Author manuscript, published in "16th International Conference on Real-Time and Network Systems (RTNS 2008) (2008)"

inria-00336526, version 1 - 4 Nov 2008

Attacking the Sources of Unpredictability in the Instruction Cache Behavior

E. Mezzetti* N. Holsti† A. Colin, G. Bernat‡ T. Vardanega*

Abstract

The use of cache memories challenges the design and verification of high-integrity systems by making WCET analysis and measurement, the central input to schedulability analysis, considerably more laborious and less robust. In this paper we identify the sources of instruction cache-related variability and gage them with ad-hoc experiments. In that light, we perform a critical review of state-of-the-art approaches to coping with and reducing the unpredictability of cache behavior. Finally we single out practices and recommendations that we deem best fit to attack the sources of unpredictability and discuss their applicability to a real processor for use in European space industry.

1 Introduction

Caches are very effective at speeding up memory accesses in the average case, yet at the cost of more variable execution time. In high-integrity systems however we need highly dependable figures for the timing behavior of applications, so that the important value is not (only) the average execution time, but the worst-case execution time (WCET) or, in practice, a trustworthy upper bound or estimate of it.

Caches make it more difficult to set bounds on the worst-case execution time. In particular, cache effects reduce the predictive value of execution-time measurements obtained by test, because the total program size, the memory layout, the execution paths and the pattern of interrupts or preemptions may all change from test to operation, and the tests themselves are unlikely to hit on the worst combination of all those factors together. Cache effects also reduce the precision of static WCET analysis because the analysis can only approximate the state of the caches at various points in the program. Moreover, schedulability analysis gets harder on two accounts: (i) the actual overhead of an interrupt or context switch is no longer constant but varies with the cache state at the time of arrival; and (ii) the WCET of a task depends on the interrupts and preemptions incurred during execution, unless the scheduler takes costly measures to preserve the state of the caches across interrupts and preemptions.

Historically thus, caches have been dispensed with in high-integrity systems altogether. At present however European space projects are transitioning to more complex and powerful processors that include a memory hierarchy with a fast, on-chip cache memory between the processor and the slower, external (off-chip) main memory. They therefore need to devise ways, whether by analysis or design or both, to cope with the inherent unpredictability of caches.

In this paper we address the instruction cache (I-cache) unpredictability problem. We do so with a view to providing the concerned industry with a pragmatic yet informed vision of a way ahead. We unfold the sources of cache-related variability and experimentally gage their magnitude on typical on-board software. We confirm that the unpredictability of I-caches does pose a significant threat to the analysis and validation of high-integrity embedded systems. On the basis of a critical review of state-of-the-art approaches to cache analysis we then make some recommendations to improve the predictability of the I-cache of the standard processor for use in European Space Agency on-board applications: the LEON family.

This paper is organized as follows: Section 2 gives our interpretation of the sources of cache-related variability, which we term "cache jitters"; Section 3 describes the approach to experimentally reveal and gage such jitters; Section 4 evaluates state-of-the-art approaches in the light of our cache jitters taxonomy; Section 5 singles out scientifically-based best practices and recommendations to cope with the I-cache unpredictability in the specific instance of the LEON processor family. Section 6 finally draws some conclusions.

Scope and contribution. We focus on the I-cache, which we can analyze with sufficient accuracy, and ignore the D-cache, which is very application-specific in use and complex to analyze since most data addresses are dynamic.

We performed experiments on the LEON AT697E chip from Atmel [2], which includes an I-cache equipped with 32 KB of memory and 32 B lines, 4-way associativity and

*University of Padua, via Trieste 63, I-35121 Padua, Italy {*emezzett, tullio.vardanega*}@*math.unipd.it*.

†Tidorum Ltd., Helsinki, Finland.

‡Rapita Systems Ltd., York, United Kingdom.

inria-00336526, version 1 - 4 Nov 2008

LRU replacement policy. In the reference configuration (50 MHz, 80 ns. memory) a fetch on a cache hit takes 1 cycle, but 4 on a cache miss, for a 3-cycle miss penalty. On more advanced processors with relatively slower memories, miss penalties range from 20 [16] to 50 cycles or more [15].

We believe this paper makes a threefold contribution: (1) we propose an original taxonomy of *sources* of cache unpredictability, which goes beyond the traditional analysis of their consequences; (2) we illustrate an approach to revealing those phenomena experimentally; (3) we review state-of-the-art recommendations for mitigating cache predictability and discuss their fitness against our taxonomy.

2 Addressing the sources of variability

Several studies [7, 3, 6] attribute the difficulties that caches cause to the timing analysis of programs to two main kinds of *interferences*. Intrinsic (intra-task) interference causes non-compulsory cache misses [11] when multiple program blocks, executed in the same task, compete for one and the same cache set and collide with one another. Extrinsic (inter-task) interference occurs in multitasking environments from similar competition between program blocks executed in different, interleaved tasks.

The intrinsic/extrinsic categorization emphasizes the "location" (intra- or inter-task) in which cache-related variability takes effect but it fails to single out the actual *sources* of such variability. In fact, compared to a cache-less processor, four new factors emerge which influence the execution time of instructions on a cache-equipped processor by changing the pattern of cache hits and misses:

1. the total size of the program code in relation to the cache size, which we term *size-related jitter*
2. the location (i.e., the memory addresses) of program code, which we term *layout-related jitter*
3. the code addresses accessed by the program in the past, which we term *path-related jitter*
4. the interrupts and preemptions incurred on execution, which we term *concurrency-related jitter*.

Size-related jitter depends on the actual program size in relation to the cache size, as both factors limit the total number of useful memory blocks that can be stored in the cache. In practice, if the repeated code in a program is smaller than the cache, most references will result in a cache hit; otherwise, numerous capacity misses may (and will) be incurred.

Layout-related jitter depends on the actual memory layout of the program code. Some memory accesses may result in a miss or a hit whether the required address was recently accessed or not. A hit can happen if another recently referenced address caused the containing memory block to be loaded in the cache. Similarly, a miss can happen if another recently referenced address caused the containing memory block to be evicted from the cache because both memory blocks map to the same cache set. This kind of misses are often called conflict misses, as the cache may still have some spare capacity in other sets.

Path-related jitter stems from the control structures in a program (i.e., conditional statements, variable-bounded loops, etc.). Control structures imply that one and the same point in the program may be reached through multiple execution paths; since the I-cache state depends on the execution path of the program, the execution time of a given instruction at a given point of the program may vary according to the execution path actually taken by the program.

Concurrency-related jitter depends on task scheduling: interruption, suspension or preemption of the running task may cause the cache state to be changed by the computation in other tasks or interrupt handlers. When the interrupted, suspended or preempted task resumes, its execution time varies accordingly. Depending on the scheduling action, we divide concurrency-related jitter into preemption- and suspension-related jitter.

As the jitter taxonomy focuses on the *source* of the variable interference, we consider it more suited than the interference categorization as the source of advice for taming the unpredictability of cache behavior. In our intent, the a priori study of cache-related jitters, rather than the a posteriori analysis of their effects, would facilitate a more conscious use of the I-cache in our application domain.

3 Revealing and gaging the cache jitters

In a recent review of this problem area [37] we surveyed several experimental evaluations of cache-related variability. In the works we reviewed, the size-related jitter is usually treated as a special case of path-related jitter, when the sequence of execution (repeatedly) accesses so much code or data that the caches overflow for any layout (e.g.: [40]).

A comparison of 21 randomly chosen layouts of the same program showed 22% layout-related jitter in total execution time [16] for a direct-mapped cache with 20-cycles miss penalty. No significant jitter was found for 4-way set-associative caches, possibly because of the low probability that a random layout causes 5-way conflicts.

Path-related jitter is studied in [31] by measuring the execution times of three cyclic tasks for many iterations of the main loop with no preemption. The observed variation in the average execution time ranged -0.5% $\cdots$ +4% for a 32 KB cache, reaching to +10% for an 8 KB direct-mapped cache. These tasks were quite simple and synthetic. Real programs should show larger path-related jitter.

Batcher and Walker report some measurements of suspension-related jitter [15]. Using six small tasks on an ARM with a 50-cycle miss penalty, they observed an in-

crease of up to 25% in the run time of individual tasks, relative to the respective average value.

Preemption-related jitter appears in a study of the SPEC92 benchmarks [13] that reports much lower cache miss rates when the benchmarks were run as the single user process, without other concurrent processes. For example, for a 64 KB unified cache, the miss ratios were about 0.9% in the former case and 5% in the latter.

The survey confirms that the use of caches causes significant variability in the execution time of a program, but no experimental evaluation was found for the I-cache jitters in our specific domain. In the next section we introduce our approach to experimentally evaluate the impact of I-cache effects on the verification and validation process of industrial-quality on-board applications. In our experiments we used RapiTime, a measurement-based WCET analysis tool from Rapita Systems Ltd. [29], on a control-flow graph generated on purpose by Bound-T, a static WCET analysis tool from Tidorum Ltd. [34].

3.1 Cache-risk patterns

Experimental evaluations of caches in the literature typically study the cache behavior from the observation of a given application on a given hardware. For this reason, some application domains tend to establish reference benchmarks, for accuracy of analysis and extent of variability likewise. No such benchmarks exist or are in sight in the space domain of interest to our study.

We exploited the jitter taxonomy to devise experiments specifically aimed at revealing I-cache variability. Our approach differs from those we surveyed in that our experiments intentionally provoked and measured I-cache jitter through a non-exhaustive set of "cache-risk" code patterns. In those patterns, program code could be laid out in memory so badly as to cause up to 100% I-cache misses, whether persistently or sporadically. We selected and configured test software that we expected to contain cache-risk patterns. The experiments targeted the AT697E, which presently is *the* processor of reference to our domain.

To understand the principles on which we devised our cache-risk pattern for the AT697E, consider a loop, the body of which consists of 32 KB of code with consecutive addresses and no inner loops. The first loop iteration loads all the body code into the I-cache. Later iterations run with no cache misses (assuming that the I-cache is not changed by preemptions or interrupts). However, if the loop body grows larger than 32 KB then some cache sets are assigned more than 4 memory blocks; those sets are consequently overloaded and cause misses in the later iterations[1]. If the loop body is 40 KB or larger then all I-cache sets are overloaded and all code fetches are in fact cache misses.

[1]We assume LRU replacement policy.

For code that is not laid out consecutively in memory, similar problems may occur even with much smaller amounts of code. Consider for example, still on the AT697E, a loop body that just calls five procedures from five distinct modules. An uncouth linker may lay those modules in memory so that the five procedures are mapped to the same I-cache sets (whereby the procedures have the same address mod 8 KB). Figures 1 and 2 illustrate a good and a bad layout respectively.

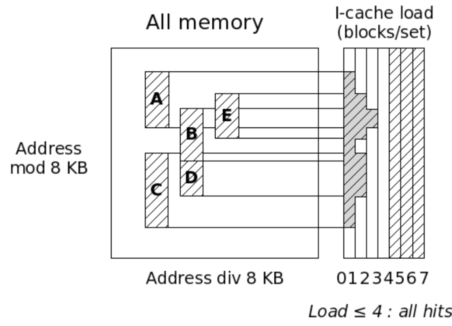

Figure 1. Good layout.

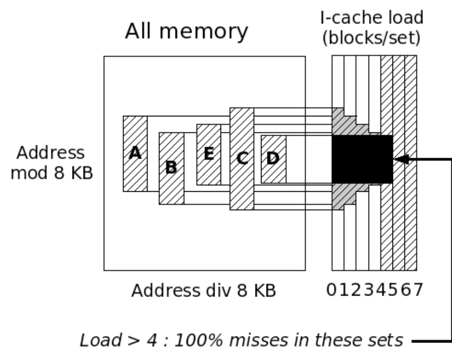

Figure 2. Bad layout.

With the good layout, the five modules lie at different memory offsets so that no cache set is loaded with more than 4 memory blocks and every memory access (other than the first) will result in a cache hit. Conversely, with the bad layout the five modules are laid at (nearly) the same memory offset so that more than 4 memory blocks are mapped to the same cache sets, which incurs 100% cache misses for the overloaded sets. The rest of the cache-risk patterns are variants of the pattern just described [36].

Test application. We used an on-board software system (OBSW) developed for the LEON processor by Thales-Alenia Space/France. The OBSW is a reusable baseline for new-generation systems, which makes its architecture most representative of a high-end industrial standard. The OBSW runs on OSTRALES, a real-time preemptive kernel

inria-00336526, version 1 - 4 Nov 2008

extended with features to: freeze the I-cache during interrupt handling; freeze the I-cache during the execution of specific tasks and unfreeze it for other tasks; freeze and unfreeze the I-cache at will by kernel calls.

3.2 Experiments

We ran over 20 experiments in the same input/output scenario, with different ways of using the cache and/or different memory layouts [36]. In all experiments the D-cache was disabled and the Instruction Burst Fetch mode[2] was enabled. We collected the execution traces, required by the RapiTime measurement-based tool, on the TSIM cycle-true simulator from Gaisler Research (professional version 1.3.9) [10] configured to simulate the AT697E. Under that configuration, the average TSIM timing accuracy is declared by its author to be better than 5% including cache effects. Ref. [36] provides a comprehensive description of the whole experimental process.

3.3 Discussion of results

All numerical results in cache-performance experiments are very specific to the application, the experimental conditions and the processor, cache and memory architectures. Thus, the results that we discuss below should not be taken to represent the "typical" nor the maximum impact of the LEON cache. They however effectively serve the purpose of providing evidence of cache-jitter effects.

Evidence of layout-related jitter. In order to gage the layout-related jitter, we chose the on-board Mission Time-Line (MTL) component of the OBSW because it contains loops that call several procedures, in a manner that resembles the pattern described in section 3.1. We concocted three layouts. The "good" layout minimizes the cache conflicts by placing all code consecutively in memory; since the MTL code is only 22 KB no cache set is overloaded. The two "bad" layouts try to create conflicts by placing procedures at similar address mod 8 KB. In the experiments we reserved the I-cache for the MTL component.

No essential difference between the layouts was observed when the cache was flushed on each MTL code activation. This is readily explained by the fact that no loop within one MTL activation calls more than 4 of the internal procedures, so that even the "bad" layouts do not overload the 4-way I-cache. Conversely, if the cache is not flushed then some contents are preserved over subsequent activations and the repeated activation of the MTL task forms an outer loop which does indeed call more than 4 procedures. In that case the 4-way I-cache can thus be partially overloaded and the execution time increases by 11% for the worst "bad" layout over the "good" one. These layouts are probably not the true worst and best case for the MTL code.

Evidence of suspension-related jitter. We measured the suspension-related jitter by comparing the execution time of a single task under three conditions: (1) only that task was allowed to change the cache; (2) all tasks were allowed to affect the cache contents; and (3) the cache was flushed on each task resumption. The experiments revealed a very clear case of the sought jitter effect: the MTL task execution time was 22% longer for case (3) than for case (1).

Evidence of preemption-related jitter. It was difficult to measure the preemption-related jitter because that depends on the point of preemption and not only on the cache conflicts between the preempted and preempting tasks. Our experiments simulated preemptions by flushing the cache at some chosen point in the "preempted" task. Notably, when the preemption point is placed very close to the start of the task, it gets similar to the suspension-related jitter. In fact, flushing the cache forces the worst case and may thus incur considerably pessimistic measurements. One could certainly be less brutal if the potentially preempting tasks could be studied for their access to the cache. Tool support to this end would be especially useful.

Overall, our experiments did not reveal very large I-cache preemption-related jitters. Most probably we failed to trigger the truly "bad" preemption points, which obviously is impractical to do without systematic tool support.

Our conclusions. Our observations show that cache jitters may hinder the reliable verification of real-time performance in OBSW. The results we obtained are proportional to the cache-miss penalty. Though low on the AT697E, it might get higher for more aggressive configurations of the LEON. For more advanced processors with larger miss penalties, a "bad" layout can easily double the execution time of a "good" layout. Dynamic effects, arising from input data values or interrupt arrival patterns, may also turn a "good" layout into "bad" one.

When real-time performance is critical to system integrity the risk of large variations in execution time is high enough to be worth considering and countering in some way. Ignoring the problem incurs the risk of occasional, unpredicted and quizzical performance effects.

4 Survey of methods for coping with cache unpredictability

Cache unpredictability may hit timing and schedulability analysis badly, as the quest for a safe and tight WCET

[2]On a I-cache miss, the referenced instruction block is loaded from the main memory, starting at the missing address until the end of the cache line. At the same time, the instructions are forwarded to the processor (i.e., streaming). If IBF was not enabled, the processor would have to wait for the whole block to be retrieved.

inria-00336526, version 1 - 4 Nov 2008

estimate (or upper bound) becomes considerably more difficult. The drastic countermeasure of disabling the cache altogether is generally not practicable since it is likely to cause an unacceptably severe impact on performance.

Where crude solutions are not an option, the only plausible route is to use some systematic method to attain a safe upper bound on the WCET, or a sufficiently reliable and precise estimate of it. Several approaches are proposed in the literature, which help one deal with the predictability threats caused by caches. We now survey those approaches from the standpoint of the jitter taxonomy introduced in Section 2 and experimentally confirmed in Section 3.

4.1 Cache-aware Static Analysis

Static analysis aims to give safe WCET bounds and safe schedulability analysis, and can include cache effects.

Static WCET analysis techniques usually build on abstract interpretation [9] and combine control-flow and call-graph information [19]. The worst-case execution path is found from an Integer Linear Programming problem with feasible paths and architectural features as constraints [39]. Those techniques build on a hardware model: the more complex the hardware, the less accurate the model. This holds for cache models too. Cache-aware WCET analysis produces a safe but often not tight WCET upper bound for a given task, as the analysis is exposed to a twofold overestimation: (1) computing abstract cache states entails a conservative loss of precision and overestimation of the number of cache misses; moreover (2) the preliminary step of control flow analysis is likely to introduce infeasible paths. From a safety standpoint, those techniques may also suffer from timing anomalies [12, 17].

Cache-aware WCET analysis targets a task at a time, on a given memory layout without considering how layout changes could affect execution time. Those techniques thus cover the path-related jitter exhaustively and only address the size-, concurrency- and layout-related jitters partially, on a case by case basis.

Cache-aware schedulability analysis [6] addresses the concurrency-related (especially preemption-related) jitter. It accounts for cache-related preemption delays in the response time of individual tasks. Yet, obtaining a tight estimate of the refill penalty is a difficult challenge and it often is a source of overestimation. We advise against this technique as being too coarse for our purposes.

Adhering to appropriate coding styles such as single-path programming [27] may reduce analysis overestimation and overcome the path-related jitter. Reducing the sections of code reachable through different execution paths will reduce both abstract state joins (as used in [9]) the principal cause of precision loss, and the complexity of the infeasible path problem. Size-related jitter may be increased because all the code in a loop is always fetched on every iteration of the loop. As a side effect, eliminating the path-related jitter will also decrease the concurrency-related one.

Informed code placement in memory [35] may reduce I-cache jitter by controlling code mapping to the cache lines. Controlling by hand the order in which object files are linked is neither a tenable nor a worthwhile engineering option. The starting location and size of every single subprogram should be controlled instead, which needs non-standard compiler/linker control features. Tool support for automatic code placement mechanisms at subprogram level thus is desirable help. Our future investigation does indeed experiment in this direction.

4.2 Measurement-based WCET Analysis

Static analysis is indeed safe but also pessimistic. An alternate approach may therefore aim at obtaining more realistic (i.e., less pessimistic) estimations by measuring execution times on the real hardware.

Measurement-based methods replace processor behavior analysis by actual measurements. They execute individual tasks on the given hardware for some set of inputs. Measurements are usually performed on program fragments that have a single execution path, independent of input data. The measured fragment execution times are combined into a WCET estimate for whole individual tasks by considering all possible paths in the task, as in the static analysis method, possibly using sophisticated statistics to compute execution-time distributions (e.g.: [5, 25]).

The initial state of the processor, including the cache state, at the start of each measured fragment of the program is a source of considerable execution-time variability. Measurement methods can either try to set the processor into a worst-case state before each measurement or simply use the state contextually reached by the processor. The former approach should be preferable, but finding and setting a worst-case initial state is quite a challenge.

In measurement-based methods, timing data are collected through either code or hardware instrumentation. Code instrumentation builds on the insertion of instrumentation points in the program so as to accumulate timing data during execution. However, the instrumentation points themselves affect the temporal behavior of the measured code, introducing the so-called probe effect. Conversely, hardware instrumentation builds on the provision of a debug support interface, e.g.: [33], which collects data passively from the processor, thus avoiding the probe effect.

Measurement-based WCET computation shares at least two common problems with static analysis approaches. As measurements are usually first performed on basic blocks and subsequently combined, they run the risk of including infeasible paths; furthermore, both methods ap-

inria-00336526, version 1 - 4 Nov 2008

inria-00336526, version 1 - 4 Nov 2008

ply to a single memory layout, thereby failing to account for the layout-related jitter. The exhaustive enumeration of all possible program executions is not feasible. Thus measurement-based methods cannot guarantee that the worst-case execution path was actually encountered and measured. The evidence discussed in Section 3 suggests that the effort invested in measurement-based methods pays off when the test cases are purposefully designed. This requires a cache-aware mentality that is seldom present (perhaps even intentionally) in current engineering practice.

4.3 Defeating the sources of cache jitters

Since neither static analysis nor measurement-based methods alone are capable of producing a safe and tight WCET estimate, we may try to increase the cache predictability by restricting the cache behavior. A more predictable cache not only reduces the overestimation of static analysis, but it also improves the safety of measurement-based WCET methods. Another approach is to use a more predictable fast memory device such as the scratchpad.

Cache partitioning. Perhaps the most immediate approach to increasing cache predictability is to partition the cache so that each task (or each main software function) is granted a piece of the cache for its own use. This allocation removes the dynamic aspect of the cache-mediated interference between tasks or functions but also reduces the usable cache size for each task or function, which may increase the unpredictability within each task or function.

Cache partitioning techniques can be implemented in hardware or in software. In hardware-implemented partitioning schemes [14, 21] the cache is divided into multiple partitions private to individual tasks and one partition that tasks may share. Each task is assigned one or more private partitions. Other approaches [38, 20] implement cache partitioning techniques by software. Bounding the memory references issued by individual tasks to a selected range of addresses effectively permits to define logical (as opposed to physical) partitions in the cache. Compiler and linker support is required for this form of cache partitioning: instructions and data are transformed to fit specific ranges of memory addresses, thereby producing a scattered memory map for each task. In order to avoid extremely small partitions one partition for each priority level may be defined [20], letting the tasks assigned at one and the same priority level share the same partition (so long as FIFO scheduling is adopted for tasks at one and the same priority level).

Cache partitioning removes the concurrency-related jitter (both suspension and preemption), though some residual concurrency-related jitter may still arise from shared partitions. If a single partition is shared among all tasks, then it will cause both preemption- and suspension-related jitter; otherwise, if partitions are shared among tasks within one and the same priority level (and the "FIFO within priorities" is adopted) then they will only suffer from suspension-related jitter. The size-related jitter may increase as partitioning reduces the amount of cache available for a given task; consequently, both layout- and path-related jitters can increase.

Partitioning creates a new factor, *partitioning-related jitter*: the execution time will also depend on the assignment of partitions to tasks. Assigning partitions to tasks can prove rather complex: naively assigning partitions in accordance with task priority (i.e., by urgency) or to task rates is unlikely to be the best choice. In principle, the assignment of the cache partitions should favor tasks whose code better exploits spatial locality (i.e., by use of local loops) and thus is more likely to benefit from the cache.

Cache locking. Another way to restrict cache behavior for the sake of improved predictability consists of exploiting hardware support to control the cache contents. Special instructions may be used to explicitly load information in the cache and to disable the cache replacement policy, thus locking all or some of the cache contents.

Since the cache content may be fixed and computed off line, cache behavior and accesses would become fairly predictable (at least for I-cache). Cache locking techniques may be static or dynamic. In static locking [22], cache contents are loaded at system start-up and remain unchanged until the system completes execution. In dynamic locking [7, 23, 24], cache contents are instead changed at specific reload points during execution. Dynamic locking techniques may be applied: (1) at system level, as statically-defined cache contents may be associated to individual tasks; and (2) at task level, as specific cache contents may be associated to designated code regions of the task. In the former case, reload points are placed at context switches, whereas in the latter case they are placed within the task.

Static locking eliminates both path- and concurrency-related jitters. Since the cache content is locked in the cache at start-up and never modified, the cache state will be the same irrespective of any execution path and despite any suspension or preemption incurred throughout. The layout-related jitter is actually reduced but it is not completely removed because in locked caches, in contrast with the scratchpad approach, the mapping between cache and main memory is still determined by the block placement policy. Cache locking avoids the size-related jitter, because the execution time of a program part depends only on its location (i.e., in the locked cache, or out of it), not on the size of the whole program (or the repeated part of a task). As the task code grows the cache-locked portion becomes a smaller fraction of the total code, and the execution time of the whole task probably grows for this reason, but this growth is still predictable.

Dynamic locking behaves quite similarly in relation to

inria-00336526, version 1 - 4 Nov 2008

cache jitters. Although the cache contents are changed at reload points, the execution time of tasks is affected by neither suspension- nor preemption-related jitters. What is affected instead is the task-switch overhead, as the cache state of the new task has to be loaded and locked before the task is resumed. However, this additional time is constant per task switch and can be handled in the schedulability analysis.

Static cache locking eases analysis but can lead to poor performance if the cache-worthy code in the application is larger than the cache size. Dynamic cache locking usually provides better performance but the (predictable) overhead imposed by dynamically reloading cache contents may not earn enough return in terms of performance improvements.

Since we do not wish to noticeably degrade cache performance we must use extreme care in selecting the portions of code to lock in the cache. This effort could prove really difficult unless adequate methodological and technological support is provided at compile time. Although several studies [7, 23, 24] suggest automatic methods to define appropriate reload points and cache contents, no industrial tools that implement the published methods are yet available.

Scratchpad memories. Scratchpad memories can provide fast memory access without affecting the overall system predictability. Scratchpad memories are small on-chip memories mapped into the hardware address space; similarly to caches, they can be unified or separated for data and instructions. Unlike caches, however, the contents of scratchpad memories are statically allocated to a separate address space or range and thus they are fully predictable.

Allocation in scratchpad memories is fully managed at software level, whether on control by the user or by the compiler. Several studies [26, 18] focus on efficient scratchpad allocation techniques, whether static or dynamic. Similarly to cache locking, scratchpad allocation techniques may turn out to be overcomplex for large applications, unless they are accompanied by specific tool support.

In terms of cache jitters the scratchpad approach is quite similar to that of locking caches. The only difference concerns the layout-related jitter: since the scratchpad uses the processor address space the content selection is not constrained by any kind of mapping. Nevertheless, the critical parts of code, which will be placed in the scratchpad, must be preventively selected and set aside from non-critical parts. This selection may force the developer to slice a subprogram into smaller chunks and to insert numerous jump or call instructions to connect them. The whole process thus verges on being overly complex and error prone unless industrial-quality automation support existed that could be relied upon (which we consider unlikely).

Table 1 summarizes the evaluation, in terms of cache jitters, of the surveyed approaches and suggests a combination of them that we consider promising.

Individual Remedies	**Cache-related jitters**									
	size		layout		path		susp		preempt	
	m	c	m	c	m	c	m	c	m	c
Static cache-aware WCET analysis		○		○		●		⊛		⊛
Meas.-based WCET analysis		○		○		●		⊛		⊛
Scratchpad	⋆		⋆		⋆		0		0	
Static cache locking	⋆		⋆		⋆		0		0	
Dynamic cache locking	⋆		⋆		⋆		0		0	
Cache partition per task	+		+		+		0		0	
Cache partition per priority level	+		+		+		±		0	
Single-path programming	+		=		0		–		–	
Combined Remedies										
Static cache-aware WCET analysis, cache partition per priority level, assume suspension clears cache	+	○	+	○	+	●	±	●	0	

Legend: m = magnitude of jitter; c = coverage of issue. Symbols: ○ = not fully covered; ● = covered; ⊛ = possibly covered; '=' = unmodified; '0' = eliminated; '+' = increased; '–' = decreased; ⋆ = controlled by allocation algorithm.

Table 1. Overview of surveyed remedies.

5 Selecting measures for better cache predictability

The cache behavior is intrinsically unpredictable since it depends on a complex set of interacting factors: execution history, memory layout and task interactions. Although a globally optimal solution is not achievable, we may still venture recommending measures for I-cache design and management that may contribute to reduced cache jitter. Our intent in this section is to formulate recommendations, drawn from the methods surveyed in Section 4, which we deem capable of attacking the sources of cache jitters discussed in Section 3. By way of example we also discuss the compliance to our recommendations of the I-cache design in the LEON2 processor family, its evolution LEON3, and the AT697E.

We differentiate between consolidated practices, as widely recognized means to improve the accuracy of both cache-aware static analysis and measurement-based methods, and practices that we recommend as worth exploring, as drawn from the scientific literature. Table 2 summarizes our recommendations and categorizes the relevant measures as practices pertaining to: system design; software design; and software development respectively.

5.1 Consolidated practices

Several studies [4, 28] suggest cache design choices the specific combination of which may be expected to improve

the overall cache predictability considerably.

Keeping separate I- and D-cache. Data and instruction caches should be separated to avoid mutual dependencies and structural hazards. Unified caches are far more arduous to analyze [4]. All LEON models and the AT697E likewise feature separate caches and independent cache controllers.

Adopting LRU replacement policy. LRU is widely recognized as the most analysable replacement policy [30]. Other policies, even those that promise optimal replacements (thus better than LRU), should be discarded on the grounds of being more unpredictable and thus less analyzable. Both LEON2 and LEON3 can be configured to implement a range of replacement policies. The caches in AT697E implement the LRU replacement policy.

Adopting set-associative caches. Set-associative caches should be preferred over direct-mapped and fully associative caches. The former should be disregarded as they incur more cache conflicts and thus yield lower performance than set-associative caches. Fully-associative caches instead entail larger information loss in static analysis since each memory block may be stored in any memory line, whereas set-associative cache bound the block placement unpredictability to cache sets. However, according to [9], fully-associative caches do not incur larger precision loss if LRU is adopted. Both LEON2 and LEON3 models can be configured to implement set-associative caches. The AT697E I-cache is 4-way set-associative.

Keeping set-associativity between 2 and 4. The LRU replacement policy is hardly implementable in hardware for set-associativity above 4 [8]. The degrees of set-associativity are thus limited to 2-4. The LEON2 and LEON3 do not allow cache associativity beyond 4-way, while the AT697E I-cache is just 4-way set-associative.

Inhibiting cache on interrupts. When a preempted task resumes execution after preemption it will find the cache state changed by either the interrupt handler or the preempting task. Interrupt handlers typically consist of short linear sequences of instructions which draw no benefit from the cache; inhibiting cache during interrupts will therefore increase predictability at virtually no performance penalty. Cache contents are preserved by freezing and unfreezing the cache prior and after each interrupt. The LEON models and the AT697E can freeze the I-cache automatically when handling an interrupt.

5.2 Recommended practices

We formulate some more proactive recommendations for improving the I-cache predictability. We refer them to the new-generation LEON3 processor model in particular, though we feel that they have wider applicability.

	Proactive measures for improved cache predictability	
System Design	Keeping separate I- and D-cache	CP
	Adopting set-associative caches	CP
	Adopting LRU replacement policy	CP
	Keeping set-associativity between 2 and 4	CP
	Inhibiting cache on interrupts	CP
	Cache partitioning	RP
	Cache locking	RP
	Scratchpad	RP
	Avoiding out-of-order execution and branch prediction	RP
Software Development	Providing an adequate debug interface	RP
	Cache flushing on activations and resumptions.	RP
	Disabling the D-cache	RP
Software Design	Adopting cache-aware design and coding styles.	RP

Legend: CP = consolidated practice; RP = recommended practice.

Table 2. Summary of measures.

Restricting, controlling or replacing the cache. Cache unpredictability may be reduced by restricting (i.e., partitioning) or controlling (i.e., locking) the cache behavior. We may also replace the cache with a scratchpad memory. Although not mandatory, architectural support for such supportive techniques is at least desirable.

Hardware support for *cache partitioning* entails a strict architectural model of the cache where both number of partitions and size thereof are statically defined, which is a rather application-specific condition. Arguably, implementation of hardware cache partitioning capable of allowing a flexible definition of number and size of partitions, to the extreme of disabling them altogether, seems most attractive to us. No LEON models currently support hardware cache partitioning: we suggest that they should.

Cache locking may prove extremely useful to improve the tightness of static analysis and, as a side effect, to obtain safer WCET measurements. The LEON2 and LEON3 both implement cache line locking. The fault-tolerant models of the LEON (LEON-FT) do not provide any locking facility since the bit used to lock a cache line is not protected against single-event upsets. The AT697E consequently does not implement cache locking. Interestingly however, the COLE processor from Saab Ericsson Space [32], a different implementation of the LEON2-FT model, does indeed support it. Cache locking should be implemented in future LEON-FT processors, in spite of the difficulty in identifying what parts of code are optimally locked in the cache.

Scratchpad memories may be a useful alternative, though the level of hardware support they need may prove problematic against the limited amount of available on-chip memory. In fact, whether it would be preferable to implement scratchpad alone or else in conjunction with caches, ultimately depends on the amount of available on-chip memory. To keep the processor model more flexible, it should be possible to disable the cache when scratchpads are used. The LEON2 provides an optional scratchpad RAM for data (sized up to 64 KB) but it does not allow one to completely disable the cache. The LEON3 fea-

inria-00336526, version 1 - 4 Nov 2008

inria-00336526, version 1 - 4 Nov 2008

tures improved support for scratchpad memories: both data and instruction scratchpads are optionally configurable (up to 512 KB) and both caches can be completely disabled. The LEON-FT models do not support scratchpad memories, supposedly because they are not fault-tolerant. We recommend that scratchpad memories and selective disabling of caches should be both supported in future releases.

Avoiding speculative execution and branch prediction. With respect to I-caches, two possible sources of hazardous inaccuracy in cache-aware static analysis stem from out-of-order execution and branch prediction. As suggested in [12], speculative execution could increase the frequency at which timing anomalies occur, whereas branch prediction may incur the so-called speculation anomalies. The LEON models and the AT697E allow neither of those accelerator features. There is of course tension between seeking the average performance improvement promised by speculative execution and branch prediction and wanting to avoid the extent of unpredictability resulting from them. In that light static branch prediction can be used to improve performance, while still preserving predictability [4].

Providing an adequate debug interface. Hardware support is required to efficiently apply measurement-based WCET methods on a given processor. The LEON3 allows non-intrusive debugging on target hardware through the Debug Support Unit (DSU3). Despite exhibiting a range of advanced features, the DSU3 does not provide adequate cache observability. For example, the DSU3 instruction trace buffer appears to be inferior to other industrial Debug Interfaces, like for example the Embedded Trace Macrocell for the ARM processors [1], which supports trace filtering.

The current LEON2 DSU fares even worse (though the COLE processor [32] does better than that) for it collects no memory access information at all. That information is crucial to analysing the cache behavior on factual evidence. Hence it should be captured, with suitable filtering options.

Cache flushing on activations and resumptions. Since during testing we are interested in measuring the WCET we often need to force the worst-case scenario on activations and preemptions. With respect to I-caches, the worst-case scenario corresponds to the situation whereby nothing is left in the cache after suspension or preemption of the task under analysis. This effect is achieved by flushing the I-cache at each task activation (after suspension) and resumption (after preemption). I-cache flushing is supported by all LEON models. The experiments we discussed in Section 3 suggest that brutally forcing a cache flush at each task resumption after preemption may be rather overkill.

Disabling the D-cache. When measuring the I-cache behavior the D-cache should be disabled. This avoids indirect interferences owing to data accesses which may affect measurements. The LEON processor models allow to selectively disable the D-cache.

Adopting cache-aware design and coding styles. Appropriate design and coding styles may certainly help improve WCET analysis by bounding the unavoidable sources of overestimation. Specific coding styles may affect cache performance since some code patterns may exploit temporal and spatial locality better than others; in fact they may attain better predictability as well, since appropriate coding styles may ease the detection of infeasible paths and incur smaller variability of execution time (e.g.: single-path-programming [27]). The current trend toward Model-Driven Engineering (MDE) may promote the factoring out of “good”, cache-friendly coding practices in the code generation engine used as part of the model transformation infrastructure which is central to the MDE paradigm. For example, shared code included in multiple paths in a manner that gives rise to vastly different execution contexts (e.g.: using far-apart bounds for loops in the shared code) could be automatically avoided, thus making it easier to identify feasible and infeasible paths.

In fact, a ramification of our investigations in this direction addresses the impact that software architectures (over and above coding styles) can have on the predictability of the I-cache. Interestingly, consideration of this possible impact seems to have been neglected so far in the literature.

6 Conclusions

I-caches may hamper the reliable verification of the real-time performance of on-board applications. A tiny modification (in memory layout only, for instance) to an already verified system can result in a system with a considerably different timing behavior. Although we can usefully rely on some best practices and recommendations to improve I-cache predictability, we fear they are still distant from being ultimate solutions. Static and measurement-based WCET analysis do not account for the layout-related jitter: tool support is undoubtedly needed to evaluate the effect of memory layouts on the cache and to help seek the least offending, if not the best, linker directives. Cache locking, partitioning or scratchpad memories and other such techniques may turn out to be frustratingly ineffective, unless some kind of methodological approach is defined to guide their use and to analyze their effect. Tool support is badly needed to tell which tasks or functions are best fit for being allocated cache space.

Acknowledgments. The work from which this paper has originated was carried out with ESA/ESTEC support under the PEAL contract (ESTEC/Contract 19535/05/NL/JD/jk) and with the precious collaboration of the Thales-Alenia

Space/France team at Cannes. To access reff. [36, 37] please contact: *maria.hernek@esa.int*.

References

[1] ARM Embedded Trace Macrocell. `http://www.arm.com/products/solutions/ETM.html`.

[2] Atmel Corp. *Rad-Hard 32-bit SPARC V8 Processor AT697E*, rev.4226d-aero-02/06 edition, 2006.

[3] S. Basumallick and K. Nilsen. Cache Issues in Real-Time System. In *ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, 1994.

[4] C. Berg, J. Engblom, and R. Wilhelm. Requirements for and Design of a Processor with Predictable Timing. *Design of Systems with Predictable Behaviour, Dagstuhl Seminar Proc., Internationales Begegnungs und Forschungszentrum (IBFI)*, 2004.

[5] G. Bernat, A. Colin, and S. Petters. WCET Analysis of Probabilistic Hard Real-Time Systems. In *Proc. of the 23rd Int. Real-Time Systems Symposium*, 2002.

[6] J. Busquets-Mataix and A. Wellings. Adding Instruction Cache Effect to Schedulability Analysis of Preemptive Real-Time Systems. In *Proc. of the 2nd Real-time Technology and Application Symposium*, 1996.

[7] A. Campoy, A. Ivars, and J. Busquets-Mataix. Dynamic Use of Locking Caches in Multitask, Preemptive, Real-time Systems. In *IFAC 15th World Congress*, 2002.

[8] J. Engblom. On Hardware and Hardware Models for Embedded Real-Time Systems. In *IEEE Workshop on Real-Time Embedded Systems*, 2001.

[9] C. Ferdinand and R. Wilhelm. Fast and Efficient Cache Behavior Prediction for Real-Time Systems. *Real-Time System*, XVII:131–181, 1999.

[10] Gaisler Research. `http://www.gaisler.com`.

[11] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan-Kaufmann, 4th ed., 2007.

[12] J. Reineke et al. A Definition and Classification of Timing Anomalies. In *Proc. of the 6th Int. Workshop on WCET Analysis*, 2006.

[13] J.D. Gee, M.D. Hill, D.N. Pnevmatikatos, and A.J. Smith. Cache performance of the SPEC92 benchmark suite. *Micro IEEE*, XIII(4), 1993.

[14] D. Kirk. SMART (Strategic Memory Allocation for Real-Time) Cache Design. In *Proc. of the the Real-Time Systems Symposium)*, pages 229–239, 1989.

[15] K.W. Batcher and R.A. Walker. Interrupt triggered software prefetching for embedded CPU instruction cache. In *Proc. of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2006.

[16] L.P. Bradford and R. Quong. An empirical study on how program layout affects cache miss rates. *SIGMETRICS Perform. Eval. Rev. 27, 3*, (Rev. 27, 3):28–42, 1999.

[17] T. Lundqvist and P. Stenström. Timing Anomalies in Dynamically Scheduled Microprocessors. In *Proc. of the 20th Real-Time Systems Symposium*, 1999.

[18] P. Marwedel, L. Wehmeyer, and M. Verma. Cache-aware Scratchpad Allocation Algorithm. In *Proc. of Design, Automation and Test in Europe*, 2004.

[19] F. Mueller. Predicting instruction cache behavior. *Language, Compilers and Tools for Real-Time Systems*, 1994.

[20] F. Mueller. Compiler support for software-based cache partitioning. In *ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems*, 1995.

[21] H. Muller, D. May, J. Irwin, and D. Page. Novel Caches for Predictable Computing. Technical Report CSTR 98-011, Dep. of Computer Science, University of Bristol, 1998.

[22] I. Puaut. Cache Analysis vs Static Cache Locking for Schedulability Analysis in Multitasking Real-time System. In *Proc. of the 2nd Int. Workshop on WCET Analysis*, 2002.

[23] I. Puaut. WCET-Centric Software-controlled Instruction Caches for Hard Real-Time Systems. In *Proc. of the 18th Euromicro Conference on Real-time Systems*, 2006.

[24] I. Puaut and A. Arnaud. Dynamic Instruction Cache Locking in Hard Real-Time Systems. In *Proc. of the 14th Int. Conference on Real-Time and Network Systems*, 2006.

[25] I. Puaut and J. Deverge. Safe Measurement-based WCET Estimation. In *Proc. of the 5th Int. Workshop on WCET Analysis*, 2007.

[26] I. Puaut and C. Pais. Scratchpad Memories vs Locked Caches in Hard Real-time Systems: a quantitative comparison. In *Proc. of the Conference on Design, Automation and Test in Europe*, pages 1484–1489, 2007.

[27] P. Puschner. The Single-Path Approach towards WCET-analysable Software. In *Proc. of the IEEE Int. Conference on Industrial Technology*, 2003.

[28] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm. The Influence of Processor Architecture on the Design and the Result of WCET Tools. *Proc. of IEEE*, XCI(8), 2003.

[29] Rapita Systems Ltd. `http://www.rapitasystems.com/wcet.html`.

[30] J. Reineke, D. Grund, C. Berg, and R. Wilhelm. Timing Predictability of Cache Replacement Policies. *Real-Time Systems*, 37(2):99–122, November 2007.

[31] R.G. Scottow and K.D. McDonald-Maier. How to manage determinism and caches in embedded systems. In *Proc. of the 2th UK Embedded Forum*, 2005.

[32] The COLE ASIC by Saab Ericsson Space. `http://www.saabgroup.com`.

[33] The Nexus 5001 Forum. `http://www.nexus5001.org`.

[34] Tidorum Ltd. `http://www.bound-t.com`.

[35] H. Tomiyama and H. Yasuura. Optimal code placement of embedded software for instruction caches. In *In European Design and Test Conference*, pages 96–101. IEEE, 1996.

[36] T. Vardanega, G. Bernat, A. Colin, J. Estevez, G. Garcia, C. Moreno, and N. Holsti. PEAL Final Report. Technical Report PEAL-FR-001, ESA/ESTEC, 2007.

[37] T. Vardanega, N. Holsti, and J. Estevez. Study of Cache Usage Effects in Typical OBSW. Technical Report PEAL-TN-WP13, ESA/ESTEC, 2006.

[38] A. Wolfe. Software-based cache partitioning for real time applications. In *Proc. of the 3rd Int. Workshop on Responsive Computer Systems*, 1993.

[39] Y.S. Li, S. Malik, and A. Wolfe. Cache Modeling for Real-Time Software: Beyond Direct Mapped Instruction Caches. In *Proc. of the 17th Real-Time Systems Symposium*, 1996.

[40] Z. Xu et al. An analysis of cache performance of multimedia applications. *IEEE Trans. on Comp.*, LIII(1), 2004.

inria-00336526, version 1 - 4 Nov 2008