1989.

[15] A. Wolfe. Software-based cache partitioning for real-time applications. In *Third International Workshop on Responsive Computer Systems*, September 1993.

[16] D. A. Wood, S. J. Eggers, G. Gibson, M. D. Hill, J. M. Pendleton, S. A. Ritchie, G. S. Taylor, R. Katz, and D. A. Patterson. An in-cache address translation mechanism. In *13th Annual International Symposium on Computer Architecture (ISCA)*, pages 358–365, Tokyo, June 1986.