Impacts of Software Architectures on Cache Predictability in High-Integrity Systems

E. Mezzetti and T. Vardanega *

* University of Padua, via Trieste 63 I-35121, Italy, (e-mail: {emezzetti, takio.vardanega}@math.unipd.it)

Abstract: The current trend in the High Integrity Real-Time System domain steadily moves—by need if not by will—toward the adoption of processors with advanced architectural features like caches. Whilst caches have been shown to be a very effective means to speeding up memory accesses in the average case, the unpredictability of caches threatens the practicability and trustworthiness of system analysis and validation. In this paper we consider the impact that real-time software architectures may have on the cache behavior, with a view to addressing system predictability in an early stage of the software development process.

Keywords: System Architectures, Cache memories, Worst-case Execution Time, Timing Analysis, Timing Jitter

1. INTRODUCTION

In the High Integrity Systems (HIS) domain, where safety and security are paramount concerns, the verification and validation process is placed under the obligation to attain trustworthy information on the timing behavior that the system may exhibit during execution. The validation of HIS is therefore not limited to functional concerns but must also ascertain correctness in the time, space and communication dimensions. In state-of-the-art approaches to the development and verification of HIS, both functional and non functional system requirements are addressed as early in the development process as possible. Schedulability analysis techniques are increasingly applied at architectural level, rather than on the production of the actual target code, so as to acquire sound and early information of the timing behavior of the system. A crucial input to schedulability analysis is the determination, whether by analysis or measurement, of the worst-case execution time (WCET) of the tasks that are to perform the system activities at run time. The resulting values are then combined together to determine the schedulability of the system as a whole.

The adoption of advanced processors equipped with a score of acceleration features like caches, complex pipelines and branch predictors, threatens the determination of safe and tight WCET bounds making it considerably more difficult and uncertain. This may in turn jeopardize the soundness of schedulability analysis. In particular, it is widely acknowledged that the presence of caches causes applications to exhibit a disturbingly variable timing behavior as the execution time of a software program or parts thereof may vary upon the total program size, the memory layout, the order in which memory accesses are performed, and the pattern of interrupts or preemptions that occur during execution. Canonical approaches to coping with cache variability, which focus on accounting for (or restraining) the effects of such factors, typically build on the assumption that the WCET behaviour of each single task is exclusively influenced by the task code itself and only to a minor extent by the interferences from other tasks.

The increasing complexity in the software component of HIS has led to the adoption of compositional development approaches that follow some incarnations of the separation of concerns paradigm. From the standpoint of component-based design, where each software module that incorporates a distinct quota of the system functionality is regarded as a component in its own merit, the characteristic software architectures—which are inherently non-functional—play an important role. As far as this paper is concerned the software architecture impacts on the runtime behavior of the system in so far as it may affect the sources of cache-related variability, either statically or dynamically or even both.

Whereas the influence of hardware features and software components on the run-time behavior of HIS has been widely acknowledged in the literature, in this paper we conjecture on the impact that the software architecture may have on the cache-related variability, mainly focusing on the instruction cache (I-cache) behavior. From the standpoint of the cache predictability problem we discuss whether and to what extent the choice of a software architecture may affect the overall predictability of the system. We consider this paper as the initial step of a path that addresses system predictability issues at architectural level. The remainder of the paper is organised as follows: section 2 introduces the cache predictability issues and the state-of-the-art approaches to cache analysis; section 3 discusses the impact of software architectures on the cache behavior providing examples of static and dynamic architectural impacts; section 4 draws some conclusions.

2. CACHE PREDICTABILITY PROBLEM

Caches are small and fast memories, often located on chip, which store most recently accessed data and instructions (Hennessey and Patterson, 2007). By exploiting the prin-
ciple of reference locality, caches can narrow the speed gap between fast processors and slower main memories. Most modern processors define separate caches for data (D-cache) and instructions (I-cache) as opposed to unified caches. When the CPU issues a memory access, the memory location is first addressed to the cache memory: if the required data are found in there (hit), they are immediately sent to the CPU. Otherwise (miss) the request is passed to the slower main memory.

Memory data are logically organised as fixed-size memory blocks; thus, in case of cache miss, a memory block containing the required memory address is retrieved from the main memory and placed into an equally sized cache frame (cache line) according to a placement policy. A memory block can be placed into exactly one cache line (direct-mapped caches), or in a number of cache lines or set (set-associative caches), to the extent of any cache line (fully-associative caches). The number of cache lines in a set is referred to as way; thus, in a 4-way set-associative cache, the same memory block can be placed into any of 4 different cache lines (see Figure 1). Hennessy and Patterson (2007) identify three types of cache misses: compulsory misses (or cold misses), when an address is first referenced; capacity misses, when the current working set exceeds the cache capacity; and conflict misses, when multiple references map to (and compete for) the same cache line. Typically, on a cache miss, a cache line is evicted from the cache to make room for the block retrieved from main memory, according to a replacement policy. Common cache replacement policies are Least Recently Used (LRU), Random, FIFO, or some pseudo (i.e., approximate) variant of them.

2.1 Instruction Cache Unpredictability

Although caches typically improve performance in the average case, they may introduce considerable variability in the execution time of a program. At every point in the program, the execution time of the next memory access depends on both the address of the referenced address and what addresses the program recently referenced in its execution path. The cache behavior is intrinsically unpredictable since it depends on a complex set of interacting factors: execution history, memory layout and task interactions. The actual sources of I-cache variability and thus of potential unpredictability (see Mezzetti et al., 2008) can be traced back to a set of cache-related jitters:

- **size-related jitter**, which refers to variability due to the total size of the program code in relation to the cache size, as both factors limit the total number of useful memory blocks that can be stored in the cache;

- **layout-related jitter**, which addresses the actual memory layout of the program code, which may in turn cause differing conflict miss patterns;

- **path-related jitter**, which stems from the control structures in a program such as conditional statements, variable-bound loops, etc., as 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; and

- **concurrency-related jitter**, which depends on task scheduling, in that when an interrupted, suspended or preempted task resumes, its execution time may vary according to the cache state as it results from changes caused by the computation performed by the preempting tasks or interrupt handlers. A similar variation can be identified in case of voluntary task suspension.

Several approaches have been proposed in the literature to cope with cache analysability (Wilhelm et al., 2008) by either accounting for cache behavior in both WCET and schedulability analysis or making caches more predictable, at the expense of replacing them altogether with more predictable fast memories (i.e., scratchpads).

2.2 Cache-aware Static Analysis

Static WCET analysis techniques try to compute a WCET bound for a given program from abstract models of both the hardware and the software. Several methods have been proposed to cope with cache-equipped processors, based on abstract interpretation (Ferdinand and Wilhelm, 1999), static cache simulation (Mueller, 1994) or exploiting integer linear programming (ILP) (Li et al., 1996). They share three fundamental steps: hardware timing analysis, program flow analysis, and a final calculation step. Given a precise hardware model, static analysis techniques can produce a safe bound on the WCET of a given program. To avoid incurring excessive pessimism however the WCET estimate should not only be safe but also tight, with as little overestimation as possible in comparison to the actual WCET incurred at run time. Cache-aware static analysis in general suffers from inherent sources of overestimation mainly due to the abstraction of execution contexts it explicitly (e.g. Ferdinand and Wilhelm, 1999) or implicitly (e.g. Mueller, 1994) relies on, and the inclusion of infeasible execution paths in the program flow analysis step. Furthermore, from the safeness standpoint, static analysis techniques may be exposed to timing anomalies (Lundqvist and Steenström, 1999; Reineke et al., 2006); within a sequence of instructions, the variation in the execution time of a single instruction may incur a greater or counter-intuitive variation over the whole sequence.

2.3 Measurement-based methods

Static analysis is indeed safe but also pessimistic and not yet widely applied in the industrial process, where end-to-end measurements during functional testing remains the de facto standard practice in WCET estimation. Unfortunately, classical end-to-end measurements are inadequate in the face of advanced architectural features like caches.
as they incur more variable execution times and make it much harder for the test run to hit the actual worst-case scenario.

Hybrid measurement-based methods (Bernat et al., 2002; Puaut and Deveuge, 2007) try to compute tight WCET estimates by measuring actual execution times of individual tasks on the real hardware, instead of using processor models. Measurements are performed on small program fragments (or basic blocks) and then combined together to obtain a WCET estimate for the task, on the basis of the results of a control-flow analysis step, much like with static analysis approaches. Timing data are collected through either code or hardware instrumentation. In the former case, the instrumentation points in the program affect the temporal behavior of the measured code, via a phenomenon that is known as the probe effect. Bernat, Colin, and Petters (2002) suggest a promising approach that builds on probability theory. However, similarly to static analysis techniques, the inclusion of infeasible paths in combining timing information leads to less tight WCET estimates. In addition to the hurdle of controlling pessimism, the key challenge of measurement-based methods is to attain measurable coverage of the execution paths, so as to place sufficient confidence in the WCET estimate, all the more in the presence of hardware acceleration features.

2.4 Cache-aware schedulability analysis

The early methods of both static and measurement-based analysis build on the simplifying assumption that the WCET of a task solely depends on the task itself. More recently however several approaches have sought ways to account for the interference caused by other tasks. Those more advanced methods extend existing schedulability analysis techniques to consider the so-called cache-related preemption delay (CRPD), as an estimate of the time required by a preempted task to restore its cache state at resumption (Lee et al., 1998; Sasnichut and Ernst, 2005; Altmeyer and Burguiere, 2009). Although the refill penalty associated with CRPD is differently estimated in each approach (i.e., the entire cache, useful cache blocks (UCB), line intersection, etc.) the computed WCET estimate is a safe but not tight upper bound to the actual WCET.

2.5 Improving cache predictability

Provided that none of the above approaches alone can always produce a safe and tight WCET estimate, the only solution is to seek some way to increase the cache predictability. In cache locking techniques (Puaut, 2002; Campoy et al., 2002; Puaut, 2006; Puaut and Arnaut, 2006) the control over the cache content is left to the programmer, by exploiting specific hardware support. Both static and dynamic cache locking extremely reduce the cache-related variability and increase predictability, as the cache content is always statically determined. Cache partitioning can be applied, either by hardware (Kirk, 1989; Muller et al., 1998) or software (Wolfe, 1993; Mueller, 1995), to slice the cache space, reserving a piece of cache for each task. This allocation removes the cache interference between tasks, although, as a drawback, it also reduces the usable cache size for each task. Another approach is to use more predictable fast memory devices such as scratchpads (Marwedel et al., 2004; Puaut and Pais, 2007), instead of caches. Scratchpad memories are small on-chip memories mapped to the hardware address space, which can provide fast memory access without affecting the overall system predictability. In fact, the contents of scratchpad memories are statically allocated in a separate address space, which makes it always aware of its contents.

We can attain better cache predictability by avoiding hardware features or design choices that may incur more unpredictable behavior (Mezzetti et al., 2008), such as dynamic branch predictors, out-of-order execution, unpredictable cache replacement policies, etc. Also appropriate coding styles and/or code patterns may help improve WCET analysis by removing unnecessary sources of overestimation due to specific code constructs. More analysable software may allow a more effective detection of infeasible paths and reduce execution time variability, as with single-path programming (Pushchiner, 2003).

3. ARCHITECTURAL IMPACT ON CACHE

From the standpoint of the component-based engineering approach, software architectures play a critical role in determining the overall structure of a system as well as the pattern of tasks interleaving and interactions. Single tasks act as components that are immersed into the system to realize the overall system functionality. Software architecture specifications address both system configuration and actual execution. It is thus obvious that we cannot analyse single tasks timing behavior in isolation but we should account for the impact of the software architecture both in their static (i.e., compile-time) and dynamic (i.e., run-time) behavior. Cache-aware schedulability analysis techniques (e.g. Lee et al., 1998) account for cache-related inter-task interference in the response time behavior of each task: in actual fact, task interaction and interference are inherently dependent on the (dynamic) software architecture of choice.

As observed in Section 2.5, we can try to increase system predictability and analysability with respect to caches, by reducing cache variability. In that light, the software architecture should be evaluated over and above single components as a source of impact on cache behavior and timing analysis. Software architectures might affect most sources of cache-related variability: memory layout, execution path, and concurrency-related jitters. Consequently, we can reasonably argue that one and the same software component may incur variable execution time behavior when immersed in distinct software architectures.

It is widely recognised that coding styles and patterns may affect both negatively and positively the tightness of cache analysis (e.g., Pushchiner, 2003). Similarly, if some invariant correlation could be ascertained between software architecture and predictability, over and above the induced code patterns, then we could use such knowledge as a factor of choice between alternative software architectures. Moreover, from a Model-Driven Engineering (MDE) standpoint, this evidence could usefully leverage cache-awareness in automatic code generation engines, thus promising to attain greater predictability by construction as opposed to via ad-hoc measures. To the best of our knowledge, while other studies aimed at evaluating
software architectures (e.g., Clements et al., 2001), it is the first time that software architectures are systematically addressed from the standpoint of timing variability and predictability.

Software architectures may affect the timing behavior of the system by inducing variability in the execution time of its components. This architectural effect impacts the sources of cache-related variability, creeping up at either compile- or run-time. In the following, we introduce those kinds of architectural impact, especially focusing on its effect on I-cache behavior and predictability.

### 3.1 Static impact

We refer to static impact as the effects of software architectures on the execution time of a given component (i.e., task) that arise at compile time. In other words, we focus on the coupling of software architecture with those sources of variability that are determined by the static architectural configuration. In terms of the taxonomy introduced in Section 2.1, that kind of impact mainly addresses layout- and path-related jitters.

Software architectures consist of a set of constructs or mechanisms that define the architecture organization itself and enforce its intended execution behavior. Each specific software architecture may entail some changes in the code surrounding its constituting components or may require the insertion of new code just inside those components (whether explicitly by the user or automatically by the compiler). Let us consider a simple example of software architecture that undertakes run-time monitoring of the system execution. Run-time monitoring is a fairly popular technique that is adopted in several application domains to ensure that the run-time behavior of the system conforms to certain bounding parameters as well as to detect possible violations of non-functional (e.g., timing) constraints (Chodrow et al., 1991). Suppose that the example architecture implements monitoring by use of code instrumentation so that each component is augmented with some extraneous code tasked to collect the run-time traces of the component behavior. The impact of the referenced architecture is illustrated in Figure 2.

As it so happens, the execution time overhead incurred by the instrumentation code (which may consist of elements ranging from the simple invocations of tiny trace procedures to the full extent of executing fault handling actions) is not limited to the time taken by the execution of the instrumentation points themselves. We must in fact understand and consider the impact that such code insertions may have on the I-cache behavior, with respect to both the component execution path and (more subtly) to its memory layout.

With respect to the former, the insertion of instrumentation code may obviously incur a variable timing behavior of the instrumented task as the execution path is no longer the same as that of the task in isolation. The difference in the execution time varies upon the amount and complexity of the inserted code. Therefore different software architectures that implement run-time monitoring, for example, at different levels of granularity should account for different execution time variations on individual tasks. Since instrumentation code should typically not trigger overly complex procedures, they should not pose a major direct threat to I-cache predictability.

A subtler behavior can be observed instead in relation to the architectural impact on the memory layout of each task. Layout-related jitter is the most critical source of variability with respect to I-cache as a tiny modification in memory layout can result in considerably different timing behavior, as shown in (Mezzetti et al., 2008): under some circumstances a bad memory layout can incur extremely poor cache performance as a result of a pathological cache behavior (Quiñones et al., 2009) where each memory access results in a conflict miss. For example, in a 4-way set-associative cache with LRU replacement policy, a loop calling five different procedures that overlap in the I-cache will entail a chain-effect in which each procedure is always evicted from the cache before being referenced again (Figure 3). This is because the effectiveness of the cache replacement policy highly depends on the actual memory layout of the program. Avoiding unnecessary conflict misses by means of cache-aware memory layout placement directives is a comparatively straightforward measure that may incur important cache performance improvements and possibly help reduce the amount of overestimation in the estimated WCET bounds.

![Diagram of architectural impact on memory layout](image-url)

**Fig. 2.** Effects of code instrumentation.

![Diagram of code insertion and I-cache behavior](image-url)

**Fig. 3.** Avoidable pathological cache behavior within a loop calling A, B, C, D, E procedures.
Software architecture specifications often define how tasks should interact during system execution. When it comes to caches, where the static impact relates to layout- and path-related jitters, the dynamic impact of a software architecture may influence the cache behavior by affecting the concurrency-related jitter.

Since the cache behavior is determined by the cache state, every modification to the cache content may incur variable execution time on the subsequent cache accesses. CRPD approaches (e.g., Lee et al., 1998) aim at accounting the effects of task preemption on the WCET by accounting for the time required to restore the cache state on task resumption, in the response time of each single task. Software architecture considerations cannot therefore be disregarded when determining the interference in the task set that forms the system, for the patterns of possible interrupts and preemptions are actually determined by the architectural specification, by way of synchronization protocols, resource access protocols, timer interrupts, and so forth. Hence, different architectures virtually impose different concurrency behavior and thus induce concurrency-related jitter effects.

Furthermore, individual software architectures may also include and require run-time mechanisms to enforce specific protocols that reflect the dynamic specification of the architecture. This is for example the case with the run-time enforcement of timing constraints in highly critical systems that is intended to provide temporal isolation between logical partitions. Run-time enforcement is typically implemented by having dedicated run-time entities monitor system execution and take corrective actions in response to any detected misbehavior. Such control entities are likely to further contribute to the dynamic impact of the software architecture.

Let us for example consider a software architecture that provides for execution-time timers, as in Ada 2005 (ISO SC22/WG9, 2005) and POSIX, to enforce tasks obedience at run time to their stipulated timing behavior. Activating, suspending or resetting such timers would activate as many interrupts and increase the refill penalty estimate, which should be accounted for in schedulability analysis. The granularity at which such timers are applied to the task (i.e., at task level, at single function level, etc.) differently affects the overall system timing behavior with respect to cache interference.

Similarly to the static impact case, a cache-aware architecture may reduce the cache-related interference between tasks, improving the precision of cache analysis and allowing a more predictable cache behavior.

4. Conclusions

Software architecture features impact the static configuration and the dynamic run-time behavior of the system by affecting the execution time of the tasks that execute system operation at run time. The timing behavior of such tasks cannot be determined in isolation but should account for the software architecture they are immersed in.

The architectural specification affects both the behavior and the analysability of the I-cache as it influences the very sources of cache-related variability. In this paper we argued...
that software architectures may be the source of extremely different extents of variability, in particular with respect to the memory layout of individual tasks and the interference occurring between them. We maintain that uncontrolled memory layout should be avoided by all means since it can be the source of unduly poor cache performance and may further complicate cache analysis.

We do not mean software architectures to become a new subject of cache analysis per se. Conversely, we contend that they should be regarded (and evaluated) as a vehicle to better control the overall system predictability. Architectures, and more generally the whole software development process, should informally embrace a cache-aware attitude if it wants to improve system predictability and analysability by reducing the extent of cache variability.

Our immediate plan is to further support our reasoning by completing an accurate experimental evaluation of the impact of different architectural variants of a representative test application. We expect that those investigations could attain a better understanding of architectural impact on the system timing behaviour and would possibly reveal some software architecture design guidelines and principles to better cope with cache variability and predictability.

REFERENCES


