Modelling Multicore Contention on the AURIX^{TM} TC27x

Enrique Díaz
Universitat Politècnica de Catalunya and BSC

Enrico Mezzetti, Leonidas Kosmidis, Jaume Abella
Barcelona Supercomputing Center (BSC)

Francisco J. Cazorla
BSC and IIIA-CSIC

ABSTRACT

Multicores are becoming ubiquitous in automotive. Yet, the expected benefits on integration are challenged by multicore contention concerns on timing V&V. Worst-case execution time (WCET) estimates are required as early as possible in the software development, to enable prompt detection of timing misbehavior. Factoring in multicore contention necessarily builds on conservative assumptions on interference, independent of co-runners load on shared hardware. We propose a contention model for automotive multicores that balances time-composability with tightness by exploiting available information on contenders. We tailor the model to the AURIX TC27x and provide tight WCET estimates using information from performance monitors and software configurations.

CCS CONCEPTS

• **Computer systems organization → Real-time systems**; • **Software and its engineering → Software performance**;

KEYWORDS

Multicore contention, Performance counters, AURIX TriCore

1 INTRODUCTION

Automotive industry is increasingly adopting multicores as the reference computing solution for ECUs [1–3]. Yet, several academic and industrial studies show that multicores have disruptive effects on Validation and Verification (V&V) practice, shaped on single-core architectures: timing analysis techniques have to be carefully adapted to factor in multicore contention in WCET estimates [5, 14].

Integrated architectures (e.g. AUTOSAR [4]), which have a dominant position in industrial practice, allow OEMs to subcontract the development of software elements to different software providers (SWPs). In terms of timing, the OEM provides SWPs with the time budgets within which all applications must fit. SWPs must provide trustworthy guarantees on the WCET behavior of the software. This, however, clashes with the dependence of the timing behavior of each application on the load other contender applications (likely developed by other SWPs) put on multicore's shared resources. To make things worse, system-level integration and analysis cannot occur until late development stages, when the cost of handling potential budget overruns is significantly higher, thus jeopardizing the whole design and product's time-to-market. On this view, execution time characterization in isolation, i.e. without requiring multicore execution of the final deployment-time integrated tasks, is fundamental to get WCET estimates already in early design stages (e.g. at functional unit implementation). Yet, new methods to factor in multicore contention are sought.

Time-composable WCET estimates hold valid under any contention scenario (and load put by co-runners on hardware resources), without any assumption on how tasks are scheduled at system level. For this reason time-composable bounds rely on the conservative assumption that each request of the task under analysis is delayed in the worst possible way by its contenders, which may easily lead to bounds that are too pessimistic to be useful in practice [11]. Effectively coping with the effects of contention requires WCET bounds that guarantee an adequate balance between tightness and time composability. This translates into flexible approaches that can be easily tailored to derive *partially-time composable* WCET estimates that hold for a subset of contention scenarios [10].

We propose a contention model for measurement-based timing analysis (MBTA), the most widely adopted analysis approach in automotive. ① In order to increase its industrial viability, our contention model relies only on information that can be derived via standard Debug Support Unit (DSU) rather than metrics that can only be obtained in processor simulators. Further, ② it computes contention-aware WCET estimates from observations of a task running in isolation, rather than against contenders, so that to favor the derivation of WCET estimates during pre-integration design stages. Finally, ③ it is flexible enough to model different contention scenarios while providing tight contention bounds.

We tailor our model to the Infineon AURIX™ TC-27x multicore family of processors. We analyze the specific execution information made available by the TC-27x. Building on TC-27x's Debug Counters, we define a highly flexible contention model, the first one for AURIX multicores to our knowledge, that can be easily tailored to different processor deployment scenarios and fits in standard MBTA for single-cores. Our model is presented as an Integer Linear Programming (ILP) problem that finds the worst-case mapping between conflicting requests on the AURIX Shared Resource Interconnect (SRI). Results show that our model provides tighter results, under different contention scenarios and load of the contenders on shared resources, than the time-composable model.

2 PRELIMINARIES

Reference Platform. The AURIX™ TC277 [12] comprises three different TriCore™ processors: a low-power core (16E) and two higher-performance cores (16P). All processors have separated core-local memories (scratchpads and caches) for instructions and data (though the 1.6E deploys a data buffer instead of data cache). Processors are connected to a shared 'memory system' through the SRI cross-bar (see Figure 1). The shared memory system comprises

DAC '18, June 24–29, 2018, San Francisco, CA, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5700-5/18/06…$15.00
https://doi.org/10.1145/3195970.3196077

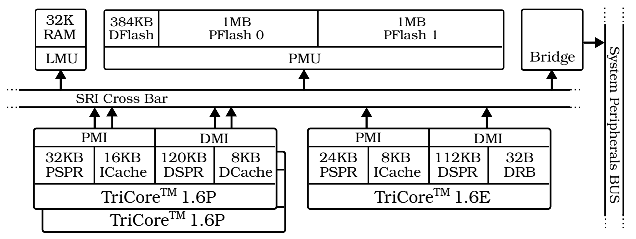


Figure 1: Block Diagram of the AURIX™ TC-27x

a SRAM device, accessed via the Local Memory Unit (LMU) and a FLASH device, accessed via the Program Memory Unit (PMU) through three independent interfaces, two for code (i.e. program) and one for data. LMU and PMU memory areas can be accessed in cacheable or uncacheable mode, depending on the address segment used. System software statically sets the *deployment configurations* defining where stack, functions, and data of the application are mapped, and their cacheability options.

Basic Notation and Assumptions. We consider one time-critical task under analysis (τ_a) for which a WCET estimate is to be derived; and a contender task (τ_b). This model can be easily extended to consider more contenders at the same time. We focus on the case in which the requests of contenders are mapped to the same SRI priority class. In this scenario, the most stressing one for our model, stalls (contention) can happen whenever τ_a issues a request to a target and the request arrives right after the arbitration turn has selected another request by τ_b on the same target. Hence, contention is determined on a per-target basis, with contenting requests arbitrated using a round-robin policy.

The exact interference τ_a suffers because of τ_b depends on how inter-core requests interleave in the SRI, which is generally not controllable during testing. Moreover, analysis is often forced to build on information obtained on τ_a in isolation, as information from joint execution cannot be derived until late design phases, when applications are integrated together. Therefore, contention models cannot determine *exactly* how tasks interleave in reality once integrated and can only conservatively assume that contenders' requests align in the worst possible way with τ_a's requests [13, 16]. In the TC-27x, contention is determined by the number of requests of τ_a (not the type of such requests) and the number and type of requests of τ_b. A conservative contention model is forced to assume τ_a is delayed by each request of its contender for request duration, which in turn depends on target resource and operation type. Table 1 summarizes the main terms we use in this work.

The relevant target resources accessible through the SRI in the AURIX platform comprise LMU and PMU. The latter is further broken down in different Flash memory interfaces for data and program (code), which we denote dfl, and $pf0$ and $pf1$ respectively. We consider the set of target resources $\mathcal{T} = \{dfl, pf0, pf1, lmu\}$. While each target might exhibit different latencies depending on the type of request (operation) processed (i.e. code/data reads or data writes), for the time being we differentiate only among code (*co*) and data (*da*) requests for all targets in $O = \{co, da\}, \forall\ t \in \mathcal{T}$. Code accesses can target the $pf0$, $pf1$, and lmu, while the data accesses can target any resource: $dfl, pf0, pf1$ and lmu as shown in Figure 2. We do not consider SRI traffic caused by code and data requests targeting scratchpads of other cores, as those requests would enable the occurrence of stalls in the memory interface even

Table 1: Definitions used in this paper.

Acronym	Description
	Target Resources and Operation Types
$\mathcal{T}$	Target resources in the SRI
O	Types of operations on a SRI target
	Access Counts
n_a,	Total access count of τ_a
n_a^{co}, n_a^{da}	Data and code access count of τ_a
$\hat{n}_a$, $\hat{n}_a^{co}$, $\hat{n}_a^{da}$	Upperbounds to previous values
$n_a^{t,o}$	τ_a's accesses of type o to resource t
	Latencies
$l^{t,o}$	Access latency of o-type requests to t
$cs^{t,o}$	Stall cycles when accessing t with an access o
cs_a^{co}, cs_a^{da}	τ_a's code and data stall cycles in isolation
Δcs_a^{co}, Δcs_a^{da}	τ_a's increment in stall cycles due to contention

when accessing core-local memories. This would invalidate any attempt to provide some level of isolation between cores.

Hardware Profiling. Our contention model builds on the AURIX™ DSU interface and TC-27x's Debug counters that can be configured to collect data on both core-local and inter-core events. We exploit the on-chip cycle counter (`CCNT`); `PMEM_STALL` and `DMEM_STALL` to count the number of cycles the pipeline has been stalled when accessing the Program/Data memory interface respectively; and `PCACHE_MISS`, `DCACHE_MISS_CLEAN` and `DCACHE_MISS_DIRTY`, related to cache performance and specifically to cache misses.

Maximum and minimum observable end-to-end latencies for SRI transactions in isolation are reported in Table 2. Note that the reported latency is always the maximum between read and write operations per SRI target, as we are only interested in discriminating between code and data operations. Table 2 also reports $cs^{t,o}$ as the amount of stall cycles incurred in the *best-case* for single accesses in isolation to each SRI target. Best-case stall counts should take into account the effects of prefetching, pipelining in the SRI, etc. so that they can be used to compute an over-approximation of the number of SRI accesses of a given application or task.

3 CONTENTION MODELS

3.1 State of the art

Modeling multicore contention has been addressed by a notable amount of works [9]. Approaches to analyze contention on COTS hardware generally build on the availability of performance monitoring counters (PMCs) [7, 8, 13, 15, 16]. Some approaches have proposed to extend single-core timing analysis frameworks to account for the effect of shared resources [6], which is generally unsustainable for COTS platforms due to the entailed computational complexity and overly pessimistic results. We also exclude from our discussion ad-hoc hardware architectures that may present specialized mechanisms for contention analysis and enforcement.

In [7] authors propose a contention model on systems with front-side bus (FSB) building upon PMCs to derive the maximum number of bus accesses of the target task. A recent work [16], builds on PMC support on the P4080 to implement a RTOS-level mechanism to enforce precomputed bounds to the maximum contention caused/suffered at operation. Some of these approaches generally focus on a predefined set of contender tasks, in an attempt to accurately model how accesses may cause interference on shared resources, often building on fragile assumptions on specific distributions of memory accesses. More importantly, obtained results are only valid under the considered context and are not flexible

	Target (t)		
	lmu	pf	dfl
l^{max}	11(21)	16	43
l^{min}	11	12	43
$cs^{t,co}$	11	6	-
$cs^{t,da}$	10	11	42

Table 2: Maximum latency and minimum stall cycles

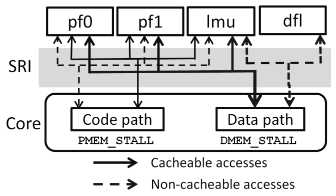


Figure 2: Code and data access paths to the SRI

enough to adapt to different deployment scenarios, hence negatively affecting time composability. Partial time composability has been recently introduced [8, 10, 13] for processors in which contention happens on the bus and hence are not applicable to AURIX. While several contention models exist, they are not applicable to our target platform: either they build on information hard to derive in industrial scenarios (while we build on DSU) or do not guarantee time composability. Furthermore they focus on the contention on a FSB-like shared interconnect while in our target AURIX platform contention happens on the specific slave interface as the SRI allows parallel transactions on distinct interfaces.

In the following we define a flexible contention model for the AURIX that is naturally amenable to handle different configurations and produce results with varying degree of time-composability. Our first model assumes all relevant information on SRI access is available. Then, we show how we deal with the lack of information and define a realistic and tight ILP-based contention model.

3.2 Ideal contention model for the AURIX

In the ideal model, each request of τ_b delays a request of τ_a. If τ_b has more requests than τ_a, those requests of τ_b with highest latency are assumed to be the ones delaying the requests of τ_a. For simplicity, we assume that all requests of a given type of τ_b have the same latency. The worst-case contention τ_b can cause on τ_a, i.e. $\Delta^{cont}_{b\to a}$, is shown in Equation 1, where $n^{t,o}_b$ is the number of τ_b requests of type o to target resource t and $l^{t,o}$ is the latency of that request.

$$\Delta^{cont}_{b\to a} = \sum_{t\in\mathcal{T}} \sum_{o\in O} min(n^{t,o}_a, n^{t,o}_b) \times l^{t,o} \tag{1}$$

In general, different requests of τ_b may have different latencies, which can be trivially captured by the model.

3.3 Coping with limited information

The ideal model builds on detailed information on (i) the latency of each operation for each target resource; (ii) the total access count per resource of the task under analysis; and (iii) total access count and operation type per target resource of the contender tasks. However, such information is not always (exhaustively) available, due to the limited hardware support in typical DSU for deriving $n^{t,o}_x$ for an arbitrary task τ_x. Furthermore, focusing on maximum $l^{t,o}$ for each resource and operation type inherently introduces pessimism by possibly discarding effects of prefetching on the SRI targets. We cope with these concerns in the TC-27x.

3.3.1 ***Latencies.*** We empirically derived the longest latency incurred by each resource when processing a code or data request, see Table 2. To measure the maximum latency to each target resource we considered the latency incurred by single accesses to a target (slave) resource in the SRI as measured by the on-chip cycle counter (CCNT). Note that dirty data misses latency on the *lmu* are reported within brackets as they apply only on limited scenarios.

3.3.2 ***Access counts of τ_a.*** AURIX TC27x lacks SRI access counters on a per-resource basis. Hence, we used the existing stall cycle counters (PMEM_STALL and DMEM_STALL) in our target AURIX platform. An upperbound to the number of SRI requests can be derived, separately for code and data, by dividing the total amount of stall cycles by the minimum amount of stall cycles per single request. We derived the latter by analyzing PMEM_STALL and DMEM_STALL under a specific set of *microbenchmarks* [10] comprising a known number of requests of a given type to a desired target resource. This allowed deriving a lower bound to the stall cycles a task can suffer while completing a code and data request to a given target, $cs^{t,co}$ and $cs^{t,da}$. It is worth noting that we are interested in lower bounds to the stall cycles in order to upperbound the number of possible accesses to the slave. The second factor for the computation of (an over-approximation of) the SRI traffic of a given task consists in the total amount of stall cycles suffered in isolation because of stalls in the memory interface. For a given task τ_x, these values can be obtained for code and data request separately (cs^{co}_x and cs^{da}_x) by running the task in isolation and collecting cumulative end-to-end values of PMEM_STALL and the DMEM_STALL counters.

From the stall cycles we can derive an upperbound to the number of code and data requests assuming that the entire stall delay has been caused by the requests of the shortest duration, i.e. $cs_{min} = min(\{cs^{t,o}\}_{\forall t\in\mathcal{T}\wedge\forall o\in O})$. As depicted in Figure 2, the lowest possible stall cycles incurred for code and data requests in the AURIX™ platform can be derived by taking into account the architectural constraints on where code and data can be deployed:

$$cs^{co}_{min} = min\left(cs^{pf0,co}, cs^{pf1,co}, cs^{lmu,co}\right) \tag{2}$$

$$cs^{da}_{min} = min\left(cs^{pf0,da}, cs^{pf1,da}, cs^{lmu,da}, cs^{dfl,da}\right) \tag{3}$$

An upperbound to code and access counts of task τ_a can be derived by assuming that all requests are of the type incurring the lowest number of stalls (hence more requests are needed to cause cs^{co}_a and cs^{da}_a) and dividing the stall cycles by the duration of the shortest request.

$$\hat{n}^{co}_a = \left\lceil \frac{cs^{co}_a}{cs^{co}_{min}} \right\rceil \qquad \hat{n}^{da}_a = \left\lceil \frac{cs^{da}_a}{cs^{da}_{min}} \right\rceil \tag{4}$$

3.3.3 ***Per Target Access Counts (PTAC) of τ_b.*** Cumulative SRI access counts for code and data do not suffice to derive a tight contention model. Since the SRI mechanism allows handling requests directed to different slaves in parallel (and each slave does incur different latencies), a good approximation of inter-core contention cannot be obtained without considering Per-Target Access Counts (PTAC). As shown in Figure 2, code and data accesses of a given task can go to different targets.

$$n_b = n^{co}_b + n^{da}_b = \sum_{t\in\mathcal{T}} n^{t,co}_b + \sum_{t\in\mathcal{T}} n^{t,da}_b \tag{5}$$

Different approximations of PTAC can be defined based on the information available and deployment configuration,. In the following we first present the fully time-composable contention model – building on information on τ_a only – and a more generic ILP formulation of the problem that can be deployed with different levels of information and supporting various degrees of composability.

3.4 fTC model in the lack of PTAC

A baseline fTC model disregards per-target information altogether, using only cumulative information at code/data access level. Despite the incurred pessimism, this fTC model is still relevant when no PTAC information is available and time-composability is the driving concern. With current TC-27x debug counters, PTAC information is only indirectly and fragmentarily available through cross-checking information on deployment configuration and cache (miss) statistics. In terms of access counts, we derive access data and code counts for τ_a and τ_b as described by Equation 4. In terms of delay, instead, the model exploits the maximum delay a code/data request from τ_a can suffer from τ_b, based on the type of requests that can go to each resource. Code accesses can address *pf0*, *pf1* or *lmu*, hence the longest delay a code access from τ_a can suffer is defined by the longest latency it can suffer owing to τ_b accessing the same interfaces for code ***and*** data, as shown in Equation 6.

$$l_{max}^{co} = max(l^{pf0,co}, l^{pf0,da}, l^{pf1,co}, l^{pf1,da}, l^{lmu,co}, l^{lmu,da}) \quad (6)$$

$$l_{max}^{da} = max(l_{max}^{co}, l^{dfl,da}) \quad (7)$$

Likewise, the maximum delay a data access can suffer is defined by Eq. 7 that matches the previous one with the exception that it factors in dflash (data) accesses from τ_b. Hence, the contention delay τ_a can suffer (Eq. 8) is defined as the number of code and data accesses of τ_a times the longest latency each request can suffer.

$$\Delta_{b\to a}^{cont} = \hat{n}_a^{co} \times l_{max}^{co} + \hat{n}_a^{da} \times l_{max}^{da} \quad (8)$$

This contention model is fully time-composable as it assumes that *all τ_a requests always suffer the longest possible contention.* The inherent pessimism of this approach is even more evident on the TC-27x on the account of its crossbar mechanism as not only latency varies depending on the specific target, but contention itself can only be incurred when requests are directed to the same target.

3.5 ILP-Based PTAC Model (ILP-PTAC)

Tighter bounds can be obtained by considering τ_b code and data requests (partial time-composability) and by exploring all possible PTAC for both τ_a and τ_b. To that end we formulate the model as an Integer Linear Programming problem to find the per-target mapping of τ_a's and τ_b's requests that maximizes the contention suffered by τ_a.

Our **objective function** maximizes the SRI stall cycles incurred by τ_a because of contention in code and data accesses (Δcs_a^{co}, Δcs_a^{da}). This is modelled in Equation 9 where $n_{b\to a}^{t,o}$ stands for the number of requests from contender τ_b targeting interface t for accesses of type o that are assumed to **interfere** with τ_a. Note that we break down interference between data and code accesses.

$$\begin{aligned}\Delta_{b\to a}^{cont} \;=\; & [\Delta cs_a^{co}] + [\Delta cs_a^{da}] = \\ & \Big[n_{b\to a}^{pf0,co} \times l^{pf0,co} + n_{b\to a}^{pf1,co} \times l^{pf1,co} + n_{b\to a}^{lmu,co} \times l^{lmu,co}\Big] + \\ & \Big[n_{b\to a}^{dfl,da} \times l^{dfl,da} + n_{b\to a}^{pf0,da} \times l^{pf0,da} + \\ & n_{b\to a}^{pf1,da} \times l^{pf1,da} + n_{b\to a}^{lmu,da} \times l^{lmu,da}\Big] \quad (9)\end{aligned}$$

Again, we assume that each interfering request of τ_b aligns in the worst manner with τ_a requests. Hence, each interfering request delays τ_a by $l^{t,o}$.

Constraints. Constraints in the ILP formulation are defined on the number of requests per target resource as follows. Equation 10 captures that the number of data requests from τ_b that can contend with τ_a on the dfl is bounded by the maximum number of requests that τ_a and τ_b make to the dfl.

The constraint in Equation 11 captures that the maximum number of *inflictive* code requests from τ_b onto $pf0$ that interfere with both τ_a's code and data requests is bounded by the minimum between τ_b's code requests and all τ_a requests (still to $pf0$). Similarly, the number of inflictive data requests from τ_b onto $pf0$ is smaller than τ_b's data requests and τ_a's data and code requests to $pf0$ (Equation 12). Finally, Equation 13 states a cumulative constraint on the total number of conflicts τ_a can suffer because of τ_b accesses to $pfl0$, which is bounded by the total number of τ_a code and data accesses to $pf0$. Equations 14-16 and 17-19 are the counterparts of Equations 11-13 but applied to the $pf1$ and the lmu respectively.

The following pairs of constraints wrap up the problem variables for the objective function. Equations 20 and 21 represent the SRI access profile (for code and data separately) from the single core execution: they reflect that τ_a makes $n_x^{t,co}$ and $n_x^{t,da}$ accesses to the different resources, which result in cs_x^{co} and cs_x^{da} stall cycles respectively. The latter values are exactly those obtained by reading PMEM_STALL and DMEM_STALL when running τ_a in isolation. Equations 22 and 23 are the equivalent constraints on τ_b execution in isolation. Note that discarding these latter constraints on τ_b would make the ILP model to be fully time-composable.

Note that, while debug counters provide unique values for cs_τ^{co} and cs_τ^{da}, there are no unique stall values for each single $cs_b^{t,o}$ as the actual stall cycles are not constant and depend on pipelining and prefetching effects. As a conservative assumption, we consider the minimum observed stall cycles per request, with the inherent drawback of potentially accounting for more requests than those actually performed by the application.

$$n_{b\to a}^{dfl,da} = min(n_a^{dfl,da}, n_b^{dfl,da}) \quad (10)$$

$$n_{b\to a}^{pf0,co} \le min(n_a^{pf0,co} + n_a^{pf0,da}, n_b^{pf0,co}) \quad (11)$$

$$n_{b\to a}^{pf0,da} \le min(n_a^{pf0,co} + n_a^{pf0,da}, n_b^{pf0,da}) \quad (12)$$

$$n_{b\to a}^{pf0,co} + n_{b\to a}^{pf0,da} \le n_a^{pf0,co} + n_a^{pf0,da} \quad (13)$$

$$n_{b\to a}^{pf1,co} \le min(n_a^{pf1,co} + n_a^{pf1,da}, n_b^{pf1,co}) \quad (14)$$

$$n_{b\to a}^{pf1,co} \le min(n_a^{pf1,co} + n_a^{pf1,da}, n_b^{pf1,da}) \quad (15)$$

$$n_{b\to a}^{pf1,co} + n_{b\to a}^{pf1,co} \le n_a^{pf1,co} + n_a^{pf1,da} \quad (16)$$

$$n_{b\to a}^{lmu,co} \le min(n_a^{lmu,co} + n_a^{lmu,da}, n_b^{lmu,co}) \quad (17)$$

$$n_{b\to a}^{lmu,da} \le min(n_a^{lmu,co} + n_a^{lmu,da}, n_b^{lmu,da}) \quad (18)$$

$$n_{b\to a}^{lmu,co} + n_{b\to a}^{lmu,da} \le n_a^{lmu,co} + n_a^{lmu,da} \quad (19)$$

$$\begin{aligned}cs_a^{co} = n_a^{pf0,co} \times cs_a^{pf0,co} + n_a^{pf1,co} \times cs_a^{pf1,co} + \\ n_a^{lmu,co} \times cs_a^{lmu,co}\end{aligned} \quad (20)$$

$$\begin{aligned}cs_a^{da} = n_a^{pf0,da} \times cs_a^{pf0,da} + n_a^{pf1,da} \times cs_a^{pf1,da} + \\ n_a^{lmu,da} \times cs_a^{lmu,da} + n_a^{dfl,da} \times cs_a^{dfl,da}\end{aligned} \quad (21)$$

$$\begin{aligned}cs_b^{co} = n_b^{pf0,co} \times cs_b^{pf0,co} + n_b^{pf1,co} \times cs_b^{pf1,co} + \\ n_b^{lmu,co} \times cs_b^{lmu,co}\end{aligned} \quad (22)$$

$$\begin{aligned}cs_b^{da} = n_b^{pf0,da} \times cs_b^{pf0,da} + n_b^{pf1,da} \times cs_b^{pf1,da} + \\ n_b^{lmu,da} \times cs_b^{lmu,da} + n_b^{dfl,da} \times cs_b^{dfl,da}\end{aligned} \quad (23)$$

4 EVALUATION

The AURIX™ TC-27x supports several deployment configurations with different code and data placement and cacheability options. Architectural constrains are summarized in Table 3, where '\$' stands for cacheable and '*n*\$' non cacheable.

The large number of deployment configurations offer high system-level flexibility. On the timing side, this results in different contention scenarios and thus restrict the PTAC feasibility region. To that end, our model takes as input information about the application code and data layout to obtain tighter results. Our generic ILP model can be easily tailored to capture any scenario by adding some constraints on target and access type.

Table 3: Constraints on code/data wrt SRI slaves.

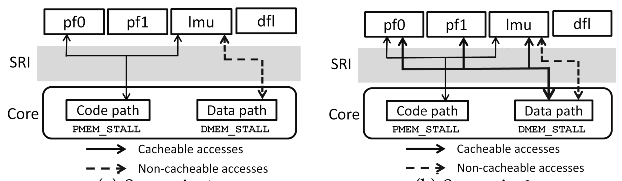
	pf0	pf1	dfl	LMU
Code \$	✓	✓	×	✓
Code n\$	✓	✓	×	✓
Data \$	✓	✓	×	✓
Data n\$	×	×	✓	✓

Table 4: Debug counters ('\$' stands for cache)

Counter	Task a	Task b
PMEM_STALL	PS_a	PS_b
DMEM_STALL	DS_a	DS_b
P\$_MISS	PM_a	PM_b
D\$_MISS_CLEAN	DMC_a	DMC_b
D\$_MISS_DIRTY	DMD_a	DMD_b

4.1 Deployment scenarios and model tailoring

As common deployment strategy, part of the application code and data is always deployed into the local scratchpads that do not generate traffic on the SRI. As a matter of fact, w.r.t. the rest of the application, some configurations, though admissible, are rarely used in practice. We focus on two deployment scenarios, see Figure 3, particularly representative of real-world deployment configurations. Without loss of generality, we assume deployment configurations equally apply to the task under analysis and contenders.



(a) Scenario 1 (b) Scenario 2

Figure 3: Scenarios deployed in this work.

Scenario 1 (see Figure 3-a): part of the code and data fit on local scratchpads, some (cacheable) code is fetched from *pf0/pf1*, and some (non cacheable) data is shared among cores in the *lmu*. In this specific case, we exploit the fact that P\$_MISS holds the exact number of code requests from a task on the SRI (as all code requests through the SRI are performed in cacheable mode): $n_a^{co} = \mathsf{PM}_a$ and $n_b^{co} = \mathsf{PM}_b$. Nothing, instead, can be argued on data requests.

Scenario 2 (see Figure 3-b): part of the code and data fit on local scratchpads, some code is fetched from *pf0/pf1* (cacheable), some data is deployed to *lmu* (cacheable and non-cacheable), and finally constant data is found in *pf0/pf1* (cacheable). This represents a challenging scenario for the fTC baseline model as it would assume all cacheable accesses to any target incur a contention delay in the amount of a dirty miss latency. Besides considering the contender accesses, the ILP model can also exploit some indirect PTAC information from the cacheable code debug counter as the P\$_MISS counter gives the exact number of code requests on *pf0/pf1*. We cannot do the same for data since the sum of D\$_MISS_CLEAN and D\$_MISS_DIRTY provides the cumulative count of cacheable data requests but does not discriminate between the target of each access, which can equally be the *pf0/pf1* or the *lmu* (also accessed in non-cacheable mode).

Table 5: Tailoring of the ILP-PTAC model

Scenario 1	Scenario 2
$n_a^{dfl,da} = 0,\ n_a^{lmu,co} = 0$	$n_a^{dfl,da} = 0,\ n_a^{lmu,co} = 0$
$n_a^{pf0,da} = 0,\ n_a^{pf1,da} = 0$	$n_a^{pf0,da} + n_a^{pf1,da} + n_a^{lmu,da} \geq \mathsf{DMC}_a + \mathsf{DMD}_a$
$n_a^{pf0,co} + n_a^{pf1,co} = \mathsf{PM}_a$	$n_a^{pf0,co} + n_a^{pf1,co} = \mathsf{PM}_a$

Table 5 shows the instantiation of the ILP-PTAC model to both scenarios, by introducing few additional ILP constraints on the PTAC. The counter used by the model and respective notation for τ_a and τ_b are reported in Table 4. It is worth noting that *indirect* PTAC information, as derived by deployment configuration options, can be incorporated on a refined fTC model, but limitedly to τ_a and thus with minor benefits.

4.2 Experimental Results

Workloads. We evaluated the contention models on an application mimicking a control loop (e.g., of an Automotive Cruise Control System). The application performs the typical sequence of signal acquisition, computation and status update, and it operates on two medium-size data structures. The application has been deployed in two variants, according to the identified deployment scenarios. We stress the application with 3 different co-runners that generate an increasing (load) number of accesses to the SRI, which hence, increasingly disturb the application under analysis. We respectively refer to these benchmarks as H-Load, M-Load, and L-Load, where 'H', 'M', and 'L' stands for high, medium and low. In all experiments, Core 1 and Core 2 (TC-1.6P) host the application under analysis and a contender respectively.

Metrics. We first executed the application and each contender in isolation to collect readings on the relevant debug counters. For the application we also collect its (observed) execution time in isolation. Then, with the counter readings we feed our model and assess the accuracy of the so-obtained WCET estimation against execution in isolation. In all experiments our model predictions upperbound the observed multicore execution time.

Debug counters readings. Table 6 reports the debug counter values observed under the two reference scenarios, for cores 1 and 2, running the target application and the H-Load contender respectively. The fact that dirty data cache misses are zeroed under both scenarios is not surprising, as cacheable data accesses are typically performed to address constant data. This provides evidence on the correct setup of the cacheability and memory deployment options.

Table 6: Counter readings for Scenarios 1 and 2.

		PM	DMC	DMD	PS	DS
Sc1	Core1	236544	0	0	3421242	8345056
	Core2	120594	0	0	1744167	4251811
Sc2	Core1	458394	200	0	2753995	86371
	Core2	233694	200	0	1404145	42826

WCET estimation. We measure how our model and the fTC model behave under the two deployment scenarios (scenario1 and scenario2) and the load added by the corunner on shared resources (H-load, M-load, L-load). Under Scenario 1, contention only happens disjointly on *pf0/pf1*, for code, and *lmu*, for data. The benchmarks are fetching part of the code from the PFlash and performing data read and writes on the *lmu*. Scenario 2, instead, requires data to be deployed to the *lmu* (in both cacheable and non-cacheable mode) and to *pf0/pf1* (constant and cacheable). Contention is suffered on the same slave because of different types of accesses (code and data). Figure 4 assesses the predictions of the different models against WCET estimates in isolation. Results from both scenarios clearly indicate that the fully time-composable bounds may end up being poorly useful in consideration of the pessimism they incur. Our model, instead, exploits the information from debug counters to derive tighter bounds on contention under specific deployment configurations. In both cases, contention cycles are below half of those for fTC bounds. Focusing on the variation across different contenders, we see that our ILP model adapts to the load introduced by the contenders, while the fTC model is unable to benefit from this information. In the first scenario, the ILP model predicts an execution time increase in between 1.49 and 1.24 while the fTC model cannot do better than 1.95. Similar results are obtained in the second scenario, where the ILP results ranges in between 1.67 and 1.34, against a 2.33 returned by the fTC model.

Flexibility and adaptability of the model is a fundamental desired property for a contention model: it provides a powerful and reactive method for OEM and SWPs to explore and evaluate different scheduling allocations and deployment scenarios with respect to the expected contention they will suffer during operation, before actual integration. The contention model should also conservatively capture the worst-case contention effects without renouncing tightness. However, whether the gap between actual measurements and model estimates corresponds to overestimation (and to what extent) cannot be determined. Triggering the worst time-alignment of memory accesses is, in general, not feasible and thus, our model relieves end users from having to exercise that level of control. On the other hand, the limited information available on AURIX PMCs forces our model to make some pessimistic assumptions on the number of contention events and their effects, which may introduce some pessimism. In any case, preliminary results on real-world automotive use cases show much lower contention bounds (~10%) than those of our benchmark (30-40%).

4.3 Adaptability to other platforms

Contention models are only as effective as much as their ability to model low-level architectural details. Defining a generic low-level contention model that fits all architectures and families of processors is an utopic endeavor, as contention is unequivocally determined by the amount and organization of shared resources, the arbitration policies in place as well as the debug counter support.

In this work we have been focusing on the AURIX TC27x, as a representative platform for the automotive domain. The contention model is naturally adaptable to other processors in TriCore family. Additionally, we consider our model to be flexible enough to be adapted to other processor models. With respect to the debug counter support, our model is exploiting a minimal set of counters and equivalent information is generally provided by modern DSU. Our model is strongly characterized by the AURIX cross-bar while other approaches presented in Section 3.1 focus on FSB-based architectures. We do not see any major obstacle in adapting our model to cover the FSB behavior as we consider the FSB model to be a reduced case for the more generic cross-bar model.

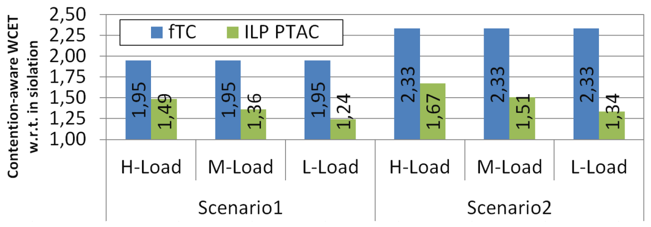


Figure 4: Model predictions w.r.t. execution in isolation.

5 CONCLUSIONS

Advanced multicore platforms are the preferred industrial solution for delivering cutting-edge functionalities in modern automotive systems. In this work we presented two analytical contention models for the AURIX TC-27x platform, building on the existing debug counters support. While fTC bounds are confirmed to be overly pessimistic, our partially time-composable model (ILP-PTAC) provides realistic bounds that are valid for a wide range of contention scenarios. Further, formulating the contention as an ILP problem, guarantees better adaptability to different configuration scenarios.

ACKNOWLEDGMENTS

The research leading to this work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 644080 (SAFURE). This work has also been partially funded by the Spanish Ministry of Science and Innovation under grant TIN2015-65316-P and the HiPEAC Network of Excellence. The Ministry of Economy and Competitiveness partially supported Jaume Abella under Ramon y Cajal postdoctoral fellowship (RYC-2013-14717) and Enrico Mezzetti under Juan de la Cierva-Incorporación postdoctoral fellowship (IJCI-2016-27396).

REFERENCES

[1] 2016. QUALCOMM Snapdragon 820 Automotive Processor. https://www.qualcomm.com/products/snapdragon/processors/820-automotive. (2016).
[2] 2017. NVIDIA DRIVE PX. Scalable supercomputer for autonomous driving. http://www.nvidia.com/object/drive-px.html. (2017).
[3] 2017. RENESAS R-Car H3. https://www.renesas.com/en-us/solutions/automotive/products/rcar-h3.html. (2017).
[4] AUTOSAR. 2006. *Technical Overview V2.0.1.*
[5] Antoine Blin et al. 2016. Maximizing Parallelism without Exploding Deadlines in a Mixed Criticality Embedded System. In *28th ECRTS.*
[6] S. Chattopadhyay et al. 2012. A Unified WCET Analysis Framework for Multi-core Platforms. In *IEEE 18th RTAS.*
[7] D. Dasari et al. 2011. Response Time Analysis of COTS-Based Multicores Considering the Contention on the Shared Memory Bus. In *IEEE TrustCom.*
[8] E. Díaz et al. 2017. MC2: Multicore and Cache Analysis via Deterministic and Probabilistic Jitter Bounding. In *Reliable Software Technologies-Ada-Europe.*
[9] G. Fernandez et al. 2014. Contention in Multicore Hardware Shared Resources: Understanding of the State of the Art. In *14th WCET Workshop.*
[10] G. Fernandez et al. 2015. Resource usage templates and signatures for COTS multicore processors. In *52nd DAC.*
[11] M. Fernández et al. 2012. Assessing the Suitability of the NGMP Multi-core Processor in the Space Domain. In *EMSOFT.*
[12] Hitex. *AURIX Application Kit TC277 TFT.* http://www.ehitex.de/application-kits/infineon/2531/aurix-application-kit-tc277-tft.
[13] J. Jalle et al. 2015. Bounding Resource Contention Interference in the Next-Generation Microprocessor (NGMP). In *ERTS.*
[14] L. Kosmidis et al. 2016. Measurement-Based Timing Analysis of the AURIX Caches. In *WCET Workshop.*
[15] T. Moseley et al. 2005. Methods for modeling resource contention on simultaneous multithreading processors. In *IEEE ICCD.*
[16] J. Nowotsch et al. 2014. Multi-core Interference-Sensitive WCET Analysis Leveraging Runtime Resource Capacity Enforcement. In *ECRTS.*