

The real cost of task pre-emptions — measuring real-time-related cache performance with a HW/SW hybrid technique

MRTC Technical report 02/58

Filip Sebek

Department of Computer Science and Engineering
Mälardalen University, Västerås, Sweden
fsk@mdh.se

12th August 2002

Abstract

Cache structures and other modern hardware is today so complex, that simulation on instruction level is very complicated and time-consuming. Real measurement is much faster, but on the other hand less observable and in most cases impossible to make non-intrusive.

To predict schedulability for a system that incorporates an unpredictable device, such as the cache, requires known safe values of task execution time, task-switch time, pre-emption delay etcetera to statically predict their schedulability. This paper proposes a hybrid HW/SW method to measure cache performance with minimum intrusion. It also presents some experiences on a real system with experimental results of setting up scenarios and measure cache performance on a high performance microprocessor system based on MPC750 CPUs.

The experimental results show that the cache related pre-emption delay for instructions can be as much as 69% of the pre-emption cost for a MPC750 system.

1 Introduction

1.1 Real-time and cache memories

Cache memories are used in today's computer systems to boost up performance by bridging the gap between response time of primary memory and CPU. Since the small cache is too small to hold all data and instructions, blocks are swapped in and out depending of what section of code in the program is handled at the moment.

The swapping results in a variable memory access time, which makes the computation of the worst case execution time (WCET) very tricky. The execution time depends of the cache contents and the cache contents depends on the execution path that depends on, among other properties, the execution time.

System developers are today in a great need of real values to implement new software based products on high-performance processors. These values must be safe but also simple and fast to get. A correctly performed real measurement on a real system might give them what they need — especially when the offered method is easy to understand.

1.2 Monitor and measurement methods

Several methods are feasible to measure or monitor cache performance in computer systems. The major issue is to make the measurement non-intrusive so the measured environment is unaffected. A second issue is to set up correct and representative scenarios to be measured. If for instance the worst case execution time (WCET) is to be measured, one must set up an execution path that leads to the WCET.

Execution time and other performance issues can either be statically analyzed [1, 2, 3, 4] or simulated[5, 6], or measured directly on the target system[4, 7].

The advantage of static methods is that they are safe if the system model and analysis method are correct and compatible with each other. The hard part is to add complex structures into the model like pipelining, cache memories, DMA and other hardware that affects the execution time. To model a real processor is very difficult

to accomplish[8, 9] and to simulate execution on a modeled complex system may take over 1000 times longer than the actual execution.

Real measurement is on the other hand much faster to perform, but requires special hardware to tap information from the system. Changing hardware parameters like cache size or bus bandwidth is often practically not possible. The major advantage is that a complex hardware is correctly “modeled” as is.

Monitoring methods can be categorized as

- **Trace driven simulation.** By feeding extracted execution traces to a simulator it is possible to monitor internal states and measure execution time. It is also common that architecture parameters (cache size, band width etc) can be altered to give a better flexibility in the analysis. One problem arises when the execution path (=the trace) can alter depending of execution time, and since execution time depends for instance on cache contents one can come to the conclusion that the simulation must model the system from where the traces were derived.
- **Hardware monitoring.** By attaching for instance a logic analyzer on the bus or taping information from the processor’s JTAG pins one could trace where the execution occurs. One problem is that JTAG has a limitation in bandwidth so for instance a processor must run at half speed. Another problem is that only addresses on the address bus doesn’t say much about what really is going on in the processor since the information is at a very low level and internal signals to registers and caches are hidden.
- **Software monitoring.** A program or pieces of code writes internal states or time values to memory, disc or screen. The information can be at a very high abstraction level, but the cost is a high utilization of CPU, memory, cache alteration, bus bandwidth etc. All these resources must be allocated and remain in the program after the measurement to eliminate probe effects.
- **Hybrid HW/SW monitoring.** With minimal code and hardware monitoring it is possible to get information at a high level with small resource utilization.

To measure with the SW or SW/HW monitoring methods, *probes* are inserted in the code. The methods proposed in this paper use three kinds of probes: *Kernel*

probes that are incorporated in the OS-code of task-switches, system functions, interrupt routines etc. *Inline probes* are placed directly in the application code and *probe tasks* that log the system state.

1.3 Cache properties to measure

In a real-time perspective some properties are more interesting and useable to measure than others.

- **Task-switch time.** It is very common in schedulability analysis to just take the WCET for the tasks as parameters and assume that the context-switch time is zero. In the real world a time will elapse during the context switch since the OS has to execute some code to make the context-switch possible. The knowledge of this time is important in systems with a high frequency of context-switches.
 - **Pre-emption delay including cache related effects.** The cache will be filled with instructions and data that belong to the executing task. Misses that occur at this point are called *intrinsic*. When a new task starts to execute, the cache will swap out the previous task with the current locality — these kinds of misses are called *extrinsic*[10]. One special kind of extrinsic miss occurs during pre-emption when a high prioritized task can interrupt low prioritized to gain faster response time. The pre-emption will swap out the cache contents of the executing task and cause a cache refill-penalty for the low prioritized task when it resumes its execution. The cost to pre-empt another task is $C' = C + 2\delta + \gamma$, where C' is the new WCET, C stands for the unmodified WCET, δ is the execution time for the operating system to make a context-switch (two are needed for a pre-emption) and γ symbolizes the maximum cache related cost by a pre-emption[11]. This *cache related pre-emption delay* (CRPD) or *cache refill penalty* that can be considered as an *indirect cost* is illustrated in figure 1 and 8.
- The CRPD can be eliminated or reduced by partitioning the cache so each task has a private part of it, but to the price of decreased over-all performance[12, 13, 14].
- **Continuously measuring cache performance.** By continuously monitoring the cache-miss ratio, maximum and average miss-ratio during a time-slice can be determined. If the time-slice is smaller than

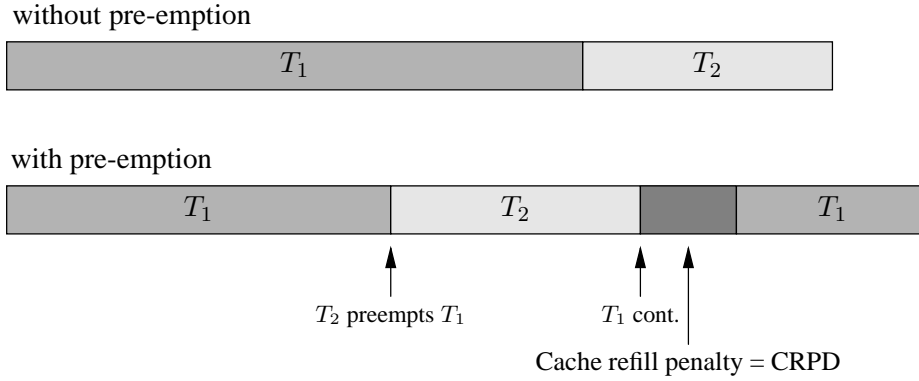


Figure 1: Cache related pre-emption delay. A pre-empted task can be drained by cached data and suffer a refill penalty. Please observe that the penalty must not come right after resuming the pre-emption — it may come later or in pieces depending of the program’s design.

the operating system’s time granularity, the average miss-ratio can be used to calculate WCET and if used in soft real-time systems the maximum miss-ratio can be used. The method is however only applicable for hard real-time systems when the worst-case execution time scenario is executed.

This paper is organized as follows: The next section describes the target system where all measurements were performed. Section 3 presents the workbenches in detail and experimental results, and the paper ends in Section 4 with conclusions.

2 The system

The complex target system “SARA”[15] has been used for experimental results and testing the measuring method. This section will describe features and special hardware components in the system. Please observe that all the features in the hardware is not necessary for the generalized measuring method — it just makes the method easier to perform.

SARA — Scaleable Architecture for Real-Time Applications — is a research project with a Motorola CPX2000 Compact PCI backplane bus with at most eight Motorola Power PC750-processor (MPC750) boards [15]. See figure 2 but also figure 4 for a complete system overview. The processor boards are equipped with a MPC750 running at 367MHz and is connected to a 66MHz bus as well as the the 64MB DRAM main memory.

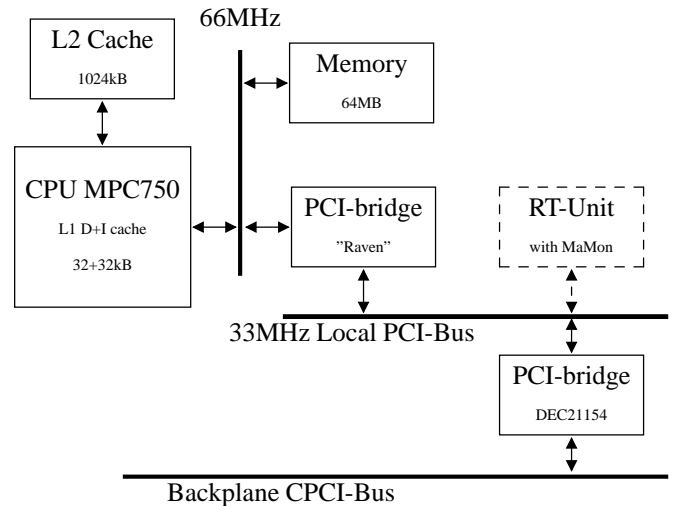


Figure 2: CPU-card. The RT-Unit is only on master cards.

2.1 MPC750

The microprocessor MPC750 is equipped with a split instruction and data cache at the first level, and a unified cache memory at the second level. The first level caches are 32kB each and organized in 8-way set-association with a pseudo LRU replacement policy. Each cache block can hold eight 32-bit words. The second level cache is in the target system 1024kB, 2-way set-associative and the block size is in this case 128 byte that is divided into sub-blocks. The caches are non-

blocking and can be locked by users during execution. Four areas in the memory that is to be cached are defined in Block Address Translation (BAT) registers. All four areas can set its own WIMG-properties, that is **W**rite-through/**W**rite-back policy, caching **I**nhibited, enforced **M**emory coherency and **G**uarded bits.

The MPC750 is equipped with an on-chip performance monitor that can monitor 48 different kind of events, but there are only 4 performance monitor counters (PMCs) available. Only 5 of the 48 events can be associated with any PMC and the rest are associated to a dedicated PMC. [16, 17]

2.2 Real-Time Unit

A special *master card* is equipped with a Real Time Unit (RTU)[18] that controls the execution of the tasks on all processor cards. The RTU is a high performance and performance predictable hardware implementation of an operating system kernel that handles scheduling and other real-time operating system services. The other processor cards are used as *slaves* to increase application performance. All communication between tasks (inter and intra-processor) is performed through a *virtual bus* which simplifies application development[19].

2.3 MAMon — an application monitor

A special device called *Multipurpose Application Monitor (MAMon)* [20, 21], can tap for instance the RTU non-intrusively on information regarding context-switching, inter process communication, task synchronization etc. There is also a possibility to write to special registers called *software probes* directly from the application. Writing to a software probe is much faster than reading a register since the processor just writes to the PCI-bridge through the 66MHz 60x-bus. The bridge will then eventually write to the RTU when the 33MHz local bus is clear.

All the collected data is sent through a parallel port to an external host for post analysis.

Today MAMon and the RTU co-exists in the same FPGA, and besides increased performance this is a very practical and cost effective way to eliminate problems with PCB-layout and other hardware manufacturing issues.

To integrate MAMon into another system than SARA that doesn't use a hardware implemented OS can easily

be performed by just adding a card with MAMon hardware, which is accessed by memory mapped addressing. In this case only software probes can be used.

3 Experimental setup and results

All experiments were performed at instructions only with synthetic workloads on the SARA system. Each of the three measurement cases are presented in their subsection. The first case describes how to measure cache miss-ratio continuously with minimal intrusion. The second case describes how to measure context-switch/pre-emption time and the third case sets up a scenario to maximize cache effects so the cache related pre-emption delay (CRPD) can be measured and included into the pre-emption time. The subsections also presents the measured results and concludes with a discussion.

3.1 Synthetic workloads

No good standard benchmark suits are available today to measure cache memory effects in real-time systems. Non-real-time benchmarks such as SPEC or Dhrystone are just single programs without (interfering) tasks. Rhexstone[22] on the other hand just tests real-time operating system issues such as task-switches, deadlock handling and task communication. The test applications are too small to test cache memory issues and were not meant to do so either.

The tests were therefore performed on synthetically generated task sets where the amount of tasks and the data and instruction size of all the tasks were generated. Priorities, miss-ratio, cache locking and cycle time can also be set. The generated code is very simple since it's only purpose is to swap out cache contents. The basic structure of a task is one simple big loop that contains a large sequence of "r1=r1+0" without jumps and ends with a delay to let lower prioritized tasks to run.

Using synthetic task sets has successfully been used in for instance Busquets-Mataix *et al*'s work[23]. More about the synthetic workload and its' different properties will be presented in the three cases.

3.2 Continuous measurement

3.2.1 Implement the probe as a task

Implementation A small, simple, cyclic probe task — "MonPoll" — polls the MPC750's performance monitor registers and passes the values to MAMon where they

get time stamped. Figure 4 illustrates the complete system with hardware and software. This small task (written in C) is as simple as in Figure 3.

The task should have the highest priority to be able to measure during all tasks' execution.

Performance requirement The OS¹ granularity to start tasks is 2 milliseconds, which means that the highest sample rate is 500Hz. The task requires 671 assembly instructions to execute which includes two context-switches in the operating system (major part) and the small code itself.

Best case is when no cache misses are present. The execution time for the MonPoll task is then 43.1 μs , which means that it consumes about 2% of the execution time. It also means that if a task is interrupted by the MonPoll task and no cache misses will occur, the execution time of the running task will be extended with 43.1 μs .

Worst case Worst case will occur if the MonPoll task is not in the cache and will pre-empt a loop for which content maps the same cache lines as the MonPoll task. This will generate a small burst of initial misses for the MonPoll task plus some new misses when the pre-empted task resumes its execution to replace the MonPoll task in the cache (=CRPD).

Specifically this means an extension of the execution time by 21.9 μs to 65 μs . 99 cache lines have been swapped out and the refill time for those is also 21.9 μs if all cache blocks were useful in the pre-empted task. The performance cost will therefore be more than doubled from 43.1 to 86.9 μs , or if the sampling is performed at maximum rate $\frac{0.0869}{2} \approx 4\%$ of CPU resources.

Analysis and Discussion A solution to reduce the performance cost of the measurement method is to keep the MonPoll task in the cache, which can be achieved by (software) partitioning. The cost is high because it allocates a task with all the inherited costs of context switches etc. Only a few registers will be used and therefore it is very expensive to store all those only to make a simple poll. The granularity of the observation by only being able to poll each second millisecond might be very poor in many situations. The MPC750 will execute almost one million instructions during this period.

¹“OS” and “RTU” will in this case referred as the same thing.

3.2.2 Implement the probe as an interrupt routine

Instead of putting the small piece of code into a task, the same code can be called as an exception routine. There are several pros and cons to do so:

- Saving registers before running own code must be performed by “user” instead of the operating system
- It is performed by the processor itself, no operating system support is needed which also means faster execution and less intrusion
- Finer granularity is possible since the interrupts are independent of the operating system's time base

Exception on timer value The MPC-family is equipped with a 32-bit decrement-register (DR) that is decreased by a step each fourth external bus-cycle which in this specific case is $66,7/4 \text{ MHz} = 16.7 \text{ MHz}$ or $T=60\text{ns}$. When DR is equal to zero an exception occurs and the program counter is set to 0x0900. If this feature is unused this address must contain `rfi` (return from interrupt) to proceed with the execution. DR can be set to any arbitrary number by user code and by this be used as an external clock. The CPU runs at 233MHz which means that a resolution of 14 clock cycles or at maximum 28 instructions is possible to achieve.

MAMon is able to handle about 3500 cache events per second. Sending performance monitor data at this pace is a severe limitation; during this period 150 000 instructions may have been executed. A workaround is to write to MAMon only when an amount of changes (for instance cache misses) has reached a limit. Since MAMon can store short bursts of data in a FIFO queue, a practical sample rate of up to 1MHz is possible but then the load at the system will be 12%²

Exception on PMC threshold value The MPC750 has a special exception routine for the performance monitor. When a PMC reaches the threshold value an exception call to 0x0f00 is performed. In this case the workaround in the previous timer value solution can be avoided with a performance increase and less intrusion as a result.

²The 12% load is best case when no cache misses occur. This value shouldn't be compared with the MonPoll 4% utilization since that only runs at 500 Hz.

```

void monitor_poller( void ){

    RTU_IO rtu_io;

    while(1) {
        asm(" mfspr 0, 938 ": "=r" (MAMON_SWPROBE_2));
        asm(" mfspr 0, 941 ": "=r" (MAMON_SWPROBE_3));
        rtu_io.delay(2);
    }
}

```

Figure 3: A small task that polls performance monitor counters and writes the results to MAMon

3.3 Workbench

The generation of code with a fixed miss-ratio has been accomplished in two ways. Either spatial locality is exploited by only executing a fix fraction of the instructions in a cache line or a fraction of reuseable code is used to exploit temporal locality. A third possibility is to combine both these methods.

- If the first instruction in all cache lines is an unconditional jump to the start of the next cache line only the first word in each block will be accessed and by this never exploit spatial locality. If the code mass is much larger than the cache memory the reuse of code will not take place and decrease miss-ratio. By this we have generated a code with 100% cache misses. To generate code with 50% misses the jump is moved from the first to the second word in the cache block, and to generate code with 33% the jump is moved to the third word and so on. 66% misses can be generated by altering the jump from the first and second word in the cache block in the complete program.
- Code within a loop that fits into the cache will gain a lower miss-ratio each iteration since it reuses the cache lines (temporal locality). If the code is a bit larger than the cache size some code will be swapped out and the amount of *useable cache lines* in the cache will decrease. With code that has 100% spatial misses the cache miss-ratio is formulated in Figure 5.

Example: A 38kB task with 100% spatial misses will in a 32kB 2-way set-associative cache memory obtain an average miss-ratio of $\frac{38-32}{32} \cdot 2 = 37.5\%$

A more correct description of “task size” is “the task’s cache-non-interfering *active* or *useful* cache lines” but in

this synthetic generated workload it is the same.

3.3.1 Results

To prove that the measurement methods work (both probe task and interrupt driven kernel probes) a task set with tasks with different average cache miss-ratios was set up, executed and measured. The theoretical values were compared with the measured and the difference was less than 1% with the interrupt-driven measurement method. The fluctuation depends for instance on DRAM refreshment and odd bus-cycle access.

Figure 6 illustrates four tasks with different miss-ratios executing. The tasks’ miss-ratio was controlled by the individual task size.

3.3.2 Discussion and conclusion

This kind of monitoring is maybe more of performance than of hard real-time interest. The monitoring has too low resolution to give any information about for instance CRPD. It can however be used for *soft* real-time systems to get a view of the *average* miss ratio of the tasks to be used for static WCET-calculation.

A periodic interrupt routine with inline probes has shown much better performance than a probe task. The granularity can increase from 500 Hz to 1 MHz in short³ bursts. CPU utilization with a probe task running at 500Hz drops from 4% to an immeasurable level if the same measurement is performed with the interrupt routine implementation running at the same frequency. When measured at 1 MHz with the interrupt

³“Short” in this case are 256 events since that is the size of the internal FIFO-queue in MAMon. Available hardware sets the limit of the queue size.

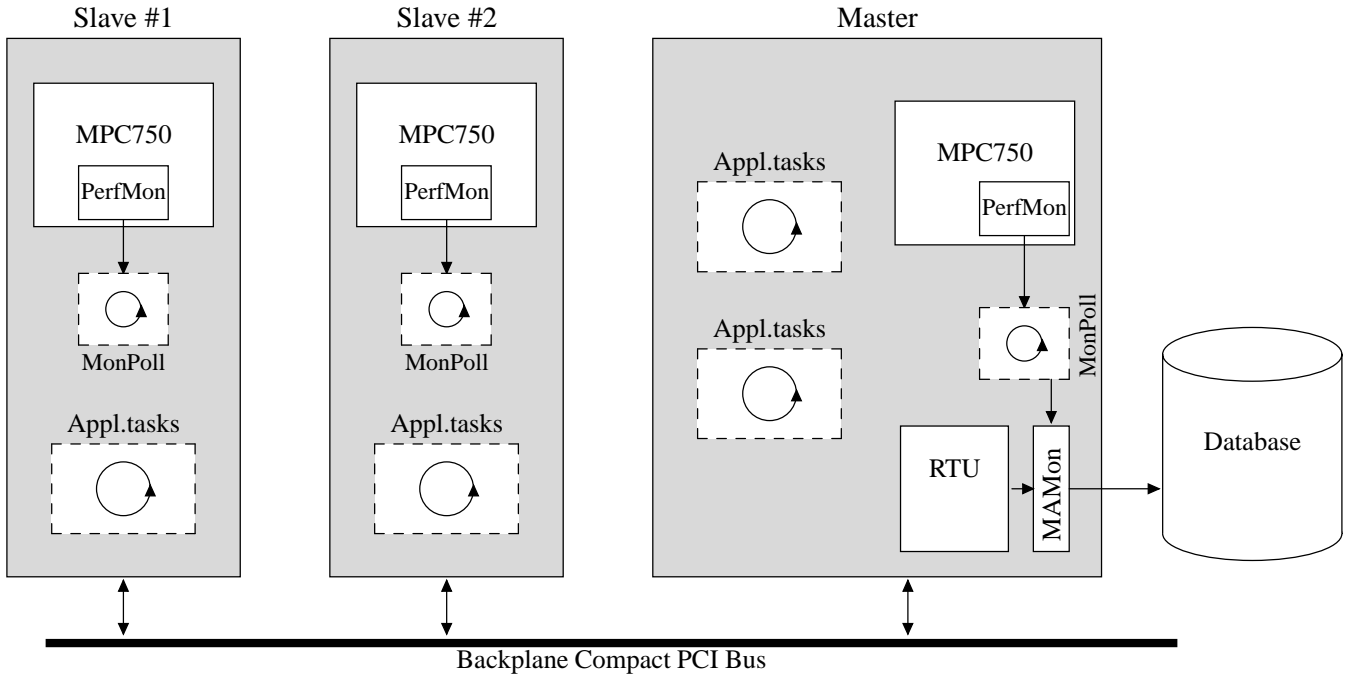


Figure 4: SARA system. Dashed boxes are software implementations.

$$\text{miss ratio} = \begin{cases} 0\%, & \text{task size} \leq \text{cache size} \wedge \text{time} \rightarrow \infty; \\ \frac{\text{task size} - \text{cache size}}{\text{cache size}} \cdot \text{set associativity}, & \text{cache size} < \text{task size} < \text{cache size} + \frac{\text{cache size}}{\text{set associativity}}; \\ 100\%, & \text{task size} \geq \text{cache size} + \frac{\text{cache size}}{\text{set associativity}}; \end{cases}$$

Figure 5: The cache miss-ratio depends in the generated code on task size, cache size and set-associativity. Each cache block begins with an unconditional jump to the next cache block to prevent utilization of spatial locality.

driven method the CPU used 12% of its computation resources to the measuring activity.

3.4 Measuring context-switch time

A pre-emption consists of two context-switches:

1. Interrupting the executing task.
 - (a) OS decides to make a context switch
 - (b) registers of the pre-empted task are saved
 - (c) registers of the pre-empting task are loaded
2. Resuming to the interrupted task.
 - (a) the high prioritized task yields to low prioritized tasks
 - (b) OS decides what lower task should run

- (c) registers of the high prioritized task are saved
- (d) registers of the lower prioritized task are loaded

Since the RTU on the target system does very much of the OS work during the interrupt and more software code is to be executed during the resuming, it is expected that the resuming in this case should take longer time.

3.4.1 Workbench and results

This type of measuring is quite straightforward since four kernel probes into the OS can perform it. The probes are placed at the first and last lines of the two context-switch routines. The best case scenario is when the routines are in the cache and the worst case is when all the code has to be loaded from main memory and

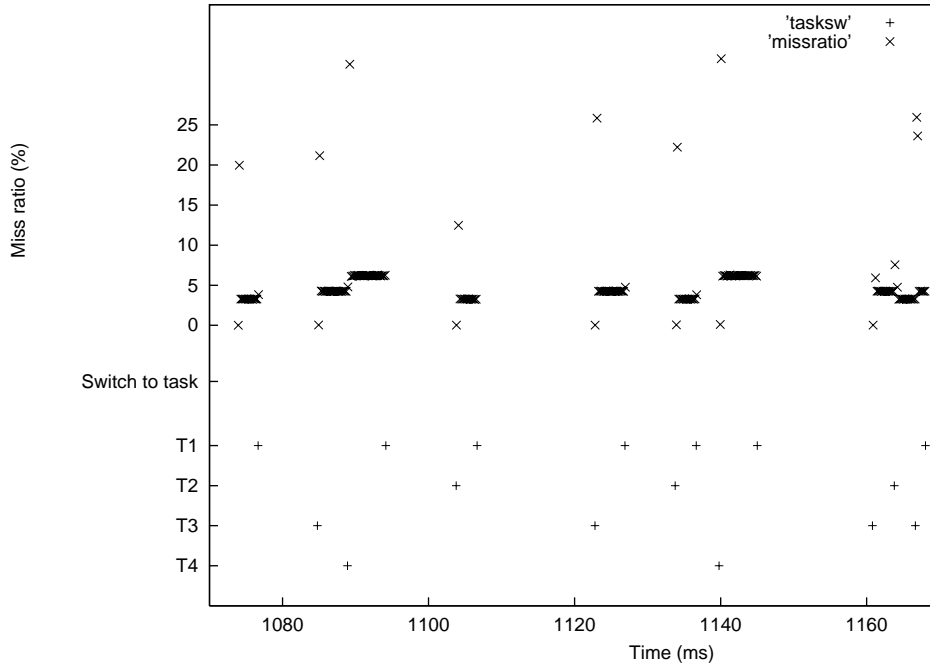


Figure 6: Continuous miss-ratio measurement with an interrupt driven kernel probe. T_2 starts to execute (miss ratio: 3.27%) and yields to the idle task T_1 (0%). Then T_3 (4.25%) executes and yields for T_4 (6.20%). At the end of this sample T_2 pre-empt T_3 . Observe the initial miss-ratio peaks at the context-switches.

swaps out some cache lines that would have been useful for the task (=CRPD).

Best case scenario is created with an empty task “pre-empting” the idle task. The worst case scenario is measured on a task set with two tasks that both are larger than the cache memory so the OS-code will be swapped out for sure. The results are presented in Table 1

3.4.2 Discussion and conclusion

Even if the context-switch time relies on many parameters such as CPU, system platform, operating system, and in this case it is more true than for others since parts of the OS kernel is implemented in hardware, the proposed measurement method is still useable for almost any computer system. The measured time can directly be used in scheduling algorithms and analysis.

3.5 Measuring CRPD

3.5.1 Workbench

The CRPD grows linearly with the pre-empted task’s size⁴ and reaches its’ maximum value when the task size is equal to the cache size assuming that the pre-empting task has replaced the suspended task completely.

One should notify that the CRPD will decrease after its maximum point since the number of useful blocks will decrease with the task’s size. If the active context of the program is twice as large as the cache size it will never reuse any cache lines. In this case there will be no CRPD what so ever.

On set-associative caches with LRU or FIFO replacement algorithm the CRPD also will depend on the set-associativity; a fully associative cache memory will with a task that is one word larger than the cache always replace the cache line that is about to be accessed and by this never have useful cachelines in the cache — poor performance but no CRPD. The relationship between CRPD, task size and set-associativity is illustrated in Figure 7 and the reasoning is similar to generation of

⁴As mentioned before; a more correct description of “task size” in this context is “the task’s cache-non-interfering *active* or *useful* cache lines”.

Situation	Number of cache misses	Interrupt (μs)	Resume (μs)	Pre-emption cost (incl. CRPD) (μs)
Best case	0	18.5	24.6	43.1+0.0=43.1
Worst case	44+55	28.4	36.6	65.0+21.9=86.9
No cache	-	52.5	66.1	118.6

Table 1: The table shows the time it takes to perform a context-switch. The cache related costs are inherited only from the context-switch code.

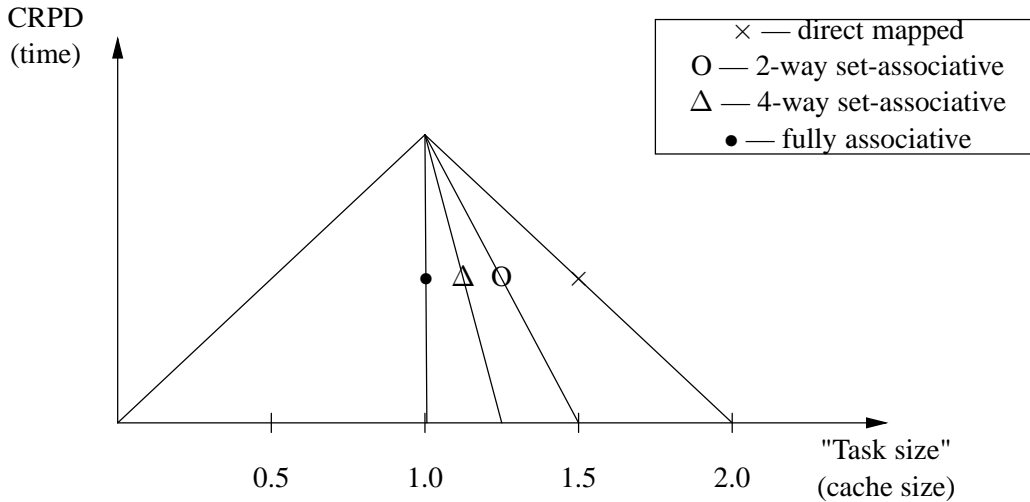


Figure 7: The theoretical maximum CRPD of a task is depending of the tasks' size and the cache memory's set-associativity.

a task with fix miss-ratio (section 3.3).

Figure 8 illustrates the scenario where the high prioritized task T_2 pre-empts T_1 and by this get $CRPD = ((f-e) + (d-c)) - (b-a)$.

To measure the *maximum* CRPD that is possible to suffer in a system, the previous scenario can however be simplified by pre-empting a task (T_1) by another task (T_2) for which both sizes are exactly equal to the cache memory. (T_1) is an endless loop that writes to a timestamped software probe each iteration to make it possible to calculate the execution time. If there are no other interfering components (other interrupts, instruction pipeline refill etc.) the CRPD is at hand. Interfering components can be detected by measuring the CRPD with different sizes of (T_1) since it in the absence of other pre-emption delay components should grow linearly with a start in the origo, and reach the top at the cache memory size.

3.5.2 Results

Several task sets as in scenario in Figure 9 with different sizes of T_1 and the result is shown in Figure 10. The maximum CRPD was measured to $195.5\mu s$ and the sloping line after 100% task size intersects the x-axis at 113.6% which is less than 1% overestimation from the theoretical 112.5% (see Figure 7). The measurement was also performed on a 233MHz MPC750 and since the main memory and busses are the same as on the 367MHz-system the CRPD was the same — a fact that verifies the method's correctness.

When the 1024kB second level cache was enabled the CRPD was decreased to $31.8\mu s$. This is however not the *maximum* CRPD of the system since the L2 was only partly used. In this case the L2 could host all tasks and OS-code and no accesses to main memory was necessary.

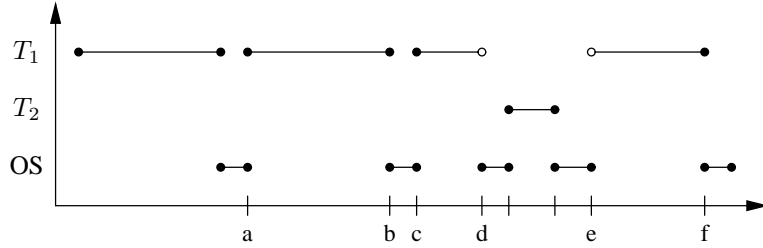


Figure 8: During the scenario T_2 pre-empts T_1 and interferes in the cache partly or completely. The CRPD effect will be maximized if all cachelines are useable for T_1 and swapped out by T_2 . $CRPD=((f-e)+(d-c))-(b-a)$

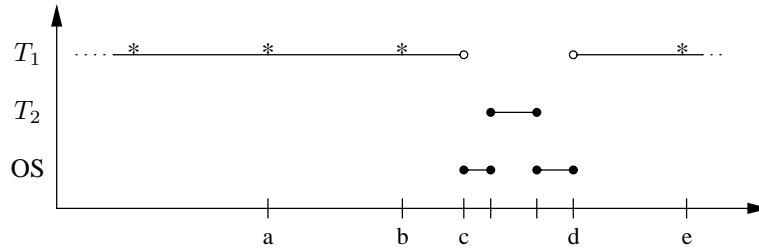


Figure 9: A scenario to get the maximum CRPD in a system: T_2 pre-empts T_1 and interferes in the cache partly or completely. T_1 is running in an endless loop and timestamps each iteration (marked as '*') in the figure. $CRPD=((e-d)+(c-b))-(b-a)$

3.5.3 Discussion and conclusion

The CRPD is at most $195.5\mu s$ on the considered SARA-system. The executing OS-code to pre-empt a task was measured to $86,9\mu s$ ⁵, which means that the total pre-emption delay is $282.4\mu s$. In relative terms the major part of the context-switch cost, or $\frac{195.5}{282.4} = 69\%$, is cache-related.

It is quite interesting that the CRPD is almost the same compared to Mogul and Borg's measurements a decade ago[5], which were $10-400\mu s$. During this time the processors have become magnitudes times faster and this means that the CRPD has grown in relative terms.

The method to get the CRPD is practicable to get a safe value that is directly useable in a scheduling algorithm. Even if the value is overestimated, the method will never fail or have any limitations that are very common in static methods.

⁵See worst case scenario in Section 3.2.1

4 Conclusions

Real-time systems are often implemented as scheduled tasks with timing constraints that must be satisfied for correct function. Even if very much research has been done in schedulability analysis, it is common to do assumptions to simplify the analysis — for instance that the cost of task pre-emption can be approximated to zero. In reality pre-emption cost, since execution of code takes time, and the indirect refill penalties with cache memories in the system can cost much more. Since the pace of instruction execution accelerates much faster than main memory access time, the penalty has increased and will increase even more in relative terms.

This paper presents a hybrid HW/SW technique that makes it possible to measure and timestamp cache events with minimal intrusion. The method has been implemented on a high-end multiprocessor system with Motorola Power PC750 CPUs controlled by a centralized real-time unit (RTU) with operating system features implemented in hardware. A hardware implemented monitor co-resides with the RTU on the same chip which makes it possible to tap information non-intrusively.

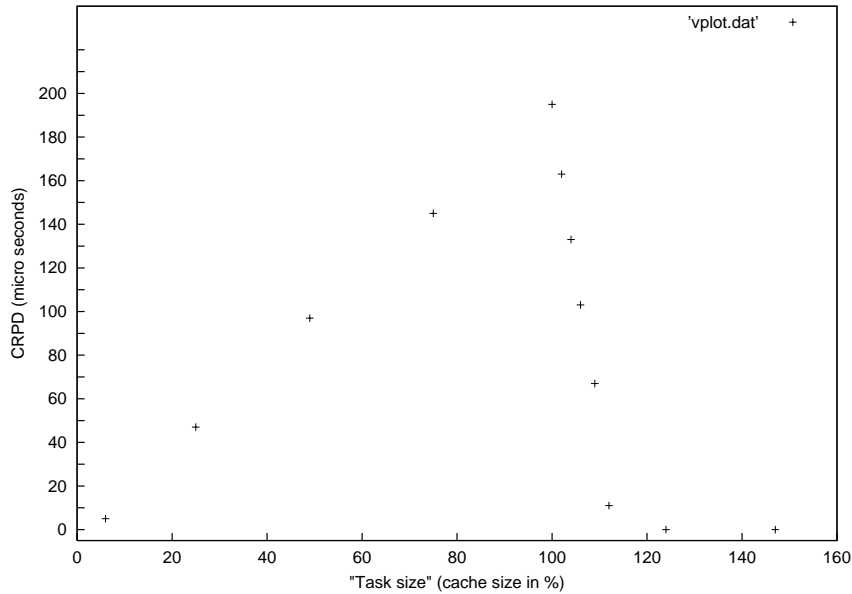


Figure 10: Measuring the instruction CRPD on a CPX2000 system with a MPC750 processor that is equipped with a 32kB 8-way set associative L1 instruction cache.

We have proposed how to continuously measure cache miss-ratio with a probe task and interrupt driven kernel probe. The interrupt driven kernel probe outcompetes the task probe in less CPU utilization and finer time granularity.

Methods has in this paper been proposed how to set up workbenches and measure the pre-emption delay including cache-related for instructions. Experimental results showed that a pre-emption could vary from 43 to 282 μ s if the cache related pre-emption delay (CRPD) is counted in. The CRPD itself will in this case stand for 69% of the time of the pre-emption execution and its indirect cause.

References

- [1] C. Healy, R. Arnold, F. Mueller, D. Whalley, and M. Harmon. Bounding pipeline and instruction cache performance. *IEEE Transactions on Computers*, 48(1):53–70, January 1999.
- [2] Hiroyuki Tomiyama and Nikil Dutt. Program path analysis to bound cache-related preemption delay in preemptive real-time systems. In *Proceedings of 8th International Workshop on Hardware/Software Codesign (CODES2000)*, pages pp. 67–71, May 2000.
- [3] Christian Ferdinand, Florian Martin, Reinhard Wilhelm, and Martin Alt. Cache behavior prediction by abstract interpretation. *Science of Computer Programming*, 35(2–3):163–189, November 1999.
- [4] Chang-Gun Lee, Joosun Hahn, Sang Lyul Min, Rhan Ha, Seongsoo Hong, Chang Yun Park, Minsuk Lee, and Chong Sang Kim. Enhanced analysis of cache-related preemption delay in fixed-priority preemptive scheduling. In *Proceedings of the 18th Real-Time System Symposium*, pages 187–198, San Francisco, USA, December 3–5, 1997. IEEE Computer Society Press.
- [5] Jeffrey C. Mogul and Anita Borg. The effect of context switches on cache performance. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 75–84, Santa Clara, CA, USA, April 1991.
- [6] Robert T. Short and Henry M. Levy. A Simulation Study of Two-Level Caches. 1988.
- [7] Stefan M. Petters and Georg Färber. Making worst case execution time analysis for hard real-time tasks on state of the art processors feasible. In *Proceedings of the 6th Real-Time Computing Systems and Applications RTCSA*, Hong-Kong, December 13–15, 1999. IEEE Computer Society.

- [8] Pavel Atanassov, Raimund Kirner, and Peter Puschner. Using real hardware to create an accurate timing model for execution-time analysis. In *Proceedings of IEEE Workshop on Real-Time Embedded Systems (RTES'01)*, London, December 3, 2001.
- [9] Jakob Engblom. On hardware and hardware models for embedded real-time systems. In *Proceedings of IEEE Workshop on Real-Time Embedded Systems (RTES'01)*, London, December 3, 2001.
- [10] Anant Agarwal, Mark Horowitz, and John Hennessy. An analytical cache model. *ACM Theory of Computing Systems*, 7(2):184–215, May 1989.
- [11] Swagato Basumalik and Kelvin D. Nilsen. Cache issues in real-time systems. In *Proceedings of the ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, June 1994.
- [12] David B. Kirk. SMART (strategic memory allocation for real-time) cache design. In IEEE Computer Society Press, editor, *Proceedings of the Real-Time Systems Symposium - 1989*, pages 229–239, Santa Monica, California, USA, December 1989. IEEE Computer Society Press.
- [13] Andrew Wolfe. Software-based cache partitioning for real time applications. In *Proceedings of the Third International workshop on Responsive Computer Systems*, September 1993.
- [14] Frank Mueller. Compiler support for software-based cache partitioning. In *ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems*, La Jolla, CA, USA, June 1995.
- [15] Lennart Lindh, Tommy Klewin, and Johan Furunäs. Scaleable architecture for real-time applications – SARA. In *Proceedings of SNART 1999*, Linköping, Sweden, 1999.
- [16] Motorola Corp. *MPC750 RISC Microprocessor Users Manual*, August 1997.
- [17] Motorola Corp. *Errata to MPC750 RISC Microprocessor Users Manual*, July 1999.
- [18] Johan Furunäs, Johan Stärner, Lennart Lindh, and Joakim Adomat. RTU94 – real time unit 1994 – reference manual. Technical report, Dept. of computer engineering, Mälardalen University, Västerås, Sweden, January 1995.
- [19] Peter Nygren and Lennart Lindh. Virtual communication bus with hardware and software tasks in real-time system. In *Proceedings for the work in progress and industrial experience sessions at 12th Euromicro conference on Real-time systems*, June 2000.
- [20] Mohammed El Shobaki. Non-intrusive hardware/software monitoring for single- and multiprocessor real-time systems. Technical report, Mälardalen Real-Time Research Centre, Västerås, Sweden, April 2001.
- [21] Mohammed El Shobaki. On-chip monitoring of single- and multiprocessor hardware real-time operating systems. In *Proceedings of the 8th International Conference on Real-Time Computing Systems and Applications (RTCSA)*, March 2002.
- [22] Rabindra P. Kar. Implementing the rheapstone real-time benchmark. *Dr. Dobb's Journal*, pages 46–55 and 100–104, April 1990.
- [23] J. V. Busquets-Mataix, A. Wellings, J. J. Serrano, R. Ors, and P. Gil. Adding instruction cache effect to schedulability analysis of preemptive real-time systems. In *IEEE Real-Time Technology and Applications Symposium (RTAS '96)*, pages 204–213, Washington - Brussels - Tokyo, June 1996. IEEE Computer Society Press.