Alma Mater Studiorum – Università di Bologna

DOTTORATO DI RICERCA IN

Informatica

Ciclo XXIV

Settore Concorsuale di afferenza: 01/B1

Settore Scientifico disciplinare: INF/01

Cache-aware Development of High Integrity Real-time Systems

Presentata da: **Enrico Mezzetti**

Coordinatore Dottorato **Relatore**

Prof. Maurizio Gabbrielli **Prof. Tullio Vardanega**

Esame finale anno 2012

"There are two ways of constructing a software design: one way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies.

The first method is far more difficult."

C.A.R. Hoare

Abstract

Cost, performance and availability considerations are forcing even the most conservative high-integrity embedded real-time systems industry to migrate from relatively simple hardware single-core processors to ones equipped with caches and other acceleration features. This irresistible migration threatens to disrupt the approach, practices and solutions that industry had developed and consolidated over the years to address the challenge of timing analysis. Industry that are confident with the efficiency and effectiveness of their verification and validation processes for old-generation hardware processors, do not have sufficiently solid insight on the effects that may be incurred by the migration to cache-equipped processors. Caches are perceived as an additional source of complexity, which has potential for shattering the guarantees of cost- and schedule-constrained qualification of their systems. The current industrial approach to timing analysis is ill-equipped to cope with the potentially large jitters incurred by the use of caches. Conversely, the application of advanced WCET analysis techniques on real-world (hence large and massively complex) industrial software, developed without analysability in mind, is often irremediably infeasible.

Building on those considerations we propose the adoption of a structured "cache-aware" development approach aimed at minimising the extent and source of cache jitters, as well as at enabling the application of advanced WCET analysis techniques to large complex systems. We defined our approach as a combination of three main constituents: (i) identification of those software constructs that may impede or exceedingly complicate timing analysis in large and complex systems; (ii) elaboration of practical means, under the model-driven engineering (MDE) paradigm, to enforce the automated generation of software that is time predictable by construction; and (iii) implementation of an incremental layout optimisation method to remove cache jitters that stem from the software layout in memory, with the intent of actively facilitating incremental software development, which is of high strategic interest to industry. The integration of those constituents in a coherent structured approach to timing analysis achieves two interesting properties: the resulting

software product is at one time usefully analysable from the earliest releases onwards - as opposed to only becoming so when the system is final - and more easily amenable to advanced timing analysis by construction, which may consequently scale to large and complex real-world systems.

Disclaimer The work that conducted to this thesis has been supported by Thales Alenia Space - France (TAS-F), a large established industry in the space domain, under grant agreement number 1520007750 and benefited from valuable discussions within the TAS-F Platform and Satellite Research Department in Cannes. However, the ideas and results hereby presented reflects only the author's opinions and do not necessarily engage those of Thales Alenia Space - France, unless explicitly stated.

Acknowledgements

First of all, I want to express my sincere gratitude to my advisor Prof. Tullio Vardanega for giving me the opportunity to enrol in the PhD program and to work all these years under his motivation, wise guidance and precious support.

I would also like to thank Prof. Fabio Panzieri and Prof. Isabelle Puaut, for being part of my supervisory committee and for their valuable advices, and Prof. Björn Lisper and Prof. Peter Puschner for serving as referees on my thesis committee and for their sharp comments on a previous version of this dissertation.

I'm grateful to all the people I met during this amazing - and sometimes frustrating - journey that is a PhD program. It would be impossible to name them all: my colleagues at University of Padua, for the uncountable coffees and the invaluable giddy conversations; my temporary colleagues in Cannes, for being more than office mates and for making me feel at home during my long-term visit at Thales Alenia Space; and all the interesting people I happened to meet during all short-term research visits, conferences and workshops. Thanks to all for being part of this!

Finally I want to heartily thank my friends and more importantly my family and my partner for their encouragement, patience and support.

Table of Contents

List of Tables

List of Figures

List of acronyms

ACS	Abstract Cache State
AH	Always Hit
AM	Always Miss
AOCS	Attitude and Orbit Control System
BSP	Board Support Package
CBSE	Component-Based Software Engineering
CCM	CORBA Component Model
CCS	Concrete Cache State
CFG	Control-Flow Graph
CRBD	Cache-Related Blocking Delay
CRPD	Cache-Related Preemption Delay
ESA	European Space Agency
FH	First Hit
FIFO	First In First Out
FM	First Miss
FPPS	Fixed Priority Preemptive Scheduling
GNC	Guidance, Navigation and Control
HIRTS	High Integrity Real-Time Systems
HMBTA	Hybrid Measurement-Based Timing Analysis
HW	Hardware
ILP	Integer Linear Programming
IMA	Integrated Modular Avionics
IPET	Implicit Path Enumeration Technique
LCT	Loop-Call Tree
LRU	Least Recently Used
M2T	Model-to-Text
MBTA	Measurement-Based Timing Analysis

MDE	Model-Driven Engineering
MIPS	Million Instructions Per Second
MMU	Memory Management Unit
OBSW	On-board software
PLRU	Pseudo Least Recently Used
RTA	Response Time Analysis
RTOS	Real-Time Operating System
RTK	Real-Time Kernel
SCM	Space Component Model
STA	Static Timing Analysis
SW	Software
TA	Timing Analysis
TAS-F	Thales Alenia Space - France
TLB	Translation Look-aside Buffer
$\overline{\text{UCB}}$	Used Cache Blocks
UCB	Useful Cache Blocks
UML	Unified Modeling Language
V&V	Verification and Validation process
WCET	Worst-Case Execution Time
WCG	Weighted Call Graph
WCP	Worst-Case Path
WCRT	Worst-Case Response Time

Chapter 1

Introduction

1.1 Problem definition

The provision of highly dependable guarantees on the timing behaviour is an ineludible concern in the qualification process of safety and mission critical embedded systems. The ongoing migration from relatively simple hardware platforms to cache-equipped processors in high integrity real-time systems is likely to disrupt the consolidated industrial approach to timing analysis. This thesis responds to the specific industrial need for the definition of a set of countermeasure and techniques to enable timing analysis of cache-equipped processors in industrial-level software development process. In the following, we introduce and elaborate on the problem we address in our investigation and provide an overview of our objectives and contributions.

1.1.1 Characterisation of High Integrity Real-time Systems

High integrity real-time systems (HIRTS) are gaining importance in several aspects of our lives: from relatively simple anti-lock braking system (ABS) devices embedded in modern cars, to extremely complex flight-control applications, to unmanned spacecraft manoeuvring systems, to accurate medical monitoring equipments.

The inherent criticality of such systems, whether exposed to physical, financial or environmental hazards, asks for strong guarantees on both dependability and timeliness of the services they are expected to provide. Besides functional correctness, HIRTS must be able to react to specific events within predefined deadlines: the time at which a functionality provided by the system is actually delivered can make the difference between success and

failure.

The strong emphasis placed on the multifaceted dimensions of dependability [13] and correctness poses additional challenges in the development of HIRTS. The provision of highly dependable guarantees on the functional and non-functional behaviour of those systems comes at the cost of stringent qualification requirements, which in turn ask for ever more intensive verification and validation (V&V) activities.

From the software development perspective, a score of certification standards have been defined either at industrial or international level to prescribe formal design, development, verification and validation processes and methods for predictable and dependable software systems. Specific certification standards accommodate the peculiarities of each application domain, such as aerospace, automotive, healthcare, public transportation or nuclear power plants. Example of domain-specific standards are the DO-178B [138] for Avionics, the EN 50128 [32] for European Railways, the IEC 880 [64] for Nuclear Power Plants, and the ECSS-E-ST-40C [44] for European space missions.

Although each standard naturally captures the peculiarities and specific requirements of each application domain, they share as a common prescription the procurement of dependable guarantees on the timing behaviour of the system. HIRTS must be *predictable*, in the sense that all system activities must be known to complete their execution within a least upper bound (and sometimes a greatest lower bound). Schedulability analysis techniques are explicitly mentioned as a means to prove the correctness of a system in the timing dimension: all jobs of each task in the system must be warranted to meet their deadline under all operational conditions.

The safety of schedulability analysis techniques in turn relies on the knowledge of the so-called worst-case execution time (WCET) of the tasks that are to perform the system activities at run time. Therefore an estimate or upper bound of the WCET of each task should be provided either by static analysis or measurements. The WCET values provided in input to schedulability analysis should be necessarily *safe* (i.e., greater than the actual WCET) and as *tight* as possible, to avoid over-dimensioning of systems, which incurs costly underutilisation.

From the hardware perspective, HIRTS often operates in an unconventional environment that requires the adoption of specialised hardware technologies. Hardware components in the aerospace domain, for example, are required to operate in particularly challenging conditions where limited power consumption and resilience to radiations are mandatory requirements. Both development and qualification of such specialised hardware are inherently more complex and costly when compared with their mainstream counter-

parts. It is not rare for hardware components in use in HIRTS to be significantly behind the current technological trends. However, as a positive effect of this technological gap, relatively simple hardware platforms allow timing analysis to stay reasonably simple. It is in fact widely acknowledged that the quality of timing analysis results strongly depend on the complexity of the underlying hardware [19]: the simpler the hardware, the easier the determination of reliable WCET bounds.

Under those conditions, part of the HIRTS industry naturally exhibits a conservative approach against any relevant change in both hardware and software design choices. Any such change in fact is likely to break the consolidated balance between compliance to strict qualification standards and pressing economical concerns. The adoption of new specialised technologies often asks for large investments with comparatively modest or long-term return on investment. On top of that, the introduction of new technologies or methods inevitably impacts on already onerous V&V activities, which are know to account for (if not exceed) the 60% of the total costs associated with a HIRTS development project.

1.1.2 Towards increasingly complex systems

Even the most conservative part of HIRTS industry is moving towards the adoption of more complex hardware platforms. The main *driver of change* behind this trend is the need for HIRTS to meet the user demands for increasingly complex functionality. The need for ever more computational power to manage and support those functionality drives HIRTS industries toward the adoption of more complex processors equipped with a number of acceleration features which have been gingerly avoided so far.

This trend towards more advanced processors representatively applies to the space domain, perhaps one of the most conservative HIRTS domain. Figure 1.1 shows the historical evolution of processor designs for use in on-board satellite systems. In the last decades, the low-paced but relentless transition from extremely simple 8-bit and 16-bit processors, adopted in the early space missions, to improved 32-bit models is now accelerating towards the introduction of increasingly complex features. The recent design of a fault-tolerant version of the LEON4 processor hints at the possible introduction of cache hierarchies and branch predictors even in high-integrity critical embedded systems.

It is worth noting that there is still a gap between the time at which a processor is designed and its actual adoption in a concrete space mission. The 32-bit radiation-hardened ERC 32 SPARC V7 processor [11], with a simple 4-stages pipeline and with no caches, has long been acknowledged as the reference processor model for European Space Agency (ESA) missions. The ERC 32 is still the processor of choice for specific missions and

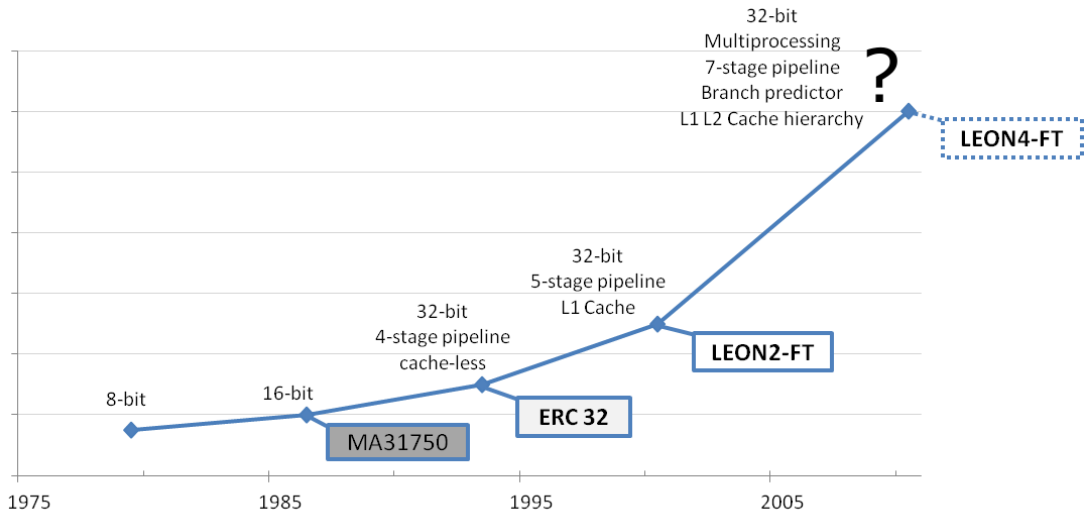


Figure 1.1: Evolution of on-board processors for European Space missions.

functionality that require small computational resources: it will be adopted for example for the data handling sub-system in the GAIA [45] astrometry mission, which will be launched in 2013.

The LEON2 [3, 12] is currently taking over the low-performance ERC 32 processor model in spacecraft on-board computer systems. The LEON2 32-bit processor model, based on the SPARC V8 architecture, provides the computational resources to meet the increased functional and performance requirements in modern space applications. It is stated to provide a 86 million instructions per second (MIPS) computing capability, thus superseding in performance its immediate predecessor (limited to 20 MIPS). The LEON2 processor will be adopted in the new Iridium NEXT [65] communication satellites mission, which involves a constellation of 66 satellites, with the launching date set to 2015.

The most perceptible innovation brought about by the LEON2 processor, over the ERC 32, is the introduction of cache memories for both data and instructions. The industrial stakeholders consequently became afraid that the transition to a cache-equipped processor is likely to affect their long-term consolidated V&V process, in particular with respect to the qualification of the system timing behaviour.

This is because although caches typically improve performance in the average case, they may introduce considerable variability in the execution time of a program. Each instruction or data fetch may incur a substantially variable timing behaviour depending on

whether an instruction or data is found in the cache or not. At every point in the program, the execution time of the next memory access depends on the cache contents, which in turn is determined by both the address of the memory reference and what addresses the program recently referenced in its execution path.

For that reason, HIRTS industry fears that the introduction of caches may complicate the effort required in analysing the timing behaviour of a system to the extent of breaking the already unsteady balance between the costs incurred by timing analysis and the quality of its outcomes.

1.1.3 Caches: friend or foe?

Industry that are confident with the efficiency and effectiveness of their verification and validation processes for old-generation hardware processors, do not have sufficiently solid insight on the effects that may be incurred by the migration to cache-equipped processors. Caches are perceived as an additional source of complexity, which has potential for shattering the guarantees of cost- and schedule-constrained qualification of their systems.

The same reasoning would also apply to other accelerating hardware features such as complex out-of-order pipelines, dynamic branch predictors, Memory Management Units (MMU), Translation Look-aside Buffers (TLB), etc. For these reasons, in fact, complex hardware features have been historically dispensed with in HIRTS.

We should first understand whether this state of uncertainty and disorientation perceived by HIRTS industry is somehow justified or not. We contend the answer to this question to be strongly affirmative, for at least three distinct reasons, on which we elaborate in the following.

Difficulties in accounting for cache-related timing variability

The first motivation comes from the observation that, compared to cache-less processors, the introduction of caches induces a variable timing behaviour that do complicates both schedulability and timing analysis [100]. The considerable variability in the execution time of a program caused by the use of caches may hamper, perhaps to a different extent, both WCET static analysis and measurements, which are expected to provide essential inputs to schedulability analysis. In the presence of caches, the execution time of the next instruction depends on the current contents of the cache (i.e., its *state*) as determined by a set of interacting factors: program size, memory layout, execution paths, and pattern of interrupts or preemptions.

As an immediate effect of this variability, the predictive value of execution-time measurements obtained by test is extremely degraded, as tests themselves are unlikely to incur the worst combination of all the above factors together (which, in addition, may even change from test to operation). Cache effects may also reduce the precision of static WCET analysis because analysis can only approximate the state of the caches at various points in the program.

Those complications indirectly affect the quality of schedulability analysis. In addition, the introduction of caches also has a more direct repercussions on schedulability analysis: in contrast with cache-less processors, the actual overhead of an interrupt or context switch is no longer constant but it also affects the WCET of the preempted or interrupted task in that it will find its cache state changed when it resumes it execution [75, 6].

The recent interest of European Space Agency (ESA) on the problematic introduction of caches in on-board processor for space applications confirms the industrial perception: several ESA-funded projects have been conducted in the last years to understand and possibly cope with cache unpredictability. During our PhD activities, we were involved in two collaborative projects: the (continuation of the) Prototype Execution-time Analysis for the LEON processor (PEAL2) project [22] and the Cache Optimisations for LEON Analyses (COLA) project [102].

Resistance of industrial practice to state-of-the-art timing analysis

Our second motivation does not directly stem from the cache predictability problem but rather addresses the conservative attitude towards timing analysis of part of the HIRTS industries. However, although not specifically related, the introduction of caches possibly exacerbates this attitude and unveils a misalignment between the state-of-the-art and the industrial state of practice.

Timing analysis of HIRTS software is typically constrained by a strict development process that is at the same time (globally) incremental and (locally) iterative. In most cases the WCET bounds are determined on the basis of past experience. A safety margin is then added to these WCET figures before they are given in input to schedulability analysis. Along the development process, WCET bounds are then periodically consolidated by testing, where a program (or part thereof) is executed and dynamically analysed (measured) a number of times with a variety of inputs that represent selected configurations and/or operation modes. Safeness of timing and schedulability analysis thus relies on adequate test coverage and safety margins. Provided that obtaining 100% coverage for complex systems is almost impossible in practice, the only option for safeness is to add

over-dimensioned and unscientific safety margins.

In spite of the clear drawbacks of this approach, its application is still reasonable for systems running on top of relatively simple hardware that do not exhibit a highly variable timing behaviour. On the contrary, in the presence of hardware acceleration features like caches, the predictive value of scenario-based measurements is drastically reduced. Typical test cases are not likely to trigger the worst-case execution scenario [159] unless explicit countermeasures to timing variability are adopted.

Provided that no simple and exhaustive solution has been devised yet to cope with the inherent unpredictability of caches [100], several approaches have been proposed in the literature to derive safe and tight bounds of the WCET of tasks running on cache-equipped processors [163]. Some of these techniques have been successfully integrated into commercial tools, such as RapiTime [133] and aiT [1], and have been successfully applied in industrial case-studies [137, 30, 146, 54, 84].

This notwithstanding, software simulation and testing continue to be the common practice for obtaining WCET values. Particularly in those application domains whose standard directives do not explicitly prescribe a specific timing analysis qualification criteria, the penetration of WCET tools and techniques in the industrial practice is limited by the perceived complexity and cost of their factual application.

Doubts on the industrial fitness of timing analysis approaches

The third and final motivation generates from practical considerations we collected during a long-term experimental activity on the timing analysis of an industrial-scale HIRTS. Thanks to our cooperation with TAS/F we were able to experiment on a significant component of the software application embedded onboard a commercial satellite system. The lessons we learnt from our experiments unfortunately seems to confirm the industrial concerns on the factual application of state-of-the-art timing analysis approaches in industrial setting [103].

Despite their undeniable theoretical soundness and the evident progress made in the last decades, state-of-the-art timing analysis approaches do have limitations that become particularly evident when applied to industrial level complex systems. The difficulties with applying state-of-the-art WCET analysis in HIRTS can be ascribed to two main facts: (i) the application of timing analysis is not fully automated (and probably will never be), hence it requires onerous and error prone human intervention in the form of flow-fact annotations to guide the analysis process; and (ii) industrial systems have been historically developed with performance in mind and do not conform to the predictability assumptions made by

timing analysis.

Even if HIRTS industry were to move to state-of-the-art WCET analysis approaches, which is the most natural solution to cope with the cache-induced variability, it would still have to cope with the complexity and costs incurred by its application to large-scale complex systems. Moreover, this methodological shift to more rigorous timing analysis approaches may also ask for an onerous and long term adaptation of the consolidated industrial development process. These factors altogether do not seem to turn in favour of an economically viable V&V process.

1.2 Thesis objectives and contribution

This thesis builds on the above observations to contend that HIRTS development cannot rely on an uninformed use of caches, for their uncontrolled effects on the timing behaviour may invalidate or, at least, degrade the results of timing analysis and, in turn, dissipate the trustworthiness of schedulability analysis.

In particular, we believe that the disruptive effect of caches in the qualification of industrial-level HIRTS can be effectively and efficiently governed only by imposing an informed use of caches and a proactive approach towards timing analysis. As we have shown in [103], software predictability problem cannot be ignored during the whole HIRTS development process and abruptly reappear while V&V activities are conducted: an extremely variable and unpredictable system will stay unpredictable, whatever technique or tool one could apply.

In our view, the development process should become *"cache-aware"* (and timing-analysis oriented) in the sense that all development activities, from high-level design to software development, should be performed with timing analysability in mind. In actual fact, this would result in the systematic adoption of a set of practices, methods and tools which should guide the whole development process. The development process should be aware of the cache impact and try to minimise any potential source of unpredictability so that to ease the overall system analysability.

Our thesis therefore aims at identifying a set of countermeasures to improve cache predictability and, in a broader sense, to facilitate the application of timing analysis in industrial setting. As fundamental requirement, the identified techniques and methods shall allow an efficient integration and application in the HIRTS industrial development process.

The strategy we devised for the accomplishment of our objectives follows three major

steps:

1. As a preliminary step, we will focus on the identification of the sources of timing unpredictability that manifest themselves at source-code level and, possibly, the proposal of analysable alternatives. We aim at identifying code constructs, patterns and design choices that hinder the application of timing analysis in general. In our reasoning, we do not limit ourselves to cache analysability issues alone as we consider the cache predictability problem as a (relevant) part of the broader problem of timing predictability.

2. Our study will then proceed with an investigation on the Model-Driven Engineering (MDE) [141] paradigm as a means to enforce predictability by construction. We aim at exploiting the results from the previous activity to both enforce the generation of analysable code patterns and to automatically convey timing information from the model to the synthesised code.

3. In a third complementary approach, we will attack the memory layout as the most impacting source of cache variability and devise a layout optimisation technique that addresses *incrementality* as a main industrial requirement.

The first two investigations will spread over two main conceptual spaces of intervention: a *task-level* dimension, where the main focus is set on the timing behaviour of each single task that contributes to the system functionality; and a *system-level* dimension, where we accommodate the effects of task interleaving and interactions, as well as the overall structure of the system and its software architecture specifications. This second dimension extends the scope of the notion of system-level timing analysis (classically limited to inter-task interference on the timing behaviour [75, 6]) to include high-level and low-level design choices.

The partial results obtained along those three directions are finally combined in a structured approach to timing analysis that exhibits, as characterising features:

- an improved degree of analysability *by construction*, which facilitates and enables the application of state-of-the-art timing analysis technique to complex industrial systems; and

- the early applicability of WCET analysis on subsequent incremental releases, as opposed to the complete system.

Contribution

In defining a comprehensive approach towards timing analysis in the industrial development, this thesis provides the following key contributions:

1. The identification and categorisation of **code constructs and design choices** that may hamper the analysability of a program, and the proposal of analysable alternatives for either code constructs or design choices.

2. As part of the previous contribution, we formalise and bound a new **source of inter-task interference** on cache behaviour, which stems from mutually exclusive access to shared resource.

3. An approach to the **automated generation of predictable systems**. Exploiting existing modelling tools [143, 51, 116] we enforce the generation of predictable task-level code constructs as well as predictable design choices at the level of software architectures. We further exploit functional and architectural models to collect any valuable information which can be expressed at model level and automatically generate flow-fact annotations for an effective application of timing analysis.

4. An **incremental cache-aware memory layout optimisation** technique which aims at the reduction of the cache-induced variability. In particular, we emphasise the incremental applicability of our technique, which meets a key requirement in the industrial development process.

1.3 Thesis organisation

This dissertation is structured as follows. In Chapter 1 we introduced the motivation of our PhD study and sketched the proposed approach and key contributions. We first provided a characterisation of the HIRTS domain and briefly introduced the cache predictability problem, as it is perceived by the industrial stakeholders. We then introduced the objectives and main contributions of our dissertation.

Chapter 2 introduces the general notions on cache operation, provides a critical review of the state of the art in cache and timing analysis, and states our research assumptions.

Chapter 3 is devoted to the presentation of our core contributions. In Sections 3.2 and 3.3 we provide a characterisation of the HIRTS development process and discuss the

industrial perception of timing analysis, particularly focusing on those issues that hinder the penetration of state-of-the-art WCET analysis in HIRTS industries. Section 3.4 elaborates our objectives and presents an industrial development approach that builds on model-driven engineering and memory layout optimisation to enable a cost-effective application of state-of-the-art WCET analysis to complex systems. In Section 3.5 we discuss code analysability issues in timing analysis of industrial level systems, and provides a taxonomy of unpredictable code constructs at task and system level. Section 3.6 details on the role of automated code generation in enforcing timing analysability by construction at task and system level. Section 3.7 introduces our incremental cache-aware memory layout optimisation technique as a means to cope with the variability incurred by caches and facilitate an early application of WCET analysis. Section 3.8 provides a qualitative evaluation of our approach.

Chapter 4, finally, concludes by recalling this thesis objectives and contributions, and by providing a brief outlook on our future activities.

Chapter 2

Background and Assumptions

2.1 Schedulability and timing analysis

In our definition of HIRTS we highlighted the fundamental role of the timing dimension in determining the correctness of such systems. Reasoning about the temporal requirements of a system is therefore an inevitable concern in HIRTS development process. Schedulability analysis techniques consist in performing a schedulability test to provide dependable guarantees that all system tasks can be actually scheduled on a processor without missing their deadlines [86]. Each schedulability test applies a specific scheduling policy.

Response Time Analysis (RTA) [68] is a classical example of schedulability test for fixed priority preemptive systems (FPPS) where the worst-case response time (WCRT) of each system task is computed and compared with the respective deadline. Assuming the classical *periodic task model* [86] where each task τ_i is characterised by an activation period T_i, a maximum computation time C_i and a deadline D_i, the worst-case response time for a task τ_i is computed by solving the following fixed-point equation that iterates over a time window w_i:

$$w_i^{n+1} = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j \tag{2.1}$$

where w_i^k refers to the time window under analysis, C_i and B_i are respectively the maximum computation time and blocking time for the analysed task, and the remaining term represents the *interference* from higher priority tasks. If the worst-case response time is not greater than the deadline for each task in a task set then the latter is said to be feasible under the specific scheduling policy.

The role of timing analysis in the context of response time analysis is clearly fundamental. The C_i term in Equation 2.1 is exactly the WCET bound for task τ_i, which is the expected outcome of timing analysis. The worst-case response time models the maximum time spent for the execution of any single activation of a task and includes interference from other task in the task set. The WCET, instead, represents a least upper bound on the computation time of each task in *isolation*.

The expectations on safeness and tightness of WCET bounds are also evident: unsafe bounds simply invalidate the above equation, whereas loose bounds may reduce the processor utilisation and thus cause undue over-dimensioning of the system.

2.2 Caches

One solution for bridging the widening performance gap between CPU and memories, is to exploit small but fast cache memories, often located on chip, to store the most recently accessed data and instructions [60]. The hierarchical relation between caches and the main memory implies that whenever the CPU issues a memory access, the memory location is first searched in the cache: if the required data are found in there (*hit*), they are immediately sent to the CPU providing a relative lower latency than that incurred by accessing the slower main memory. Otherwise (*miss*) the request is forwarded to the main memory: the retrieved data is first copied into the cache and then delivered to the CPU. Figure 2.1 below provides an high-level representation of the cache operation.

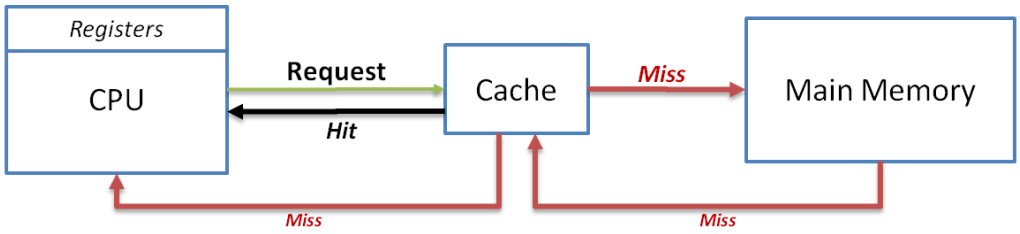


Figure 2.1: High level cache operation.

Most modern processors define separate caches for data (D-cache) and instructions (I-cache) as opposed to *unified caches*, which instead are typically defined in more complex architectures to provide additional cache levels.

Caches build their usefulness on the observation that memory references in a program typically follow a principle of *locality* [145, 60]. This concept is further decomposed into:

- *temporal locality*: memory addresses being accessed recently are likely to be accessed again in the near future; and,
- *spatial locality*: memory addresses near to a referenced address are likely to be accessed in the near future.

The former principle is clearly exploited by caches operation as they store the most recently accessed instructions and data. Caches also exploit spatial locality in a program as, in order to make use of the available bus bandwidth, they typically fetch from the main memory (and store) more data than required. Memory data in fact are logically organised as fixed-size memory blocks (typically ranging from 8 to 64 bytes); thus, in case of cache miss, a memory block containing the required memory address is retrieved from main memory and placed into an equally sized cache frame (*cache line*).

Cache placement and look-up

Where a memory block is actually placed, and then searched for, in the cache depends on the *placement policy* (or associativity) in use. With this respect, three main design alternatives are defined:

- *direct-mapped* caches: where a memory block can be placed into exactly one cache line. In this case the cache mapping is extremely simple to implement as it is determined by *(block address)* `mod` *(number of cache lines)*.
- *set-associative* caches: where a memory block can be placed into a subset of the total cache lines, referred to as *cache set*. The number of cache lines in a set is referred to as *way*; hence, in a 4-way set-associative cache, the same memory block can be placed into any of 4 different cache lines. Therefore, in set-associative caches the mapping function is *(block address)* `mod` *(number of cache sets)*.
- *fully-associative* caches: where a memory block can be placed in any cache line. The implementation cost of this placement policy makes it suitable only for extremely small caches.

It is worth noting that those three alternatives can also be considered as the instantiation of different degrees of associativity, ranging from one single way (direct mapped) to the extreme of the total number of cache lines (fully associative).

Figure 2.2 shows the cache look-up process for a 4-way set-associative cache. The cache look-up is typically performed by decomposing the referenced address A in three parts [60]: $[\mathrm{tag}_A][\mathrm{set}_A][\mathrm{offset}_A]$. The look-up process consists in comparing the tag_A part with the tags of the cache lines in the cache set set_A. The offset_A part finally selects the required data within the block. Fully associative caches have no index field.

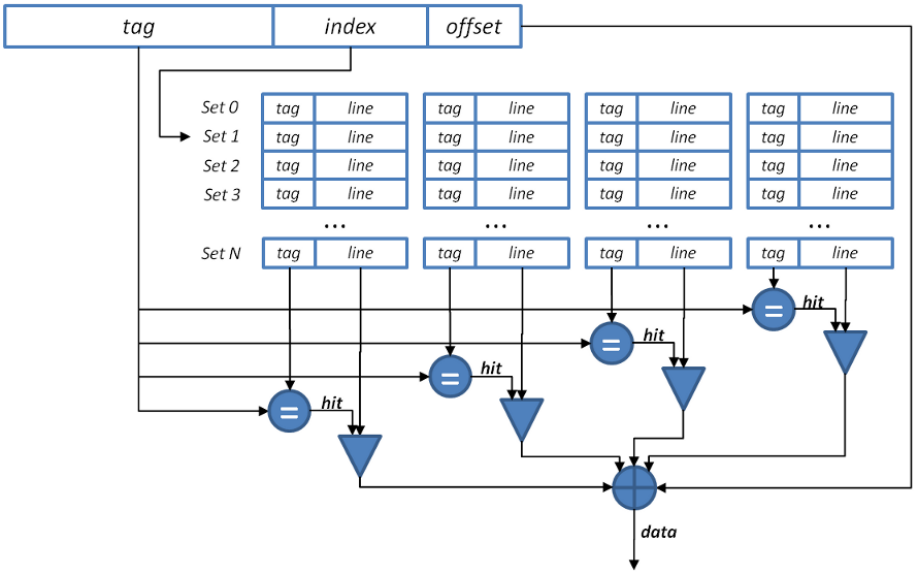


Figure 2.2: Cache look-up process.

Cache replacement

Typically, on a cache miss, a cache line is evicted from the cache to make room for the block retrieved from main memory. The selection of the cache line to be evicted is straightforward in direct mapped caches as there is a unique location for each memory block. For set-associative caches, instead, the evicted cache line is selected according to a specific *replacement policy*. Common cache replacement policies are Least Recently Used (LRU), FIFO, Random, or some *pseudo* (i.e., approximate) variant of them.

LRU exploits the notion of ageing to induce an ordering between cache lines in a set. The line selected for replacement at any specific time is the line with maximal age. LRU is typically implemented in hardware for caches with associativity up to four. The relaxed form, termed Pseudo LRU, is used instead for larger associativity as the (hardware) implementation cost of maintaining precise ageing information becomes too high.

Cache writes and allocation

Another relevant design choice addresses specifically the behaviour of data cache writes (*write* and *allocation policies*). Upon a write request, the referenced data could be in the cache or not. In case they were, changes applied to data can be written to the main memory at once (*write-through*) or else deferred until when the same cache line is evicted from the cache (*write-back*). Moreover, in case of write miss, data can be loaded in a cache line (*allocate-on-write*) or can be directly modified in the low memory (*no-allocate-on-write*). Any combination of those policies is allowed, except for "write-back" and "no-allocate-on-write" since in case of cache miss the new data cannot be held in the cache and later written back to main memory without allocating a cache line.

Cache miss categorisation

Three types of cache misses are distinguished [60]: (i) *compulsory misses* (or cold misses), when an address is first referenced; (ii) *capacity misses*, when the current working set exceeds the cache capacity; and (iii) *conflict misses*, when, depending on the actual replacement policy, multiple references map to (and compete for) the same cache line.

2.3 Approaches to cache analysis

The cache predictability problem is well acknowledged by the academic community and several approaches have been proposed in the literature on both instruction and data caches predictability [163]. In the following we survey the state-of-the-art approaches to cope with cache unpredictability. These approaches build on either accounting for cache behaviour in both timing and schedulability analysis, or making caches more predictable, to the extreme of replacing them altogether with more predictable fast memories (i.e., scratch-pads). Besides, even software and hardware issues will not be disregarded since they play a main role in determining the cache behaviour and thus its analysability.

2.3.1 Cache-aware Timing Analysis

The timing behaviour of a task depends on the task code, the execution context and the hardware on which it executes. The execution context is determined by both the task inputs and the hardware state, which includes the state of caches. The difficulties introduced by

caches in timing analysis of a program are classified in to two main kinds of interferences: intrinsic and extrinsic.

Intrinsic (intra-task) interference is related to the possibility of incurring non-compulsory cache misses as a consequence of the fact that multiple program blocks may compete for a cache set and collide with one another. *Extrinsic* (inter-task) interference occurs in multitasking environments and is related to possible non-compulsory cache misses due to suspension or preemption, as when a suspended or preempted task starts again to execute it may find its cache state changed. Both intrinsic and extrinsic interferences depend on the number of references that can potentially compete for the same cache set and on the actual number of collisions, which is dynamically determined by the execution path. Extrinsic interference also depends on possible interactions between tasks. Both kinds of interference should be taken into account for a comprehensive analysis of the timing behaviour of a program.

WCET analysis techniques, either static or hybrid measurement-based, account just for intrinsic task interference as they typically focus on single tasks in isolation and assume no interference from other tasks or any other scheduling issue (e.g.: task communication, synchronisation, etc.). The extrinsic cache interference is then handled by advanced schedulability analysis techniques which are extended to account for the cache impact in the response time of individual tasks.

2.3.1.1 Static WCET analysis

Dynamic analysis techniques can only provide a *maximum (minimum) observed execution time* which is likely to harmfully under-estimate the actual WCET. Provided that computing the exact WCET of a given application in presence of caches is not generally feasible in practice, WCET static analysis methods aim at obtaining a safe and tight upper bound on the WCET. Several approaches in the literature extend the static timing analysis techniques to account for the cache behaviour.

Static WCET analysis techniques try to compute a WCET bound for a given program from an abstract model of a processor and an executable. Most of these approaches, however, greatly benefit from the availability of the application source code.

All these approaches share three fundamental steps for the WCET analysis of a program:

1. *High-level analysis* step, which aims at determining the control-flow graph of a program. The nodes of such graph, termed *basic blocks*, are small linear sequences of code characterised by single entry and exit point and that contain no branches;

2. *Low-level analysis* step, where hardware-level timing is taken into account and the timing behaviour of each instruction is modeled. Different techniques can be exploited to perform this step; and

3. *WCET calculation* step, which allows to compute an upper bound to the WCET, typically exploiting an Integer Linear Programming (ILP) formulation of the problem optimised through the Implicit Path Enumeration Technique (IPET).

Whereas the timing behaviour of caches is explicitly addressed in step (2), the results from step (1) may positively or negatively influence the results of cache analysis. For example, the accuracy of loop bounds and a precise detection of indirect calls would improve the precision of the analysis results. In the following, we will focus on state-of-the-art cache analysis techniques without addressing more general WCET analysis issues, unless of major relevance with respect to caches.

Cache analysis by Abstract Interpretation Ferdinand and Wilhelm [48] propose an approach which builds on *abstract interpretation* [37] to obtain tight bounds of instruction and data cache behaviour. In order to classify memory accesses, they define an abstract domain to represent an over-approximation of all possible cache states, called abstract cache state (ACS). Building on the results of *path* and *address* analysis, ACS accumulate the information on cache accesses using two operators defined in the abstract domain:

- *Update*: the ACS is updated according to the memory accesses performed in each basic block along the program execution path; or

- *Join*: since ACS can be split on a branch instruction, different ACS are conservatively combined together when distinct control flow paths merge together.

This approach allows to collect *Must* and *May* information, which are respectively upper and lower approximations of the concrete cache state at every program point. *Must* analysis is used to determine safe information on the cache content (cache hits) whereas the complement of the *May* information is used to determine cache misses. Based on the analysis results, each memory access can be classified as Hit, Miss or unknown. In [154], *Persistence* analysis has been introduced to avoid the pessimism entailed by the original approach in determining the cache behaviour within loops. A block is *persistent* when its first execution may result in a cache hit or miss, but all consecutive references are known to be in the cache. The results of *Persistence* analysis has been recently refined in [15]. Defined in [48] for LRU fully-associative caches, Ferdinand's technique has been extended to different replacement policies and set-associative caches in [59].

Cache analysis by Static Cache Simulation Mueller et al. in [106] introduce *static cache simulation* for direct-mapped instruction caches. This approach, based on data-flow information, classifies each instruction fetch as *Always-Hit*, *Always-Miss*, *First-Miss* or *Conflict*, which intuitively corresponds to the classification obtained by the Ferdinand's approach extended with *Persistence* information. The same approach has been extended in [162] to analyse data caches and apply to set-associative caches.

Cache analysis by ILP formulation Li et al in [82] manage the problem of finding the WCET by modeling an ILP problem with feasible paths and architectural features as constraints. Cache, pipeline and path analysis are combined together in a single ILP formulation. An obvious drawback of this approach is that gathering all the relevant timing information may easily get intractable when the system becomes complex.

Data Cache analysis by Cache Miss Equations Gosh et al. in [52] present a framework for providing global bounds on the data cache behaviour of loop nests. This approach builds on a complex analytical method using the so-called Cache Miss Equations (CME), a sort of Diophantine equations where each solution corresponds to a possible cache miss, to gather as much information as possible on data cache access patterns, which are known to be more irregular and difficult to predict than those of instruction cache. The framework, which applies to set-associative LRU caches, is more oriented to find proper code optimisations to improve cache performance rather than obtaining precise WCET figures. Ramaprasad and Mueller in [131] refine the CME-based approach by focusing more on WCET analysis of data caches. This more precise approach still applies to set-associative LRU caches.

Hierarchy of caches The increase of hardware complexity in real-time systems has lead to processors equipped with hierarchical multi-level caches, which further complicate the cache-aware WCET analysis. Mueller in [108] extends the static cache simulation approach [106] to hierarchies of instruction caches. This method has been proved to be unsafe by Hardy and Puaut in [57] where the multi-level cache classification is complemented by a *cache access classification* (CAC) which can be used to determine if a cache access would entail an access to the next hierarchy level or not. The same techniques is extended in [79] to multi-level data caches.

Discussion The main strength point of static analysis approaches is that they are supposed to always provide safe WCET bounds. Actually, the safeness of those techniques is threatened by at least two issue. First, all static analysis techniques build on an abstract model of the target hardware: the more complex the actual hardware, the less accurate and cumbersome may be to model it. Safe WCET bounds can be guaranteed if and only if the abstract model they build upon is actually correct. Providing a precise and sound timing model may prove to be a tough challenge when the target hardware features exceeding complexity or when we lack complete knowledge of it (i.e. not everything is disclosed).

Another safeness issue for static analysis techniques is related to *timing anomalies* [92, 135]: 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. In terms of WCET computation, for example, a cache miss may result in a shorter global execution time than a cache hit (scheduling anomaly) or a cache access may result in a cache miss just because a branch misprediction has unnecessarily evicted the referenced instruction (branching anomaly). Further counter-intuitive cache behaviour may be incurred by unpredictable replacement policies, like PLRU (cache anomalies). Timing anomalies can occur whenever there is a source of non-determinism in the analysed model. Although timing anomalies mainly occur in out-of-order processors, speculation and cache anomalies can also happen on in-order processors [135].

However, although in most circumstances static analysis techniques can produce a safe bound on the WCET of a given program, the computed WCET estimate is not only expected be safe, but should also be tight, with as little overestimation as possible in comparison to the actual WCET incurred at run time. Cache-aware static analysis in general may suffer from, at least, two main sources of overestimation:

- *the join operation between abstract states*: several approaches gather information on cache accesses by computing abstract cache states explicitly (e.g. [48]) or implicitly (e.g. [106]). Whenever a control-flow join occurs, a new abstract cache state is conservatively computed, which introduces overestimation;

- *the infeasible path problem*: flow analysis may derive from the program control-flow graph (CFG) some execution paths that may not be feasible in any program execution.

A possible way to overcome the precision loss problem is to adhere to appropriate coding styles; reducing the sections of code reachable trough different execution paths will reduce abstract state joins which are the principal reason of precision loss. For example

the idea behind single-path coding and WCET oriented programming [128], although they intuitively incur serious performance drawbacks, can be interpreted in that light. Appropriate coding style may also reduce the complexity of the infeasible path problem since reducing the number of paths in the program also reduces the search space for infeasible paths. This strategy could reduce the complexity of determining infeasible paths in a program and hence result in tighter WCET.

In order to facilitate and improve the precision of WCET analysis flow-facts annotations are required to explicitly exclude some infeasible paths, define branch and indirect call targets, refine loop bounds and refine path-analysis in general. A classification of flow-facts annotations can be found in [74]. However, the annotation process requires a deep knowledge and understanding of the program behaviour. Since manual intervention is both laborious and error-prone, it is preferable to gather as much knowledge as possible by automated methods (e.g.: [62, 16, 29]).

2.3.1.2 Hybrid Measurement-based Approaches

Despite the progress with static analysis in the last two decades, interest in measurement-based approaches to estimate the WCET has grown, mainly due to the increasing complexity of modeling modern processors. Hybrid measurement-based analysis, conversely to classical dynamic analysis, aims to avoid optimism in the WCET estimation by measuring the WCETs of small program fragments (typically basic blocks or groups of them) and then combining them using static analysis techniques [20, 125, 85] to compute a WCET estimate. Therefore, loop bounds can be added a posteriori (through annotations) to the measurement stage and the WCETs of program fragments can be triggered in different test runs. In particular the approach by Bernat et al. [20] defines a probabilistic approach where measurements are combined according to some probability distributions.

Classification of hybrid measurement-based approaches Measurement-based methods differ in several ways. First, measurements can be performed at different levels of granularity. Measurements are usually performed over the execution of program fragments characterised by having a single execution path. Such fragments can be single basic blocks, which contain no branches, or a small group of them, which may contain branches but whose execution is input-data independent.

Also the initial state of the processor (including the cache content) is a source of significant variability on the execution times of program fragments. Measurement methods can either try to set the processor into a worst-case state [118] before each measurement

or simply use the state contextually reached by the processor. The former approach would potentially ensure a safe WCET estimate for each measured execution path; however it may be quite difficult to find a worst-case initial state. Conversely, the latter approach does not provide safe WCET estimate by construction, since the initial states are just a sample of all possible states.

Finally, timing data need to be collected whilst the program executes. Two differing approaches can be adopted involving either code or hardware instrumentation. Code instrumentation builds on the insertion of instrumentation points in the program in order to accumulate timing data during its execution. However the instrumentation points themselves affect the temporal behaviour of the measured code introducing the so-called *probe effect*. The probe effect also impacts cache usage, as instructions or data normally placed in cache are disturbed by the instrumentation code. The overhead of the probe effect can be prevented with the assistance of on-chip debug interfaces, such as Nexus [112] or the ARM Embedded Trace Macrocell [10]. In these cases, the trace data are either written to an on-chip trace buffer for subsequent download, or exported directly in real-time through an external port. In order to limit the size of traces, only program flow discontinuities (i.e., conditional and unconditional jumps) should be monitored.

Discussion Measurement-based methods provide real measurements of the execution time of a given task or parts of it, on the hardware of interest (or on a representative simulator of it) for a given set of inputs and initial states. As an advantage over static analysis, measurement-based methods do not need to model the processor behaviour exactly and does not require additional effort to adapt to different target processors.

However, the major limit of this approach is in that measurements of a subset of all possible executions will produce estimates or distributions, which despite of possibly being accurate, cannot define bounds for the execution times. We cannot tell for sure whether the worst-case execution path has in fact been traversed or not.

Furthermore, similarly to static analysis techniques, the inclusion of infeasible paths in combining basic blocks measurements and possible overestimation of loop bounds may lead to loose WCET estimates. For example, combining timing information of different program fragments may include infeasible paths. As discussed in the previous section, some approaches aim at improving flow analysis techniques to cope with the infeasible path problem [62, 16, 29].

Since only a subset of the possible initial processor states (including cache states) are considered for each measured code segment, trustworthiness of measurements themselves

relies on a high degree of test coverage, which is likely to be too expensive under normal circumstances. Furthermore, coverage metrics that are typically adopted in industrial software testing are inherently focused on the functional properties of a program and are thus inadequate with respect to assessing the coverage of timing properties, like the WCET. An attempt to define some WCET-oriented coverage metrics for processors with pipelines is introduced in [21].

With respect to the probabilistic approach in [20], it is not fit for hard real-time system requirements as it gives probability distribution of the WCET of tasks rather than safe bounds. However, since real-time applications typically consist of multiple components within distinct criticality levels, it could be effectively applied to the less critical (i.e. soft real-time) parts of an application.

2.3.2 Cache-aware Scheduling Analysis

In general, static analysis and measurement-based approaches focus on intrinsic cache behaviour and do not account for extrinsic interference. Schedulability analysis techniques are therefore extended to account for safe and possibly tight estimates of the cache impact on extrinsic interference.

Cached Rate Monotonic Analysis Basumallick and Nilsen in [18] define the Cached Rate Monotonic Analysis (CRMA) which includes the so-called cache related preemption delay (CRPD) into the classical Liu-Layland's Rate Monotonic Analysis. The execution time of each preempting task incorporates the context switch delay and a CRPD term that accounts for the time required to restore the cache state of the preempted task. The CRPD is estimated as the cost to completely refill the cache. This approach is rather pessimistic since the estimate of the CRPD fails to take into account that some cache lines may not be evicted, either because they are not affected by the preempting task or because they are shared by the preempting task with the preempted task. Furthermore it is indeed possible that not every cache line will be referenced again by the preempted task.

Cached Response Time Schedulability Analysis Busquets-Mataix and Wellings [28] propose to incorporate the CRPD in the Response Time Schedulability Analysis (RTA) thus defining a Cached Response Time Analysis (CRTA). The worst-case response time for a task is affected by the execution of higher-priority tasks in two ways: it may suffer a delay because of the CRPD paid upon its resumption or even because of the CRPD paid by

higher-priority tasks. Thus CRTA accounts for direct and indirect extrinsic interference: indirect interference is suffered by task τ_i when a higher-priority task τ_j is released, as a consequence of the direct interferences suffered by intermediate priority tasks τ_k. The refill penalty is computed as the time required to restore the cache lines evicted by the preempting task. Therefore this approach is less pessimistic than the Basumallick and Nilsen's one but it is nevertheless prone to overestimation: as we noted earlier, not every cache line will necessarily be referenced again by the preempted task.

Useful Cache Blocks In order to reduce the pessimism incurred by the CRPD estimates, Lee et al. in [77] were the first to introduce the concept of useful cache blocks (UCB) to compute the CRPD. A UCB is defined as a cache block that will be referenced again before it could be evicted by another memory block, according to the cache replacement policy. The approach has been further refined in [75, 147] where the analysis has been modified to take into account that only useful blocks which intersect with cache blocks of the preempting tasks are to be refilled. All the above approaches apply to direct-mapped or LRU set-associative caches: Burguiére et al. in [26] show that the previous techniques based on UCB and evicting cache blocks (ECB) (also referred to as used cache blocks) are not safely applicable to FIFO and PLRU replacement policies. More recently, with respect to LRU set-associative caches, the idea of *resilience* has been introduced [8] to exclude from the CRPD computation those UCB that can be guaranteed to persist in the cache, thanks to the specific replacement policy.

Other approaches Other approaches try to cope with the inherent pessimism of CRPD estimates by limiting or minimizing the possible interferences. Nemer et al. in [111] present a task timing analysis for statically scheduled multi-tasking systems that accounts for the interleaving of non-preemptable tasks in computing the abstract cache states at each task activation after a suspension.

Zamorano and de la Puente in [166] aim at reducing the pessimistic estimate of the refill penalty by actually limiting the number of cache refills. The authors suggest discarding cache refills due to interrupt handlers which do not imply actual preemption of the active task; this is obtained by inhibiting caching for interrupt handlers. This should not overly degrade the overall performance as execution of interrupt handlers are supposed not to largely benefit from caches: interrupts typically consist in sequential code which exploits only cache boosts from spatial locality.

A different approach is proposed by Altmeyer and Gebhard in [7], where the authors

focus exclusively on computing the WCET for single tasks apart from schedulability analysis. After forcing a quasi-optimal memory layout that minimises cache conflicts between tasks, they apply static analysis to classify persistent and endangered cache lines and then compute WCET bounds for each task.

Discussion Cache-aware schedulability analysis aims at including cache effects in the schedulability analysis; this inclusion is generally accomplished by accounting for CRPD overheads in the response time of individual tasks. The refill penalty associated with CRPD may be estimated in several ways: it may be measured on the entire cache size or on different estimates of the actual lines to be refilled (i.e., useful blocks, line intersection, etc.). To any rate, the computed WCET estimate is a safe upper bound to the actual WCET but it is not a tight one.

2.4 Restraining the cache behaviour

Alternate approaches to increasing the level of cache predictability aim at either bounding or tailoring the cache behaviour so as to make caches more suited for WCET and schedulability analysis. The increase in predictability would not only reduce the overestimation of static analysis, but may also improve the safety of measurement-based WCET methods.

2.4.1 Cache Partitioning

Cache partitioning techniques aim at increasing cache predictability by partitioning the cache in a way that a cache segment can be reserved for a specific task or group thereof. Cache partitioning techniques can be implemented in hardware or in software.

Hardware Cache Partitioning Kirk in [70] suggests a hardware-implemented partitioning scheme named SMART (Strategic Memory Allocation for Real-Time). In SMART, the cache is divided into several segments private to individual tasks; and a shared partition. Each task is assigned one or more private partitions. Assigning private segments to tasks reduces extrinsic cache interferences; since we also have a shared segment, however, there would be a (possibly small) CRPD at context switches. The implementation of SMART requires a hardware flag to tell private versus shared cache partitions and hence a custom cache controller. The author also supplies an algorithm to choose the size and the assignment of partitions [71].

Muller et al. in [109] introduce a similar cache partitioning technique, hardware implemented as well, but more D-cache oriented. Unlike Kirk's approach, the cache is split into several partitions, which operate like small direct-mapping caches. No shared segment is required. Load and Store instructions include a partition operand which selects the appropriate partition; the compiler must be aware of the cache architecture and should provide an appropriate allocation of the cache partitions. This approach aims at also reducing the intrinsic cache interferences by mapping data structures into cache partitions (and into lines within partitions) according to a compiler analysis of data accesses.

Software Cache Partitioning Since hardware partitioning is not usually supported in commercial processors, other approaches aim at implementing cache partitioning techniques by software. Software partitioning builds on optimising memory mapping of code through specific compiler and linker support: instructions are placed in the address space so as to reduce or eliminate inter-task interference.

Wolfe in [164] suggests dividing the cache space into partitions, each of which is only used by certain tasks. A direct-mapping cache is partitioned by altering the address translation process at each cache access, also requiring some hardware tweaks. Bounding the memory references issued by tasks to a selected range of addresses effectively permits to define logical (as opposed to physical) partitions in the cache.

Mueller in [107] takes Wolfe's approach and focuses on the compiler and linker support required for cache partitioning. Mueller aims at defining how to assign tasks to addresses and thus assembling the code in a manner that permits to eliminate extrinsic interferences. Cache lines are grouped into partitions, each assigned to a specific (real-time) task; a single partition is reserved for a shared access. In order to map task memory accesses to a specific cache partition some code transformations are applied. Instructions and data are transformed to fit a certain range of memory addresses, producing a scattered memory mapping for each task. The scattered memory mapping is performed through the compilation and linking processes to ensure that every task will only access its own cache partition, except in case of synchronisation, when the shared partition is accessed.

The compiler must restrict the code of a task to only those memory addresses that map into the cache partition. Mueller also suggests breaking the tight one-to-one mapping between tasks and partitions: in order to avoid extremely small partitions it suffices to define one partition for each priority level, letting the tasks at one and the same priority level share the same partition (if FIFO scheduling is adopted for tasks at the same priority level). Assigning multiple tasks to a single partition permits to define bigger partitions and

consequently reduce intrinsic interferences. On the other hand, sharing partitions within priorities will introduce some extrinsic interference: since when a suspended task will execute again, it may find its cache state modified by some task sharing the same partition.

Discussion Cache partitioning is likely to reduce extrinsic cache interferences by assigning separate cache segments to given tasks. Residual intrinsic interference may still be incurred owing to access to shared partitions (if any). The most important drawback of both hardware and software partitioning approaches is the reduction of cache space per task: since each task is granted a smaller amount of addressable cache, the number of capacity (and conflict) misses will inevitably increase.

Furthermore, this kind of techniques leaves one central problem unattended: they do not advise on how we are supposed to define both size and assignment of partitions. This issue is critical as it will have a major impact over cache performance. In fact, the performance of cache partitioning is strictly related to number, size and actual assignment of cache partitions. In particular, assigning partitions to tasks can prove quite a complex job: naively assigning partitions in accordance with task priority (i.e., by urgency) or to task rates is unlikely to be the best choice.

Some studies noticed those open problems and tried to find an automated (i.e., algorithmic) way to solve them. Sasinowsky and Strosnider in [140] define a dynamic programming algorithm for allocating cache segments to a set of periodic tasks. Their algorithm, which runs in a polynomial time, is claimed to be optimal where optimality consists in finding the allocation which produces the minimum processor utilisation for the given task set. As a single task utilisation depends on how large its partition is, the algorithm tries to find a global partitioning and assignment to get a minimum utilisation for the entire task set. The determination of single task utilisation in each possible partition size is obtained through simulations or experiments.

Tan and Mooney in [153] propose a different allocation scheme strictly related to task priorities; for this reason, that approach is often referred to as *prioritised cache*. A set-associative cache is partitioned at the granularity of sets and each partition can assume a priority in the same range as task priorities. Each task is then allowed to only access cache partitions with equal or lower priority. The priorities of the cache sets are originally set to the lowest priority level; when a task uses a cache set, the set priority is raised to the task priority and is then downgraded to the lowest priority level after the task has completed its execution. The main drawback of this approach is that it guarantees high cache performance to higher-priority task at the cost of degrading lower-priority tasks,

which will not necessarily yield an optimal overall performance.

2.4.2 Cache Locking

Cache locking techniques consist in exploiting hardware support to explicitly control the cache contents. Special instructions may be used to explicitly load information in the cache and to disable the cache replacement policy, thereby locking cache contents. However, cache locking does not inhibit cache accesses.

Since all or some of the cache contents are actually locked in the cache, the cache behaviour and accesses would become fairly predictable, at least with respect to instruction cache. Furthermore, locking techniques only require a set of cache management operations to explicitly load instructions in the cache and to inhibit the cache replacement policy. Fortunately most commercial processors provide those special operations. Locking techniques can be classified as *static* or *dynamic*.

Static Cache Locking In static locking, cache contents are loaded at system start-up and remain unchanged until the system completes execution: all tasks can compete for the cache space to be locked. Hence the cache may store code from different tasks (to the extreme of all tasks).

Puaut in [122] tries to exploit the advantages of static locking techniques over static analysis. Static cache locking implies predictable memory access times, which in turns caters for simpler WCET analysis. Cache locking addresses both extrinsic and intrinsic cache interferences: cache-related preemption delay is indeed suppressed and intra-task block eviction is inhibited. Experimental results show that in most cases the worst-case performance obtained by static cache locking outperforms the one obtained by classical static analysis techniques exploiting static cache simulation. Of course, this would also depend on the size of the analysed application.

Other more recent studies focus on static cache locking, at least with respect to I-cache. Falck et al. in [55] and Liu et al. in [83] present content selection algorithms for I-cache static locking which can compute an optimal I-cache content with respect to minimising the WCET behaviour.

Dynamic Cache Locking Conversely to the static technique, in dynamic locking, cache contents are changed at specific reload points during execution. Dynamic locking techniques may be applied at different level, depending on how these points are defined:

- at *system level*, as statically-defined cache contents may be associated to individual tasks; or
- at *task level*, as specific cache contents may be associated to designated code regions of (each) single tasks.

In the former case, reload points are placed at context switches, whereas in the latter case they are placed within the task body.

Dynamic locking, however, requires reload points and respective cache states to be defined off-line before execution. Defining appropriate reload points and cache contents within a task is indeed a critical point. Since leaving this complex decision to the human operator is inconvenient, several studies [123, 124, 31] try to explore algorithmic approaches.

Puaut in [123, 124] suggests a genetic algorithm to appropriately define task regions and cache contents for dynamic cache locking. Puaut's approach introduces multiple reload points, each one associated with a statically selected cache state, with a view to minimising the WCET estimation of the program. Reload points are placed at function entry points and loops to enhance code locality and thus improve WCET performance. Cache contents are defined so as to minimise the worst-case execution path (WCEP), which is dynamically computed along the control flow graph. Experimental results show that the worst-case WCET estimates obtained by locking the instruction caches is very close to the WCET estimates obtained by uncontrolled caches (it depends on the actual extent of code locality).

A similar approach is suggested by Jain et al. in [67]: they provide a software-assisted replacement mechanism for cache blocks, which augments a basic LRU policy. Special instructions are defined for unconditionally/conditionally "keeping" (i.e., locking) or "killing" (i.e., marking as LRU) cache blocks. The issue of where to insert *keep* and *kill* instructions to a program is left to a compiler-based static analysis strategy. Experimental results show that performance (in terms of hit rate) is never worse than that obtained by applying the LRU policy. Unfortunately this approach requires specific hardware as well as software modifications.

A comprehensive approach exploiting both cache partitioning and dynamic locking has been proposed by Vera et al. in [160]. The positive effects of cache partitioning in eliminating inter-task cache interference are combined with the use of dynamic locking as a means to limit the precision loss stemming from unpredictable memory accesses. A set of lock/unlock instructions are automatically inserted in the CFG of a program to define *lock regions* where the cache content is statically determined and thus fully analysable.

Discussion From the instruction cache standpoint, the use of lockable, and software-managed caches in general, could in principle be quite effective since we are exactly aware of whether a statically-addressed instruction is stored in the cache or not. The same in not trivially true for data caches because of dynamic references.

Dynamic locking techniques may be applied to a subset of the system tasks to the extreme of the whole task set. If the cache locking technique is applied just to a subset of tasks then cache variability still affects residual tasks. On the other hand, if the cache contents were statically selected for whole task sets then it would be more suitable to use a scratchpad memory instead of cache locking.

As Campoy et al. point out in [31], static and dynamic locking techniques are not equivalent, as static locking ensures easier analysability while dynamic locking usually ensures better performance. Nevertheless, the (predictable) overhead imposed by dynamically reloading cache contents may not be paid back by actual performance improvements. At task level, the delay caused by reloading the cache between task regions may become as large as the delay potentially caused by cache misses: depending on the actual execution path taken after the reload points, some cache contents may even not be used at all. Even at system level, cache reload at context switches may incur considerable overhead.

Consequently, choosing between static and dynamic locking involves some considerations on the actual program code. In particular, when the cache size is close to the cache-worthy code size, static cache locking should be preferred; otherwise implementing dynamic locking at system level may be prove more convenient. Furthermore, if the task code exploits clearly different "working sets" for different code regions then dynamic locking at task level may be worthwhile too.

At any rate, since we are also interested in obtaining a performance similar to that provided by a conventional cache, the parts of code to be locked must be carefully selected. Whenever the application is not a simple and small one, it could prove really difficult to select the code parts to be locked unless some kind of advanced support is provided at compile time. This is particularly evident with regard to industrial-scale systems. Freezing the cache content, as a result of static cache locking, may help reducing the negative effects of unpredictable data access on cache analysis. On some hardware platforms, however, freezing the cache also inhibits any burst load mechanism, and yields to overly degraded performance. Although the combination of cache partitioning and locking could in principle improve the overall system analysability (as suggested in [160]), the industrial application of this kind of approach is still conditional on the effects on performance. Besides the already discussed limitations of the partitioning approach, dynamic locking relies on

specific hardware support to be able to load cache content at the required granularity level and, more importantly, without unacceptable performance loss. Such hardware support may not be available in simpler processors such as those in use in the HIRTS domain.

2.4.3 Scratchpad Memories

Scratchpad memories are small on-chip memories mapped to the hardware address space, which can provide fast memory accesses without affecting the overall system predictability. In fact, the contents of scratchpad memories are statically allocated in a separate address space, which makes one always aware of its contents. The considerations we made on the cache locking approach still hold for scratchpad memories, as the way contents is loaded into the scratchpad is quite similar to that used in cache locking techniques.

Memory allocation in scratchpad memories is completely software managed, whether on control by the user or by the compiler. Several studies [127, 94] focus on efficient scratchpad allocation techniques, either static or dynamic. Static allocation techniques are discouraged as they imply a plain performance loss if the code size is much bigger than the scratchpad size.

Puaut and Pais in [127] propose a generalisation of the allocation algorithm introduced in [123] for cache locking. As we mentioned above, this algorithm uses multiple reload points, each one associated with a statically defined memory state. Memory contents are selected depending on the execution frequencies of basic blocks of code along the worst-case execution path, obtained through an external WCET estimation tool. In the same work, the scratchpad memory approach is compared with the dynamic cache locking technique. The difference between scratchpad memories and dynamic cache locking caches is described as twofold. The addressing scheme in scratchpad memories is completely software-controlled whereas in locked caches it is transparent to the software. From the granularity standpoint, in locked caches the smallest lockable unit usually consists of a cache line whereas in scratchpad memories the smallest memory unit is not bounded to the cache block size but corresponds to task basic blocks. This may lead to scratchpad fragmentation since some space may be wasted when the basic blocks to be allocated are too large compared to the residual free space in the scratchpad. It is worth to note that on-chip memory may also be split into cache and scratchpad memories: a suitable algorithm for those architectures is proposed in [94].

Discussion Replacing the cache with a scratchpad memory leads to a fully predictable timing behaviour for fast memory accesses; however the actual scratchpad size limits the

amount of potential fast accesses, thus leading to poorer overall performance than that obtainable with caches. Splitting the on-chip memory into a scratchpad and a cache may improve the overall performance at the cost of an even more variable system behaviour.

However, a general methodology should be devised to properly determine which parts of code should be mapped to the scratchpad. The surveyed approaches focus on efficient (static or dynamic) scratchpad allocation techniques, aiming at reduced energy consumption or better WCET performance rather than improved system predictability. Moreover, the WCET-driven allocation algorithms presented in the above studies exploit a greedy approach since each time a different allocation scheme is considered, a complete re-evaluation of the WCET is required.

2.5 Hardware Predictability Issues

Improved cache predictability can also be obtained by avoiding hardware features or design choice that may incur more unpredictable behaviour, such as dynamic branch predictors, out-of-order execution, unpredictable cache replacement policies, etc. [19, 136, 100]. Also appropriate coding styles and code patterns may help improve WCET analysis by removing unnecessary sources of overestimation due to specific code constructs.

Hardware Design Choices Several studies [19, 59] suggest cache-specific design choices whose combination is expected to considerably improve the overall cache predictability. With respect to predictability of cache replacement policies, LRU is widely recognised as the most analysable replacement policy [59, 136]. Reineke and Grund in [134] introduce the concept of *relative competitiveness* of cache replacement policies. Using competitive analysis, they compute a bound on the additional cache misses that different replacement policies incur with respect to those obtained by applying LRU.

Data and instruction caches should be kept separate to eliminate inter-dependencies: unified caches are vastly more difficult to analyse [19]. The same line of reasoning suggests implementing separate memory interfaces for code and data (Harvard style memory architecture).

Two possible sources of overestimation in cache-aware static analysis stem from out-of-order execution and dynamic branch prediction which thus should be avoided. Both features overly increase the complexity of WCET analysis. Out-of-order execution could also increase the frequency at which timing anomalies occur whereas dynamic branch prediction may entail the so-called speculation anomalies [135]. Complex pipelines, for

example, performing superscalar or out-of-order execution, allowing hazards, etc., force static analysis to keep track of the whole pipeline states and of the interactions between instructions at different stages. From the analysis standpoint it means that complex abstract pipeline states have to be computed that accurately model the pipeline behaviour. The more complex the pipeline, the larger the number of possible pipeline states to be considered (with increasing loss in precision) and the more likely it may incur unbounded timing effects. Some approaches in the literature aim at analysing processors equipped with complex pipelines. For example, Li et al. [81] have modeled a pipeline with out-of-order execution. However, although the authors claim that the execution context of surrounding basic blocks are taken into account, they have not proved the absence of long-timing effects in their model.

Dynamic branch prediction schemes build on buffers or tables that store control flow information to predict the outcome of any branch instruction. Therefore, they expose a dynamic behaviour which inherently depends on the execution history (similarly to caches). Conversely, static branch prediction can be used instead to improve performance (though with a relatively higher misprediction rate than dynamic predictors), while still preserving predictability, as each branch outcome is statically predicted at compile time as *taken* or *not-taken*.

Timing analysis is also hampered by the use of virtual memory as the WCET analysis of virtual memory accesses have to account for the time required by the Memory Management Unit (MMU) to translate a virtual address to a physical address on every memory access. With respect to timing analysis, MMU poses the problem to statically predict both the Translation Look-aside Buffer (TLB) behaviour and the possible occurrence of page faults. However, it should be noted that, in high-integrity domain, virtual addressing and pagination via MMU are typically regarded as a mean to achieve spatial isolation (i.e. memory protection) among different processes and not as a mechanism for augmenting the address space.

Puaut and Hardy in [126] focus on the paging issue and state that the complexity of the memory paging system cannot be analysed just exploiting classical cache analysis techniques because of the inherent complexity of page replacement policies (typically some sort of PLRU). They therefore resort to a compile-time approach based on graph coloring to statically determine the program points at which a page in or page out may occur. The cited authors present an ILP algorithm variation of the same approach in [56].

Multicore processors The approaches surveyed so far address single-core architectures which has represented the main-stream architecture in the last decades. Obtaining dependable figures of the WCET can became even more difficult on multi-core platforms. Analysing the latter currently represents the current frontier in academic research and much work has still to be done in this direction.

Analysis issues related to thread interferences in multitasking environments are somehow exacerbated by possible interferences in shared physical resources, such as the memory hierarchy, buses, etc. Several recent studies have focused on real-time scheduling in multi-core systems [9] but they do not discuss how to obtain the WCET values required by schedulability analysis.

A partitioning approach for multicore systems is presented by Chang and Sohi in [33, 34] which exploits cooperative caching and cache partitioning allocation scheme in multicore systems. Local private caches also cooperate as an aggregate shared cache to keep in cache globally active data. However that approach is average-performance-driven and thus does not address timing predictability. Suhendra and Mitra in [148] propose a predictable combination of cache locking and cache partitioning in multi-core systems. They evaluate possible combination of static/dynamic locking and thread- and core-based partitioning with respect to WCET performance.

With respect to instruction scratchpads, a scratchpad-based approach is presented by Metzlaff et al. in [99], which consists of a fully predictable dynamic allocation scheme for instruction scratchpads in multi-threaded processors. Scratchpad allocation is automatically done at the granularity level of complete functions (assuming they can completely fit in the scratchpad); therefore the scratchpad contents are always and exactly determined by flow analysis.

2.6 Predictable Software

Several efforts to cope with cache predictability aim at devising a sound and computationally feasible analysis approach or try to avoid those hardware features or design choices that may incur less predictable behaviour. Over and above the adoption of sound analysis approaches and hardware countermeasures, however, we must note that cache behaviour is also highly affected by the actual program code, both in term of performance and predictability.

It is worth noting that both cache-aware code patterns and coding styles would be more easily enforced through automatic code-generation approaches (as well as compiler

transformations), thus to guarantee analysability by construction.

Predictable Code Constructs Appropriate code structures may contribute to improving WCET analysis by removing those unnecessary sources of overestimation that hamper both static analysis and hybrid measurement-based approaches to cache analysis. More generally, coding constructs and styles may also affect, both negatively and positively, the tightness of WCET analysis caches.

Some code constructs are known to be more difficult to analyse. For example, indirect calls pose a threat to static analysis, which may not be always able to automatically detect call targets. The same holds for the switch/case construct, when implemented through indexes in a dynamic table. When it comes to data caches, static analysis is unable to cope with dynamic allocation and is penalised by any load/store instructions that reference multiple memory locations (e.g.: arrays indirection, pointers, etc.). When static analysis fails or incur too much overestimation the user is required to provide additional information (typically through annotations) to refine and guide the analysis. As observed in 2.3.1.1 user intervention in the analysis process is both laborious and error-prone.

However, the set of code constructs that should be avoided to facilitate WCET analysis is commonly known but, to the best of our knowledge, no classification of "bad constructs" is proposed in the literature. We will further detail on this topic in Chapter 3.

Analysis of Software Structure Besides the previous influence factors, the timing behaviour of a program is affected by structure of the software itself. Increasing complex software may hamper the viability of WCET analysis. The adoption of state-of-the-art development approach for complex systems, as Component-Based Design (CBD) have gained enormous attention in recent years. CBD allows to facilitate the economic development of complex software by reuse of pre-built components. However, it poses a threat to timing analysis, due to complex control flow inside (possibly black-box) components as well as dependencies between components. Szulman in [149] focuses on data-flow and control-flow dependencies between complex components.

Fredriksson et al. in [49] observe that due to the generality of reusable components, the analysis of Component-based software may lead to a severe overestimation of each component WCET. They suggest a method that allows to associate different parametric WCET bounds to the one and the same component, thus resembling to the actual component behaviour. However, to limit the complexity of the problem, they assume no hardware influence in the component WCET determination. For example, they do not consider ef-

fects from different memory layouts.

Code Optimisations It is worth noting that the final application code is obtained through a set of compiler transformations which could be transparent to the programmer. Some studies [88, 46] aim at devising WCET-aware compiler optimisations to be automatically applied at compile time. In particular, Lokuciejewski and Falk in [89] take into account the effect of memory layout on the cache behaviour and aim at computing a WCET-aware memory layout for a program, that minimises the number of possible conflict misses.

2.7 WCET Tools in Industrial Experiences

Although several approaches have been proposed in the literature to derive safe and tight bounds of the WCET of tasks running on cache-equipped processors [163], those techniques and tools are yet to be fully embraced by HIRTS industry, most of all by its most conservative members.

Several prototype tools are currently maintained by the most active WCET research groups: for example the SWEET tool [93] from Mälaradalen University, the Eptane tool from University of Rennes I, the Chronos tool [80] from the National University of Singapore (NUS) and the several prototypes from the Vienna University of Technology (TUW).

Some cache analysis techniques, however, have been successfully integrated into commercial tools and progressively forced their way into industry: RapiTime [133] and aiT [1] represent the current main-stream industrial level tools. RapiTime is based on the hybrid measurement-based approach introduced in [20] whereas aiT builds on the static analysis approach based on abstract interpretation defined in [48]. Those tools have been successfully applied in industrial case-studies [146, 54] especially related to the automotive and avionic domains. Other industrial-level commercial tools, such as Bound-T [155], do not support the analysis of cache equipped processors.

Automated code-generation facilities, which follow the Model-Driven Engineering (MDE) paradigm, are increasingly adopted in the HIRTS industrial domain (e.g. SCADE [42], Matlab/Simulink [96], etc.). As a matter of fact, industrial-quality HIRTS routinely integrate generated code from different modeling tools together with manual code. To the best of our knowledge, however, no industrial-quality code generation engines exists that is expressly focused on generating predictable cache-aware code. This notwithstanding, this topic has recently gained some interest in the WCET academic community (e.g., [72, 151]). Some advanced software development suites do address the integration of

WCET analysis tools in the development process. The integration of the aiT WCET analysis tool with SCADE and Simulink has been addressed in [47, 151].

2.8 Research assumptions

As discussed in section 2.5, the more complex is the processor, the more difficult the analysis, whether by static analysis or hybrid measurement-based methods. In our study we will mainly address relatively simple single-core hardware architectures, thus excluding all those hardware features that overly complicate WCET analysis (e.g., out-of-order pipelines, dynamic branch predictors, unpredictable cache replacement policies). Although we are aware that the mainstream research is currently moving towards more advance processor and multi-cores in particular, in Section 1.1.2 we already observed that the current processor baseline in HIRTS is still a relatively simple single processor system.

In order to reduce time and cost of the development of HIRTS, component-based and model-driven engineering approaches are increasingly adopted to facilitate an early V&V of a system and to inject good practice in the automated generation of part of the system itself. Our study will leverage on the increasing penetration of the MDE paradigm in the HIRTS domain.

We also assume non-partitioned preemptive systems and avoid any consideration on other models, though aware of their adoption in some HIRTS application domain (e.g., IMA in avionics). Moreover, when considering predictability of code constructs and patterns we may need to define the target programming language and compiler as the executable code highly depends on both of them. In remainder of this work, we try to be as independent as possible from a specific language or compiler; however, on specific issues, we address the Ada programming language and the GNU GCC compiler.

Chapter 3

Cache-aware Development Process

3.1 Introduction

HIRTS industry fears that the adoption of cache-equipped processors may threaten their already onerous and complex verification and validation process. The High Integrity domain asks for guarantees that the timing behaviour of a program would meet the expectations made at design time, even in presence of caches. This can be hardly accomplished without a proactive attitude towards caches and timing analysis in general already as of the early stages of the development process. In order to facilitate a conscious adoption of caches in HIRTS, a systematic approach is needed which should involve and influence the whole development process.

In this chapter we reason on the current role of timing information in the development process and elaborate on a systematic approach that enables timing analysis and inoculates cache-awareness into the industrial-level development process of HIRTS.

3.2 Characterisation of industrial development process

In the high-integrity domain, the software development process must meet the requirements set by the applicable qualification/certification standard, which determines the design, development, verification and validation criteria for a fully predictable and dependable application. The more critical the application, the stronger the requirements to be met and the larger the amount of formal activities and documentation prescribed.

Although domain-specific standards do not explicitly impose a particular life-cycle model, their set of rules and directives, as well as consolidated practices, may bring about

the adoption of specific process models in each application domain. By and large, the software development process typically follows the classical "V model" in Figure 3.1. This widespread model is characterised by the specular placement of two set of activities preceding and following the implementation phase. The descending part on the left comprises requirement definition, high-level and detailed design; whereas the ascending part on the right includes test, integration and verification phases. A strict relationship (represented by dashed arrows in the picture) holds between activities on the left branch and the corresponding verification and validation steps.

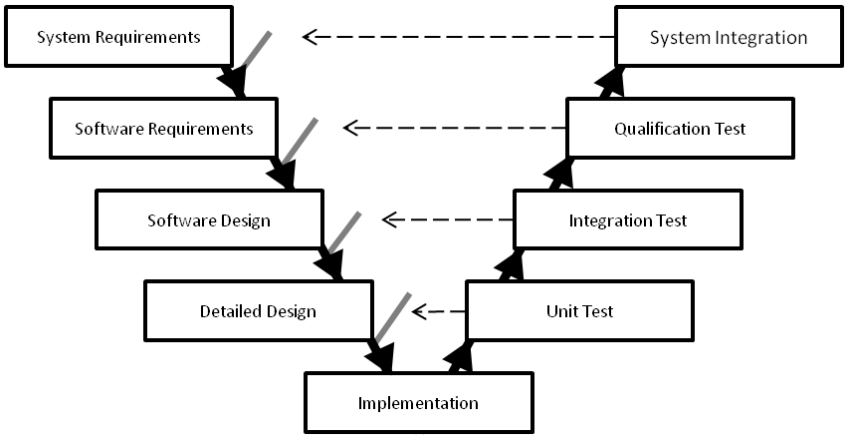


Figure 3.1: The V model development cycle.

The small branches protruding from the left arm of the V represent *incrementality*, which is a distinguishing feature of HIRTS development process. The development in fact proceed through incremental software releases, which individually follow the life-cycle model. Incrementality is required both to better master the overall complexity of the software and to enable the cooperation of different contractors on the same project. Iterations, instead, may be locally required to reassess aspects of the development; in HIRTS development, however, they are feared (and possibly avoided) because of the potential destructive effects of regressions on the project economy.

3.2.1 Role of timing information

As we already sketched in Chapter 1, in spite of the considerable progresses in timing analysis, a combination of software simulation and testing with safety margins still continues

to be the current industrial baseline. Rough initial WCET estimates obtained from past experience are augmented with safety margins and fed to schedulability analysis. These estimates are then refined against available prototypes and legacy code. It is however from the implementation and unit testing phases on that several measurement instruments (e.g., cycle-accurate simulator, oscilloscope, logic analyser, etc.) are used to dynamically analyse various pieces of software under a selection of inputs, in attempt to exercise realistic conditions.

As a matter of fact, timing analysis is pushed and compressed near the end of the development process with a massive amount of measurement-based analysis performed by testing. Although early measurements can be performed along with the implementation phase to consolidate the WCET bounds, their safeness still relies on a poorly representative test coverage, backed by safety margins. Although the prominent role of timing information is well perceived by industry, the way this information is collected and exploited is not commensurate with the required level of confidence. Moreover, the predictive value of scenario-based measurements is drastically reduced by the impact of caches in the timing behaviour of a program. Measurements are thus not necessarily safe and it is difficult to use them with confidence in schedulability analysis.

High-integrity industry cannot simply accept that the level of uncertainty incurred by the unrestricted use of caches can only be remedied a-posteriori on costly feedback cycles triggered by analysis results. As timeliness of the system is a major concern in HIRTS, timing information should instead pervasively contribute to every single step in the development process, from early considerations on timing requirements, to schedulability concerns in architectural and design phases, to predictable code implementation, to timing assessment in the final V&V activities. Ideally the system timing behaviour should be determined as early as possible in the development process, to allow an early detection and correction of timing problems and faults. In this setting, the dynamic testing campaign typically performed near the end of the development process should only confirm the expectation made on the system timing.

3.3 Timing analysis: the industrial perception

HIRTS industry currently stands in an awkward position with respect to timing analysis. On the one hand, they perceive that the current development approach is jeopardised by the migration to more complex cache-equipped processors that expose to potentially large jitters: they actually agree on the point that a more structured approach to timing analysis

would bring an added value to their process. On the other hand, however, they fear the costs that a factual application of state-of-the-art timing analysis would add to an already onerous development process. Their perception of advanced WCET analysis is that the quality of its results may not match the costs it incurs for large complex systems

As often happens, the truth lies in the middle. The current industrial approach to timing analysis is poorly equipped to cope with the variability incurred by caches. The strong requirements on timing predictability can hardly be fulfilled if timing concerns are only addressed in the final steps of the development process. It holds equally true, however, that the introduction of (cache-aware) timing analysis techniques and tools in a consolidated process may incur costly and demanding efforts of integration in so far as they may require the introduction of new design-time decisions or the allocation of new process activities and tasks. From the industrial perspective, any amendment to industrial practice must be carefully evaluated with respect to attainable benefits and induced costs.

In the scope of the COLA project [102], the author has been involved in an evaluation of a score of approaches to cope with cache unpredictability, enriched with relevant industrial feedback on their industrial fitness. What clearly emerged from that evaluation is that, in order to be embraced by industry, any countermeasure or techniques should not have disruptive effects in the software life cycle, but should be easily integrated in the consolidated industrial practice. Without casting any doubt on their correctness, each proposed approach and technique should be in fact be evaluated against the prospective benefits, in particular with respect to their effectiveness, efficiency, scalability and required knowledge and skills.

With respect to WCET analysis, building on a sound and precise model of the target processor (which is increasingly less obvious), static analysis techniques allow to compute a safe WCET bound for a program. Industry, however, is often discouraged from adopting static analysis tools and techniques by the added complexity that their use is expected to inject in the final verification and validation process. Since static analysis approaches and tools may suffer from exceeding overestimation, complex low-level annotations (e.g., on loop bounds or addresses of memory accesses) may be required to improve the precision of the analysis results. The provision of effective manual annotations, besides being an onerous and error-prone task, requires sharp skills as well as a deep knowledge of the program itself. Knowledge about the program behaviour, which is naturally kept by the system designer and by the software developer, may be unavailable at the end of development and difficult to retrieve. This is the case, for example, when the program analysis is not performed by the same person that designed or developed it, or even when the program

includes black-box outsourced components or external libraries. As we observed in a long term investigation on a large piece of real industrial-scale software [103], the limitations (in terms of scalability and costs) of WCET analysis may abruptly emerge when facing complex industrial software, developed without analysability in mind. Under that condition, in fact, the analysis process reveals itself to be overly complex and costly and the quality of its results rapidly degrades.

In contrast with static analysis, hybrid measurement-based techniques produce WCET estimates or distribution that may be realistic but are not necessarily safe. Since hybrid analysis builds on measurements performed on the actual hardware, the latter should be available to the user. This prerequisite rules out all cases when the hardware is developed in parallel with the software. However, this circumstance is not that frequent in the HIRTS domain where at least a cycle-accurate simulator is typically available. Hybrid analysis is intrinsically a less demanding approach compared to static analysis as it typically does not require deep knowledge about the analysed program. Therefore, analysis can even be performed on black-box components, simply triggering their execution by the test harness. On the other hand, and for a similar reason, the results of hybrid analysis are less trustworthy: if some extreme or unexpected timing behaviour is observed during measurements, the analysis results cannot help us determine the exact reason of such behaviour. As compared to static analysis approaches, hybrid analysis requires that all program code must be available, since measurements does not allow to declare execution-time bounds for unimplemented parts of a program. Finally, from an industrial perspective, the fact that the hybrid analysis process does not diverge too much from dynamic analysis may prevent industry from appreciating its benefits.

3.3.1 Approaches to predictable caches

Besides timing analysis techniques, several approaches have been proposed to specifically cope with cache-induced variability and unpredictability. These approaches, which have been extensively introduced in Chapter 2, should be evaluated against their fitness to the industrial development process: in the following we briefly review their pro and cons from the industrial perspective.

Restricting the cache behaviour. In contrast to canonical analysis approaches, which tend to be applied near the and of the development process, restricting the cache behaviour for the sake of a reduced variability and unpredictability is typically regarded as a design-time activity that will be effective just at the end of the development process. However,

cache partitioning and locking, as well as the adoption of a more predictable scratchpad instead of caches, also affect how the system is actually developed and permit to make assumptions on the results of the unit test campaign.

The main perceived drawback entailed by the decision of partitioning the cache to minimise the interference between tasks is the reduction of the usable cache size for each task. The performance of cache partitioning is thus strictly related to number, size and assignment of cache partitions. Due to the intrinsically tiny size of caches with respect to the size of a program, the overall system performance can be greatly penalised. Especially for large systems, assigning cache partitions to tasks can prove to be quite a complex job, as naively assigning partitions in accordance with task priority (i.e., by urgency) or to task rates is unlikely to be the best choice. Moreover, from the concrete implementation standpoint, the software partitioning approach may prove cumbersome for complex systems, whereas hardware partitioning requires explicit hardware support, which is not always available in the HIRTS processors baseline.

When it comes to cache locking, both static and dynamic approaches rely on the explicit selection of the parts of the application code or data that shall be explicitly loaded in the cache. Unless the application is simple and small, making that selection could be very difficult. Performance issues should also be accounted for in making that decision, as static locking incurs poor cache performance while dynamic locking incurs overhead when reloading the cache content. It is worth noting that the provision of predictable cache accesses is relatively simple for the I-cache while is not an easy matter at all for the D-cache, primarily because of dynamic references. A combination of cache partitioning and locking, while potentially improving the overall system predictability and analysability, may still expose to unsought performance loss.

Similarly to cache locking, the use of scratchpads requires the user to explicitly single out application code or data to store in them. A general methodology, which does not exist yet, should be devised to properly determine which parts of an application should be mapped to the scratchpad separate address space. With respect to I-caches, the reduced dimension of a scratchpad is likely to produce inadequate performance for large systems. The scratchpad approach could be more efficiently applied to data rather than to code, assuming a sound method to content selection.

Predictable coding patterns and styles. Cache-aware coding is straightforwardly applicable at the implementation stage of the development process as it explicitly addresses software implementation issues. However, code predictability should inform both task-level

code constructs and software architectural-level design choices. Therefore, when it comes to software architecture issues, cache-awareness should also address architectural-level decisions taken in the high-level and low-level design phases. Appropriate code patterns and coding styles may certainly help improve WCET analysis by bounding the sources of overestimation. However, this would require a major change in the predominant way of programming, which is typically geared to optimising the average case, obliviously of its effects on the worst case.

Layout optimisations. Layout optimisations have been proved to be quite effective, at least with respect to I-caches. As proposed in the literature, the application of layout optimisations needs to account for the structure of the whole system as local improvements for a software module or task may penalise other modules. The natural allocation in the development process is then in the integration phase when the final system is available. Provided that automated support exists, the enforcement of a controlled layout does not require special skills.

The computation of a cache-aware layout is most effective when performed on the final executable at the end of the development process. However, the desired cache behaviour may not be preserved when the system is modified or other modules are added to the system. This observation clashes with the incremental nature of the typical HIRTS development process.

Selective caching. The explicit run-time control of the cache utilisation may help improve both the performance and the predictability of cache-worthy tasks and code. Similarly to the approaches for a controlled cache behaviour (partitioning, locking and scratch-pads) the run-time management of the cache operation builds on design-time decisions and affects both the development and unit testing phases.

Again, however, the criteria to follow in the selection of the parts of the application that may be allowed to use the cache are not easy to determine and automate. This is in fact a critical choice which should be taken early in the design process, where the most interesting information on loop- and data-intensiveness, as well as criticality of tasks, may not be available. Cache inhibition should also be guaranteed not to overly degrade the memory latency, as it happens for the LEON2 processor [3] where the resulting time penalty discourages its adoption.

Cache-aware compiler support. Compilers play a crucial role in the determination of the timing behaviour of a program as they generate the object code that will be actually executed at run time. Compiler cache-aware optimisations/transformations and support for cache analysis make it possible to enforce and preserve predictable source code constructs in the source-to-object mapping. Those activities can be naturally mapped to the software implementation phase.

Although the effectiveness of compiler transformations and optimisations has been proved, it is worth noting that their introduction touches the innermost workings of the compiler. Its deployment therefore brings about the use of non standard compilers or at least important modifications to them. This in practice is a cumbersome and challenging endeavour from an industrial standpoint.

Although the surveyed approaches may help improve the predictability and analysability of the cache behaviour, none of them lends itself to a straightforward integration or application in the industrial development. All those approaches in fact share as a common drawback the lack of either a methodological approach or an adequate tool support, which cannot be dispensed with for an extensive industrial application. Table 3.1 summarises pro and cons of the considered approaches from the industrial perspective.

Approach	**Perception**	**Pro/Cons**
Cache partitioning	Doubts on performance	✗
	Lack of methodologies for partition size and allocation	✗
Cache locking and scratchpads	Doubts on performance	✗
	Lack of methodologies for code/data selection	✗
Predictable coding patterns and style	Ease the application of timing analysis	✓
	Difficult to enforce	✗
Layout optimisations	Almost straightforward when automated	✓
	Wrecks incrementality	✗
Selective caching	Difficult to select cache-worthy tasks	✗
	Potential performance loss	✗
Cache-aware compilers	Effortless application	✓
	Implies use of non-standard compilers	✗

Table 3.1: Industrial perception of common approaches to improve cache predictability.

3.3.2 Industrial requirements on timing analysis

The perceived gap between state-of-the-art timing analysis techniques and their industrial adoption can be formulated as a set of high-level applicability issues. Those issues, in turn, should be intended as a set of preconditions on the full industrial applicability of timing analysis approaches.

Issue #1 - Scalability. The computational complexity in both the time and space dimensions of the technique cannot be disregarded when considering its industrial fitness. The applicability of state-of-the-art timing analysis may be in fact hindered by the dimension of the problem at hand. In respect to static timing analysis, tractability problems are known to arise in the presence of complex hardware features (e.g.: caches, out-of-order pipelines), which cause the state space of the abstract processor model to explode in the attempt to fully and accurately account for the inner state of all the hardware features with bearing on the WCET. However, computational issues may even occur for relative simple processors, such as those adopted in HIRTS (e.g., the LEON2 [3] processor). Industrial-scale software artifacts are large complex systems, composed of a mixture of automatically generated and manual code. It is not rare for such systems to exhibit a massive call graph: in a recent experiment on a relevant part of a real OBSW [103] we stepped into a huge graph of more than 1,300 procedures for a single task entry point (see Figure 3.2).

In this context, since most procedures are executed within several execution contexts, inter-procedural analysis shall be applied for the sake of reducing the pessimism that would otherwise be incurred by considering always the worst-case context. The clear drawback of a massive application of inter-procedural analysis to a complex call graph, such that reported in Figure 3.2, is a state space explosion similar to that experimented with more complex hardware.

Even hybrid analysis, although at a different degree, is not exempt from tractability issues. In the presence of a complex system, as that in Figure 3.2, the size of the program traces, which are collected to derive timing information of single program segments, may easily and swiftly become unmanageable.

Beside complexity in space, also the computation time required by the analysis steps may dramatically increase along with the complexity of the analysed software.

Issue #2 - Required skills and knowledge. Industry is also sensitive to the costs that may be incurred by a more rigorous application of timing analysis, both in terms of underlying skills and capabilities of the personnel involved, and time spent in the analysis process. Although current best-of-breed analysis tools pull for an increasing level of automation in

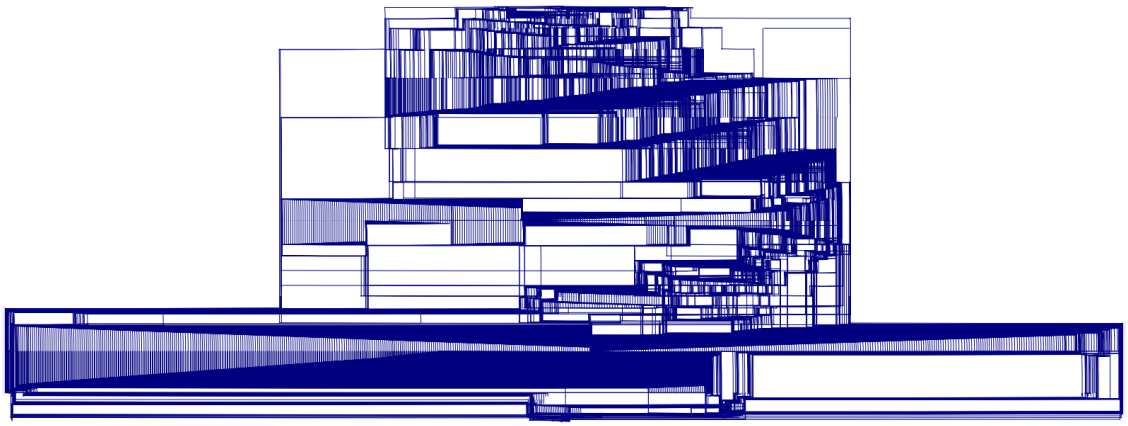


Figure 3.2: Complexity of an industrial-scale call graph (from aiT).

the analysis process, a magical one-button solution does not exist yet. Regardless of the analysis approach (either static or hybrid timing analysis), a tool user is expected to supply information on loop bounds, dynamic jumps, etc., that could not be automatically extracted from the program. This is not a negligible issue in those industrial settings where WCET analysis is not a consolidated practise. Besides the unavoidable training on a specific analysis tool, the collection and provision of trustworthy timing information to guide the analysis process ask for specific skills and knowledge of the program under analysis.

The assumption that the user can be a trusted source of flow facts is extremely fragile for large, complex and long-lived industrial programs. It is worth considering that the engineers that should analyse the timing properties of the program are seldom those who designed or produced it. This is even more true for the large amount of legacy code, which is a most valuable industrial asset.

Moreover, the sought-after information is often not available at source-code level as it depends on the code generation engine of compiler back ends, which for example determine how the program control flow is reorganised, introduce loops and branches not traceable to the source code. The information may also be scattered over large slabs of the code base; or may not even be available at all in the program at hand as it may depend on higher level design choices, far removed from it. Especially in the latter case, it may even be impossible to accurately and precisely reconstruct the sought information.

In consideration of its critical role in determining both safeness and tightness of the analysis results, collecting timing information in the form of flow facts to be fed to the

analysis tool may prove to be a challenging and onerous task.

Issue #3 - Relationship with schedulability analysis. Timing analysis is typically kept separate from system level issues as the focus in WCET analysis is set on the timing behaviour of individual tasks, under the simplifying assumptions that they run in isolation. It is up to feasibility analysis to account for the costs of context switching and external interference suffered by the task owing to the effects of preemptive concurrency on history-dependent shared physical resources, caches above all.

The apportionment of inter-task interferences in advance schedulability analysis techniques, as proposed by increasingly advanced approaches (cf. Chapter 2), may not be straightforward in complex and task-intensive preemptive systems. The separation of intra- or inter-task analysis gets blurred as a consequence of the common occurrence of self-suspending tasks, timer alarms and watchdogs in industrial software. This unclear separation manifestly clashes with the consolidated industrial approach to timing analysis, which is historically based on response time testing.

Issue #4 - Perceived quality of the results. HIRTS industry expects timing analysis to produce both safe and tight WCET bounds. This makes static timing analysis the most natural candidate for use in critical systems. Probably the plainer hindrance to the industrial application of static timing analysis originates from the observed difference between the longest measured execution times and the computed (safe) WCET bounds. To some extent, such difference is justified by the unavoidable conservative attitude of WCET analysis. However, the necessary conservative approach of timing analysis in combination with the industrial-scale of the analysis problem and poorly analysable code constructs often results in extremely large differences, which are hardly justifiable. The perception of such a significant gap is likely to shed some scepticism on the accuracy of the computed WCET, which, at its extreme, may lead to ignoring those WCET bounds altogether.

In contrast, the results of hybrid analysis, do not generally suffer from exceeding overestimation, though some pessimism may still be incurred by the inclusion of infeasible paths when combining measurements on program segments. However, hybrid analysis cannot guarantee to produce safe bounds (but only realistic estimates) unless a fully exhaustive test coverage is achieved.

Issue #5 - Cost-efficiency. The application of timing analysis to complex industrial systems, developed without any particular attention to the timing analysability dimension, typically poses strong requirements on the user's capabilities and efforts. As observed in

Issue #2, regardless of the specific analysis tool, the user is in fact required to peruse an impressive amount of source and object-level code to get an adequate knowledge of the program flow facts. The more time is spent in collecting such information, the less overestimation will be incurred in the analysis results. Economical considerations may ask for a compromise between analysis costs and results.

The costs incurred by the provision of precise manual annotations in a scenario with more than 1,300 procedures (just for a single task entry point), inclusive of poorly analysable code constructs and software architectures, easily exceed those typically required by the current industrial practice. In particular, the user is often forced into an extremely time-consuming and trial-and-error process of uncovering and fine-tuning the annotations required by the analysis. This observation, in addition to the considerations on the quality of the analysis results, may make timing analysis scarcely cost-effective and difficult to defend against the more familiar maximum-observed execution times.

Issue #6 - Extensive tool support. A timing analysis techniques is hardly ever taken into serious consideration for industrial application if it lacks an adequate tool support. Industrial-level advanced tools are currently available for static timing analysis [1, 155] and hybrid analysis [133]. The same does not hold, however, for the reviewed techniques for an improved cache predictability. The industrial applicability of those techniques is heavily conditioned by the provision of comprehensive - and as automated as possible - tool support, to guide the user in the correct application of the proposed method. For example, tool support is hardly needed to devise proper scratchpad and cache locking allocation schemes, or to optimise the memory layout. The costs that industries may be willing to afford for such tools varies on a subjective basis. Those tools as well as their outcomes should also be easy to integrate each other and with the consolidated tool-chain (cf. following issue).

Issue #7 - Integration in the software life cycle. Finally, the adoption of a new approach to timing analysis is likely to ask for additional development activities and tasks. The introduction of new approaches and tools should not break the industrial practice but should rather fit in the preexisting development process. This consideration also holds with respect to the consolidated industrial tool-chain: specific timing analysis approaches or tools should conform to the development methods, hardware components, compilers, tools, etc. in use. It is worth noting that static analysis sets comparatively more rigid requirements in this respect than hybrid analysis.

The timing analysis method should also be able to accommodate the peculiarities of

the industrial development process, among which incrementality is the main characterising trait in the HIRTS domain. The HIRTS development process, in fact, proceeds incrementally in the implementation, integration and qualification of the system. Incrementality is prerequisite to return on investment from the reuse of architectural building blocks. Current timing analysis approaches are inherently not incremental, especially in the presence of caches, as they critically relies on the availability of information that can only be safely determined on the final executable, near the end of the development process (e.g.: actual code and data layout).

The above issues on the industrial applicability of timing analysis tools and techniques (summarised in Table 3.2), are inevitably interrelated with each other. Scalability, required skills and knowledge, as well as the available tool support, all participate in determining the quality of the results and the costs incurred by the analysis process. A comprehensive approach that simultaneously addresses several requirements (rather than a single one) is more likely to effectively contribute to narrowing the gap between state-of-the-art timing analysis and HIRTS practice.

Id	**Issue**
I 1	Scalability
I 2	Required skills and knowledge
I 3	Relationship with schedulability analysis
I 4	Perceived quality of the results
I 5	Cost-efficiency
I 6	Extensive tool support
I 7	Integration in the SW life cycle

Table 3.2: Issues on the industrial applicability of timing analysis.

3.4 Enabling a structured approach to timing analysis

Inherent limitations in combination with the intrinsic complexity of real-world systems still hinder the extensive industrial application of state-of-the-art cache-aware WCET analysis. Apparently a methodological gap still exists between state-of-the-art approaches and industrial state of practice: most of the industrial requirements on timing analysis still remain unfulfilled. This gap can be hardly filled by extemporaneous approaches aiming

exclusively at reducing the cache-induced variability. Despite all the proposed approaches may positively affect cache predictability and analysability, none of them seems to lend itself to a straightforward integration or application in the industrial development process.

We contend that the only viable solution for industry to account for the variability introduced by caches is the adoption of a structured "cache-aware" approach aimed at a twofold objective:

(i) allowing a cost-effective application of state-of-the-art WCET analysis to complex systems; and

(ii) minimising the variability and unpredictability incurred by the cache behaviour.

We maintain, in fact, that objective (ii) can be hardly fulfilled other than by way of a novel and more rigorous attitude of industries towards timing analysis, that should reconcile the current industrial approach to timing analysis with the industrial expectations on timing predictability. We contend that this objective can be achieved only through (i).

The first step towards this direction consists in breaking the unbalanced accommodation of timing concerns which are typically addressed as an afterthought to software implementation. Timing concerns are currently addressed near the end of the development process where timing analysis is typically applied, regardless of the employed technique. We contend, instead, that all development activities, from high-level design to software implementation, should account for timing concerns. Earlier considerations of timing concerns (Figure 3.3) allows to steer the whole development process towards the implementation of analysable and predictable systems. Only under those premises we can expect the final timing analysis step to confirm the information collected during the preceding phases.

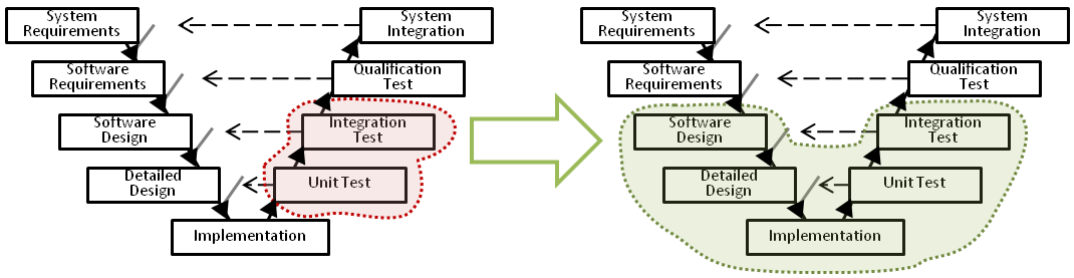


Figure 3.3: Early addressing of timing concerns.

Timing information, as well as an early application of timing analysis tools and techniques, is relevant even in the early stages of development when architectural choices and later detailed design decisions are taken. At these stages, early figures on the timing behaviour, from prototypical implementations and even reused software components, may guide the system designer in taking decisions on the overall system configuration, like task allocation, execution-time budgeting, scheduling policy, etc. The timing dimension is obviously critical in the software implementation step and in the successive unit testing phase, where (partial) WCET estimates can be computed for the available system components considered in isolation.

All these development activities should instead be performed with timing analysability in mind. The software development mantra should be that of minimising any potential source of unpredictability so that to ease the overall system analysability.

Provided the effectiveness of a more comprehensive approach to timing analysis, however, we still need a practical approach to enable this sought change of attitude in the industrial practice. In this respect we identify two practical means for promoting a cache-aware and timing analysis oriented development process:

- the identification and enforcement of *predictable design choices and code constructs*, as we deem software analysability to be the key enabler for a cost-effective application of timing analysis in industrial setting; and

- the identification of an *incremental cache-aware layout optimisation* that better fits the industrial development process, as we consider the memory layout to be the main source of cache variability.

Both those techniques have been already assessed in Section 3.3.1 as the most promising approaches to improve cache predictability and, in a broader sense, to enable the application of timing analysis in industrial setting. In our investigation we focus on removing their identified drawbacks so that to make them effectively applicable in the industrial development process. In the following we briefly elaborate the motivations that support the main constituents of our approach.

Identification and enforcement of analysable software. The application of state-of-the-art timing analysis approaches to complex industrial system is hampered by the presence of adverse code constructs and the implementation of timing-unaware design choices that may overly complicate the analysis process and gravely impair its results. As a preliminary step, we thus identify and provide a taxonomy of those code constructs and design

choices that may negatively affect the software analysability. In a consolidated development industrial practice, however, switching to a timing-aware attitude in both design choices and implementation requires an onerous and long-term effort. We propose to overcome this practical issues by leveraging on the model-driven engineering approach and its code generation facility to enforce an improved degree of analysability by construction, where those code constructs and design choices that actually hinder the analysability of a system are automatically avoided. We intend our approach to ease the application of WCET analysis by improving the overall system analysability and by supporting the analysis process with automatically generated annotations. This would enable a cost-effective application of WCET analysis techniques even to complex large-scale systems.

Incremental cache-aware layout optimisation. The memory layout of a program determines the pattern of hits and misses that will be incurred at run time. In particular conflict misses, arising when multiple references map to the same cache line, are known to be the main source of variable cache behaviour. Layout optimisation techniques, which are quite effective in coping with this source of variability, are typically applied at the end of the development process, on the final executable. This late applicability crashes with the incremental nature of the industrial development process model, where early guarantees on the timing behaviour of incomplete releases of the system are required to facilitate a prompt detection and reaction to potential timing hazards. We propose a novel layout optimisation techniques that allows an incremental application to successive software releases with guarantees on the already analysed software modules. In our intention, an incremental layout optimisation would allow at the same time to control the cache variability stemming from the memory layout, and to enable the application of state-of-the-art WCET analysis techniques already in the early stages of the development process without the need to rerun all analyses upon each increment.

3.4.1 A cache-aware development process

Our idea of cache-aware development process consists in a development process that is extensively oriented to timing analysability and predictability. The combination of the proposed approaches fosters the adoption of a proactive attitude towards WCET analysis and cache variability, where timing and analysability concerns are addressed along the whole software development.

The timing-analysis oriented MDE framework we propose as part of our approach straightforwardly addresses timing concerns from software design to implementation. The

preliminary identification of inattentive design choices and code constructs is used to restrict the design space of the software designer so as to allow only those software design that will result in the automated generation of analysable code by construction. This makes both software design and implementation almost transparently informed to timing analysability concerns.

A general improvement in software analysability also straightforwardly affects the analysability of the cache behaviour, as part of WCET analysis. Our approach, however, also specifically addresses the reduction of cache variability as a means to further improve its analysability and predictability. Our layout optimisation technique allows to govern the cache variability originating from the memory layout of the program. Moreover, the incremental applicability of our approach better fits the characteristics of the HIRTS development process and facilitates an early application of timing analysis.

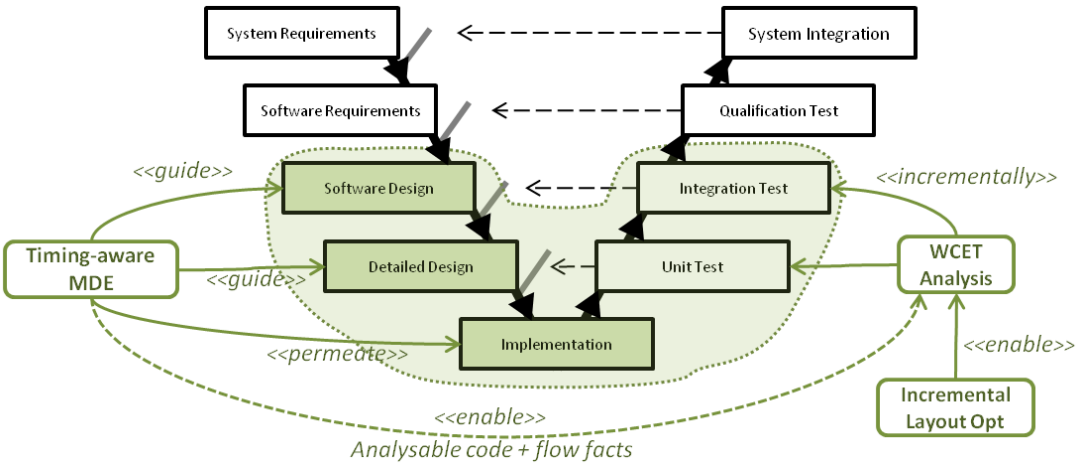


Figure 3.4: The amended V model.

The devised approaches can be seamlessly accommodated in the classical V model to configure a cache-aware and timing-oriented development process, as summarised in Figure 3.4. The comprehensive MDE framework we propose, in fact, guides the design process of both task-level and system-level concerns and, as its final outcome, enforces the generation of highly predictable code and relieves the user from defining most of the flow-fact annotations required by WCET analysis. The incremental application of our layout optimisation technique allows to collect early guarantees on the cache behaviour of each incremental release of the system. The combination of improved code analysability,

supported by the automatically generated flow facts, with the guarantees that the cache behaviour will not change on subsequent software releases is expected to enable an early application of WCET analysis.

As a fundamental industrial requirement the proposed set of countermeasure or techniques should not have disruptive effects in the software life cycle, but should be easily integrated in the consolidated industrial practice. In this respect, both the proposed approaches are characterised by a high degree of automation. By leveraging on the MDE code generation framework, the production of analysable code comes at virtually no cost for the system and software designer. Similarly, the application of our incremental layout optimisation approach is enabled by a fully automated support tool that allows a straightforward enforcement of an optimised layout.

The following sections will be devoted to the elaboration of the main constituents of our proposed approach.

3.5 Predictable Software Systems

The quality of the results of timing analysis critically depends on the complexity of the underlying hardware, regardless of the adopted analysis approach. Several studies moved from this observation either to devise hardware countermeasures to unpredictability or to identify those hardware features that should be avoided to improve the analysability of a system. This is particularly true for caches as several cache design choices directly affect its predictability: studies on replacement policies [136], cache locking and partitioning, and alternative scratchpads all seek improved cache analysability (e.g., [124, 94, 70]). However, just focusing on hardware predictability does not guarantee the analysability of a system: the role played by the software dimension itself in analysing a system should not be disregarded.

The detrimental effects on timing analysis of some code constructs and patterns are widely acknowledged in the literature. This notwithstanding, it is not rare for these constructs to be assumed to not be present in the program under analysis. Unfortunately, the assumption that software developers are somehow educated to produce only predictable code is poorly justified in practice, even in HIRTS.

In a recent study [103], we reported on an attempt to statically analyse the WCET behaviour of a large scale piece of software, part of a real on-board satellite system. We provided evidence that timing analysis of complex industrial software *as-is*, developed without any particular attention to code analysability, is hardly feasible in practice. The

reasons of the issues we encountered were twofold: they were in part ascribable to the presence of unpredictable code constructs, and, in the remaining part, derived from domain specific design choices and the overall complexity of the software itself.

Several software development standards have been proposed for the sake of improved safety in HIRTS software. Some of them come in the form of coding guidelines, as MISRA-C [105] in the Automotive domain. Others, instead, consist in the explicit definition of programming language subsets, as the SPARK Ada approach [17], which also supports semantic annotations. Although the set of guidelines defined by most approaches may also facilitate timing analysis, their main focus is set on functional correctness of programs and they very seldom explicitly aim at improving timing analysability or predictability. As an exception, the Ravenscar profile [27] for high integrity systems is a standard subset of the Ada language that focuses on the definition of systems that are amenable to schedulability analysis.

We deem the software analysabililty dimension to be of utmost importance in the HIRTS domain as it is evident that poorly analysable code will lead to as much poor WCET bounds, if not prevent analysis altogether. Interestingly, a taxonomy of unpredictable code constructs has not been defined in the literature yet. Such a categorisation would make developer aware of the effects of coding constructs on timing predictability, suggest their avoidance or identify valuable alternatives. Furthermore, we contend that, besides code constructs and patterns, also system-level design choices do influence the system analyisability in that they may complicate the analysis process or ask for more intrusive user intervention, regardless of the analysis approach.

We therefore focus on the role of good coding patterns and design in determining more analysable applications. In doing so, we differentiate between task-level and system-level code, where the former addresses the functional part of a system (i.e., the task code that is subject to timing analysis) and the latter, instead, identifies the non-functional part of a system, that defines the actual organisation of the functional one. In this respect, we particularly focus on code patterns and design choices that may positively or negatively affect task interleaving and interactions. In the following we first provide a brief overview on the related work in this topic; then we propose a taxonomy of adverse constructs based on the effects they may incur on the development of predictable systems. Finally we identify some software design choices and coding patterns either at task or system level that may complicate timing analysis and propose, when possible, alternative implementations.

3.5.1 Positioning of our work

The stringent constraints set on the industrial development process pose strong expectations on the overall quality of the software product. Accordingly, several coding rule, best practice and guidelines have been defined either local to single industrial concerns or agreed between stakeholders in a specific domain (e.g., MISRA-C [105] in the automotive domain). The idea of quality, however, lends itself to different interpretations when applied to software. The fact that the most natural interpretation of software quality overlaps with its functional correctness justifies the fact that those coding standard typically aim at defining coding rules that may reduce the probability of introducing errors in the code or ease their detection. The use of semantic pre-condition and post-condition annotations in the SPARK [17] approach are representative in this respect.

The role of software constructs in affecting both positively and negatively the timing analysability of a program as long been acknowledged. Information on unpredictable code constructs is sparsely present in the literature. The single-path approach [128] and cache-aware compiler optimisations [46] can be seen as attempts to make software more predictable instead of trying to analyse it as-is.

Nonetheless, the idea of timing analysability as a quality aspect has only recently emerged. Few attempts to formalise coding guidelines for timing analysability have been produced as partial result of several european projects. Examples of such outcomes are the identification of *Cache risk patterns* in the PEAL and COLA projects [159, 102], and the definition of coding guidelines in the MERASA project [110]. The guidelines produced by the latter study, however, are quite specific to the analysis tool adopted in the course of the project itself and include some code patterns that are typically (and transparently) removed by compiler transformations. Other studies [161, 50] instead highlight the positive effects that the adoption of standard coding guidelines may have even on timing analysis.

Our work differentiates from the previous approaches in two respects. First, we distinguish between code unpredictability that stems from task-level code patterns and system-level design choices, where the latter dimension has not been addressed by former studies. Second, we try to collect a representative set of unpredictable constructs and propose a taxonomy based on their effects on timing analysability.

3.5.2 Taxonomy of Source-level WCET Analysis Issues

WCET analysis has undergone important theoretical and technical achievements in the last decades. The maturity level achieved by both analysis approaches and support tools is

the driving force behind a growing industrial interest on structured approaches to timing analysis [137, 30, 146, 54, 84]. However, in spite of those significant theory advances and the availability of powerful tools [1, 133], analysing the WCET behaviour of real industrial-scale programs remains an extremely onerous and challenging task, even for comparatively simple hardware architectures, regardless of the used method and technique.

On the one hand, static analysis in general suffers from inherent sources of overestimation, which impair the tightness of the computed bounds. The principal difficulties come from the need to conservatively abstract the execution contexts the analysis has to consider, and the inclusion of infeasible execution paths in program flow analysis. On the other hand, measurement-based WCET computation shares a common problem with static analysis approaches: as measurements are usually first performed on basic blocks and subsequently combined, they run the risk of including infeasible paths. Moreover, since triggering all relevant hardware and input states of a program is not generally feasible, measurement-based methods cannot guarantee safe estimates.

These limitations, which seldom become apparent in laboratory experiments, are exacerbated by the complexity of industrial-scale systems. Complex industrial systems in fact are typically developed without timing predictability in mind and often include those code constructs and patterns that are known to dramatically hit on the quality of the results of timing analysis. This observation has been largely confirmed by the difficulties we encountered in our experience in trying to analyse the WCET behaviour of a satellite on-board software system [103].

Those adverse code constructs can be categorised with respect to a twofold criterion, namely the origin and the effects of their inclusion. We identify two different sources of inclusion of those adverse code constructs. In fact, software systems are typically organised into two distinct dimensions: a *task-level* dimension, that comprises the functional specification of each task, and a *system-level* one, that governs the task functionality and organises them in the time, space and communication dimensions. This conceptual separation is clearly perceptible in the partitioning between the task code and the architectural part of a system. Accordingly we observe that unpredictable code may originate either at task level, stemming from the adoption of specific code patterns, or at system level, stemming from definite design choices.

With respect to their effects, instead, WCET analysis issues at software level are mainly related to the reconstruction and analysis of the control flow of a program but may hamper the analysability of a systems under the following aspects:

1. *Feasibility*: some code constructs may impede the computation of safe WCET bounds

or estimates. This category includes those code constructs (e.g., irreducible loops) that cannot be generally handled by the analysis steps involved in both static and hybrid analysis approaches;

2. *Precision*: some constructs may degrade the quality of the results of timing analysis. Constructs that yield imprecise memory accesses or inclusion of infeasible paths will inevitably incur additional overestimation in static analysis and hinder the application of hybrid measurement-based approaches;

3. *Computational complexity*: some constructs may lead to an increase in the analysis complexity, in both space and time dimensions. Complex loop bounds, data-dependencies and unstructured code may ask for computationally intensive advanced analysis techniques such as virtual loop unrolling and interprocedural (context-dependent) analysis;

4. *Labour-intensiveness*: some constructs may ask the user for additional efforts in conducting the analysis. Beside plain avoidance, the common approach to cope with unpredictable code constructs typically consists in providing additional information, in the form of manual annotations (a.k.a. *flow-facts*). Providing trustworthy manual annotations, however, requires very profound knowledge and understanding of the program behaviour, which are both fragile (owing to incorrect beliefs) and volatile (owing to personnel mobility) assets even in high-integrity industry;

5. *Interference*: some constructs may expose to possible interference in history-dependent hardware features like caches. These constructs, which are typically introduced by system level design choices, may hamper the soundness of the analysis results (e.g., cache interference in case of blocking).

It is worth noting that this categorisations do not configure disjoint sets of constructs: the same construct may in fact affect analysability under multiple aspects. For example, constructs that yield precision loss are typically labour-intensive as they ask for more user intervention.

In the following we discuss the most relevant constructs that adverse software analysability, without any assumption on the adoption of specific tools or techniques: although some of the following code-level issues may be better handled by a specific tool and/or technique, the important point is that they may actually complicate or hinder WCET analysis. We separately address problematic program features that are introduced by task-level code patterns and system-level design choices.

3.5.2.1 Adverse code patterns

Several industrial coding guidelines [105, 17] agree on the identification of a set of ordinary code patterns that hinder the analysability of a program in general. Those patterns evidently affects also timing analysis as the latter also relies upon common analysis techniques. However, systematically avoiding those constructs, while improving the system analysability, may reduce the expressiveness of modern high level programming languages. Thus deciding whether to use such constructs often calls for a compromise between functionally complex features and the onerous user annotations required to cope with them.

In the following we review a set of code patterns with respect to timing analysability. We categorise each construct with respect to the proposed taxonomy. As a preliminary remark, we observe that the role of compilers cannot be disregarded as analysis is typically performed at object level and the mapping from source code to object code may introduce (as well as remove) unpredictable code constructs. Therefore, for the sake of simplicity and in line with industrial practice, we assume that no aggressive compiler optimisation is performed, except for some optimisation steps that are known to have positive effects on code analysability such as loop unswitching, splitting, peeling, etc.

P 1 - Recursion

Recursion (either direct or indirect) is considered to be a harmful construct in high integrity system as the maximum depth of the recursion chain and thus the incurred time and memory consumption are hardly predictable. This is even more critical in embedded HIRTS, where the maximum stack consumption is a critical aspect that is thoroughly addressed in system verification. Besides the fact that recursive calls are generally difficult to bound by simple code inspection, they may also be translated by the compiler into irreducible loops (cf. pattern P 2).

Whenever possible, the best way to handle recursion is simply to avoid it at all. Some simple form of recursion (so-called tail recursion) can be rewritten into a simpler loop structure [104]. Figure 3.5 below suggests a how this transformation can be applied to a simple example function that uses tail recursion to compute the n^{th} integer in the Fibonacci's series.

In presence of more complex recursion schemes, limited support to recursive functions may be provided through manual annotations or bothersome workarounds. The first solution consists in asserting the maximum recursion depth, at the risk of defining a wrong bound or, at least, an overly conservative one. This feature is supported, for example, by

```
1  function fib(n : integer)
2           return Integer is
3  begin
4    if n < 2 then
5      return n;
6    else
7      return fib(n-1) + fib(n-2);
8    end if;
9  end fib;
```

```
1  function fib(n : integer)
2           return Integer is
3    first  : Integer := 0;
4    second : Integer := 1;
5    tmp    : Integer;
6  begin
7    for i in 1..n loop
8      tmp    := first + second;
9      first  := second;
10     second := tmp;
11   end loop;
12   return first;
13 end fib;
```

Figure 3.5: Tail recursion transformation.

the aiT static analysis tool [1]. A possible workaround, instead, applies to indirect recursion (e.g., where function `A` calls `B` which in turn calls `A`) consists in slicing the recursion and analysing separately the involved functions. This approach, however, assumes that the recursion chain WCET is given by somehow summing up the WCETs of each function, which does not hold true, for example, in the presence of caches.

P 2 - Irreducible loops

Some programs may exhibit non structured loops with multiple entry points. These particular constructs, often referred to as *irreducible loops* [4], impede the reconstruction of the program control flow. Figure 3.6 reports a canonical example of irreducible flow-graph where the loop involving $\texttt{BB}_3$ and $\texttt{BB}_4$ is either entered from $\texttt{BB}_3$ or $\texttt{BB}_4$. These kind of constructs are more easily found in binary code, as a result of aggressive compiler optimisations. However, they can be more or less intentionally introduced by low-level assembly code or weird use of the `goto` statement.

The consequences of having unstructured loops in a program are at least twofold. First, in the presence of reducible loops the CFG reconstruction process cannot exploit *dominator analysis* to detect loops. Second, irreducible loops complicates data-flow analysis as they prevent the application of interval-based analysis and cause the use of much more complex iterative approaches.

Interestingly, the aiT tool allows to define specific annotations on flow relations between basic block to attenuate the problem of irreducible loops. Since defining this kind of annotation is extremely complex, the first countermeasure still consists in preventing them. The first root cause of the introduction on irreducible loops is the activation of complex

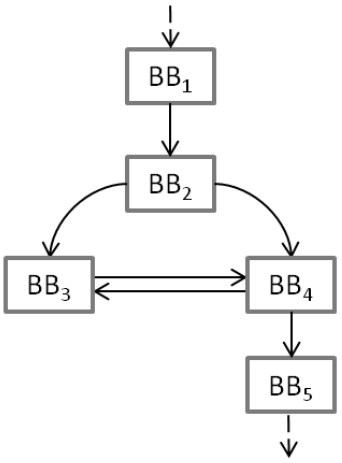


Figure 3.6: Example of irreducible loop.

compiler optimisations: in principle, these should be already avoided in HIRTS as they largely break the correspondence between source code and object code, thus complicating software verification in general. Improper use of the `goto` statement are typically intercepted by syntax and semantic checks performed by the compiler. However a semantic transformation from a `goto` formulation to a canonical branch structure is recommended. At the same time, the low-level assembly code typically used for low-level hardware support, such as board support packages (BSP), or other low-level libraries (e.g., `divide` procedure) should be carefully inspected to verify that they do not contain unstructured loop regions.

P 3 - Dynamic allocation

Dynamic memory allocation of data structures is commonly discouraged in HIRTS. The effects of memory allocation on cache analysis are simply devastating for both static analysis and measurement-based methods. From the static analysis perspective, dynamic allocation actually hides information on the affected memory locations and thus causes massive loss of information. Recalling the classification of memory accesses in cache static analysis (i.e., always hit, always miss, first miss, unknown), every access to memory allocated data are forcibly classified as unknown. When it comes to measurements, the cache behaviour may incur a variable amount of cache conflicts, depending on the cache lines the allocated memory area is mapped to.

The precision loss entailed by dynamically allocated data is hardly remedied by user

annotations. Recent studies [61] propose the adoption of cache-aware memory allocators as a work-around to static analysis. The modified allocator enforces allocated data to statically predefined addresses, thus avoiding unpredictability. In HIRTS, however, dynamic memory allocation has been historically discouraged for the sake of deterministic memory usage.

P 4 - Data dynamic references

Besides dynamic memory allocation issues, static analysis is hampered by the use of dynamic references (i.e., pointers) as the actual referenced address cannot be always statically resolved. the fact that the same instruction may access different memory addresses, depending on the actual state of the data structure, eventually leads to the computation of imprecise memory accesses, if any at all.

From the cache perspective, since the dynamic reference determines which cache sets the referenced data will be loaded or written to, imprecise memory accesses lead to a loss of information on the cache state. This information may be difficultly reconstructed by user annotations stating the affected range of memory addresses.

WCET analysis in the presence of pointers is further complicated by pointer aliasing and complex pointer-based data structures. Classical examples of these data structures are linked lists or complex tree-like aggregates used to navigate large and dynamic data structures with the aggravating circumstance of exploiting dynamic allocation (cf. pattern P 3).

P 5 - Function pointers

Similarly to data pointers, dynamic referenced functions are hard to tackle by static analysis. Indirect calls need to be resolved in order to reconstruct the program control flow. Although some simple occurrences can be automatically resolved by state-of-the-art static analysis tools by a pattern-oriented analysis approach (e.g., when function pointers are defined in a static array), user annotations are generally required.

However, multiple indirect call targets, though defined by user annotations, are still a source of overestimation as a conservative timing analysis is forced to always select the function (from a set of candidate targets) that leads to a supposed worst-case path, which may also be actually infeasible in a specific calling context. Besides the precision loss, considering multiple alternative call chains also introduces additional context-dependent information that negatively affects the analysis complexity in both time and space.

The use of dynamic calls in a program may be necessary, for example, to handle data-

dependent contingencies. Interestingly, however, some indirect calls are introduced to satisfy architectural constraints rather than pure algorithmic requirements. This is the case, for example, of function that are declared as pointers in the functional interface of a module while they are statically defined in the module implementation. The simple example in Figure 3.7 resembles an architectural false dynamic call. The function pointer `My_Handler` actually targets the statically defined procedure implementation `My_Handler_Impl`. This kind of indirect call, once identified, can be precisely resolved with a relatively simple flow-fact annotation.

```
1  -- Type definition
2  type Handler_Ptr is access
3    procedure (x : A_Type);
4
5  -- Interface definition
6  My_Handler : Handler_Ptr := null;
7
8  procedure Initialize;
```

```
1  -- Implementation
2  procedure My_Handler_Impl (x: A_Type) is
3  begin
4     -- do something
5  end My_Hanlder_Impl;
6
7  procedure Initialize is
8  begin
9     My_Handler := My_Handler_Impl'access;
10 end Initialize;
```

Figure 3.7: False dynamic call.

P 6 - Multi-way branches

The multi-way branch construct (i.e., the `switch` and `case` statements in C and Ada respectively) is typically used to control the execution flow when several alternate actions should be executed depending on the value of a relatively simple variable or expression. This is a recurrent construct in embedded control systems where the performed action highly depends on the input stimuli. Besides the source-level complexity that may arise from non exclusive branches, the analysability of this construct may be further complicated by compiler intervention.

The multi-way construct is mapped by the GCC compiler (and GNAT) into three different object-code constructs: (i) a heuristically balanced branch tree; (ii) a statically defined external jump table; or even (iii) a set of bitwise-computed offsets internal to the procedure body. The compiler decision on which object-code pattern has to be applied is transparent to the developer as it is based on both the number of branch alternatives and the distributions of the possible values in the case variable (or expression) range. Figure 3.8 shows the bitwise implementation of a multi-way branch.

```
1  procedure Multi_Cond_5
2       (c   : in X_Type;
3        res: in out Natural) is
4   begin
5     case c is
6       when CASE_0 => res := 0;
7       when CASE_1 => res := 1;
8       when CASE_2 => res := 2;
9       when CASE_3 => res := 3;
10      when CASE_4 => res := 4;
11      when others => res := 100;
12    end case;
13 end Multi_Cond_5;
```

```
1  <multi_cond_5>:
2    and  %o0, 0xff, %g1
3    cmp  %g1, 4
4    bgu  <return>
5    mov  0x64, %o0
6    sll  %g1, 2, %g1
7    sethi %hi(base_addr), %g2
8    or   %g2, 0x68, %g2
9    ld   [ %g2 + %g1 ], %g1
10   jmp  %g1 !!dynamic jump
11   nop
12   ...
13   <local jump table>
14   ...
15   retl <return>
16   nop
```

Figure 3.8: A mutli-way branch implemented with bitwise offsets (SPARC).

The `expand_case` procedure in `stmt.c` (GCC internals) is responsible for the object code generation for a switch/case statement. The compiler first tries to implement the case statement as a short sequence of bit-wise comparisons: this is feasible only when the set of case alternatives can be selected using a bit-wise comparison. Whenever the bit-wise implementation is not feasible, the compiler may decide to generate either a balanced branch tree or a table jump, based on a predefined value threshold (currently 5 in GCC 4.5) and the ratio between allowed range and actual number of distinct values of the conditional variable or expression. In fact, a jump table implementation implies the definition of a branch target for each possible value in the variable range: if the variable can assume just few values in its range then a jump table would waste a huge amount of memory space.

This construct, especially in its nested and unstructured forms, should be moderately used as it increases the number of paths in the program to be accounted for in flow-analysis and may introduce infeasible paths that are hard to detect. Moreover, in case of bit-wise offsets and external jump tables, the effects on code analysability are comparable to those of indirect calls (cf. pattern P 5). The user is forced to manually inspect the code and provide flow-fact annotations for each branch target. Leveraging on compiler-specific patterns may allow the automatic detection of branch targets: aiT, for example, seems to be able to automatically analyse patterns (i) and partially (ii). However, additional information on conditional execution should be provided to prevent the inclusion of infeasible paths in the analysis.

P 7 - Data-dependent loop bounds

Input-data dependency is the major obstacle to automatic WCET analysis as it generally complicates path analysis. In particular, analysing those loops whose number of iterations is correlated to an input parameter is generally infeasible. Even in case analysis is able to precisely account for the input parameter, the obtained bound is valid solely for the specific combination of the input data and calling context. Therefore, every possible such combination should be accounted for each time the loop is executed, which seriously affects the complexity of the analysis process (e.g., in case of multiple nested loops).

Avoiding input-data dependent loops altogether may not be a reasonable option in practice. In some simple cases it may be worth removing the input-data dependency by transforming a data-dependent loop into a (constant) counter-based one, with limited consequences on the average-case performance. Whenever loop transformations cannot be applied, the only solution is to manually define sound and precise flow facts, which is not always a trivial task.

P 8 - Template and generic

Some high level programming languages provide a generic mechanism that allows to define software modules that are parametric on some discriminating feature, typically data types. That mechanism is realised for example by the `generic` package construct in Ada and function or class `templates` in C++. The use of that constructs may incur additional data-dependency in the program since, depending on the specific compiler in use, distinct instantiations of the same package/template may partially share object code. An example is the implementation of distinct lists whose size depends on the type of the elements they should contain.

The use of generic package instantiation does not only compromise the tightness of analysis results but may seriously affect its complexity, as it increases the number of execution contexts that have to be accounted for. Their use is thus conditioned upon verification that the compiler does not allow code to be shared between different instantiations.

P 9 - Shared data structures

It is not rare for statically defined shared structures, like lists or buffers, to be shared between different procedures or even different tasks in a system. The size of such data structure is statically determined to accommodate a certain maximal amount of data and is perfectly accountable by static analysis.

However, whenever such structure is traversed in a loop (to access the contained data), the number of iterations performed at run-time may actually depend on the number of elements in the structure which may be considerably less than then the size of the structure itself. Reducing the incurred overestimation by providing useful manual annotations has to cope with the fact that this kind of information originates from system-wide properties that are not detectable through simple code inspection.

P 10 - User-defined data types

Programming languages typically allow the user to extend the type system by creating user-defined types. The use of user-defined data types and ranges as loop iterators allows to perform run-time checks (e.g., in Ada) but, at the same time, may complicate the automatic detection of loop bounds. This is because the information on the data type is not generally exposed by the compiler, unless range checks are enabled.

The left side of Figure 3.9 below reports the definition of a simple 8 bits unsigned integer and its usage as arithmetic range for a loop. The type definition can be reformulated so that it is more explicitly related to the basic Integer type, which allows for the automatic detection of the loop bound (e.g., in aiT).

```
1  type UINT8 is mod 2**8;
2  for I in UNIT8 loop
3   -- do something
4  end loop;
```

```
1  type UINT8 is range 0..255;
2  for I in UNIT8 loop
3   -- do something
4  end loop;
```

Figure 3.9: Loop over user defined-data range.

P 11 - Array copy

Some object code generation strategies implemented by the compiler may lead to overly complex object code constructs. This is the case, for example, when the compiler reorganises the control flow of a program, or introduces loops and branches that were not defined in the source code, thus breaking the traceability of object code back to the originating source code.

Array slice assignment is a clear example of this kind of code generation pattern. For array slice assignment we refer to an assignment that involves two parts of different arrays.

In Ada, this corresponds to an assignment in the form `A(X..X+n):=B(Y..Y+n)` which is mapped by GCC into different object code constructs: either a simple call to `memcpy` or a loop over the source and target arrays. In the latter case, when the compiler cannot know for sure if the source and target memory addresses overlap, two loops are implemented. In fact, to ensure a semantically correct array copy the source array slice must be copied from its starting address to the end or the other way around, depending on the address overlapping.

Those code generation patterns can complicate static analysis in that, when not automatically detected, they ask for a deeper understanding of the program at object code level as a prerequisite to providing flow-fact annotations.

P 12 - Variable-size data structures

The only concession to dynamicity in HIRTS is perhaps the declaration of dynamic-sized data structures, which are typically allowed inside functions. An example of such structure is the declaration of an array, whose size is dynamically determined and correlated to a function parameter. These data structure are allocated on the function stack. Typically, the dynamically-initialised data structure will be later involved in a data dependent loop, whose dependence on data is subtly different from that observed for pattern P 7.

```
1  type Foo_Array is array (Positive range <>) of Foo_Type;
2
3  procedure Foo (struct: ComplexStructure; item: Item) is
4    temp_arr : Foo_Array (1..struct.size);
5    begin
6      for I in temp_arr'Range loop
7        -- do something
8      end loop;
9  end Foo;
```

Figure 3.10: Dynamically initialised array.

Procedure `Foo` in Figure 3.10 contains a loop that iterates over the size of a dynamically initialised array. In this case the loop bounds is directly related to a `size` attribute of the allocated data structure, but in general it may depend on many other factors. The effects of this code pattern are similar to that of construct P 7, as analysis may fail to account for the actual size and thus yield degraded bounds.

P 13 - Logically-controlled vs counter-based loops

Simple counter-based loops are generally easier to bound than logically-controlled loop, where the exit condition does not depend on a constantly incremented iteration variable but depends on a more or less complex condition [63, 41]. The introduction of additional `exit` conditions in a loop (e.g., for the sake of program optimisation) makes it even more difficult to analyse the exact loop behaviour.

The analysability of a loop, however, essentially depends on the analysis technique in use: a loop that cannot be automatically bounded by a tool could be analysable by another. Earlier syntactic-based approaches were certainly much less powerful than more recent techniques, which are typically based on a combination of pattern-matching, data-flow, invariants and semantic-based analysis (e.g., interval-based abstract interpretation).

However, in principle, a method capable of automatically analyse all kind of loops cannot exist. Moreover, it is worth noting that the actual implementation of a loop is determined by the compiler and it is not rare for the compiler to transparently generate such additional exit conditions. Similar effects can be determined by the use of additional, deeply nested `return` statements in a function.

P 14 - Cache-risk patterns

These code patterns, which were originally defined in the scope of the PEAL project [159, 100], specifically address the I-cache behaviour. The main focus is set on the avoidance of potential source of uncontrolled cache jitters, in particular related to the combined effects of code size end conditional execution. An example of such patterns consists in the definition of a loop that under certain conditions may invoke a number of procedures (greater than the I-cache associativity) that overlap in cache, as shown in Figure 3.11. Therefore under LRU replacement policy, such loop may generally exhibit good cache performance except for those rare cases when the condition is satisfied and the execution incurs overly poor cache performance. In this case, in fact, procedures `A`,`B`,`C`,`D` and `E` will evict one another repeatedly, thus incurring an extremely large amount of conflict misses.

These patterns do not address any specific unpredictable construct but rather highlight some cases of extreme cache variability. They may complicate the application of measurement-based approaches as the rare condition may never be observed during measurements. However, even with respect to static analysis, if the actual value of the rare condition is difficult to analyse then analysis may conservatively assume that it is always satisfied, thus incurring a large overestimation. In Section 3.7 we will introduce in a mem-

```
1  for I in 1..N loop
2    call to A;
3    call to B;
4    call to C;
5    -- Rarely triggered condition
6    if rareCond then
7      call to D;
8    end if;
9    call to E;
10 end loop;
```

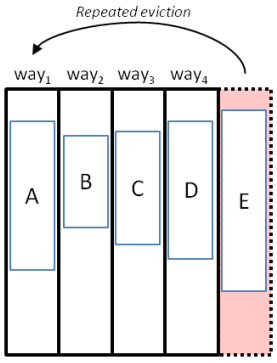


Figure 3.11: Cache-risk pattern under LRU.

ory layout optimisation technique aimed at removing this source of variability.

P 15 - Unstructured patterns

Control flow and data accesses that follow complex unstructured patterns largely complicate timing analysis in many respects. With respect control flow, every twisted combination of loops and branches (either simple or multi-way) is detrimental to timing analysis as it is likely to introduce hardly detectable infeasible paths and yield an increase in complexity.

Similarly, when it comes to data accesses, non-trivial access patterns are hard to analyse with sufficient precision and reasonable complexity. A classification of data structures and access patterns from the standpoint of D-cache predictability is provided in [91]. The effect of non-contiguous and variable-stride data accesses combined with data-dependency leads to the computation of largely imprecise memory accesses. Bounding these detrimental effects thorough manual annotations is extremely onerous and complex. It is worth noting, however, that most modern compiler optimisations address these adverse patterns in common average-case optimisation.

Summary of task-level constructs

The following Table 3.3 summarises the effects of the addressed code constructs and patterns on timing analysis. Each pattern is categorised against our proposed taxonomy by placing a $\times$ or - symbol to respectively express whether a specific pattern falls or not into the identified categories: *Feasibility*, *Precision*, *Complexity*, *Labour-intensiveness* and *Interference*. It is worth to note that the no task-level pattern actually belongs to the latter category as inter-task interferences naturally generate from system-level design choices.

ID	Pattern	F	P	C	L	I
P1	Recursion	×	×	-	×	-
P2	Irreducible loops	×	-	×	×	-
P3	Dynamic allocation	×	×	-	-	-
P4	Data dynamic references	-	×	×	×	-
P5	Function pointers	-	×	×	×	-
P6	Multi-way branches	-	×	×	×	-
P7	Data-dependent loop bounds	-	×	×	×	-
P8	Template and generic	-	×	×	×	-
P 9	Shared data structures	-	×	-	-	-
P10	User-defined data types	-	×	-	×	-
P11	Array copy	-	×	-	×	-
P12	Variable-size data structures	-	×	-	×	-
P13	Logically-controlled vs counter-based loops	-	×	-	×	-
P14	Cache-risk patterns	-	×	-	-	-
P15	Unstructured patterns	-	×	×	×	-

F = Feasibility; P = Precision; C = Complexity;
L = Labour-intensiveness; I = Interference.

Table 3.3: Categorisation of task-level patterns.

3.5.2.2 Adverse system-level design choices

Focusing exclusively on the analysability of task-level code patterns does not suffice to guarantee the system predictability. System-level design choices, and their realisation in a specific software architecture, actually governs task interleaving, interactions and communication in a system. These choices are relevant not only from the schedulability analysis standpoint but also from the WCET analysis one, as long as execution-history dependent hardware features, like caches, are adopted.

Hence, system-level design choices may be even more critical than the task-level code patterns reviewed in Section 3.5.2.1. This is because the code constructs induced by system design choices may also invalidate any effort made to guarantee software predictability at task level. For similar reasons, the compliance to coding guidelines or standards that specifically address system-level concerns may also ease timing analysis. The set of rules defined by Ravenscar profile [27], for example, although explicitly devised to enable schedulability analysis, have positive effects also on timing analysis. All the following design choices fall into the *Interference* category, as defined in Section 3.5.2.

D 1 - Impact of RTOS

System level considerations are typically kept separated from WCET analysis as a major assumption of the latter is that tasks execute in isolation. Residual inter-task interference, for example on caches, are delegated to advanced schedulability analysis techniques (e.g., [76, 147, 8].

As foreseen in [142], however, the WCET of a task may not be completely unrelated to the underlying real-time operating system (RTOS). The influence of the RTOS on the task timing behaviour is not generally limited to scheduling primitives. Despite the progresses made in calculating the context-switch cost, complex real-life scenario are actually difficult to handle. The effects of common constructs, like self-suspension as well as the firing of timer alarms and watchdogs on history-sensitive hardware (e.g., caches) are difficult to apportion to intra- or inter-task analysis, either of which must be very sophisticated to handle them all satisfactorily.

In this setting there is no clear separation between the task code and the kernel code, which is also typically much more difficult to analyse. Kernel primitives are often implemented in small chunks of assembly code for which flow facts annotations may result to be overly complex. More importantly, this undermines the assumption on tasks running in isolation.

The principle of *separation of concerns* [39], which predicates on the separation of a program into distinct cohesive features that should not overlap, may help. Separation of concern in this context means that the functional behaviour of a system (and thus of its tasks) to be kept logically and physically separate from the implementation of the non-functional specification. We conducted some preliminary experiments on the positive effects of separation on both the cache variability and the degree of overestimation incurred by mingling RTOS code to application code [101]. The obtained results confirmed that an increase in both cache variability and overestimation is incurred when no separation is enforced.

D 2 - Task communication and synchronisation

The way tasks are allowed to communicate and synchronise with each other has important effects on the inter-task interferences in a system. The implementation of strict synchronisation between tasks, for example, forces tasks to wait one another thus making it more complicate to account for inter-task interferences.

Consider, for example, a task that needs to acquire some data from the environment.

It may send data requests to a serialised communication channel, set a time-out timer and then self-suspend, waiting for the required data arriving over the same channel. In a preemptive system, since the task suspends itself, some other task with either higher or lower priority may execute and this incurs a serious form of inter-task interference on the cache. A safer design choice would then consist in implementing the same functionality with two separate tasks, where the second one is triggered once data are available or the time-out timer has expired.

With respect to the Ada language, powerful synchronisation constructs such as task `entry` are explicitly forbidden by the Ravenscar profile [27] to preserve time-deterministic execution. Task synchronisation and communication is allowed exclusively via protected object, whose effects will be considered in the next point.

D 3 - Effects of resource sharing

As observed in point D 1, with the adoption of caches the context-switch cost is no longer constant as it must account for the interferences between tasks: interrupt handling and preemption may influence the execution time of a preempted task. On resumption, in fact, the preempted task may incur a number of additional cache misses as some useful cache content may have been evicted by the preempting task. The inclusion of those effects are accounted for separately in advanced schedulability analysis by accounting for an upper bound on the so-called Cache-Related Preemption Delay (CRPD) in the response time of individual tasks [76, 28, 6].

However, interference caused by task interleaving and interactions is not limited to task preemptions. As observed in D 2, the assumption of task independence rarely holds in practice and real systems often include shared resources that can be accessed by multiple tasks in mutual exclusion. When a high priority task needs to access a resource that is already locked by a lower priority task, it cannot proceed until the lower priority task completes execution inside the resource and relinquishes its lock.

Therefore, from the standpoint of caches, task blocking may cause effects similar in kind to task preemption, in that some useful code or data blocks already loaded in the cache may be evicted while the task is being blocked. Very few works [132] consider the additional time spent in reloading the evicted cache contents, which is referred to as Cache-Related Blocking Delay (CRBD).

Whenever a lower priority task prevents the execution of a higher priority task, the system experiences potentially unbounded priority inversion. This phenomenon can be bounded with the use of a resource access protocol [144]. In a fixed-priority preemptive

system with shared resources equipped with a synchronisation protocol, three different kinds of blocking may arise [87]:

- *Direct blocking* occurs when a higher priority task requests a shared resource held by a lower priority task; another form of direct blocking, *transitive* or *chain blocking*, occurs when nested resources access is permitted, and a blocked task transitively suffers from the blocking incurred by the blocking task itself.
- *Inheritance or push-through blocking* occurs for a task τ_m that does not need any shared resource, when a lower priority task τ_j blocks a task τ_i with priority $\pi(\tau_i) > \pi(\tau_m)$ and executes at a priority higher than $\pi(\tau_m)$ due to some priority inheritance rule.
- *Avoidance blocking* occurs when a task τ_i is denied access to an *available* resource to prevent deadlock.

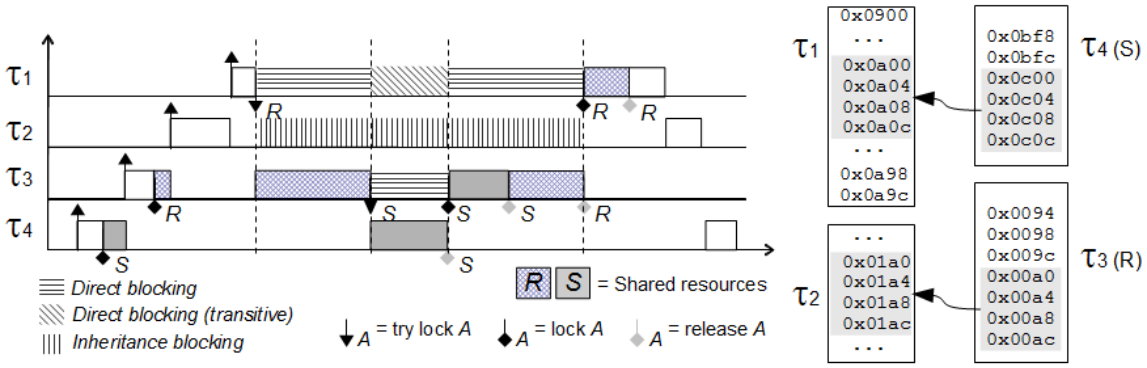


Figure 3.12: Different kinds of blocking.

The scenario depicted in Figure 3.12 illustrates the different types of blocking under the Priority Inheritance Protocol [144] and shows how lower priority tasks may affect the cache content of higher priority tasks. In particular, task τ_1 and τ_3 suffer direct blocking, while task τ_2 suffers inheritance blocking. Assume that τ_1 has loaded four cache blocks that would be shortly reused (i.e., the shaded memory addresses in Figure 3.12) in a small direct mapped instruction cache. Unfortunately, task τ_1 is blocked when trying to access shared resource R currently held by τ_3, which in turns is blocked by τ_4 on the shared resource S. While τ_3 has no effect on the useful cache content of τ_1, the code executed by τ_4 in its critical section accessing S maps exactly to the same cache sets and evicts all the four useful blocks of τ_1. When τ_1 resumes, it will incur four additional cache misses.

A subtler penalty is experienced by task τ_2 due to the execution of τ_3: whereas it does not share any resource with other tasks, τ_2 is blocked due to priority inheritance. In the example, the useful cache content of τ_2 is evicted during the execution of τ_3 inside its critical region. It is worth noting that τ_2 would not have suffered any interference (CRPD) due to the higher priority task τ_1.

We observe that different patterns and durations of blocking and thus the amplitude of the CRBD can be induced by the specific resource access policy in use. It is therefore important to understand which resource access policy may introduce less interference and thus facilitate the system analysability.

Exploiting the similarity between CRPD and CRBD, we build on the same concepts of *Useful Cache Blocks* (UCB) and *Used Cache Blocks* ($\overline{\text{UCB}}$) (cf. Section 2.3.2), for blocked and blocking task respectively, and sound theoretical bounds [144] on the number of blocking events to compute a safe bound on the CRBD suffered from each task under three well-known protocols: the Priority Inheritance Protocol [144], the Priority Ceiling Protocol [144], and the Immediate Ceiling Priority Protocol, a derivative of Baker's stack resource policy [14].

In our approach we assume that no timing anomalies (cf. Section 2.3.1) can ever occur in the analysed system. In fact a separate analysis of the number of additional misses due to preemption and/or blocking cannot yield safe results in the presence of timing anomalies. In addition, it has been observed that the notions of useful and used cache blocks cannot be safely used in the computation of the CRPD (and thus the CRBD) under FIFO and LRU cache replacement policies [26].

Under those premises, the obtained bound β_i is a safe bound and can be straightforwardly included in the iterative equation of response time analysis:

$$w_i^{n+1} = C_i + B_i + \beta_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil \times (C_j + \gamma_j) \tag{3.1}$$

where both B_i and $CRBD_i$ depend on the resource access protocol of choice, and γ_j represent the cache interference from preemption (CRPD).

We omit to detail here the computation of those bound as it would need a fair amount of notation. However, a detailed explanation on how to compute CRBD bounds is reported in Appendix A.

The important conclusion we draw from analysing the CRBD under those protocols is that the Immediate Ceiling Priority Protocol does not incur any CRBD interference and, thus, represents the most natural choice when variability in the cache behaviour is the main

concern.

3.5.3 Summary

Whereas the adoption of predictable hardware platforms is a prerequisite for the definition of predictable HIRTS, it does not generally guarantee the timing analysability of a program running on top of it. The application of timing analysis approaches is known to be hampered by the presence of adverse code constructs and patterns that can degrade the quality of the analysis results to the extent of possibly impeding its application. As a matter of fact, complex industrial systems, even in the HIRTS domain, often include those code constructs and patterns, as systems are typically developed without timing predictability in mind.

In this section we singled out a number of code patterns that hinder the overall analysability of a system. We proposed a twofold classification of adverse code constructs based on the design space they originate from (either task-level or system-level) and the effects of their inclusion on the application and the results of timing analysis. Based on that categorisation, we surveyed the most common constructs and design choices that complicates the analysability of a system. In our mind, a categorisation of poorly analysable constructs would make developer aware of the effects of coding constructs on timing predictability, suggest their avoidance or identify valuable alternatives. Moreover, that same information enables the use automated code generation as a means to enforce predictable code patterns and coding styles that guarantee code analysability by construction, as we will discuss in the next chapter.

3.6 Code Generation for Timing Analysis

As discusses in Section 3.5, the application of state-of-the-art timing analysis approaches to real complex system is hampered by the presence of adverse code constructs and the implementation of inattentive design choices. Hence the user is often required to provide manual annotations to assert infeasible paths, define branch and indirect call targets, refine loop bounds and support path analysis in general. Providing such annotations, is not a trivial issue for they require a deep understanding of program behaviour and are both onerous and error prone.

The identification of proper coding styles and constructs may facilitate the timing analysis of industrial systems, in the wake of what has been historically done with respect to

their functional correctness, readability and maintainability. Despite the undeniable value of such guidelines, however, the timing predictability dimension is far apart from the common average-case oriented programming practice. The transition to a WCET-aware coding in a consolidated development infrastructure requires an onerous and long-term effort for educating developers to commit to a completely new approach.

The increasing penetration of the Model-Driven Engineering (MDE) [141] approach even in HIRTS opens some interesting avenue to develop software products that are amenable to timing analysis. The MDE paradigm builds on the definition of domain specific abstractions (or models) which can automatically undergo a series of model-to-model transformations [98], up to the generation of the application code. Both the model definition and the code generation steps of the MDE approach offer an ideal ground for promising solutions to guarantee code *analysability by construction*.

In fact, the transformation from model to application code intuitively offers an ideal ground for promoting programming constructs that are amenable to timing analysis as these can be transparently injected into the transformation process itself. Even more interesting solutions can be devised at model level, which is not only the natural place for the enforcement of specific design choices, but also allows to collect valuable flow information (e.g.: on dynamic calls and loop bounds) that can be automatically translated into flow fact annotations.

The benefits that can be drawn from the application of the MDE approach encompass, tough at different degree, the design and production of both the functional (algorithmic) specification or that of the overall system (architectural) specification. The open-source GeneAuto/Ada framework [51] and the Space Component Model Editor [116] were selected a the practical evaluation of our approach in both contexts.

In this section we first introduce the main concepts on the MDE approach and its application to HIRTS. We then outline a threefold approach that leverages on the MDE paradigm to enforce software analysability by construction. Finally we provide an example application of our proposed approaches as part of the selected tool-chains.

3.6.1 Positioning of our work

As a matter of fact, industrial-quality HIRTS routinely integrate code generated from different modeling tools together with manual code. Automated code-generation facilities, which follow the model-driven paradigm, are increasingly adopted in the industrial domain as a means to reduce costs and complexity in software design and implementation. Such adoption has been favoured by the extensive availability of high-quality tool sup-

port for system design and simulation, as well as actual code generation. SCADE [42], from Esterel technologies, ASCET [43] from ETAS, and Mathworks Simulink and State-Flow [96, 97] are representative examples of industrial-level modelling tools. Accordingly, model-driven approaches also arose the interest of the WCET community in the possible interaction and interoperability between such tools and state-of-the-art WCET analysis tools. The integration of the aiT WCET analysis tool with SCADE, Simulink and Statflow has been addressed in [47, 151]. The Simulink environment has been extended in [72] to accommodate WCET annotations on the model. An interesting approach to enable automatic detection of infeasible paths for WCET analysis of Esterel specifications has been introduced in [69]. The proposed techniques, however, exploit the highly favourable peculiarities of synchronous specification languages and Esterel programs in particular (e.g., basic blocks executed at most once) and cannot be straightforwardly applied to other modeling and code generation frameworks.

Our approach is similar to those presented in [151] and (partially) [72] as we collect flow fact information from the abstract model. However, we differ from previous approaches in several respects: first, we separately address modelling and code generation for both algorithmic and system-level code, whereas other approaches address mainly the functional part of a systems; and, second, we directly operate on the model-to-code transformations to enforce predictable constructs. The latter aspect is much more similar in intent to the set of modelling style guidelines defined by the Mathworks Automotive Advisory Board (MAAB) [95], which restrain the expressiveness of Simulink and Stateflow models to abide by the safety standard requirements. Since we wanted full control over all model transformations we selected two open source tools (GeneAuto/Ada and the Space Component Model Editor) for a practical demonstration of our approach.

3.6.2 Model-driven engineering in HIRTS

Model-Driven Engineering [141] is a development approach that promotes the systematic use of models as the primary artifact through the development cycle. When applied to software, MDE fosters the adoption of software models to express domain specific concepts while abstracting away from implementation details. Product development is seen as a series of *model-to-model* transformations that proceed from a platform independent model (PIM), which specifies the system behaviour in terms of both its functional and non-functional properties but still independent of any implementation technology (programming language, execution platform, middleware, etc.), to possibly multiple platform specific models (PSM), which instead provide the necessary implementation details. The

implementation details of the PSM allow for the early application of advanced simulation and analysis techniques on an abstract representation of the software, and thus ahead of production. Code synthesis in this setting can be obtained through automatic transformation from a validated PSM to a the specific programming language (Figure 3.13).

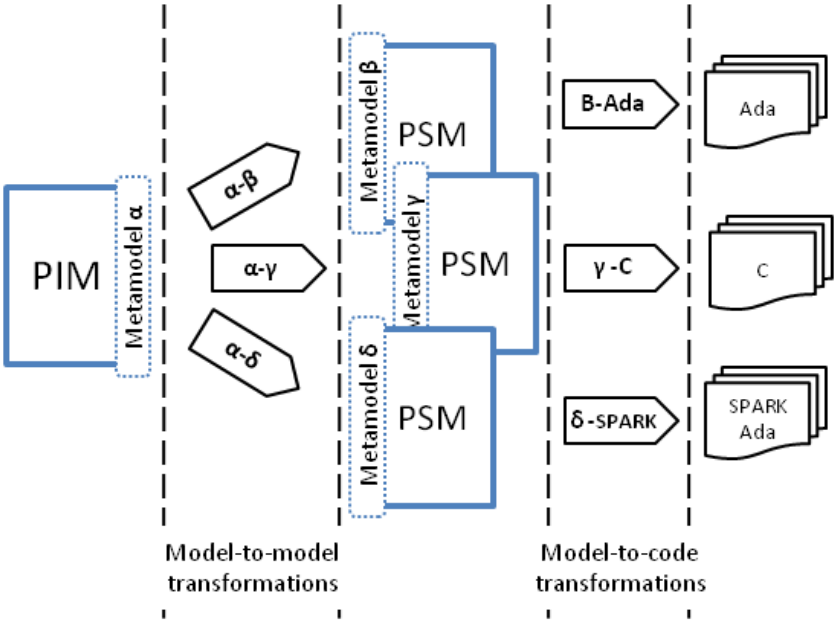


Figure 3.13: PIM, PSM and model transformations.

The application of the MDE paradigm requires the characterisation of a domain-specific modelling languages through *metamodel* definitions, which describe the entities allowed in a model, as well as their syntactic and semantic specification. Multiple intermediate models and transformations can occur along the development process,

The use of abstract descriptive models is a consolidated approach to cope with the complexity of software systems. A score of Computer-Aided Software Engineering (CASE) tools have been developed starting from the late 80ies to support software modeling and code synthesis. These tools incarnate the same principle as those of MDE with the only limitation that models and transformations are proprietary to the tool vendor and poorly extendable and customisable.

Several modeling tools are increasingly adopted in HIRTS to model, simulate and synthesise complex algorithms [96, 42, 97]. The industrial penetration of these tools is far

consolidated. Relatively more recent, instead, is the application of the same approach in combination with the Component-based Software Engineering (CBSE) approach [150] to model the overall system organisation through a domain-specific *component model*. This model addresses the organisation and cooperation of functional (algorithmic) components for the realisation of the overall system functionality. The CORBA Component Model [113] is a consolidated standard maintained by OMG. The definition of a component model for specific use in HIRTS is the main expected outcome of the ARTEMIS CHESS project [36].

Although algorithm specification and component model address different dimensions, they are equally interesting from our perspective as they both offer promising means to ease the application of timing analysis and improve its results.

3.6.2.1 Leverage points to favour timing analysis

Our main goal consists in improving the analysability of the code, which is automatically generated as final outcome of MDE approaches. We maintain that an improved software analysability can be effectively achieved along two complimentary directions: by focusing on the code that can be actually generated within the modeling framework, and by collecting valuable timing information along the various model transformations. The role of timing information is in fact as critical as the actual code constructs from the standpoint of timing analysis. WCET analysis often asks for the user to stipulate flow facts, in the form of manual annotations, to assist the analysis process (e.g., on control flow analysis, loop bounds, indirect calls, etc.) or just to shed pessimism off analysis results. Unfortunately, the assumption that the user can be a trusted source of flow facts is extremely fragile for large, complex and long-lived industrial programs.

Both code characteristics and relevant timing information are addressed in what we identified as the main leverage points for the enforcement of more analysable code: restrictions to the metamodel, definition and automated extraction of timing information, and exploitation of model transformations.

These aspects point out all the distinctive features of the MDE approach:

- *metamodel*: as the easiest way to enforce predictable code is to inhibit or restrain the use of those model elements that, according to transformation rules, yield unpredictable constructs;

- *model*: since some kind of timing information may be more easily defined at model level rather than on the source and object code, the model should be augmented so

that to accommodate user defined flow facts;

- *transformation*: as model transformations and in particular the final model-to-code transformation, are the most promising MDE steps where timing information can be automatically extracted and alternative code constructs can be enforced.

The identified approaches can be more or less effective whether applied to functional or architectural code due to the characteristics of the two dimensions. For example, we expect algorithmic code to be much more loop-intensive and the architectural code to exhibit more indirect calls. In the following we separately detail on each of the leverage points. In the next sections, instead, we provide, as a proof of concept, the application of our approach to both algorithmic and architectural model-driven engineering.

Restrictions to the metamodel. A model always conforms to a unique set of rules that defines the legal entities that can populate the model itself, their syntactic and semantic specification as well as the way these entities are allowed to interact. This set of rules are formalised into a so-called metamodel. The modeling tool is typically responsible for enforcing the syntactic and semantic rules defined by the metamodel, so that a software designer is allowed to describe only models that do not violate them.

Once the complete chain of model transformations has been understood, it is possible to devise a mapping between specific metamodel elements (or aggregation thereof) and code patterns. Therefore, according to such mapping, those entities or aggregation rules that lead to adverse code patterns (see Chapter 3.5) may be categorised as harmful and explicitly forbidden in the metamodel. The identification of harmful entities and aggregation rules are specific to the application domain and the model transformations.

Definition and extraction of flow-fact information. The definition of flow facts in the form of manual annotation is onerous and error prone. Gathering accurate and exhaustive timing information retrospectively may be complicated by the fact that the sought information may not even be available at all in the program at hand as it may depend on higher level design choices that are not reflected in the final implementation code.

Representative examples has been addressed in Chapter 3.5 where we discussed the effects of user-defined data types (pattern P 10) and the occurrence of some indirect calls that are barely introduced to satisfy some architectural concerns (pattern P 5). In these cases the information on data-types or call targets can be easily collected (or asserted) by perusing the model, whereas they may be hardly resolved by code inspection. Part

of this information, in fact, is likely get lost along multiple legs of model transformations. Similarly, information that depends on the characteristics of underlying kernel or operating system (e.g., loops iterating on the number of tasks in the systems) are clearly not easily re-constructable from the code.

In order to accommodate this kind of information, an abstract model can be extended to accommodate user annotations. Depending on whether the information can be mapped to a specific model element or not, it can be conveyed directly into the code (in the form of source code annotation) or into an external annotation file respectively.

Exploitation of model transformations. The synthesis process of code generation typically consists in a sequence of intermediate model-to-model transformation that moves from the abstract model to the concrete implementation. Whereas some system-level concerns are exclusively expressed at model level, a large part of the information relevant to timing analysis is progressively built along this transformation chain.

The final transformation in particular is responsible for the realisation of the model into a set of predefined code patterns. Having code predictability and analysability in mind, model transformations can be exploited to adjust either the mapping from model elements to code patterns or the generative code patterns themselves. In the former case, one may want, for example, to switch from a while-loop to a more predictable counter-based for-loop. This can be easily done, if the loop semantic allows, by setting a counter variable and changing the mapping function to point to the for-loop generative pattern. Another example would consist in normalising the condition variable that is fed to a multi-way switch so as to guarantee that the final code will be compiled into a bitwise-offset geared jump, instead of a complex nest of conditional branches (cf. pattern P 6 in Chapter 3.5).

Besides analysis-oriented customisations, model transformations straightforwardly allows the extraction of timing information. Code generation patterns, in fact, are necessarily parametric so that they can be used to instantiate different scenarios. This is the case, for example, of a matrix multiplication pattern which includes a loop over the matrix elements: we may be interested in extracting information on the maximal loop iterations. By intercepting this parametrisation, we are able to make it explicit, re-formalise it according to a specific annotation syntax (e.g., the AIS specification for the aiT tool), and finally generate it as a comment in the proper place in the code. Figure 3.14 depicts the extraction of timing information from within the model-to-code transformation process.

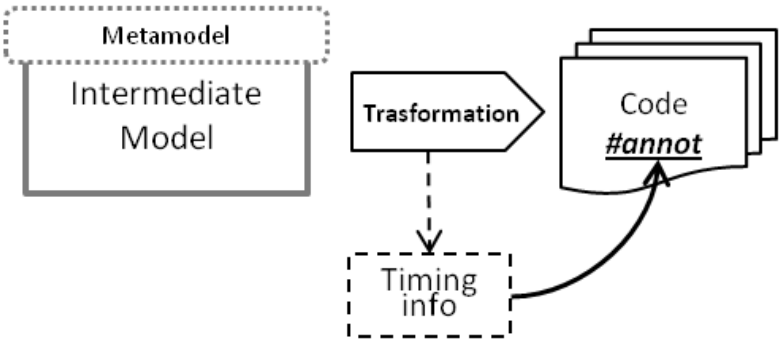


Figure 3.14: Information extraction and code annotation.

3.6.3 Proof of concept

Model-based development and tools are increasingly adopted in HIRTS industries, seeking for a reduction in both time and costs of industrial software development. This trend acknowledges that the increasing complexity in the design and verification of industrial-scale systems can more efficiently governed by the adoption of abstract models for both the algorithmic and architectural specifications of a system.

In respect to the former, in fact, model-based development provides a modular approach to design complex dynamic systems, including control systems, signal processing, and communications systems. Modeling proceeds through the definition of different kinds of diagrams whose main entities model either algorithmic or structural, and dynamic behaviour. State-of-the-art modeling tools, such as Matworks Simulink [96] and Stateflow [97] or Scade [42], differ in the description language they use, which may include block diagrams, state-transition diagrams and other proprietary constructs. Modeling functionalities are extended with advanced simulation engines for early verification of the functional behaviour of the modeled system. Consolidated models can then undergo automated code generation process that eventually yields a semantically equivalent software product, which is thus correct-by-construction, with respect to the model they derive from, and ready to be deployed.

From the architectural specification standpoint, when the MDE paradigm meets the reuse-oriented Component-Based Software Engineering (CBSE) approach, the abstract model of a system consists in the definition of a set of interacting *software components*. An interesting definition of component in given in [150]:

> *A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third-parties.*

According to that definition, CBSE sets its main focus on the (hierarchical) organisation and composition of components, intended as highly reusable building blocks. Components are standardised software entities that are characterised by a set of services that are provided to other components (*Provided Interface*) and a set of services that must be provided for the component to execute (*Required Interface*). Figure 3.15 provides a graphical representation of a component with its set of provided and required interfaces.

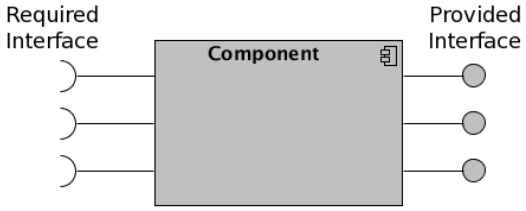


Figure 3.15: Classical (UML) graphical representation of a component.

Provided and required interfaces do not only define the functional behaviour of a component but they are typically extended to address non-functional system properties that determine how the components services will be executed at run-time. From the CBSE standpoint a system is modeled according to a *component model* - typically comprising different views - that define the implementation, documentation and deployment of the set of components that realise the overall architectural specification of the system. The conformance of the component model to a proper computational model allows to perform different forms of analysis at the model level (in its PSM representation) instead of the actual implementation. For example, a model-based schedulability analysis approach has been presented in [23]. Finally, the definition of a proper *programming model*, consistent with the computational model, informs the final model-to-code transformation that automatically yields to the code implementation of the software architecture. In this setting, the concrete functional behaviour of a component is not the main concern as long as all functional and non-functional interface contracts are fulfilled. The algorithmic behaviour of the generated run-time entities is typically not included in the code generation process, where a simple skeleton is generated instead.

The separation of functional and non-functional concerns [39] favoured by the CBSE approach straightforwardly enables the separate application of the MDE approach to both algorithmic and architectural dimensions, and the later combination of the synthesised code. Assuming that the algorithmic specification does not address any non-functional concern, the automatically generated functional code will in fact replace the functional placeholders in the architectural implementation so that to obtain the complete system implementation.

While addressing the timing analysability of the automatically generated code, we cannot disregard that the algorithmic and architectural specifications, though both defined within a MDE framework, may intrinsically raise different analysability issues and thus need different solutions. In the following, we therefore consider two distinct representative modeling frameworks and detail possible means to improve the analysability of the generated code.

The automatic extraction of flow facts and the automated generation of timing annotations (either in the source code or in an external file), which is part of our approach, requires the selection of a specific annotation format. In our investigation we selected the *ais* [1] file annotation format, for use with the aiT tool. We are also aware of the role of compilers in mapping the source code to object code, which is the actual target of timing analysis. Thus the preservation of code analysability and correctness of flow facts is not guaranteed and depends on the compiler of choice as well as the applied compiler optimisations. In the following, we assume that the generated code will be compiled with the GCC-based GNAT Pro for LEON compiler from AdaCore [2], with no optimisation. The selected simplifying configuration allows the preservation of the source-level control flow on the generated object code (besides some know exceptions) and does not require any complex remapping of flow facts into object code [73].

3.6.3.1 Algorithmic specification

Several MDE industrial-level tools are available for the design of the functional behaviour of (parts of) a system. We already mentioned a score of commercial tools that provide both modeling and code generation functionality [96, 97]. With respect to our objective, however, the main issue with those tools is that their internals are more or less transparent to the user and they do not easily allow modifications to either the metamodel or the model transformations (included the model-to-code one).

We partially overcame these limitations by selecting GeneAuto [139]. GeneAuto is an open-source framework for model-to-code transformation, that allows to generate code

from different modeling artifacts based on data-flow or state models, such as Simulink and Scicos [143] (an open-source alternative to Simulink) block diagrams and Stateflow diagrams. The set of transformation that proceeds from the input model to the implementation code is organised into several elementary tools, each one performing a model transformation or refinement. GeneAuto exploits two intermediate modeling formalisms: the `GASystemModel`, which is the internal pivotal formalism the supported input models are initially transformed to, and the `GACodeModel`, which is an intermediate representation, much close to the implementation code but still generic enough to allow a final transformation step to generate the implementation code in different programming languages (C and Ada are currently supported). Figure 3.16 shows the GeneAuto framework and hints at the main entities that inhabit the intermediate languages metamodels.

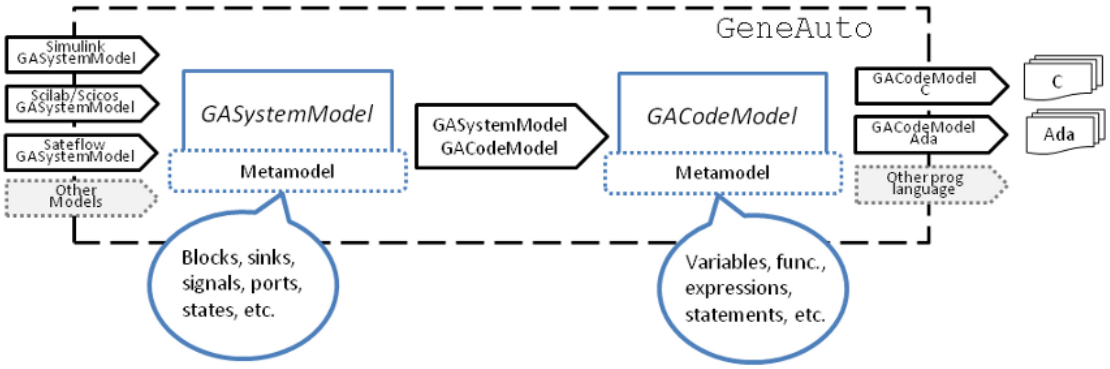


Figure 3.16: Core model transformations in the GeneAuto tool set.

Although GeneAuto is not a complete modeling framework, the availability of its source code allowed us to inspect and modify its internal model transformations. We restricted our investigation to the generation of Ada source code, which was not included in the original release of GeneAuto and is part of a side project effort led by AdaCore under the Open-DO initiative [51]. We focused on the application of our approach to data-flow block diagrams designed in Scicos [143], a modeling, simulation and code generation toolbox for Scicoslab (formerly Scilab) an open-source alternative to Matlab, developed at INRIA and ENPC.

Functional diagrams in Scicos are similar to their Simulink counterparts, with just a slightly different notation. The main entities in the Scicos data-flow diagram metamodel consist in a set of basic building blocks, corresponding to combinatorial or sequential functions, ports, and signals. Blocks are thus interconnected through input and output

ports and can be hierarchically organised in subsystems. Scicos data-flow block diagrams allow to model complex data-intensive algorithms that fit the CBSE concept of algorithmic specification.

As discussed in Section 3.6.2.1, the metamodel definition is the first leverage point to enforce code analysability. Accordingly, a set of restrictions should be applied directly to the Scicos metamodel so that only analysable models can be constructed. We did not need to apply any further restriction to the Scicos metamodel as a set of restrictions is implicitly set by GeneAuto in its model import step. GeneAuto has in fact been especially tailored to the development of high integrity systems and deliberately supports only a *safe* subset of the Scicos metamodel, excluding those constructs in the input model that may complicate the analysis of the final code. As a matter of fact, the GeneAuto native code generation engines enforces MISRA C [105] compliant code. In particular, although the data-flow diagram formalism in Scicos supports the definition of hybrid systems that can include at the same time continuous-time and discrete-time components, only the latter are supported in GeneAuto. Although these limitations are meant to improve software quality and analysability in general, they naturally entail an improved timing analysability.

The restriction to Scicos data-flow diagrams, combined with those metamodel-level constraints, enforces the corresponding `GASystemModel` to not include harmful model elements such as recursion, dynamic allocation, function pointers or any form of unstructured code pattern. This favourable condition, however, also narrows our space of intervention to the `GACodeModel` to Ada code transformation. Automated code generation from other formalisms is likely to imply the generation of more complex constructs (e.g., state junctions in GeneAuto may result in the generation of `goto` statements). At the same time, other formalisms may also allow the application of other approaches: as suggested in [151] information on state transitions in Stateflow models can be exploited to automatically detect infeasible paths and different operation modes. We were also unable to experiment with the preservation of model-level annotations up to the generated code as the Scicos to GeneAuto bridge does not currently preserve model annotations.

As a proof of concept of our approach, we focused on the `GACodeModel` to Ada code transformation to ease timing analysis of the final code. The `GACodeModel` already holds all the information on variables, functions, expressions and statements, which are generated along with the transformation from `GASystemModel`. The final model-to-code transformation in GeneAuto is implemented through a specialisation of a generic `Printer` module that simply translates the implementation-level entities into programming language specific constructs.

In the GACodeModel a loop statement can be represented either as a ForStatement or a RangeIteratorStatement. The C code Printer translates both elements to a straight counter-based for loop. The Ada Printer implementation instead provides two different generative patterns where a RangeIteratorStatement is translated into a for-loop ranging over a user-defined Ada Range type, and a ForStatement is translated into a while-loop (Figure 3.17).

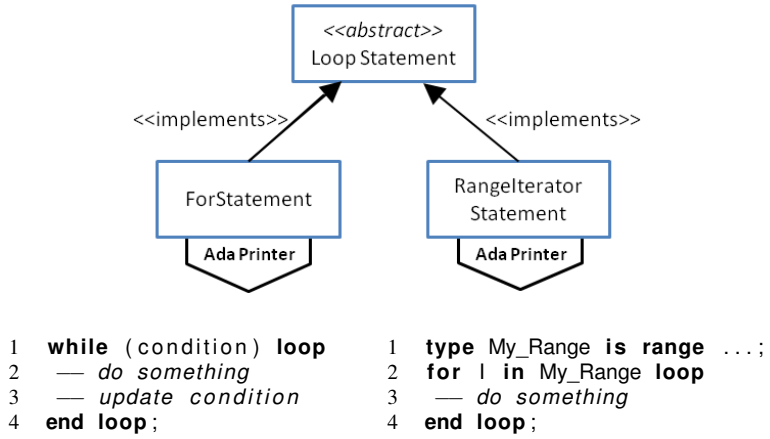


Figure 3.17: Generative patterns for loop statements.

In contrast with model-to-model transformations, model-to-code transformations do not typically rely on a stringent formal specification of the adopted transformation rules [38]. The GeneAuto framework is a representative example of a pragmatic (and rather informal) approach to model-to-code transformation. In fact, automated code generation is performed according to the so-called *visitor-based* approach [38], where the source code is generated along a simple traversal of the the internal representation of a model.

The GACodeModel ForStatement is modelled as a set of sub-entities: a loop variable, a termination condition (over the loop expression), a loop body and loop pre- and post-statement (i.e., the increment step). Figure 3.18 below shows how the AdaPrinter module (written in Java) actually implements the model-to-code transformation for the ForStatemnt as an Ada while-loop (Ada keywords are highlighted).

However, we noticed that within the GeneAuto framework the condition variable in the while-loop pattern is often bounded by construction to an implicit counter variable (with no additional exit condition). Under these premises, the loop condition can be reformulated as a triple (or binary expression) <x,op,y>, where x is an explicit counter variable, y

```
1  // Prints ForStatement as a While-loop:
2  public String print() {
3    String result = "" + Formatter.newLine();
4    result += preStatement.print() + Formatter.newLine(); // Init condition
5    result += Formatter.indentLine()
6           + "while ("+ conditionExpression.print() +") loop"
7           + Formatter.newLine();
8    Formatter.incIndentLevel();
9    result += bodyStatement.print() + Formatter.newLine(); // Loop body
10   result += postStatement.print() + Formatter.newLine(); // Update condition
11   result += Formatter.indentLine() +"end loop; " + Formatter.newLine();
12   return result;
13 }
```

Figure 3.18: Visitor-based model-to-code transformation for the ForStatement.

is the maximal value for `x` and `op` is the binary operator $\leq$. Therefore, the model-to-code transformation rule for the `ForStatement` can be modified to ease loop analysis (cf. pattern P 13 in Section 3.5.2.1) by generating explicit counter-based for-loops, as reported in Figure 3.19.

```
1  // Prints ForStatement as a For-loop:
2  public String printAsFor() {
3    String result = "";
4    BinaryExpression bExp = (BinaryExpression)this.conditionExpression;
5    // Do not need the preStatement (loop var initialised to 0)
6    // --> for Var in 0 .. MaxValue loop
7    result += Formatter.newLine() + Formatter.indentLine()
8           + "for "+ bExp.getLeftArgument().print() // Loop variable
9           + " in 0 .. "+ bExp.getRightArgument().print() +" loop" // Max value
10          + Formatter.newLine();
11   Formatter.incIndentLevel();
12   result += bodyStatement.print() + Formatter.newLine();
13   // Do not need the postStatement (loop var automatically incremented)
14   result += Formatter.indentLine() +"end loop;" + Formatter.newLine();
15   return result;
16 }
```

Figure 3.19: Alternative model-to-code transformation for the ForStatement.

The outcomes of the two alternative code generation strategies for the `ForStatement` are reported in Figure 3.20.

Computational intensiveness is the main characteristic of data-flow diagrams. At the source code level, those computations often result in loops iterating over n-dimensional data that are typically exchanged (via signals) between blocks. From the timing analysis standpoint, the user is often required to manually define flow facts on the maximum number of iterations, which is an annoying and error prone task. Building on the information in the `GACodeModel` we are able to automatically extract flow facts on the upper bound for

```
1  vars := ...; -- Setup initial conditions
2  while ( cond(vars) ) loop
3    -- loop body
4    vars := ...; -- Update conditions
5  end loop;
```

```
1  -- No need to init x
2  for x in 0 .. max loop
3    -- loop body
4    -- No need to increment x
5  end loop;
```

Figure 3.20: Alternative Ada code production for the ForStatement.

each loop. As shown in Figure 3.21, the maximum number of iterations of each loop that implements a computational block is determined by the actual size of the block input ports.

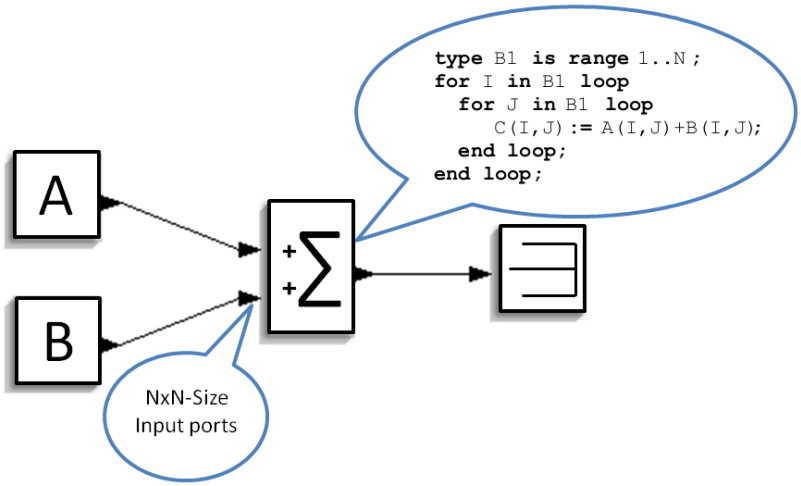


Figure 3.21: Relationship between ports size and loop bound for a summation block.

Depending on the analysis tool, flow facts should be generated in place (i.e., interleaved with the code) or in a separated file. We implemented both solutions and generated the loop bound annotations in the *ais* format, for the aiT static analysis tool.

The automated production of source code annotations straightforwardly fits in whit the model traversal that yields to the implementation code. The information on block port size is represented at `GACodeModel` level as a set of ranges that determines the maximum iteration for all loops in the block computations. Each `Range` can be exploited to generate a loop bound annotation that is seamlessly included in the generator output. Lines 1-5 in Figure 3.22 shows how a loop bound annotation in the *ais* format can be inserted into a code generation template. The generated *ais* annotation consists in a code macro like:

```
/* ai: loop here max N; */
```

where `N` is the extracted loop bound.

When defining the annotations in an external file, instead, we had to account for the naming convention of the compiler in use to identify the annotation target. According to the naming convention in the adopted compiler, in absence of overloading a procedure named `Bar`, declared in the package `Foo`, will have `Foo__Bar` as full link name. According to the *ais* annotation format, a loop in `Bar` is unequivocally identified by the enveloping procedure full link name and a loop index. For example, `loop "Foo__Bar" + 2` identifies the second loop in procedure `Bar`, declared in package `Foo`. To keep consistent information on the annotation scope we implemented a simple state machine whose execution follows the `AdaPrinter` module. This new module, named `FlowFactPrinter`, allows to record the exact scope along the code generation process. Thus an external loop annotation can be defined just by providing the loop bound (lines 6-9 in Figure 3.22): the `FlowFactPrinter` is responsible for generating the correct scope for the flow fact (i.e. Ada package, procedure and loop index). The external flow fact is generated according to the following syntax:

```
loop "Foo__Bar" + k loops max N begin by default ;
```

where `N` is the extracted loop bound, `begin` (or `end`) identifies the shape of the assembly representation of the loop and `by default` just tells the analysis tool to check the correctness of the annotation itself.

```
1  // Insert an aiT code annotation stating the loop bound
2  result += Formatter.indentLine()
3         +  "— /* ai: loop here max "
4         + (((RangeExpression)range).getEnd()-((RangeExpression)range).getStart())
5         + "; */";
6  // Produce a loop bound annotation in an external .ais file
7  // E.g. loop "Foo__Bar" + 1 loops max N [begin|end] by default;
8  FlowFactPrinter.getInstance().PrintLoopFlowFact(
9       (((RangeExpression)range).getEnd()-((RangeExpression)range).getStart()),
10      true);
```

Figure 3.22: In place generation of loop bounds.

As observed in Section 3.5.2.1 (P 10), the use of user-defined data types may complicate timing analysis as the information on the data type is typically not exposed by the compiler. Collecting the same information by code inspection may be complicated by the fact that user-defined data types are typically organised in complex data-type hierarchies whose definition is spread over a large amount of code. For example, the GeneAuto model-to-code transformation exploits user-defined (Ada) `Range` types for those loops that are generated following the (already mentioned) `RangeIterator` pattern. Since the

data-type information is clearly available at model level, we are able to resolve any complex data-type hierarchy and to generate precise loop bounds annotations, again within the code or in an external annotation file. For the sake of a better code readability, we always generate the same information also in the simple form of comment in the code.

3.6.3.2 Architectural specification

Correctness and overall quality of a system are also determined by its architectural description. MDE and CBSE approaches are increasingly adopted for the design and implementation of the architectural specification of industrial applications. The combination of those approaches is currently attracting the interest of the HIRTS industrial sector. Similarly to what we have suggested for the algorithmic design, we aim at exploiting the MDE approach to improve timing analysability of the automatically-generated architectural code.

The stringent requirements on both functional and non-functional concerns in high integrity systems prevents from applying a generic modeling framework and asks for domain-specific solutions. In our investigation we selected the Space Component Model (SCM) Editor [116], a prototype tool currently under development at University of Padua, which is especially oriented to the development of on-board space applications. The SCM Editor is grounded on a comprehensive methodology [115] aiming at the design and implementation of software systems that abide by the principles of composability and compositionality, and guarantees the preservation of non-functional system properties from design to implementation and execution (principle of *composition with guarantees* [158]). The main constituents of this approach are a component model (SCM), that accommodates functional and non-functional domain-relevant attributes; a computational model, that allows to relate the component model to specific analysis techniques (e.g., schedulability analysis); a programming model, consistent with the computational model; and an execution platform that permits the preservation of the system and component properties at run time.

SCM is designed on top of a domain-specific metamodel spreading over several design views, each one addressing specific concerns. At design level, the main model entity is the classical CBSE concept of component. In SCM, however, components are pure functional units that provide an algorithmic specification and do not directly address any non-functional concern. The latter, in fact, are exclusively expressed as declarative attributes on components instances or their provided and required interfaces. We will not detail here the allowable extra-functional attributes in the model, which are discussed in detail in [115]. Although SCM differentiates between different refinements of a compo-

nent (*type*, *implementation* and a *instance*), component bindings (through interfaces) can be designed only between component instances. Figure 3.23 shows how bindings can be defined between component instances through a decorated interface.

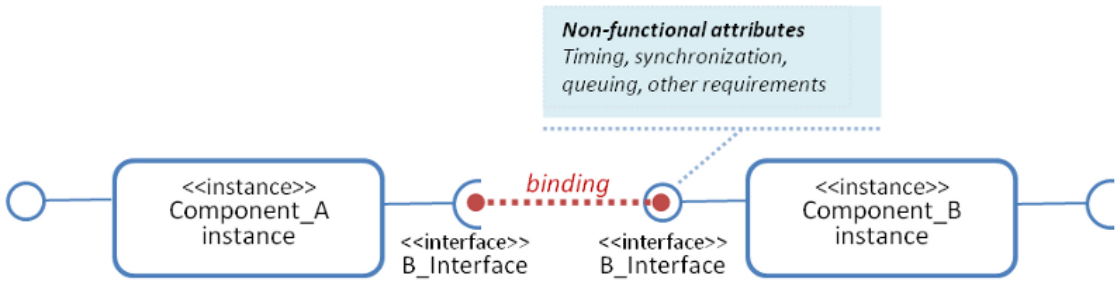


Figure 3.23: Binding between component instances.

A hardware/deployment view allows to describe the assumed execution platform and to allocate design-level components to specific processing unit. The deployment information, in conjunction with a proper computational model, allows to undergo a model-to-model transformation that yields the definition of an implementation-level platform-specific model (view) that accounts for the non-functional attributes in the model. SCM adopts the Ravenscar Computational Model (RMC) [24], originating from the Ravenscar profile [27]. The constraints imposed by RCM on system design and implementation enforce the definition of systems that are amenable to static analysis by construction.

The RCM-compliant implementation model builds on the main concepts of *containers* and *connectors* to realise the non-functional system properties. Containers realise the deployment of a component by encapsulating it on top of the execution platform of the processing unit and by implementing a predetermined run-time entity (i.e., cyclic or sporadic tasks and protected objects) depending on the timing and synchronisation properties of the components. Connectors, instead, implement the bindings between different components and between components and the underlying execution platform (middleware services, communication channels, etc.), which typically consist in function/procedure calls, remote message passing or data accesses.

A final code generation step may then stem from the implementation view and yield C++ or Ada source code conforming to a set of predefined code archetypes. Those code archetypes, in turn, abide by the programming model subsumed by RCM. The automated code generation step is typically performed on a consolidated model that has undergone

some form of analysis: as part of the SCM framework, the implementation-level view can also be automatically transformed, for example, in a schedulability analysis model (SAM) that can be statically analysed with mainstream schedulability analysis techniques.

In our investigation we focused on the code archetypes, aiming at identifying possible means to improve the timing analysability of the generated code. The SCM framework relies on a set of reference implementations of the Meta-Object Facility (MOF) meta-modeling and model management standard [66, 114], provided by the Eclipse Modeling Project [40]. In particular, according to the MOF Model To Text (M2T) transformation specification, the SMC model-to-code transformation step is formalised as a set of *textual templates*, characterised by fixed parts and placeholders that are intended to accommodate data extracted from the source model. Within the SCM framework, those templates adhere to and implement a set of RCM code archetypes.

```
1  [template public generateInterface(model: Model, aInterface : Interface)]
2   [file ((aInterface.name).concat('.ads').toLower(), false)]
3  with Datatype; use Datatype;
4  package [aInterface.name/] is
5    type [aInterface.name/] is Interface;
6  [for (op : Operation | aInterface.ownedOperation)]
7   [if (op.ownedParameter->size() = 0)]
8      procedure [op.name /] (Self : in out [aInterface.name/]) is abstract;
9   [else]
10     procedure [op.name /] (Self : in out [aInterface.name/]
11    [for(par:Parameter|op.ownedParameter)];[par.name/]:[par.direction/][par.type.name/]
12    [/for]) is abstract;
13  [/if]
14 [/for]
15    type [aInterface.name/]_ptr is access all[aInterface.name/]'class;
16 end [aInterface.name/];
17  [/file]
18 [/template]
```

Figure 3.24: Simple M2T template for an interface specification. Credits to SCM [116].

The simple textual template in Figure 3.24 allows to generate the source code for the Ada package specification (thus with the *.ads* file extension) of a given interface. Similar textual templates are defined for each code archetypes in the SCM framework [116]. The final source code is determined as the combination of transformation directives, iterators, conditionals, manipulations and queries on the model (reported in italic within square brackets) with recurrent code parts. It is worth noting that the representation of model-to-code transformation rules by MOF textual templates is just slightly more structured than that provided by the visitor-based approach in the GeneAuto framework (Section 3.6.3.1) and it does not lend itself to a stringent formal representation. However, textual templates

generally provide a cleaner separation between model dependent and independent code parts.

With respect to timing analysis, the SCM framework assumes full composability of the WCET of each component: the WCET of a task is computed as the summation of the WCET of the functional specification of each component involved in the task body call chain, augmented with a known overhead stemming from the execution of non-functional code ascribed to the container/connector mechanism. However, the WCET values of each components, which can be fed to schedulability analysis at this early design stage, are mainly estimates from previous experience or development prototypes and need to be eventually replaced with dependable WCET bounds. The same holds even for non-functional code: although the execution time incurred by containers and connectors is assumed to be known and fixed, we still need to confirm their WCET behaviour.

We first acknowledge that the separation of functional and non-functional concerns enforced by SCM and the adoption of the Ravenscar profile as programming model does not only make a system amenable to schedulability analysis, but also positively affects its timing analysability. As already observed in Section 3.5.2.2, allowing task synchronisation and communication exclusively via protected object excludes complex synchronisation patterns that are possible source of non-determinism in schedulability analysis. At the same time, the adoption of the immediate ceiling priority protocol [14] to provide mutually-exclusive access to protected resources allows to exclude any interference from task blocking on the cache behaviour (see Section 3.5.2.2, point D 3).

Probably the most relevant effect of the SCM implementation model on the final generated code is the implementation of the interface promotion mechanism between containers and components. The component provided and required interfaces, in fact, are respectively delegated and subsumed by the container envelop, as shown in Figure 3.25 below.

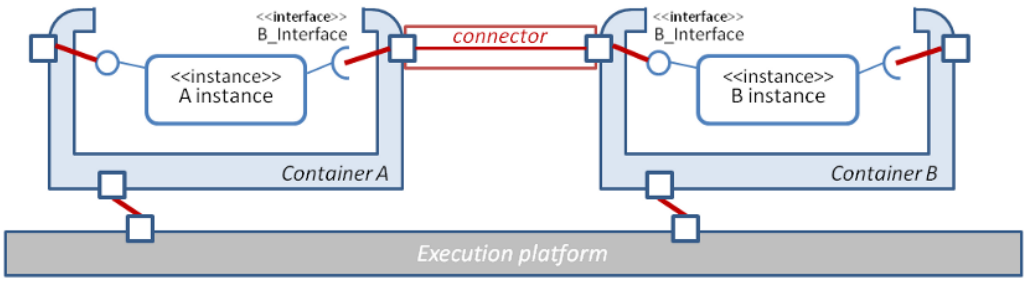


Figure 3.25: The container-component delegation mechanism.

In this respect, the generated code exactly mirrors the implementation model as the delegation mechanism is implemented through interface implementations. According to the code archetypes, however, the set of required interfaces of a component are implemented as procedure pointers, to guarantee the necessary degree of genericity and reuse among different instances of the same component type.

The positive point with this approach is that it does not exploit the instantiation of generic types (or templates in C++) to express genericity (cf. pattern P 8 in Section 3.5). The other side of the coin is that also function pointers complicate timing analysis, as explained in Section 3.5.2.1 (P 5). Those procedure pointers are actually initialised at system start up when each binding between components is implemented by exchanging procedure pointers from the container that provides a service to the container that requires it. Returning to Figure 3.25, a pointer to the procedure implementing the interface `B_interface` in component `B instance` is retrieved from `Container B` and given to `Container A`, which in turn initialises the respective pointer in `Component A`. From the timing analysability point of view, those procedure references cannot be statically resolved and a flow fact annotation is required to reconstruct the control flow of the analysed program. Clearly this is a non-standard use of indirect calls as the use of a pointer is not justified by any dynamic algorithmic behaviour and is instead tied to architectural concerns. In fact, since the SCM model also holds information on the relationship between the component provided procedure and the required ones, we are able to automatically generate a proper annotation (currently in the *ais* format) in a separate file that statically resolve this form of dynamic calls.

The textual template reported in Figure 3.26 generates an external *ais* file (unique for the model) that includes the flow facts required to resolve all the dynamic calls in the model between a component and the containers implementing its required interfaces. The adopted syntax is exactly the same M2T-compliant specification language used to define the set of SCM code archetypes. For each component implementation instance (as each component in SCM can be instantiated multiple times) we iterate over all component operations and identify the respective required interfaces (lines 11-22). We then use the full link name of both the component instance and the container that, according to the component-container bindings (*connector*), provides the required interface implementation. This information are used to generate a flow fact annotation in the form:

```
instruction "component_RI_operation_full_link_name" + 1 computed
        calls "connected_container_provided_operation_full_link_name";
```

which states that the first dynamic call in *"component_RI_operation_full_link_name"* shall

be resolved by the analysis tool to a call to the specified procedure *"connected_container_provided_operation_full_link_name"*. It is worth noting that a component operation may invoke more than one operations from its set of required operations. In this case, the template will generate a separate flow fact annotation addressing, for example, the second dynamic call in *"component_RI_operation_full_link_name"* by exploiting the `[/i]` counter (line 24 in Figure 3.26).

```
1  [template public ComponentContainerCall(model : Model,
2                                          compInstInputList : Sequence(OclAny),
3                                          connectorInstInputList: Sequence(OclAny)) {
4      compImplList : Sequence(Component) = getComponentImplList();
5      compInstList : Sequence(InstanceSpecification) =
6                          compInstInputList->filter(InstanceSpecification);
7      connectorInstList : Sequence(InstanceSpecification) =
8                          connectorInstInputList->filter(InstanceSpecification);
9       }]
10 [file ((model.name.concat('_dynCalls.ais')).toLower(), false)]
11 [for (cImpl : Component | compImplList)]
12 [let cImplInstList : Sequence(InstanceSpecification) =
13                          getComponentInstanceList(cImpl, compInstList)]
14   [for (opImpl : Operation | cImpl.ownedOperation)]
15    [if (getICB(opImpl) <> null)]
16    [let RIOpName : String = ((cImpl.clientDependency->filter("Realization")->
17                               any(true).supplier->any(true).name).concat('s__')
18                               .concat(cImpl.name).concat('s__').concat(opImpl.name))
19                               .toLower()]
20 try {
21 # Dynamic calls flow fact for routine. "[RIOpName/]"
22    [for (cOp : CallOperationAction | getCalledOperationList(getICB(opImpl))
23                                      ->filter("CallOperationAction"))]
24 instruction "[RIOpName/]" + [i/] computed calls
25     [for (cImplInst: InstanceSpecification | cImplInstList)]
26      [for (riSl : Slot | cImplInst.slot->select(definingFeature = cOp.onPort))]
27       [let boundSlot : Slot = getBoundPIslot(riSl, connectorInstList)]
28 "['CT_'.concat((boundSlot.value->first().oclAsType(InstanceValue))
29                               .instance.name).concat('s').toLower()/]__
30                               [getLinkName(cOp.operation.name).toLower()/]";
31       [/let]
32      [/for]
33     [/for]
34    [/for]
35 }
36    [/let]
37    [/if]
38   [/for]
39 [/let]
40 [/for]
41 [/file]
42 [/template]
```

Figure 3.26: Textual template for the generation of flow facts.

The only issue in generating this kind of flow facts comes from the need to unequivocally identify the link names of the involved procedure: this is because, both interface definitions and delegation pattern induce an overloading of those procedures. Naming rules are specific to the compiler in use: in our case the GNAT Pro LEON compiler simply append a numerical suffix (e.g., `__2`) to the normal procedure link name, following a set of specific rules. The impact of the automatic generation of this kind of annotation is not negligible in the analysis of industrial-scale systems: in a recent experiment on part of a real on-board satellite system [103] we were forced to manually define more than one hundred annotations to resolve this architectural pattern.

We observed a similar naming resolution problem for procedures in protected objects (shared resources with an access protocol) as the compiler expansion introduces a wrapper to manage a mutual exclusion mechanism (lock) on the procedure itself. In this case, again according to the specific compiler, wrapper and original procedures are suffixed with a `P` and a `N` respectively. In order to support the generation of correct flow-fact scopes we implemented an external procedure (termed query in the MOF M2T specification) that can be invoked from within the code generation process to compute the exact link names, according to the compiler internal rules (see `getLinkName(cOp.operation.name)` in Figure 3.26). Exploiting the automated generation of these simple flow facts we were able to provide all the architectural-level annotations for a simple producer-consumer toy example.

Another interesting means to improve the system analysability could consist in exploiting the architectural description to allow a separated timing analysis of different operation modes. In fact, following the principle of separation of concerns, discriminant factors on the different operation modes is kept separated from the functional specification of the system and is handled from within the container implementation. Unfortunately, we were prevented from extensively investigate this approach as well other promising solutions due to the current incomplete and prototyping state of the modeling tool. However, the SCM Editor is currently under completion and we expect that we will soon be able to extend our investigation.

3.6.4 Summary

The timing analysability of a program is inherently influenced by the characteristics of the software in that poorly analysable code prevents the determination of safe and tight WCET bounds. To counter the effects of those constructs, the user is forced to undergo the onerous and error prone process of manually defining a score of flow fact annotations to support the analysis process. The model-driven engineering approach, with its auto-

mated code generation facility, could in principle provide an effective way of imposing predictable code patterns and coding styles that guarantee code analysability by construction. Current MDE modeling frameworks, however, are rarely informed to system timing predictability and, more importantly, their final code generation steps does not account for timing analysability concerns.

In this chapter we reasoned on how MDE approach can be effectively exploited to improve the system analysability. We focused on model definition and transformations as the most promising spaces of intervention for improving software analysability, either by enforcing predictable construct or by automatically generating valuable timing annotations. We then provided interesting, though still incomplete, evidence of potential benefits of our proposed approach within two modeling frameworks, addressing the functional and architectural dimensions of a system. The obtained system is expected to be more easily analysable by construction without relying on overly intrusive and onerous user intervention.

3.7 Cache-aware Incremental Layout Optimisation

In addition to making WCET analysis harder, the context-dependent timing behaviour of caches also acutely clashes with the incremental nature of the software development practices sought by HIRTS industry. In that industrial setting in fact, the hardware and software development and their integration and qualification proceed in incremental steps to better master the complexity of the process and thus contain schedule and cost hazards. As a key part of that process, trustworthy information on the timing behaviour of the software must be had from as early in its development as possible. The later that information emerges the more hazard may impend on system integration. The later that information changes for the worse the more costly the fixes.

Cache-aware timing analysis, whether based on static techniques [48] or hybrid measurement-based methods [20] does not lend itself to incremental development. Furthermore, it also critically relies on the availability of information that can only be consistently determined on the final executable, and thus near the end of the development process. The way the code and the data of the program are laid out in memory is a crucial information item for cache-aware timing analysis. The memory layout of the program in fact determines the pattern of hits and misses incurred by individual executions at run time [25], while it also contributes to inter-task interference [75, 6]. Both phenomena manifest themselves as cache jitters.

More specifically, the memory layout determines the amount of conflict misses [60] that occur when memory blocks in the working set of a program compete for cache space. Incrementally adding a module to a software system may thus affect the cache behaviour of the preexisting modules as a consequence of changes in their memory layout. Even small changes may cause significant jitter in the observed timing behaviour [25, 100]. As the memory layout may naturally change upon subsequent software releases, no timing guarantees can be actually obtained from a system subject to incremental development. The bound obtained on a previous release for a given module is in fact likely to become invalid in the subsequent release.

Avoiding unnecessary conflict misses by means of cache-aware memory layout placement directives is a comparatively straightforward countermeasure that may improve both cache performance and predictability. Several approaches have been proposed in the literature to compute an optimised memory layout that guarantees better cache performance in the average case [156, 58, 53, 119] or in the worst case [89, 35]. However, while being provably effective in reducing the number of conflict misses, those approaches may prove unfit for industrial application.

The fact that classic layout optimisation techniques are meant to be applied at the tail end of development, on the final system, intrinsically contradicts the incremental slant of the industrial development process and fails to understand the role played in it by timing information. Besides being dauntingly intractable for large systems, the application of layout optimisation techniques as part of final verification would occur just too late: guarantees on the timing behaviour must instead be acquired incrementally. Applying those techniques on each incremental release would also be no use unless measures were taken to ensure that the memory layout of the previous release was preserved onto the new increment.

In our activities we focused on the instruction cache to start with. We contend that the same approach could also be applied to reducing cache conflicts between data structures. At the same time, we are aware that data cache optimisations are more effective when applied at intra-procedural level to address spatial locality in data access patterns, instead of temporal locality alone.

We revisited the use of procedure positioning, a well-known layout optimisation technique, to control the variability in the memory layout and thus limit its negative effects on the cache behaviour. Our ultimate goal is to enable an incremental approach to cache-aware WCET analysis, which arguably fits the industrial process well. We exploit information on the program structure to first compute and then enforce a memory layout that both minimises the number of potential conflict misses and also preserves the cache behaviour of software module across incremental releases.

In the following we introduce a novel cache-aware layout optimisation technique that differs from previous approaches in both the program representation and layout computation. We then present a prototype tool that enables an incremental application of our technique. The prototype is finally used to provide quantitative evidence of the effectiveness of our approach at improving cache behaviour and avoiding cache jitters across development increments.

3.7.1 Positioning of our work

The code layout optimisation problem has been extensively studied as a means to improve cache performance or to limit power consumption. Several studies aim to reduce the number of cache (conflict) misses by exploiting profile-based information to rearrange the program code.

Code reordering approaches can be applied at different levels of granularity: from basic blocks [156], to whole procedures [58, 53], or both [119]. Basic blocks allow fine-grained control over the number of cache misses as the latter depend on which basic blocks

are actually executed. Procedures, although strongly correlated to cache conflicts, are straightforwardly rearrangeable in memory by mainstream compilers whereas basic block reordering requires special support.

Classic procedure positioning approaches use profiling information to build a more or less complex graph structure, e.g.: a Weighted Call Graph (WCG) [119, 58] or a Temporal Relationship Graph (TRG) [53], to represent call relations and call frequencies between procedures. The WCG data structure (Figure 3.27) can be formally defined as follows:

Definition 1 (WCG)**.** Given a program P, the weighted call graph $\mathrm{WCG}_{\mathtt{P}}$ is an (undirected) weighted graph consisting of a set of nodes $V = \{p \mid p \text{ is a procedure in } \mathtt{P}\}$ and a set of edges $E \in V \times V = \{(p, p') \mid p \text{ calls } p' \vee p' \text{ calls } p\}$. A label $W_{p,p'}$ is associated to each edge $(p, p') \in E$ to indicate the call frequency between p and p' in P.

The placement algorithm is guided by a simple heuristic based on call frequency, which exploits the WCG to select the procedures that more frequently call one another. Placing those procedures as near as possible in memory would reduce the likelihood of cache conflicts. As memory blocks are placed in the cache according to a *(block address)* mod *N* mapping function, where *N* is the number of cache lines or cache sets in a direct-mapped or set-associative cache respectively, nearby memory addresses do not compete for the same cache lines.

In practice, procedure nodes are pairwise merged according to the greatest edge weight ($\max W_{p_i,p_j}$) until a single chain of procedures is obtained. With reference to the example WCG in Figure 3.27, the merging process would first produce two macro-nodes [AC] ($W_{A,C}$=20) and [DE] ($W_{D,E}$=12), and eventually end up with the procedure chain [BACDE]. The relative addresses of each procedure within that chain are translated into

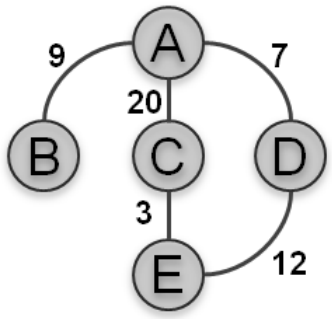


Figure 3.27: A Weighted Call Graph.

absolute addresses, thereby defining the memory layout.

The main defect of these approaches is that they base on profiling information; hence they can only seek average-case execution time (ACET) improvement. For HIRTS instead, focus is on the WCET behaviour. In this respect, an interesting approach has been presented in [89], which extends the techniques introduced in [119] with worst-case path information to assign edge weights in a WCET-centric WCG. That work describes an iterative and a heuristic approaches which use WCGs to compute a memory layout that optimises the cache behaviour along the worst-case path.

Our approach, much like the WCET-oriented one, uses information on the program structure as well as statically pre-computed loop bounds to derive call frequencies between procedures. We further use an improved program structure representation and implement an original procedure selection algorithm. All in all, our approach differs from all the surveyed techniques in that it lends itself to incremental application, as it stores the current optimised layout as a set of constraints which is preserved in determining the placement of new software modules.

As regards the modular application of cache analysis, several studies focus on analysing software components or modules in place of the full executable, e.g.: [117, 130, 15]. Partial cache behaviour is separately computed for each software module, whether a component or an object file, and later composed to account for the way modules are aggregated in the final executable. The main concern of those approaches is to reduce the computational complexity of cache analysis; our key motivation instead is to attain early guarantees on the timing behaviour and to use later analysis to confirm the previous stipulations (as opposed to determine them). Our main focus is thus set on attaining *composability* [129] of cache behaviour during software construction.

3.7.2 Incremental procedure positioning

We consider a software increment to contribute a new software module to an existing, incomplete system. Depending on the software engineering approach in use, the term software module may take a different meaning: from a software package up to a software component downright. For the purpose of this discussion we just regard it as a cohesive set of procedures accessed by an entry point. Moreover we initially assume that the additional functionality offered by a software increment is associated to the entry point of individual tasks, and that tasks are only allowed to interact via shared resources.

The principle of composability that holds for functional modularity implies that the system functionalities can be considered in relative isolation when assessing functional

correctness. Hence, the functional correctness ascertained for earlier software releases is preserved across increments unless regressions occur, which are painful development hazards. The same does not hold however for correctness in the timing domain: each software increment may in fact invalidate the analysis result of previous releases.

As observed in Section 3.7, the memory layout may change every time a software module is added to a system, which in turn may cause preexisting modules to incur a different timing behaviour. This phenomenon becomes pathological when individual basic blocks or procedures evict one another from the cache with devastating effects on cache performance [100]. Such a pathological layout change may happen as the new module gets somehow interspersed with the previous code perhaps because one shares text with the other or the linking order has changed for some reason.

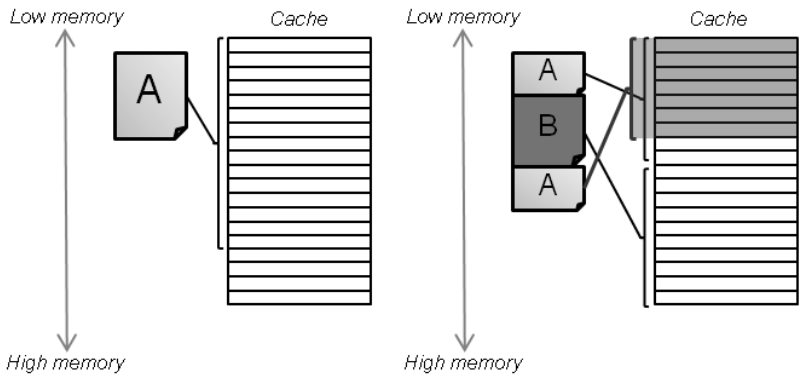


Figure 3.28: Critical layout change.

As a motivating example, consider a software module `A` whose code fits completely in a direct mapped cache without incurring any conflict miss, as shown in the left side of Figure 3.28. White cache lines in the figure do not suffer any conflict between memory blocks. Assume now that a new software module `B` is added to the system as a result of a development increment. As depicted on the right side of Figure 3.28, the memory layout of module `A` can be disrupted in a way that code belonging to it does compete for some cache lines (darker area). As a result, if the conflicting code is repeatedly executed (e.g.: inside a loop) the execution of `A` will incur potentially many additional conflict misses. Hence, any previous timing bound for `A` becomes unsafe and its WCET behaviour must be re-analysed, adding unwelcome costs to an already taxing development.

Classic layout optimisation techniques, aimed at avoiding or at least reducing the amount of conflict misses are typically applied at the end of the development process and

cannot guarantee that a specific memory layout is preserved across subsequent incremental releases. Our approach, which is explicitly incremental, exploits layout optimisations to preserve the WCET behaviour of software modules by avoiding any disturbing change in the memory layout. A locally optimised cache-aware procedure ordering is computed for each software module to enforce a global controlled memory layout which accounts for previously released modules and is preserved on successive releases.

Let us now discuss the specifics of our approach with respect to both layout optimisations and preservation. For the sake of simplicity, we will assume a direct mapped cache although our approach can be straightforwardly extended (as we show in our experiments) to set-associative caches too.

3.7.2.1 Layout Optimisation

Finding the optimal procedure placement would involve the computation of all possible permutations, which is exponential in the number of procedures. Procedure positioning techniques guide the placement algorithm by a simple heuristic which builds on the observation that two procedures that frequently call each other and map to the same cache set are a potential source of conflict misses. As a consequence, those procedures should be laid out as near as possible in memory, to avoid that the caller and callee procedures could overlap in the cache. As anticipated in Section 3.7.1, a WCG is typically used to understand the call relations between procedures and to compute a memory layout that minimizes the number of potential conflict misses in either the average [58, 53, 119] or the worst case [89]. Our view however is that the assumptions behind the procedure selection heuristic used in previous approaches should be revised. We argue in fact that the most critical source of conflict miss should be found in loops that envelope call relations, rather than in call frequencies alone. Not only loops determine the call frequencies but they also define the structural relations between calls, which WCGs do not capture.

Let us define p, p' as elements of $Proc(\mathtt{P})$, the set of procedures involved in a program $\mathtt{P}$, and l_i^p as the i-th loop in p. We discriminate whether the invocation of a procedure p' from within p happens inside a loop (i.e., $l_i^p \rightarrow p'$) or outside any loop (i.e., $p \rightarrow p'$). Accordingly, we recognize three types of structure for a call chain, with regard to the amount of potential cache conflicts that it may incur. The first type of structure is the simple direct call $p \rightarrow p'$ outside of any loop nest; no conflict miss can occur in this case, except for cache blocks possibly preloaded in p. When the call relation between p and p' involves a loop structure, instead, the number of potential conflict misses depends on the structural relation between p and p'. In fact, the second type of call relation occurs when

$l_i^p \rightarrow p'$: the maximum number of conflict misses is then determined by the instructions in the l_i^p body. The third type of relation happens when p and p' are repeatedly called from within the same loop structure: $\exists\ l_i^{p*} |\ l_i^{p*} \rightarrow p \wedge (l_i^{p*} \rightarrow p' \vee p \rightarrow p')$. This is the most critical relation with respect to conflict misses, as the effect of p and p' overlapping in the cache may cause a larger number of misses to be incurred at each l_i^{p*} iteration.

Since the size of a program normally far exceeds the cache size, a placement algorithm is not likely to find an optimal placement that is able to avert all conflicts. Some decisions should be taken instead to address the most critical sources of potential cache conflicts first. In our approach when we select the procedures to be ordered we pay more attention to those our classification ranks as highly related.

Loop-Call Tree structure

Our procedure placement algorithm uses a Loop-Call Tree (LCT) to capture the structural relationship between calls, representing the program structure as a set of procedure nodes and loop nodes.

Definition 2 (LCT)**.** Given a program P, the loop-call tree LCT_{P} is an ordered directed tree consisting of a set of nodes $V = \{p \mid p \in Proc(\text{P})\} \cup \{\ l_i^p \mid l_i^p$ is the i^{th} loop in $p\}$ and a set of edges $E \in V \times V = \{(p,\ p') \mid p \rightarrow p'\} \cup \{(p,\ l_i^p)\} \cup \{(l_i^p, p') \mid l_i^p \rightarrow p'\} \cup \{(l_i^p, l_j^p)$ | loop l_j^p is nested inside $l_i^p\}$. The LCT is rooted in a special entry node that stands for the program root procedure, for example a task entry point. A label $B_{l_i^p}$ is defined for each loop node l_i^p, representing its statically computed loop bound, which makes it possible to derive procedure call frequencies.

This richer representation enables a more informed procedure selection for layout optimisation than achieved with classical structures. The benefits of a LCT compared to a WCG can be inferred from the simple example shown in Figure 3.29. The call relation information represented by the WCG, (a) on the left of the figure, is partial and ambiguous since it may correspond to at least two different LCTs, (b) and (c) in the figure. The key point here is that the two LCTs show structurally different call relations: the most critical overlap, which should guide the placement algorithm, would involve procedures X and Z in (b), and procedures X and Y in (c).

Unless the program fits completely in the cache, which is unlikely in the general case, a misinterpretation of the program structure may accidentally lead to the computation of bad layouts. Similar limitations can be observed for the TRG-based representation. In fact,

while it refines the WCG representation to capture the importance of procedure interleaving and phasing, it still fails to explicitly account for the loop-induced call structure.

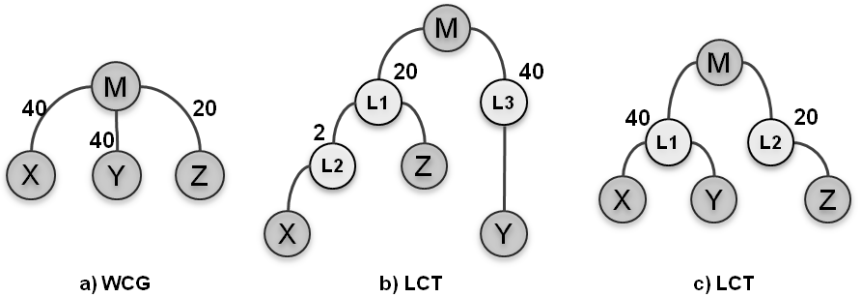


Figure 3.29: WCG vs. LCT expressiveness.

For example, returning to Figure 3.29, canonical approaches that just rely on call frequencies, as in (a), may position the procedures in memory as [MXYZ]. However, assuming the structure depicted in (b), this ordering misses out that the invocation of procedure Y is structurally unrelated with the invocation of procedure X. Although X and Y show exactly the same call frequency, the call chain [MY] is somewhat independent of the relation [MXZ]. Procedure Y can overlap in the cache with either X or Z without incurring any conflict miss, as it does not share any common loop nest with them, as shown in Figure3.29 (b). Conversely, the invocation of procedure X is structurally related with the call of Z, as they share the loop node L1 as common ancestor.

Based on the LCT structure, the selection phase of our placement algorithm is guided by a new heuristic which involves both call frequencies and loop nest relations. Our strategy strives to avoid that procedures belonging to the same loop nest overlap in the cache. In fact, this is a potential source of conflict misses, far more critical than a mere, though frequent, call relation. Procedures that are executed more frequently and share an ancestor loop node will be placed as near as possible in memory to reduce the likelihood of overlapping in the cache. In the example above it would result in placing the procedure in memory in the order [XZYM].

Figure 3.30 displays the effects of applying the two different methods to the referenced example. The fact that in both cases the procedures overlap in the cache does not necessarily imply that they will incur any conflict miss. As shown on the leftmost side, the memory layout obtained only relying on frequency information does not prevent the potential conflicts between X and Z (darker area), while the structure-based layout on the right does.

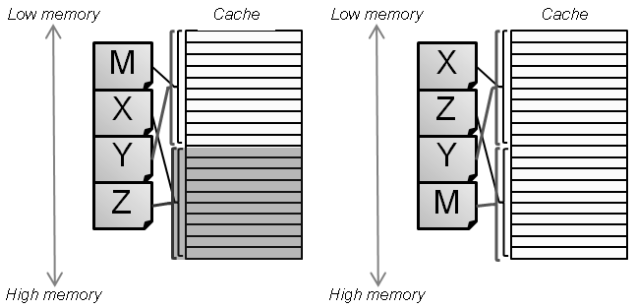


Figure 3.30: WCG vs LCT-based positioning.

The LCT of a program is built starting from the control flow graph (CFG) of the program root procedure (i.e., entry point) as well as that of each invoked subprocedures. The classic dominator analysis [78, 5] is applied to each CFG in isolation to identify loop headers and loop nests. It is worth noting that dominator analysis is only able to detect natural loops; our approach can therefore not be applied if the analysed program contains irreducible loops. However, is worth noting that irreducible loops are typically introduced by aggressive compiler optimisations, which are disabled in the high-integrity domain. Dominance and call-graph information are then used to construct a LCT for each procedure; finally all LCTs are merged according to the program call graph producing a single LCT.

Profile-based approaches (e.g.: [119]) augment the program representation with call frequencies that capture the average case, but do not form a reliable information base for determining the WCET. In this respect, our approach is closer to the WCET-oriented approach in [89] as each loop node in the LCT is assigned a statically computed loop bound. The authors of [89] actually use their WCET-oriented call graph to hold the frequency information just along the worst-case path. Notably instead, we focus on neither average execution nor the worst-case path, since we always account for *every* possible execution path in the program. As a result of that, while our procedure selection step does not directly aim at WCET reduction, it is in fact quite likely to achieve it all the same as a natural effect of its global conflict miss avoidance effort.

Devising a good layout

The LCT provides a hierarchical view of a program or task where the root node represents the main procedure or entry point and procedure nodes are positioned as far from the root as their invocation is nested inside loop structures. This structure is particularly suited for

a structure-based procedure selection since it naturally exhibits the loop-induced relation between procedures.

As a preliminary step, to enforce structural priorities in the procedure selection phase, we recursively order the LCT subtrees according to both depth and execution frequency. In this respect, several heuristics can be defined that privilege the structure of the call chain or the frequency induced by the involved loops. In this study we adopted a combined measure that assigns more weight to deeper loop nests and use execution frequency as an auxiliary criterium to prize nests with higher loop bounds.

Once the LCT has been ordered according to a specific heuristic, we approach the ordering problem by a *divide-and-conquer* strategy where each LCT subtree recursively configures smaller ordering subproblems. Figure 3.31 shows a pseudo-code representation of the core algorithm in the procedure selection process.

The `Process_Sub_LCT` procedure operates on a LCT node and a *pool* of procedures, where the latter is simply an ordering of procedures with relative addresses: we use the term pool as the amount of conflicts inside it is independent of the absolute memory address it would be placed at, as long as the subtree is self-contained (includes all involved subprocedures) and its ordering (i.e., all the inter-procedure offsets) is preserved. In fact the cache conflicts within a pool remain unchanged because the *(address)* `mod` N mapping changes uniformly for all memory blocks.

Building on the LCT structure and ordering we are able to populate pools by executing a simple depth-first recursion (Lines 6-12). Collecting procedures in a bottom-up traversal of the LCT guarantees that procedures that are descendants of a common loop node (most critical source of cache conflicts) are placed as near as possible in memory.

A local pool $\mathcal{P}'$, given as a parameter in the recursive call on the children nodes (Line 10), is used to collect the set of procedures involved in every sub-LCT. Once all the subtrees of the input node have been visited, $\mathcal{P}'$ is merged with the input pool $\mathcal{P}$ that holds the set of procedures that have been gathered so far (Line 21). The algorithm terminates after all nodes have been visited and the recursion chain returns to the LCT root node. The memory layout of the program is then computed by a simple translation of the ordering of procedures $\mathcal{P}$ into absolute memory addresses.

Handling multiple invocations

Multiple invocations of the same procedure along different execution paths are quite common in practice. Figure 3.32 shows an ordered LCT where the two leftmost subtrees share `X` as a common procedure node (which is collapsed into a single node for the sake of bet-

```
Inputs: n : Node of an ordered LCT (initially the root node);
        P : Pool of procedures, initially ∅;

1  procedure PROCESS_SUB_LCT (n, P) is
2    P' := ∅
3    sharing := False;
4    disp := null;
5    if Children(n) ≠ ∅ then
6      for each c ∈ Children(n) loop
7        if c is ProcedureNode and c ∈ P then
8          sharing := True;
9        else /* either LoopNode or unconstrained procedure */
10         PROCESS_SUB_LCT (c, P');
11       end if;
12     end loop;
13   end if;
14   if n is ProcedureNode then
15     P':=P' ⋃ n;
16   end if;
17   if sharing then
18     disp := COMPUTE_DISPLACEMENT (P, P');
19     P := P ⋃_disp P';
20   else
21     P := P ⋃ P';
22   end if;
23 end procedure;
```

Figure 3.31: Core algorithm pseudocode.

ter viewing). Multiple invocations make the positioning more complex, independently of how the program structure is represented (whether call-graph or LCT). In WCG-based approaches, multiple invocations are represented with more than one incoming or outgoing edge on the same node.

Disregarding the fact that those multiple invocations may be involved in distinct call chains is likely to induce non-optimal choices in positioning procedures. In [119] procedures are ordered following a greedy merging process guided by the call frequency alone. At a certain point of that process, a multiple invocation of a procedure will result in two macro-nodes being connected by a weighted edge. These nodes are then merged together simply trying to reduce the distance between the procedures involved in call relations.

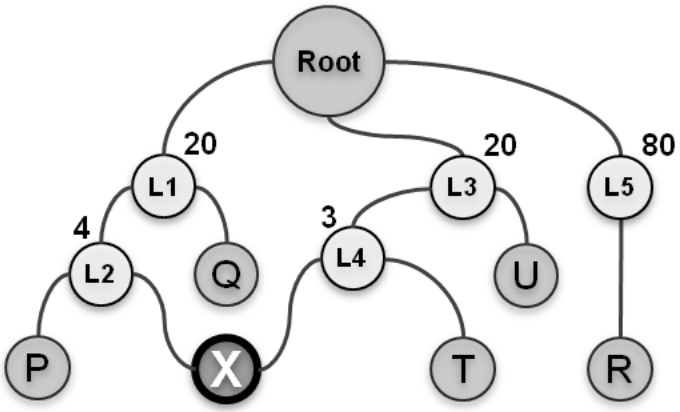


Figure 3.32: LCT with shared procedures.

This may result in an unsuccessful conflict avoidance when each possible merging does not succeed in placing those procedures sufficiently near. The same problem is implicitly handled by the WCET-oriented approach in [89] within the iterative evaluation of the partially computed positioning. The same authors also suggest cloning shared procedures, which however increases the program footprint.

Similarly to [58] we explicitly account for multiple invocations and try to reduce the potential cache conflicts by introducing an ad-hoc offset in the memory layout. If a procedure has already been mapped we stop the recursion along the respective subtree and set a flag (Lines 7-8 of Figure 3.31). Then, at the merging step, an attempt is made to compute a displacement that can avoid cache conflicts between the involved procedures: this computation involves shared procedures in $\mathcal{P}$ and $\mathcal{P}'$ (Line 18 of Figure 3.31). Since the number of intrinsic conflicts inside a pool is independent of its absolute address, we evaluate the number of potential extrinsic conflicts for each cache line displacement $d \in 0 \ldots N-1$ of $\mathcal{P}'$. The displacement that incurs less extrinsic conflicts will be translated into an offset for the current pool of procedures $\mathcal{P}'$.

The introduction of such displacements may of course cause memory fragmentation and increase the memory footprint of the program. In order to circumvent this problem, the profile-based approach in [58] captures the procedures not frequently invoked according to the profiling information (and calls them unpopular). Those procedures are used to fill the gaps resulting from positioning displacements. That idea is not applicable to our structural approach since every procedure in the LCT is assumed to execute according to its worst-case frequency.

We can identify a similar concept by reasoning on the structural properties of the LCT. For example, according to our taxonomy of call relations, if a procedure is invoked by the root procedure outside of any loop, then that invocation can only incur a negligible amount of cache conflicts, regardless of the actual memory layout and cache associativity. We refer to these calls as *relatively-independent* and use the involved subprocedure to fill the gaps possibly left by previously arranged displacements. Note that this step is not reported in Figure 3.31 for the sake of simplicity. Other call relations may fall into the relatively-independent category, depending on the I-cache associativity. Information on the relatively-independent calls is detected without any additional effort during the reordering pass of the LCT.

Furthermore, discontinuities in the memory space can also be filled with procedures that are not invoked during the nominal execution of a program (e.g.: exception handlers), which are typically ignored in the analysis. In fact, the vast majority of multiple invocations involves calls to low-level libraries and a careful adoption of procedure inlining may help us reduce the magnitude of that problem.

3.7.2.2 Incremental Optimisation

This far we have introduced a structure-based procedure positioning algorithm and shown how it may yield a memory layout that reduces the conflict misses resulting from procedures that overlap in the cache. We now illustrate how this approach can also guarantee that the computed layout is preserved across incremental releases where software modules or tasks are added to a system.

We assume that at every incremental step of development, the software modules that form the system at that time are committed. We acknowledge that modifications to previously committed modules might in real life still be required to rectify functional or timing errors, thus incurring a development hazard. In the presence of inter-module dependencies due to shared procedures, the effects of any such modifications may degrade the benefits of our procedure placement, thus failing to mitigate the hazard.

In this setting, in order to enforce a controlled memory layout and to provide guarantees on the timing behaviour upon subsequent software releases, we need to inhibit layout changes on the preexistent code and at the same time account for the layout constraints stemming from procedures shared between incrementally added software units. In this respect, our LCT-based approach is intrinsically incremental as the procedure ordering is computed locally for each subtree of the LCT representation (i.e., $\mathcal{P}'$) and is incrementally merged into a global pool $\mathcal{P}$. The merging step $\mathcal{P} \cup \mathcal{P}'$ preserves the ordering of both local

and global pools, regardless of any intentionally introduced displacement.

In order to preserve the timing behaviour of the system modules across distinct increments we simply account for the global ordering computed up to that point as a set of constraints over all the relevant procedures. These constraints are represented as an initial preexisting pool $\mathcal{P}$ and fed to the very first invocation of our core algorithm. They will therefore be considered when computing the optimised layout for a new software module.

In case a new module includes procedures that have already been positioned in memory to accommodate preexisting modules, the problem is not to preserve the previously computed layout, but rather to account for it while computing a procedure ordering for the new element. By construction, this is not an issue in our approach as the pool $\mathcal{P}$ will also hold the procedures from previous releases. As in the non-incremental case, when a pool $\mathcal{P}'$ exhibits some dependencies with the procedures in $\mathcal{P}$, a displacement of $\mathcal{P}'$ that would minimize potential cache conflicts is computed.

Again, the gap introduced in the memory layout can be filled by those procedures that have been explicitly excluded or are involved in *relatively-independent* call relations. The same also holds for those gaps that can be traced back to procedure positioning of previous releases.

3.7.3 Experimental Evaluation

To evaluate the effectiveness of our approach we developed a prototype tool. With the prototype we assessed the extent to which our structural procedure positioning improves cache performance and studied how effectively it suits incremental development. In the following we introduce our tool and discuss an experimental evaluation of our approach on a representative application case.

3.7.3.1 Prototype tool

Our prototype tool is as independent as possible of any specific target platform or compilation tool-chain. It just relies on a GCC-based compiler which can enforce a specific memory layout by controlling the linking process.

Our tool computes an optimised procedure placement from a description of a program structure (typically one or more CFGs) and a statically computed bound for each loop in the program. We do not deal with the provision of such inputs as we assume they can be generally obtained using available static analysis tools, for several target platforms and compilers. In our experiments, we obtained that information from Bound-T [155], a static

analysis tool that can export an XML representation of the program structure and compute the maximum iterations for each loop.

The overall structure of our tool is shown in Figure 3.33.

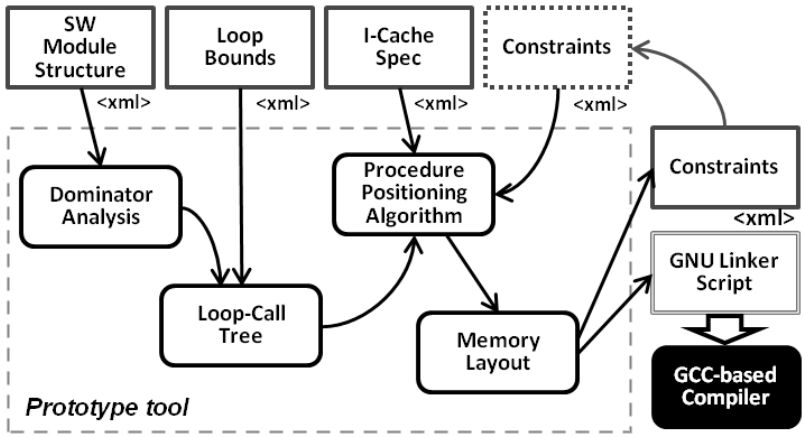


Figure 3.33: Architecture of our prototype tool.

The information on the program structure is used to build a CFG for each procedure in the program. Dominator analysis is performed on the obtained CFGs to find all natural loop headers, to compute their dominance relations, and to construct a dominator tree. An overall LCT structure is built from the dominator tree by merging the information on loop headers with call relations between procedures. This LCT is augmented with the loop bounds and used to compute a procedure ordering that minimizes the number of conflict misses, as described in Section 3.7.2.

Information on the I-cache size and line size is also exploited in the computation of a concrete layout, including those displacements required to avoid potential cache conflicts. As a fundamental input, a set of constraints caters for both procedure placements from previous releases as well as other requirements (e.g.: memory areas reserved for interrupts). Based on actual cache configuration and constraints, the procedure ordering is translated into an absolute placement which specifies the memory layout of the program. The eventual memory layout is then reflected in a linker script and an updated set of constraints. The linker script can be fed to a GNU linker to enforce the computed memory layout in the new executable. The new set of constraints, instead, serves as the baseline for the application of the algorithm to the next incremental release.

Our tool is extremely flexible as it works under different cache configurations and does not need to be tailored to a specific target processor. At present, the only limitations to

our prototype descend from the structural features of the program to be optimised as our approach treats neither irreducible loops nor recursion. Those limitations, however, are commonly suffered by many WCET analysis approaches.

3.7.3.2 Experiments

Our experiments had a twofold goal. First we wanted to assess our procedure positioning technique against conflict miss avoidance. As our strategy is structure-based, we expected it to be effective for every possible execution of the program rather than solely in the worst case. We therefore simulated a random execution of a test application and compared the cache miss ratio obtained with either the default layout produced by the compiler or ours.

We further wanted to prove that the improved cache performance from procedure positioning extends to the worst-case execution and, more importantly, is preserved across incremental releases. In this respect a statically computed WCET behaviour is a far better metric than simulated execution. Hence we focused on WCET bounds and evaluated the variability of the WCET behaviour, due to layout changes across incremental software releases. The experiments show that our approach preserves the WCET bounds.

For the sake of completeness, we also evaluated our approach with respect to a WCG-based optimisation. Building on the greater expressiveness of LCTs over WCGs, our prototype is also capable of switching to a WCG representation of the program to perform a basic non-incremental WCG optimisation (also introduced in [89]). It is worth noting, however, that the results obtained with our approach are not directly comparable with either trace-based (ACET) techniques [58, 53, 119] or the WCET-oriented approach in [89]. Our optimisation builds on a data structure that represents all possible program executions rather than the average execution or a single worst-case path alone.

We wanted our experiments to be as representative as possible of an industrial development process in our target domain in terms of both hardware and software. We therefore chose the LEON2 [3] model as reference platform, since it is highly representative of our reference domain of interest and is currently adopted in several European Space Agency (ESA) projects. In our experiments we configured the I-cache to 32 KB, 4-way set-associative, with 32 B lines and Least Recently Used replacement policy.

The assessment of layout optimisation approaches is extremely sensitive to both the analysed code and the cache size as the evaluation may be biased by large caches and relatively small benchmarks. To avoid misleading results, we conducted our experiments with a large-scale piece of Ada software representative of part of the Attitude and Orbit Control System (AOCS) component of a typical satellite system, which is responsible for

managing the communication between the ground segment (Earth communication station) and the satellite itself.

The AOCS functionalities are provided by a set of modules, among which: the Guidance and Navigation Control module (GNC), the Propulsion module (PRO), the Telemetry and Telecommand module (TMTC). Whereas all modules have a footprint much larger than the I-cache size, we chose the GNC for our experiments, which we rated the most interesting from the layout optimisation standpoint. The GNC module in fact involves a considerable amount of loop nests and subprocedures. The PRO and TMTC modules instead were later added to the system to mimic successive incremental releases.

For our experiments we used the GCC-based GNATforLEON compilation tool-chain, developed by the Technical University of Madrid, which includes ORK+ [157], an open-source real-time kernel of small size and complexity, especially suited for mission-critical space applications.

3.7.3.3 Evaluation

We now discuss the results obtained in our experiments. In doing so we observe that when caches are involved the evaluation tends to be very specific to the memory architecture of the target processor. For example, in the LEON2 processor, on an I-cache miss, the referenced instruction block is loaded from main memory in a so-called burst-fetch mode (starting from the missing address till the end of the cache line) and is simultaneously forwarded to the processor. This provision considerably reduces the latency on memory accesses and, more importantly, influences the actual miss rate [120].

Cache performance

We tried our procedure positioning technique on the GNC module measuring the cache performance in terms of the hit/miss ratio observed on the same random execution under different layout configurations. We developed a highly configurable cache simulator to collect the I-cache statistics from a full execution trace of the program, obtained with the TSIM Pro cycle-accurate simulator developed by the LEON2 processor vendor [3].

Table 3.4 relates the cache behaviour observed when the application is executed with the default layout produced by GNATforLEON (tagged *uncontrolled*) to the results obtained with the layout computed by our tool (tagged *LCT opt*). For the sake of comparison, we also provide cache statistics for a WCG-optimised layout (tagged *WCG opt*) and an artificially constructed bad layout (tagged *overlapping*) where all subprocedures overlap in

the cache. The latter was obtained by aligning procedures at a cache set boundary, which is 8 KB in the target I-cache configuration.

Test case	Hits	Misses	Miss ratio
GNC uncontrolled	92,926	3,524	3.65 %
GNC LCT opt	95,666	484	0.50 %
GNC WCG opt	95,204	946	1.01 %
Overlapping	86,660	9,490	9,86 %

Table 3.4: Comparison of cache performance.

The GNC version compiled with our layout optimisation achieves good cache performances as it incurs only 484 misses (less than 1%) over the 96,150 memory accesses captured in the execution trace (slightly better than the WCG-based optimisation). This compares to 3,524 and 9,490 misses in the default and bad layout versions respectively. Since the number of compulsory and capacity misses is almost the same for all versions (i.e., modulo code alignment), the different cache behaviour may be ascribed to a variable amount of cache conflicts. The relative improvement depends on the goodness of the default layout.

WCET behaviour preservation

Preserving the timing behaviour of a software module throughout different incremental releases is the main goal of our proposed approach. We used aiT for LEON2 by AbsInt [1], an industrial-quality static analysis tool, to analyse the binary executable of our application under different layout configurations. To simulate an incremental development process, we first considered the GNC module as part of an intermediate software release and then observed the variation in its statically computed WCET behaviour when the PRO and TMTC modules were added to the system.

The WCET bounds computed for the GNC module alone confirm the effectiveness of our approach even in the worst case (Figure 3.34a). The GNC compiled with our LCT-based optimisation, with a 478,431 cycles bound, outperforms by a 10.5% factor the WCET behaviour observed when no layout optimisation is applied (526,459 cycles). Our approach also slightly improves (1.97%) the WCET observed with the WCG-optimised layout (487,849 cycles). The 603,627 cycles bound observed with intentional overlaps hints at the potentially devastating effect of an uncontrolled layout.

When it comes to WCET behaviour preservation, our incremental layout optimisation enforced the constant WCET bound (478,431 cycles) for the GNC module across all the software releases. This is explained by the fact that, as long as the GNC code is fixed and unmodified, the computed bound only depends on the memory layout. Conversely, when we left the memory layout uncontrolled the WCET increased and varied per release. The WCET bound changed from the original 526,459 cycles with the GNC alone, to 524,768 cycles when the PRO module was joined to it, to 531,581 cycles when the TMTC module was added too. Interestingly, the WCET variation was not monotonically increasing across successive releases as the effects of layout changes are obviously unpredictable.

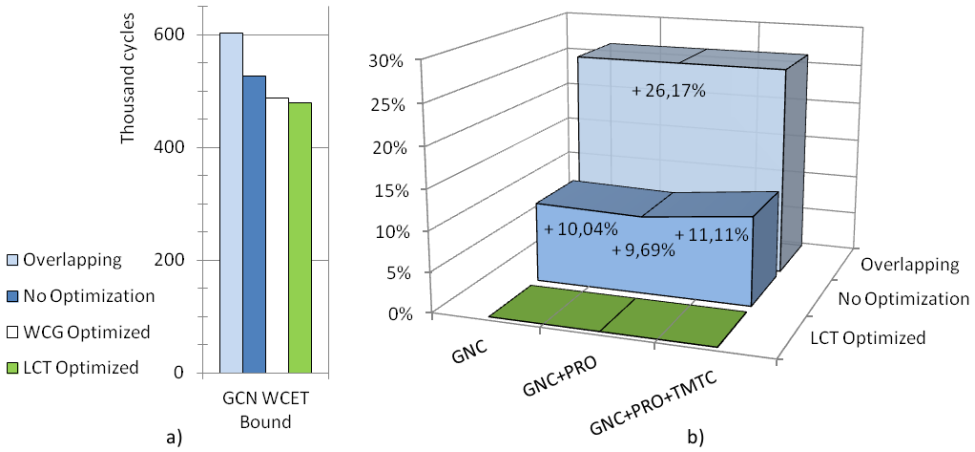


Figure 3.34: WCET performance and variation.

Figure 3.34b contrasts the constant behaviour we obtained with our approach with the relative increase and variation in the WCET bound of the GNC module on different releases when no layout optimisation is applied. Although the observed degradation in the WCET bound across subsequent software increments was limited in between 9.69% and 11.11%, it may inadvertently get worse, as suggested by the 26.17% variation in case of intentionally overlapping procedures. It is worth noting that overlaps occurring at the level of basic blocks could easily cause many more cache conflicts. An even more variable WCET behaviour (up to a 55.5% variation) was reported for individual subprocedures within the module.

Our incremental procedure placement makes it possible to avoid the risk of incurring such extreme variability while at same time improving the WCET performance of all system modules. The results of applying the task-level WCET analysis to the GNC module

are thus preserved when other modules are incrementally added to the system.

Memory fragmentation

The merging step of our optimisation algorithm may introduce corrective memory displacements to avoid cache conflicts between pools of procedures. Although the potentially incurred memory fragmentation is partially remedied by filling those memory offsets with relatively-independent procedures, an increase in the overall program footprint is unavoidable. An assessment of our approach with respect to the increase in the program size would require a deeper and more intensive statistical evaluation. However, the variation we observed in our experiments was limited to a 4% increase in the overall executable size with our optimised layout. We deem such footprint variation to be reasonably acceptable.

3.7.4 Summary

Industrial practice favours the adoption of modular and incremental development process models. This in turn requires that guarantees on the timing behaviour of incomplete releases of the system should be collected as early as possible so as to facilitate prompt detection and reaction to potential timing hazards. In the presence of caches, however, the timing figures obtained for earlier software releases, regardless of the means, are disrupted by the innate sensitivity of the cache behaviour to the memory layout of the executable.

In this Section we presented and evaluated an approach intended to govern the I-cache variability resulting from changes in the memory layout incurred across incremental software releases. We built on a novel structure-based procedure placement technique to enforce an optimised memory layout which is preserved across incremental releases of the system. This provision enables the application of state-of-the-art WCET analysis techniques, like cache-aware static analysis or hybrid measurement-based methods, to collect trustworthy timing information already in the early stages of the development process, and dispenses with the need to rerun all analyses upon each increment.

3.8 Qualitative Evaluation

Among the main constituents of the proposed approach, only our incremental layout optimisation technique has been evaluated on a representative application. A practical assessment of the combination of our contributions against a significant case study would have been desirable to confirm their effectiveness. In this respect, an ideal evaluation would have consisted in reengineering a representative part of a real industrial application following our development approach, and then compare the resulting application against the original system with respect to both analysis cost and quality of the results. Unfortunately, the prototyping state of the tools we selected to support our investigation prevented us from realising a sufficiently complex software system. We are confident that we will soon be able to use a more mature release of the SCM modeling tool, which should allow us to carry on an extensive evaluation of our approach.

This notwithstanding, it is still possible to evaluate our contribution against its industrial applicability. When defining our approach, in fact, we claimed that it lends itself to an effective application to the industrial development process. More importantly, we also contended that the proposed techniques facilitate the application of timing analysis techniques in the industrial development. To evaluate our approach from a qualitative (as opposed to quantitative) standpoint, we leverage on the high-level industrial issues on the applicability of timing analysis, that have been introduced in Section 3.3.2, as evaluation criteria to assess whether our approach contributes to narrowing the gap between state-of-the-art timing analysis and HIRTS practice. Table 3.5 evaluates our approach with respect to the identified industrial issues.

Scalability. Although our approach does not explicitly address the scalability problem, the MDE framework we propose may help in cutting the complexity incurred by inter-procedural analysis and context-dependency, which are the root causes of the (software-related) state space explosion. Both the architectural and functional specifications contribute to the overall complexity of the software and can be tuned to disallow ill-formed software structures (e.g., huge call-graphs) or adverse constructs (e.g., unstructured multi-way branches).

Required skills and knowledge. As we observed in Section 3.3.2, the provision of trustworthy and precise annotations to guide the analysis process typically ask for specific skills and deep knowledge of the program under analysis. The generation of better analysable

Id	Issue	Contribution
I 1	Scalability	Indirectly reduced as a side effect of the proposed MDE framework
I 2	Required skills and knowledge	Mitigated by the enforcement of better analysable software
I 3	Relationship with schedulability analysis	Addressed in the functional specification of a system
I 4	Perceived quality of the results	Reduced pessimism by construction and by the extraction of precise flow-fact information directly from the software model
I 5	Cost-efficiency	Reduced impact of the time-consuming definition of flow facts
I 6	Extensive tool support	Addressed with respect to the proposed layout optimisation technique
I 7	Integration in the SW life cycle	Improved support to incremental development

Table 3.5: Evaluation of our approach against the industrial requirements.

software and the automatic extraction of flow facts both contribute to reducing the amount of user intervention and relieve the user from reconstructing the information on the program. No particular skills are required, besides those generally assumed for a software designer.

Relationship with schedulability analysis. Inattentive system-level design choices may complicate the separation of intra- and inter-task timing analysis. The factorisation of timing-aware design choices, as those identified in Section 3.5.2, in the architectural specification of a system facilitates a clear separation between system-level and task-level concerns.

Perceived quality of the results. The application of our approach may help improve the quality of the results of timing analysis in two respects. Firstly, the avoidance of poorly predictable code constructs is expected to reduce the level of pessimism incurred by timing analysis. Secondly, the automatic extraction of flow-fact information from the model guarantees a high level of precision in the applied annotations.

Cost-efficiency. Again, the impact of the time-consuming task of collecting and defining flow-fact annotations in the overall costs incurred by timing analysis is explicitly addressed and mitigated by our approach through the enforcement of more analysable code and the automatic generation of a large part of the required annotations.

Extensive tool support. The quality of the support provided by current timing analysis tools is out of question. Our approach instead explicitly addresses the need of automated support for memory layout optimisations. In Section 3.7 we provide a fully automated prototype tool for the computation and enforcement of our incremental layout optimisation approach.

Integration in the SW life cycle. On the one hand, our contribution relies on the realisation and adoption of a non-standard MDE framework. Although the MDE paradigm is increasingly adopted in HIRTS, the strong orientation of our approach towards timing analysis concerns does not allow an effortless integration in a consolidated industrial toolchain. However, evaluating the distance of our approach from a generic MDE framework against the benefits that can be obtained may justify an investment that would be applied once for all.

On the other hand, our layout optimisation approach, besides reducing the cache-induced variability, explicitly accounts for incrementality as a main characterising trait of the industrial development and facilitates an earlier application of WCET analysis on incremental releases, instead of the final system.

Chapter 4

Conclusions and Future Work

4.1 Recapitulation of study objectives

The need for more computational power to meet increasingly complex user demands is driving even the most conservative high integrity real-time systems industry towards the adoption of more complex processor equipped with caches and other acceleration features. The introduction of caches, in particular, induces a highly variable timing behaviour that do complicates both schedulability and timing analysis.

The industrial stakeholders understand the fact that the migration to cache-equipped processors is likely to hike the effort required in analysing the timing behaviour of a system, to the extent of breaking an already delicate balance between time and cost of timing analysis and the quality of its results. On the one hand, in fact, the current industrial approach to timing analysis, often still based on simulation and testing, is definitely poorly equipped to cope with the variability incurred by caches. On the other hand, the application of advanced WCET analysis techniques on large-scale complex industrial software developed without analysability in mind, hits on the inherent limitations of those approaches.

In our thesis, we contend that the disruptive effect of caches in the qualification of industrial-level HIRTS can be effectively and efficiently governed only by imposing a paradigm shift in the industrial practice towards an informed use of caches and a proactive approach towards timing analysis. In particular, we propose the adoption of a structured "cache-aware" approach aimed at (i) allowing a cost-effective application of state-of-the-art WCET analysis to complex systems; and (ii) minimising the variability and unpredictability incurred by caches. We maintain, in fact, that any countermeasure to the cache-induced variability is unavoidably tied to a more rigorous attitude towards timing analysis,

more commensurate with the industrial requirements on timing predictability.

We intend such cache-aware approach as the structured combination of a set of countermeasures to improve cache predictability and, in a broader sense, to facilitate the application of timing analysis in industrial setting. As fundamental requirement, the identified techniques and methods shall allow an efficient integration and application in the HIRTS industrial development process.

4.2 Summary of contributions

The main motivation that guided our investigation was the evident disproportion between the strong requirements set on the timing behaviour of industrial HIRTS systems and the relatively inadequate attention on timing concerns along the development process. This disproportion, which is clearly unveiled by the introduction of caches, is partially justified by a non negligible gap that still exists between state-of-the-art approaches to timing analysis and their concrete applicability to industrial-scale complex systems.

The main contribution of our thesis is the definition of a structured approach that aims at effectively narrowing that gap and enables a more pervasive role of timing analysis in the industrial development process. Timing analysis concerns should inform all development activities, from high-level design to software development. Our approach makes the following contributions.

Identification of adverse design choices and code constructs. In Chapter 3.5 we identified a set of code constructs that may hamper the application of state-of-the-art timing analysis approaches. We differentiated between unpredictable code constructs that originate either at task level, stemming from the adoption of specific code patterns, or at system level, resulting from inattentive design choices. The latter in fact may have as much disruptive effects on timing analysability.

We proposed a classification of poorly analysable code constructs based on the issues they bring to timing analysis: *Feasibility*, *Precision*, *Complexity*, *Labour-intensiveness* and *Interference*. In most cases, we also suggested more analysable alternatives.

Formalisation of the CRBD. Still in Chapter 3.5, as part of the investigated design choices, we formalised and provided a bound for a new source of inter-task interference on cache behaviour, namely the Cache-Reated Blocking Delay, which stems from mutually exclusive access to shared resource. From the standpoint of caches, the CRBD may cause

effects similar in kind to task preemption (i.e., CRPD), in that some useful code or data blocks already loaded in the cache may be evicted while the task is being blocked. In contrast to the preemption case, a task may suffer from a CRBD due to lower priority tasks.

We computed a safe bound on the CRBD under three well-known resource access protocols: the *Priority Inheritance Protocol*, the *Priority Ceiling Protocol*, and the *Immediate Ceiling Priority Protocol*. In particular, we showed that the latter does not incur any CRBD interference.

Automated generation of timing-analysis aware code. We leveraged on the MDE approach to enforce the design and automated generation of timing predictable and analysable systems. In Chapter 3.6 we proposed a comprehensive approach that accounts for the modeling of both the algorithmic and architectural specifications. Aiming at the generation of code that is more easily analysable by construction, we deepened our investigation on two MDE modelling frameworks: a combination of Scicos and GeneAuto to automatically generate analysable functional code; and SCM to generate predictable architectural code.

We further exploited the MDE paradigm to collect any valuable model-level timing information and to automatically generate flow-fact annotations, for an effective application of timing analysis.

Incremental cache-aware memory layout optimisation. We investigated on the memory layout as the most impacting source of cache variability. In Chapter 3.7 we presented and evaluated an approach intended to govern the I-cache variability stemming from changes in the memory layout incurred across incremental software releases. We focused on industrial applicability of layout optimisations and proposed a novel structure-based procedure placement technique to enforce an optimised memory layout which is preserved across incremental releases of the system.

We introduced a prototype tool that allows a fully automated application of our technique. By providing early guarantees on the cache behaviour, our approach enables the application of state-of-the-art WCET analysis techniques to collect trustworthy timing information already in the early stages of the development process.

The combination of the above contributions allowed us to define a structured approach that fulfils our initial objectives. The enforcement of analysable code patterns and design choices and the provision of automatically defined flow-fact annotations allows a cost-

effective application of state-of-the-art WCET analysis. More analysable code, in fact, is expected to both improve the quality of the results of timing analysis and reduce the onerous and error-prone user intervention in the analysis process. The application of our incremental layout optimisation, offers a means to reduce the variability and unpredictability incurred by caches and, at the same time, enables an earlier application of WCET analysis on incremental releases, instead of the final system. Our approach lends itself to a relatively easy integration within the consolidated industrial development, thus fulfilling a fundamental industrial requirement: both timing-aware code generation and layout optimisation, in fact, rely on automated tool support that do not require particular skills, that are not already employed in the industrial development process.

4.3 Future work

During our investigations we stepped into a number of open problems that hinder a cost-effective application of timing analysis within the industrial software life-cycle. Although we were able to propose some promising approaches we are still aware that a lot of research directions need to be investigated.

The most natural continuation of our approach would consist in its evaluation on the development cycle of a realistic part of an industrial software. The main difficulties that prevented us to conduct this experimental evaluation, besides the actual time and effort that it would have required, were related to the limitations entailed by the use of specific modeling tools. However, the SCM Editor is currently under completion and we expect that we will soon be able to extend our investigation.

In respect to the idea of a timing-aware MDE approach, it could be also interesting to extend our investigation to different tools and formalisms. During our investigation, we had to face some limitations that were ascribable to the tool architecture and implementation. For example, the bridge between Scicos and GeneAuto prevented us from exploiting model-level user annotations. A challenging but interesting direction would consist in implementing a fully customisable modeling tool so as to fully exploit the MDE paradigm. When it comes to the modeling formalism, we already observed that each formalism may offer new spaces of intervention and new solutions. In particular, we expect that the use of a state-based formalism (e.g., Stateflow or state-machines) may allow to collect valuable flow facts for the exclusion of infeasible paths, which are a known source of overestimation in flow analysis.

Our investigation has been mainly focused on source-level software analysability, un-

der simplifying assumptions on the role of compilers: in particular, we assumed the avoidance of complex compiler transformations so that the program control-flow would be basically preserved in the final object code. This notwithstanding, we are aware of the role of compilers in determining both analysability and predictability of a program. We consider the new plug-in infrastructure exploited by standard GCC-based compilers to be worth studying in the view of timing analysability, in a similar way to what has been done in [90]. The same plug-in infrastructure could be also exploited to further develop our layout optimisation approach as an inner compiler optimisation.

We will also continue our cooperation within the FP7 EU founded Probabilistically Analysable Real-Time Systems (PROARTIS) project [121], in which we have been recently involved. PROARTIS is proposing an innovative approach based on probabilistic hardware and analysis techniques to promotes a paradigm shift that aims to eradicate the dependencies of timing behaviour on the execution history, while at the same time benefiting from complex hardware acceleration features. Our investigation within this project is mainly focused on removing the sources of timing dependencies affecting and stemming from kernel-level services.

Bibliography

[1] Absint GmbH. aiT WCET Analyzer, 2011. http://www.absint.com/ait/.

[2] AdaCore. GNAT Pro Toolsuite, 2011. http://www.adacore.com/home/products/gnatpro/overview/.

[3] Aeroflex Gaisler. LEON2 Processor Family, 2011. http://www.gaisler.com.

[4] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Prentice Hall, 2 edition, 2006.

[5] S. Alstrup, D. Harel, P. W. Lauridsen, and M. Thorup. Dominators in linear time. *SIAM Journal of Computing*, 28(6):2117–2132, June 1999.

[6] S. Altmeyer and C. Burguiere. A new notion of useful cache block to improve the bounds of cache-related preemption delay. In *Proceedings of the 21st Euromicro Conference on Real-Time Systems (ECRTS)*, 2009.

[7] S. Altmeyer and G. Gebhard. Wcet analysis for preemptive scheduling. In *Proceedings of the 8th International Workshop on WCET Analysis*, 2008.

[8] S. Altmeyer, C. Maiza, and J. Reineke. Resilience analysis: Tightening the crpd bound for set-associative caches. In *LCTES '10: Proceedings of the ACM SIGPLAN/SIGBED 2010 conference on Languages, compilers, and tools for embedded systems*, pages 153–162. ACM, 2010.

[9] J. Anderson, J. Calandrino, and U. Devi. Real-Time Scheduling on Multicore Platforms. In *Proc. of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2006.

[10] ARM Embedded Trace Macrocell, 2011. http://www.arm.com/products/solutions/ETM.html.

[11] Atmel Corp. *TSC695F SPARC 32-bit Space Processor*, rev. 4148h-aero-12/03 edition, 2003.

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

[13] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING*, 1(JANUARY-MARCH 2004):11–33, 2004.

[14] T. P. Baker. Stack-based Scheduling for Realtime Processes. *Real-Time Systems*, 3(1):67–99, 1991.

[15] C. Ballabriga, H. Casse, and P. Sainrat. An improved approach for set-associative instruction cache partial analysis. In *Proc. of the 2008 ACM symposium on Applied Computing (SAC 08)*, pages 360–367. ACM, 2008.

[16] D. Barkah, A. Ermedahl, J. Gustafsson, B. Lisper, and C. Sandberg. Evaluation of Automatic Flow Analysis for WCET Calculation on Industrial Real-Time System Code. In *Proceedings of the 20th Euromicro Conference on Real-Time Systems (ECRTS)*, 2008.

[17] J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[18] 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.

[19] 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.

[20] 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.

[21] A. Betts, G. Bernat, R. Kirner, P. Puschner, and I. Wenzel. WCET Coverage for Pipelines. Technical report, ARTIST2 Network of Excellence, 2006.

[22] A. Betts, A. Colin, N. Holsti, Y. Gouy, G. Garcia, C. Moreno, and T. Vardanega. Prototype Execution-time Analyser for the LEON (PEAL2) Final Report. Technical Report RS-PEAL2-001, ESA/ESTEC, 2009.

[23] M. Bordin, M. Panunzio, and T. Vardanega. Fitting Schedulability Analysis Theory into Model-Driven Engineering. In *Proc. of the 20th Euromicro Conference on Real-Time Systems*, pages 135–144, 2008.

[24] M. Bordin and T. Vardanega. Correctness by Construction for High-Integrity Real-Time Systems: a Metamodel-driven Approach. In *Proceedings of Reliable Software Technologies - Ada-Europe*, 2007.

[25] L. 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.

[26] C. Burguière, J. Reineke, and S. Altmeyer. Cache-related preemption delay computation for set-associative caches - pitfalls and solutions. In *Proc. of the 9th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, 2009.

[27] A. Burns, B. Dobbing, and T. Vardanega. Guide for the Use of the Ada Ravenscar Profile in High Integrity Systems. *Technical Report YCS-2003-348, University of York*, 2003.

[28] 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.

[29] S. Bydge and B. Lisper. Towards an Automatic Parametric WCET Analysis. In *Proc. of the 8th Int. Workshop on Worst-Case Execution Time Analysis (WCET)*, 2008.

[30] S. Byhlin, A. Ermedahl, J. Gustafsson, and B. Lisper. Applying Static WCET Analysis to Automotive Communication Software. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems*, 2005.

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

[32] CENELEC. EN 50128:2011 - Software for railway control and protection systems, 2011.

[33] J. Chang and G. Sohi. Cooperative cache partitioning for chip multiprocessors. In *Proc. of the 21st annual international conference on Supercomputing*, 2006.

[34] J. Chang and G. Sohi. Cooperative caching for chip multiprocessors. In *Proc. of the 33th Annual International Symposium on Computer Architecture (ISCA)*, 2006.

[35] S. Chattopadhyay and A. Roychoudhury. Unified cache modeling for WCET analysis and layout optimizations. In *Proc. of the 30th IEEE Real-Time Systems Symposium*, RTSS '09, pages 47–56, 2009.

[36] CHESS: Composition with guarantees for High-integrity Embedded Software components assembly . ARTEMIS Joint Technology Initiative (JTI) 2009-2011 `http://http://chess-project.ning.com/`.

[37] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1977.

[38] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621 –645, 2006.

[39] E. W. Dijkstra. On the role of scientific thought. In *Selected writings on Computing: A Personal Perspective*, pages 60–66, Springer–Verlag Ed., 1982.

[40] Eclipse. Eclipse Modeling Project, 2012. `http://www.eclipse.org/modeling/`.

[41] A. Ermedahl, C. Sandberg, J. Gustafsson, S. Bygde, and B. Lisper. Loop bound analysis based on a combination of program slicing, abstract interpretation, and invariant analysis. In *Proc. of the 7th Intl. Workshop on Worst-Case Execution Time Analysis (WCET)*, 2007.

[42] ESTEREL SCADE© Suite, 2011. `http://www.estereltechnologies.com/products/scadesuite/`.

[43] ETAS. ASCET, 2011. `http://www.etas.com/en/products/ascet_software_products.php`.

[44] European Cooperation for Space Standardization (ECSS). ECSS-E-ST-40C - Space engineering - Software, 2009.

[45] European Space Agency. ESA space & science: GAIA, 2011. `http://gaia.esa.int`.

[46] H. Falk and M. Schwarzer. Loop nest splitting for wcet-optimization and predictability improvement. In *Proc. of the 6th International Workshop on Worst-Case Execution Time Analysis (WCET)*, 2006.

[47] C. Ferdinand, R. Heckmann, T. Sergent, D. Lopes, B. Martin, X. Fornari, and F. Martin. Combining a high-level design tool for safety-critical systems with a tool for WCET analysis

on executables. In *Proceedings of the 4th European Congress on Embedded Real Time Software and Systems*, 2008.

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

[49] J. Fredriksson, T. Nolte, A. Ermedahl, and M. Nolin. Clustering Worst-Case Execution Times for Software Components. In *Proc. of the 7th International Workshop on Worst-Case Execution Time Analysis*, 2007.

[50] G. Gebhard, C. Cullmann, and R. Heckmann. Software structure and WCET predictability. In P. Lucas, L. Thiele, B. Triquet, T. Ungerer, and R. Wilhelm, editors, *Bringing Theory to Practice: Predictability and Performance in Embedded Systems*, volume 18 of *OpenAccess Series in Informatics (OASIcs)*, pages 1–10, Dagstuhl, Germany, 2011. Schloss Dagstuhl.

[51] GeneAuto/Ada. Verifying model compiler for Simulink/StateFlow and Scicos models, 2011. `http://www.open-do.org/projects/geneautoada/`.

[52] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations:a compiler framework for analyzing and tuning memory behavior. *ACM Transactions on Programming Languages and Systems*, pages 703–746, 1999.

[53] N. Gloy and M. D. Smith. Procedure placement using temporal-ordering information. *ACM Trans. Program. Lang. Syst.*, 21(5):977–1027, 1999.

[54] J. Gustafsson and A. Ermedahl. Experiences from Applying WCET Analysis in Industrial Settings. In *Proceedings of the 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, 2007.

[55] S. P. H. Falk and H. Theiling. Compile time decided instruction cache locking using worst-case execution paths. In *Proc. of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2007.

[56] D. Hardy and I. Puaut. Predictable paging in real-time systems: an ILP formulation. In *Ecole d'été Temps Réel (ETR'07)*, 2007.

[57] D. Hardy and I. Puaut. WCET Analysis of Multi-level Non-inclusive Set-associative Instruction Caches. In *Proc. of the 2008 Real-Time Systems Symposium (RTSS)*, 2008.

[58] A. H. Hashemi, D. R. Kaeli, and B. Calder. Efficient procedure mapping using cache line coloring. *ACM SIGPLAN Notices*, 32(5):171–182, 1997.

[59] 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.

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

[61] J. Herter, P. Backes, F. Haupenthal, and J. Reineke. CAMA: A predictable cache-aware memory allocator. In *Proceedings of the 23rd Euromicro Conference on Real-Time Systems (ECRTS '11)*. IEEE Computer Society, July 2011.

[62] N. Holsti. Computing time as a program variable: a way around infeasible paths. In *Proc. of the 8th Int. Workshop on Worst-Case Execution Time Analysis (WCET)*, 2008.

[63] N. Holsti, T. Långbacka, and S. Saarinen. Worst-case execution time analysis for digital signal processors. In *European Signal Processing Conference (EUSIPCO 2000)*, 2000.

[64] International Electrotechnical Commission (IEC). Software for Computers in the Safety Systems of Nuclear Power Stations, Standard IEC-880, 1986.

[65] Iridium Communications Inc. Iridium NEXT Satellite Constellation, 2011. http://iridium.com/About/IridiumNEXT.aspx.

[66] ISO/IEC. Information technology —Meta Object Facility (MOF). ISO/IEC 19502, 2005.

[67] P. Jain, S. Devadas, D. Engels, and L. Rudolph. Software-assisted cache replacement mechanism for embedded systems. In *Proc. of the Int. Conference on Computer Aided Design (ICCAD)*, 2001.

[68] M. Joseph and P. K. Pandya. Finding response times in a real-time system. *The Computer Journal*, 29(5):390–395, 1986.

[69] L. Ju, B. Khoa, H. Abhik, and R. S. Chakraborty. Performance debugging of Esterel specifications. In *In International Conference on Hardware Software Codesign and System Synthesis (CODES-ISSS*, 2008.

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

[71] D. Kirk, J. Strosnider, and J. Sasinowski. Allocating SMART cache segments for schedulability. In *Proc. of the EUROMICRO Workshop on Real Time Systems*, 1991.

[72] R. Kirner, R. Lang, G. Freiberger, and P. Puschner. Fully Automatic Worst-Case Execution Time Analysis for Matlab/Simulink Models. In *Proc. of the 14th Euromicro Conference on Real-time Systems (ECRTS)*, pages 31–40, 2002.

[73] R. Kirner and P. Puschner. Transformation of Path Information for WCET Analysis during Compilation. In *Proc. of the 13th Euromicro Conference on Real-time Systems (ECRTS)*, 2001.

[74] R. Kirner and P. Puschner. Classification of Code Annotations and Discussion of Compiler-Support for Worst-Case Execution Time Analysis. In *Proc. of the 5th International Workshop on Worst-Case Execution Time Analysis*, 2005.

[75] C. Lee, J. Hahn, Y. Seo, S. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Transactions on Computers*, 47(6):700–713, 1998.

[76] C. G. Lee, K. Lee, J. Hahn, Y. Seo, S. L. Min, R. Ha, S. Hong, C. Park, M. Lee, and C. S. Kim. Bounding cache-related preemption delay for real-time systems. *IEEE Transaction on Software Engineering*, 27(9):805–826, 2001.

[77] Lee, C. et al. Enhanced Analysis of Cache-related Preemption delay in Fixed-priority Preemptive Scheduling. Technical report, Dep. of Computer Engineering, Seoul National University, 1997.

[78] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1:121–141, 1979.

[79] B. Lesage, D. Hardy, and I. Puaut. WCET analysis of multi-level set-associative data caches. In *Proc. of the 9th Int. Workshop on Worst Case Execution Time Analysis*, 2009.

[80] X. Li, Y. Liang, T. Mitra, and A. Roychoudhury. Chronos: A timing analyzer for embedded software. In *Science of Computer Programming*, 2007.

[81] X. Li, A. Roychoudhury, and T. Mitra. Modeling out-of-order processors for wcet estimation. *IEEE Real-Time Systems Journal*, 34(3), 2006.

[82] Y. 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.

[83] Y. Li, S. Malik, and A. Wolfe. Minimizing WCET for Real-Time Embedded Systems via Static Instruction Cache Locking. In *Proc. of the 15th Real-Time and Embedded Technology and Applications Symposium*, 2009.

[84] B. Lisper, A. Ermedahl, D. Schreiner, J. Knoop, and P. Gliwa. Practical experiences of applying source-level WCET flow analysis on industrial code. In *Proc. of the 4th Int. Conference on Leveraging applications of formal methods, verification, and validation*, ISoLA'10, pages 449–463. Springer-Verlag, 2010.

[85] B. Lisper and M. Santos. Model Identification for WCET Analysis. In *Proc. of the 15th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2009.

[86] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *J. ACM*, 20:46–61, 1973.

[87] J. W. S. Liu. *Real-Time Systems*. Prentice Hall PTR, 2000.

[88] P. Lokuciejewski, H. Falk, and P. Marwedel. Tighter WCET Estimates by Procedure Cloning. In *Proc. of the 7th International Workshop on Worst-Case Execution Time Analysis (WCET)*, 2008.

[89] P. Lokuciejewski, H. Falk, and P. Marwedel. WCET-driven Cache-based Procedure Positioning Optimizations. In *Proc. of the 20th Euromicro Conference on Real-Time Systems (ECRTS)*, 2008.

[90] P. Lokuciejewski and P. Marwedel. *Worst-Case Execution Time Aware Compilation Techniques for Real-Time Systems*. Springer, November 2010.

[91] T. Lundqvist and P. Stenström. A Method to Improve the Estimated Worst-Case Performance of Data Caching. In *Proc. of the 20th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'99)*, 1999.

[92] T. Lundqvist and P. Stenström. Timing anomalies in dynamically scheduled microprocessors. In *Proc. of the 20th Real-Time Systems Symposium (RTSS'99)*, 1999.

[93] Mälardalen University. SWEET, the SWEdisch Execution-time Tool. *http://www.mrtc.mdh.se/projects/wcet/sweet.html*.

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

[95] MathWorks Automotive Advisory Board (MAAB). Control algorithm modeling guidelines using MATLAB, Simulink, and Stateflow - Version 2.1, 2007. `http:\www.mathworks.com`.

[96] Mathworks Simulink©, 2011.
http://www.mathworks.com/products/simulink.

[97] Mathworks Stateflow©, 2011.
http://www.mathworks.com/products/stateflow.

[98] T. Mens, K. Czarnecki, and P. V. Gorp. A taxonomy of model transformations. In *Language Engineering for Model-Driven Software Development*, Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2005.

[99] S. Metzlaff. Predictable dynamic instruction scratchpad for simultaneous multithreaded processors. In *Proc. of the 9th Workshop on Memory Performance: Dealing with Applications, Systems and Architecture*, 2008.

[100] E. Mezzetti, N. Holsti, A. Colin, G. Bernat, and T. Vardanega. Attacking the sources of unpredictability in the instruction cache behavior. In *Proc. of the 16th Internat. Conference on Real-Time and Network Systems (RTNS08)*, 2008.

[101] E. Mezzetti and T. Vardanega. Towards a Cache-aware Development of High Integrity Real-time Systems. In *Proceedings of the 16th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2010)*, pages 329–338. IEEE Computer Society Press, August 2010.

[102] E. Mezzetti and T. Vardanega. Cache Optimisations for LEON Analyses (COLA) Final Report. Technical Report COLA-FR-001-i1r1, ESA/ESTEC, 2011.

[103] E. Mezzetti and T. Vardanega. On the industrial fitness of WCET analysis. In *Proceedings of the 11th International Workshop on Worst-Case Execution Time Analysis (WCET 2011)*. OCG, Austrian Computer Society, July 2011.

[104] J. Mitchell. *Concepts in Programming Languages*. Cambridge University Press, 2003.

[105] Motor Industry Software Reliability Association (MISRA). Guidelines for the use of the C language in critical systems, 2004.

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

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

[108] F. Mueller. Timing predictions for multi-level caches. In *ACM SIGPLAN Workshop on Languages, Compilers and Tools SUpport for Real-Time Systems*, 1997.

[109] 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.

[110] Multi-Core Execution of Hard Real-Time Applications Supporting Analysability (MERASA), 2011. `http://www.merasa.org`.

[111] F. Nemer, H. Cassé, P. Sainrat, and J. Bahsoun. Inter-Task WCET Computation for A-way Instruction Caches. In *Proc. of the Int. Symposium on Industrial Embedded Systems(SIES)*, 2008.

[112] Nexus. The Nexus© 5001 Forum,. `http://www.nexus5001.org`.

[113] Object Management Group. CORBA Component Model, v4.0, 2006. `http://www.omg.org/technology/documents/formal/components.htm`.

[114] OMG. Meta Object Facility (MOF) Specification, v. 2.4.1, August 2011. `http://www.omg.org/spec/MOF/2.4.1/`.

[115] M. Panunzio. Definition, realization and evaluation of a software reference architecture for use in space applications. Technical Report UBLCS-2011-07, Department of Computer Science, University of Bologna, 2011.

[116] M. Panunzio. Space Component Model (SCM) metamodel 0.4.0, November 2011.

[117] K. Patil, K. Seth, and F. Mueller. Compositional static instruction cache simulation. In *Proc. of the SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 2004.

[118] S. Petters and G. Färber. Making Worst Case Execution Time Analysis for Hard Real-Time Tasks on State of the Art Processors Feasible. In *Proc. of the 6th Int Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 1999.

[119] K. Pettis and R. C. Hansen. Profile guided code positioning. *ACM SIGPLAN Notices*, 25(6):16–27, 1990.

[120] M. Prieto, D. Guzmán, D. Meziat, S. Sánchez, and L. Planche. LEON2 cache characterization. A contribution to WCET determination. In *IEEE Int. Symposium on Intelligent Signal Processing (WISP)*, 2007.

[121] PROARTIS: PRObabilistically Analysable real-Time System. EU FP7 2010-2013 http://http://www.proartis-project.eu/.

[122] 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.

[123] 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.

[124] 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.

[125] I. Puaut and J. Deverge. Safe Measurement-based WCET Estimation. In *Proc. of the 5th Int. Workshop on WCET Analysis*, 2005.

[126] I. Puaut and D. Hardy. Predictable Paging in Real-time Systems: a Compiler Approach. In *Proceedings of the 19th Euromicro Conference on Real-Time Systems*, 2007.

[127] 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.

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

[129] P. Puschner and M. Schoeberl. On composable system timing, task timing, and wcet analysis. In *Proc. of the 8th Int. Workshop on WCET Analysis*, 2008.

[130] A. Rakib, O. Parshin, S. Thesing, and R. Wilhelm. Component-wise instruction-cache behavior prediction. In *Automated Technology for Verification and Analysis*, 2004.

[131] H. Ramaprasad and F. Mueller. Bounding worst-case data cache behavior by analytically deriving cache reference patterns. In *Proc. of Real Time and Embedded Technology and Applications Symposium*, 2005.

[132] H. Ramaprasad and F. Mueller. Bounding worst-case response time for tasks under PIP. In *Proc. of the Real-Time and Embedded Technology and Applications Symposium*, 2009.

[133] Rapita Systems Ltd. Rapitime©, 2011. http://www.rapitasystems.com/rapitime.

[134] J. Reineke and D. Grund. Relative Competitiveness of Cache Replacement Policies. In *Proc. of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 2008.

[135] J. Reineke, D. Grund, C. Berg, and R. Wilhelm. A definition and classification of timing anomalies. *Proc. of the 6th Int. Workshop on WCET Analysis*, 2006.

[136] J. Reineke, D. Grund, C. Berg, and R. Wilhelm. Timing predictability of cache replacement policies. *Real-Time Syst.*, 37:99–122, November 2007.

[137] M. Rodríguez, N. Silva, J. Esteves, L. Henriques, D. Costa, N. Holsti, and K. Hjortnaes. Challenges in Calculating the WCET of a Complex On-board Satellite Application. In *In Proceeding of the 3rd Intl. Workshop on Worst-Case Execution Time Analysis (WCET03)*, 2003.

[138] RTCA SC-167/EUROCAE WG-12. RTCA/DO-178B: Software considerations in airborne systems and equipment certification, 1992.

[139] A. E. Rugina, D. Thomas, X. Olive, and G. Veran. Gene-Auto: Automatic Software Code Generation for Real-Time Embedded Systems. In *Proc. of DAta Systems In Aerospace (DASIA)*, 2008.

[140] J. Sasinowsky and J. Strosnider. A dynamic programming algorithm for cache/memory partitioning for real-time systems. *IEEE Transactions on Computers*, 42:997–1001, 2006.

[141] D. Schmidt. Model-driven engineering. *IEEE Computer*, 39(2):25–31, 2006.

[142] J. Schneider. Why You Can't Analyze RTOSs without Considering Applications and Vice Versa. In *In Proc. of the 2nd International Workshop on Worst-Case Execution Time Analysis (WCET 2002)*, 2002.

[143] SCICOS. Graphical dynamical system modeler and simulator, 2011. `www.scicos.org`.

[144] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Trans. Computers*, 39(9):1175–1185, 1990.

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

[146] J. Souyris, E. Le Pavec, G. Himbert, G. Borios, V. Jgu, and R. Heckmann. Computing the worst case execution time of an avionics program by abstract interpretation. In *Proc. of the 5th Intl. Workshop on Worst-Case Execution Time Analysis (WCET05)*, 2005.

[147] J. Staschulat and R. Ernst. Scalable precision cache analysis for preemptive scheduling. In *Proc. of the 2005 ACM SIGPLAN/SIGBED conference on Languages*, 2005.

[148] V. Suhendra and T. Mitra. Exploring Locking & Partitioning for Predictable Shared Caches on Multi-Cores. In *Proc. of the 45th annual conference on Design automation*, 2008.

[149] P. Szulman. WCET Analysis of Data Dependent, Component Oriented, Embedded Software Systems. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 203(7), 2009.

[150] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.

[151] L. Tan, B. Wachter, P. Lucas, and R. Wilhelm. Improving timing analysis for matlab simulink/stateflow. In *Proceedings of the 2nd International Workshop on Model Based Architecting and Construction of Embedded Systems (ACES-MB)*, pages 59–63, 2009.

[152] Y. Tan and V. Mooney. Integrated intra- and inter-task cache analysis for preemptive multi-tasking real-time systems. In *Proc. of the 8th International Workshop on Software and Compilers for Embedded Systems (SCOPES)*, 2004.

[153] Y. Tan and V. Mooney. A prioritized cache for multi-tasking real-time systems. In *Proc. of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, 2005.

[154] H. Theiling and C. Ferdinand. Combining abstract interpretation and ilp for microarchitecture modelling and program path analysis. In *Proc. of the 19th IEEE Real-Time Systems Symposium*, 1998.

[155] Tidorum Ltd. Bound-T tool,. `http://www.bound-t.com`.

[156] H. Tomiyama and H. Yasuura. Code placement techniques for cache miss rate reduction. *ACM Trans. Des. Autom. Electron. Syst.*, 2(4):410–429, 1997.

[157] Universidad Politécnica de Madrid. GNAT/ORK+ for LEON cross-compilation system. `http://polaris.dit.upm.es/~ork`, 2011.

[158] T. Vardanega. Property Preservation and Composition with Guarantees: From ASSERT to CHESS. In *Proceedings of the 12th International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, 2009.

[159] 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.

[160] X. Vera, B. Lisper, and J. Xue. Data cache locking for tight timing calculations. *ACM Trans. Embed. Comput. Syst.*, 7(4):4:1–4:38, December 2007.

[161] I. Wenzel, R. Kirner, M. Schlager, B. Rieder, and B. Huber. Impact of dependable software development guidelines on timing analysis. In *Proc. of the 2005 IEEE Eurocon Conference*, pages 575–578, 2005.

[162] R. White, C. Healy, D. Whalley, F. Mueller, and M. Harmon. Timing analysis for data caches and set-associative caches. In *Real-time Technology and Application Symposium (RTAS'97)*, 1997.

[163] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G.Bernat, C. Ferdinand, R.Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, G. Staschulat, and P. Stenström. The worst-case execution time problem: overview of methods and survey of tools. *Trans. on Embedded Computing Systems*, 7(3):1–53, 2008.

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

[165] V. Yodaiken. Against Priority Inheritance. Technical report, Finite State Machine Labs, 2004.

[166] J. Zamorano and J. de la Puente. Kernel Support for Reducing Cache Refills in Response Time Schedulability Analysis. In *Proc. of the 23rd Int. Real-Time Systems Symposium*, 2002.

Appendix A

Bounds on CRBD

A.1 CRBD computation

In the following we first provide a formal characterisation of the CRBD potentially incurred by a task; and then we exploit well-known bounds on the number of blocking events suffered by a task, under different resource access protocols

Assumptions. In the following we assume total ordering between tasks such that $i < j$ if $\pi(\tau_i) > \pi(\tau_j)$: hence τ_0 is the highest priority task. A task τ_i self-suspends only at the end of every execution of its jobs, and may access a shared resource $R \in SR_i$, where SR_i identifies the subset of the system resources (SR) that get accessed by τ_i.

Since a task τ_i may access a shared resource R through different critical sections, we define cs_i^R to be the set of critical sections in τ_i accessing the resource $R \in SR_i$. Similarly, $cs_{i,k}^R$ identifies the k^{th} critical section in τ_i accessing the resource $R \in SR_i$. In any case, we assume critical sections to be properly nested so that they can never overlap. For every pair of critical sections $cs_{i,k}$, $cs_{i,z}$ in τ_i either $cs_{i,k} \subset cs_{i,z}$, $cs_{i,k} \supset cs_{i,z}$ or $cs_{i,k} \cap cs_{i,z} = \emptyset$.

CRBD computation. The determination of the CRBD incurred by a task exploits similar concepts as when computing the CRPD, involving the computation of UCB and $\overline{UCB}$ for blocked and blocking task respectively. Useful and used blocks are defined as follows:

- *Useful Cache Blocks* (UCB): cache blocks that may be referenced again before they could be evicted by other memory blocks, according to the cache replacement policy;

- *Used Cache Blocks* ($\overline{UCB}$): cache blocks that may be accessed during the execution of the preempting task.

We recall that the set of UCB and $\overline{UCB}$ for a task τ_i are dependent on each specific node n in the Control-Flow Graph (CFG) of τ_i, where each node represents a basic block. In fact, UCB and $\overline{UCB}$ can be safely computed at basic block level, as proved in [75].

According to [76, 152, 147] the UCB_i^n for a task τ_i at node *n* can be computed as the intersection between the sets of *ReachingBlocks* (RB) and *LiveBlocks* (LB) at node *n* where RB is the set of cache blocks potentially cached at node N, whereas LB is the set of blocks that could potentially be reused in the successors of *n*. Intuitively, instead, $\overline{UCB}_i^n$ can be computed as $RB_i(n)$. Thus, $UCB_i^n = RB_i(n) \bigcap LB_i(n)$ and $\overline{UCB}_i^n = RB_i(n)$.

In case of blocking, we are interested in determining UCB and $\overline{UCB}$ for a task τ_i blocked on a critical section $cs_{i,k}^R$. For example, let us consider a simple case of direct blocking between two tasks. Task τ_i is blocked when trying to access critical section $cs_{i,k}^R$ because a lower-priority task τ_j is executing inside a critical section $cs \in cs_j^R$ accessing the same shared resource R. In this case, the set of UCB for the blocked task τ_i is to be computed with respect to the node n_R trying to enter $cs_{i,k}^R$.

$$UCB_{i,k}^R = RB_i(n_R) \cap LB_i(n_R), \text{ where } n_R \text{ is the entry node of } cs_{i,k}^R$$

The set of $\overline{UCB}$ for the blocking task τ_j must be computed with respect to the critical section $cs_{j,h}^R$ it is executing within, as only the RBs in $cs_{j,h}^R$ can affect the cache state of τ_i. For this reason, we extend the notion of RB to address *intervals* of nodes in the CFG instead of single nodes.

Given an interval $[n_1, n_2] = \mathcal{I} \in CFG(\tau_i)$, we define $RB_i(\mathcal{I})$ as the contribution to $RB(n_2)$ of all possible paths in $CGF(\tau_i)$ from node n_1 to n_2. Accordingly,

$$\overline{UCB}_j(cs_{j,h}^R) = RB_j(cs_{j,h}^R) = RB_j([first\_node, last\_node]_{cs_{j,h}^R})$$

In the example, the execution of τ_j inside $cs_{j,h}^R$ may evict some useful cache blocks that τ_i may have loaded in the cache before its attempt to enter $cs_{i,k}^R$. The incurred CRBD can be computed as a function of the $UCB_{i,k}^R$ and $\overline{UCB}_{j,h}^R$ terms just defined:

$$CRBD = \otimes_\sigma \Big(UCB_{i,k}^R, \overline{UCB}_j(cs_{j,h}^R) \Big) \times \text{miss penalty} \tag{A.1}$$

where the $\otimes_\sigma$ operator accounts for the actual cache associativity and replacement policy in combining the information on useful and used cache blocks, cf. [152, 6]. For example,

for direct-mapped caches, $\otimes_{DM}(UCB, \overline{UCB})$ will include those cache sets which at least one cache block in both UCB and $\overline{UCB}$ is mapped to (set intersection). For LRU *n*-way set-associative caches, instead, the $\otimes_{LRU}$ operator must account for the number of additional cache misses for each cache set. In case of a non-empty $\overline{UCB}$ set, those misses are bounded by the minimum between the cache associativity (*n*) and the number of UCB mapping to that cache set [26].

In case τ_i and τ_j share more than one resource, we can generalize Equation A.1 to determine an upper bound on the delay suffered by τ_i, due to a *single* direct blocking by τ_j for any critical section accessing any shared resource as follows:

$$CRBD_{i,j} \leq \max_{\substack{R \in SR_i, k \in [1, |cs_i^R|] \\ cs \in cs_j^R}} \left\{ \otimes_\sigma \left(UCB_{i,k}^R, \overline{UCB}_j(cs) \right) \right\} \times \text{miss penalty} \tag{A.2}$$

However, Equation A.2 just holds in this simple case where neither transitive direct blocking nor other types of blocking are taken into account. In terms of CRBD, determining the effects of inheritance blocking is much more complex, as the computation of UCB for the blocked task cannot make any simplifying assumption on when the task actually gets blocked.

A more comprehensive bound on the CRBD incurred by a task can be computed by leveraging on the bounds that a specific resource access protocol places on blocking. An upper bound on the worst-case number of blocking events incurred by a task is given in [144, 14] for each protocol. That bound is then combined with the worst-case duration of each critical section to derive a bound on the blocking time potentially suffered by a task. Those bounds typically rely on the notion of potentially blocking critical sections to account for any type of blocking that may occur under the protocol itself. To this end, $\beta_{i,j}$ is defined in [144] as the set of critical sections of a lower-priority task τ_j which can block τ_i in any way. The bounds on the number of blocking events and blocking time exploit the $\beta_{i,j}^*$ set which identifies the set of *outermost* critical sections of τ_j that can block τ_i. More formally:

$$\beta_{i,j}^* = \{(cs_{j,k} | cs_{j,k} \in \beta_{i,j}) \wedge (\neg \exists cs_{j,m} \in \beta_{i,j}, cs_{j,k} \subset cs_{j,m})\}$$

We will exploit the same concepts, with the only difference that we are not interested in the critical section that may incur the maximum blocking time since we focus on the CRBD, which is independent of the duration of the critical section. Instead, we are interested in the critical section $cs_{j,k} \in \beta_{i,j}^*$ which causes the eviction of the greatest number of useful blocks for the blocked task, for all lower-priority tasks τ_j.

In the following, we will combine given bounds on the number of blocking events with the same concepts as used in CRPD analysis to provide a *safe* upper bound on the CRBD under different protocols.

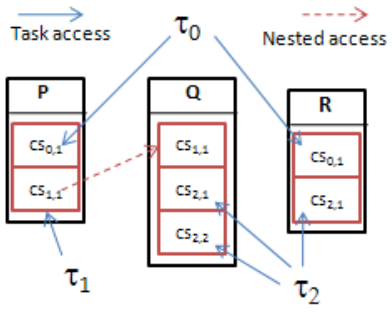
A.2 CRBD under the Priority Inheritance Protocol

When access to shared resources is managed with PIP [144], whenever a task that holds the lock of a resource blocks a higher-priority task, it inherits the priority of the highest-priority task it is blocking. When a task releases the lock of a resource, its priority is lowered to the *highest inherited* priority value [165].

PIP is interesting as it does bound priority inversion and also does not require any knowledge on the system's tasks and their priorities, since the priority value to inherit is determined dynamically. Unfortunately, PIP does not prevent deadlock (which may occur in case of nested critical sections) and a task can be blocked multiple times during a single activation. In fact, a task τ_i can be blocked for the duration of at most $\min(n, m)$ outermost critical sections, where n is the number of lower-priority tasks that may block τ_i and m is the number of semaphores that can be used to block τ_i. In the following we re-elaborate both bounds from the standpoint of the CRBD.

Bound on lower priority tasks. Under PIP, a high priority task τ_H can be blocked by a lower priority task τ_L for at most the duration of one critical section of $\beta^*_{H,L}$. Therefore, given a task τ_i for which there are n lower priority tasks $\{\tau_{i+1}, \ldots, \tau_{i+n}\}$, τ_i can be blocked for at most the duration of one critical section in each $\beta^*_{i,k},\ i+1 \leq k \leq i+n$ [144].

If we assume that all shared resources and critical sections are statically known, we can define a resource access graph and table, similar to that shown in Figure A.1.



	P	Q	R
τ_0	$cs^P_{0,1}$		$cs^R_{0,1}$
τ_1	$cs^P_{1,1}$	$cs^Q_{1,1}$	
τ_2		$cs^Q_{2,1}$ $cs^Q_{2,2}$	$cs^R_{2,1}$

Figure A.1: Resource graph and corresponding resource access table.

Note that the critical section $cs_{1,1}$ of resource P performs a nested access to critical section $cs_{1,1}$ of resource Q. In this case, the $\beta_{i,j}$ sets derived from Table A.1 are as follows: $\beta_{0,1} = \{cs^P_{1,1}, cs^Q_{1,1}\}$ due to resource nesting, $\beta_{1,2} = \{cs^Q_{2,1}, cs^Q_{2,2}, cs^R_{2,1}\}$ (by inheritance blocking), and $\beta_{0,2} = \{cs^R_{2,1}, cs^Q_{2,1}, cs^Q_{2,2}\}$ as τ_2 could transitively block τ_0 by blocking τ_1. The $\beta^*_{i,j}$ sets, instead, removes redundant innermost critical sections; thus, for example, $\beta^*_{0,1} = \{cs^P_{1,1}\}$.

We recall that computing the UCB of the blocked task τ_i in case of inheritance blocking needs to consider any possible node in $CFG(\tau_i)$, similarly to task preemption. To avoid the overestimation in considering all possible nodes, we will threat inheritance blocking separately.

An upper bound on the CRBD in case of direct blocking of τ_i due to τ_j is the maximum $\otimes_\sigma$ applied to UCB and $\overline{UCB}$ for any resource accessed by τ_i, every critical section in τ_i accessing that resource and every outermost critical section of τ_j potentially blocking τ_i. Hence, it can be formalized as:

$$CRBD^{base}_{i,j} \leq \max_{\substack{R \in SR_i, k \in [1, |cs^R_i|] \\ cs \in \beta^*_{i,j}}} \left\{ \otimes_\sigma \left(UCB^R_{i,k}, \overline{UCB}_j(cs) \right) \right\} \times \text{miss penalty} \tag{A.3}$$

With regard to inheritance blocking, we need to account for the most penalizing blocking point for τ_i (i.e., node in the CFG). To this end we define $\widehat{\beta}_{i,j}$, a subset of $\beta^*_{i,j}$ including all critical sections in τ_j which can block τ_i due to inheritance blocking. Thus, $\widehat{\beta}_{i,j} = \{cs | cs \in \beta^*_{i,j} \wedge cs \text{ can block } \tau_i \text{ due to inheritance blocking}\}$. We can now compute the maximum CRBD incurred by τ_i due to inheritance blocking by τ_j as follows:

$$CRBD^{inherit}_{i,j} \leq \max_{\substack{cs \in \widehat{\beta}_{i,j} \\ n \in CFG(\tau_i)}} \left\{ \otimes_\sigma \left(UCB^n_i, \overline{UCB}_j(cs) \right) \right\} \times \text{miss penalty} \tag{A.4}$$

However, since a lower priority task τ_j can block τ_i because it is executing inside at most one $cs \in \beta^*_{i,j}$, each τ_j can induce solely one of either inheritance or "non-inheritance" blocking on τ_i. Hence, we can safely account for the worst-case blocking (inheritance or not), that is:

$$CRBD_i \leq \sum_{j>i} \max \left(CRBD^{base}_{i,j}, CRBD^{inherit}_{i,j} \right) \tag{A.5}$$

Bound on semaphores. A second upper bound on blocking, based on the number of semaphores potentially blocking a task under PIP is given in [144]. Under PIP, if there are m semaphores which can block task τ_i, then τ_i can be blocked at most m times, as it can be blocked at most by one critical section for each potentially blocking semaphore. Since

we assume that each semaphore corresponds exactly to a shared resource, then τ_i can be blocked at most by one critical section for each potentially blocking resource.

Similarly to the previous case, [144] defines $\xi_{i,j,k}$ as the set of critical sections of a lower-priority task τ_j guarded by a semaphore S_k and which can block τ_i (due to any type of blocking). Subsequently, $\xi^*_{i,j,k}$ identifies the set of all potentially blocking outermost critical sections guarded by S_k, that is $\xi^*_{i,j,k} = \{cs^{S_k}_{j,m} | cs^{S_k}_{j,m} \in \beta^*_{i,j}\}$.

For example, recalling Table A.1, $\xi^*_{0,1,P} = \{cs^P_{1,1}\}$, $\xi^*_{0,1,Q} = \{cs^Q_{1,1}\}$, $\xi^*_{0,2,R} = \{cs^R_{2,1}\}$ and $\xi^*_{1,\cdot,R} = \{cs^R_{2,1}\}$ (inheritance). Similarly to the first bound, we define $\widehat{\xi}_{i,j,k}$, a subset of $\xi^*_{i,j,k}$ including all critical sections in $\xi^*_{i,j,k}$ guarded by the semaphore S_k which can block τ_i through inheritance blocking. Thus, $\widehat{\xi}_{i,j,k} = \{cs | cs \in \xi^*_{i,j,k} \wedge cs$ can block τ_i due to inheritance blocking$\}$ can be used to separately account for the direct and inheritance cases. First we provide a means to compute the maximum CRBD for each resource that accounts for any lower priority task and any cs in those tasks that may incur both forms of blocking.

$$CRBD^{base}_{i,R} \leq \max_{\substack{j>i, k\in[1,|cs^R_i|] \\ cs\in\xi^*_{i,j,R}}} \left\{ \otimes_\sigma \left(UCB^R_{i,k}, \overline{UCB}_j(cs) \right) \right\} \times \text{miss penalty} \tag{A.6}$$

$$CRBD^{inherit}_{i,R} \leq \max_{\substack{n\in CGF(\tau_i) \\ j>i \\ cs\in\widehat{\xi}_{i,j,R}}} \left\{ \otimes_\sigma \left(UCB^n_i, \overline{UCB}_j(cs) \right) \right\} \times \text{miss penalty} \tag{A.7}$$

Again, since task τ_i can be blocked at most once for each semaphore (resource), we can compute a safe upper bound on the blocking delay by summing the $|S|$ worst-case penalties over the $S \subset SR$ semaphores (resources) potentially blocking τ_i:

$$CRBD_i \leq \sum_{R\in S} \max \left(CRBD^{base}_{i,R}, CRBD^{inherit}_{i,R} \right) \tag{A.8}$$

The actual bound on the CRBD under PIP is then determined by the minimum between the bounds on lower priority tasks and semaphores (i.e., Equations A.5 and A.8).

A.3 CRBD under the Priority Ceiling Protocol

With PCP [144], each resource is assigned a *ceiling priority* which is set to at least the priority value of the highest-priority task that uses that resource. Since ceiling priorities are assigned statically, all the tasks of the system and their priority must be known statically.

For a task τ_i to be able to access the critical section of a resource, its current priority must be higher than the ceiling priority of any currently locked resource (i.e. semaphore). Otherwise, the task that blocks τ_i inherits the ceiling priority of the resource it is locking.

PCP introduces *avoidance blocking*: a task, when trying to access a resource that is *currently available*, is blocked if its current priority is not higher than the highest ceiling of all semaphores currently locked by other tasks. This protocol rule is used to warrant the absence of deadlock. Furthermore, transitive blocking is not possible, a task τ_i can be blocked at most once per activation, and the duration of the priority inversion is minimized.

Similarly to PIP, a bound on the delay incurred by the effects of blocking on the cache state must account for inheritance blocking separately from direct and avoidance blocking as only the latter ones are triggered when a task attempts to access a resource. Provided that the computation of the $\beta^*_{i,j}$ set includes all critical sections of τ_j that may block τ_i due to direct, inheritance or avoidance blocking, an upper bound for the CRPD can be computed in a similar way to the first bound on PIP. The CRBD suffered by a task τ_i can be bounded by the following equation:

$$CRBD_i \leq \max_{j>i} \left\{ \max \left(CRBD^{base}_{i,j}, CRBD^{inherit}_{i,j} \right) \right\} \tag{A.9}$$

where $CRBD^{base}_{i,j}$ and $CRBD^{inherit}_{i,j}$ are exactly as defined in the PIP case (Eq. A.3 and A.4 respectively). As opposed to the PIP case, we are interested just in the most penalizing critical section among all critical sections and all lower-priority tasks, due to Theorem 12 in [144].

A.4 CRBD under the Immediate Ceiling Priority

The Immediate Ceiling Priority Protocol (ICPP) (direct derivative of Baker's stack resource policy [14]) is similar to PCP, as ceiling priorities are assigned to resources with the same rules. Under ICPP however, a task that enters in a critical section always inherits the ceiling priority, while under PCP only when it is *blocking* a higher-priority task; therefore all tasks with a priority lower than or equal to the ceiling priority cannot be scheduled until the resource has been released. ICPP retains the advantages of PCP: absence of deadlock, tasks can block at most once during each activation and the blocking duration is minimized.

The maximum blocking time for a task τ_i is bounded by the longest outermost critical section executed by a lower-priority task τ_j using a resource with a ceiling priority greater than or equal to the priority of τ_i.

More importantly from the CRBD standpoint, the rules of ICPP prevent any disturbing effects on the cache state of the blocked task. In fact, if blocking occurs, it is always before the affected job begins execution; this implies that cache analysis does not need to account for any effect and can continue to assume the worst-case initial cache state (empty or chaos state, depending on the analysis approach). More formally: $CRBD_i = 0, \forall \tau_i$.

Appendix B

List of scientific publications

1. E. Mezzetti and T. Vardanega: "On the industrial fitness of WCET analysis" *Proc. of the 11th International Workshop on Worst-Case Execution Time Analysis* (WCET 2011) Ed. Austrian Computer Society (OCG).

2. E.Mezzetti and T.Vardanega: "Towards a Cache-aware Development of High Integrity Real-time Systems" *Proc. of the 16th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications* (RTCSA 2010) Ed. IEEE Computer Society, ISBN 978-0-7695-4155-6.

3. E.Mezzetti, M.Panunzio and T.Vardanega: "Bounding the Effects of Resource Access Protocols on Cache Behavior" *Proceedings of the 10th International Workshop on Worst-Case Execution-Time Analysis* (WCET2010) Ed. Austrian Computer Society (OCG), ISBN 978-3-85403-268-7. (Also published online by Schloss Dagstuhl OASics series)

4. E.Mezzetti, M.Panunzio and T.Vardanega: "Preservation of Timing Properties with the Ada Ravenscar Profile" Proceedings of the 15th International Conference on Reliable Software Technologies Ada-Europe 2010, Lecture Notes in Computer Science (LNCS), Vol. 6106, XII, Ed Springer, ISBN 978-3-642-13549-1.

5. E.Mezzetti, A.Betts, J.Ruiz and T.Vardanega: "Cache-aware Development of High-Integrity Systems" *Proceedings of the 15th International Conference on Reliable Software Technologies Ada-Europe 2010*, Lecture Notes in Computer Science (LNCS), Vol. 6106, XII, Ed. Springer, ISBN 978-3-642-13549-1

6. E.Mezzetti, M.Panunzio and T.Vardanega: "Temporal Isolation with the Ravenscar Profile and Ada 2005" The 14th International Real-Time Ada Workshop (IRTAW '09) ACM-SigAda Ada Letters, Vol.XXX, n.1, pagg 45-54. October 2009.

7. E.Mezzetti and T.Vardanega: "Impacts of Software Architectures on Cache Predictability in High Integrity Systems" Proceedings of the 30th IFAC Workshop on Real-Time Programming and 4th International Workshop on Real-Time Software (WRTP\ RTS '09)
October 2009.