Improving Early Design Stage Timing Modeling in Multicore Based Real-Time Systems

David Trilla[†,‡], Javier Jalle[†,‡], Mikel Fernandez[†], Jaume Abella[†], Francisco J. Cazorla[⋆,†]
[†] Barcelona Supercomputing Center (BSC). Barcelona, Spain
[‡] Universitat Politècnica de Catalunya (UPC), Barcelona, Spain
[⋆] Spanish National Research Council (IIIA-CSIC). Barcelona, Spain.

Abstract—**This paper presents a modelling approach for the timing behavior of real-time embedded systems (RTES) in early design phases. The model focuses on multicore processors – accepted as the next computing platform for RTES – and in particular it predicts the contention tasks suffer in the access to multicore on-chip shared resources. The model presents the key properties of not requiring the application's source code or binary and having high-accuracy and low overhead. The former is of paramount importance in those common scenarios in which several software suppliers work in parallel implementing different applications for a system integrator, subject to different intellectual property (IP) constraints. Our model helps reducing the risk of exceeding the assigned budgets for each application in late design stages and its associated costs.**

I. Introduction

During the early design phases (EDP) of an RTES, critical decisions are taken including the processor to use and the time budget assigned to each application (task[1]). These decisions affect not only subsequent designs phases (i.e. the design plan) but also the final delivered product.

The EDP are characterized by the uncertainties (i.e. lack of information) in terms of functional and non-functional properties/requirements of the system. During the EDP, the set of tasks that implement a given functionality is preliminary and so are their timing requirements, which are bounded with early estimates. On the one hand, overestimating those timing requirements helps removing the uncertainties but may result in an over-designed system with a lot of spare (lost) capacity. On the other hand, underestimation may lead to changes in the task structure in the late design phases (LDP) of the system, which are hard and costly to implement.

This situation is compounded by the use of multicores as the reference computing platform for future RTES. When a task (τ_j) runs on a multicore, its execution time – and hence its execution time estimates – does not only depend on τ_j itself but also on τ_j's co-runner tasks. This is so because τ_j's access delay to shared hardware resources depends on the load its co-runners put on those resources. This heavily limits the ability of individual software suppliers to provide accurate bounds to their tasks' non-functional behavior (timing). Further in the embedded-system domain, integrators (e.g. original equipment manufacturer or OEM) increasingly incorporate in their products applications coming from different software suppliers, which complicates deriving tight timing estimates. This is caused by the fact that, usually, suppliers keep the IP rights of their software, preventing the source code from being shared among them or with the OEM.

[1]In this paper we use the terms *task* and *application* indistinctly. Instances of tasks, which are usually periodic, are called *jobs*.

In the scope of this paper we focus on the scenario in which the target computing platform is known. In the case of the space domain, the NGMP [7] is a strong candidate for European Space Agency's future missions and it will be maintained for years. We assume that each supplier is provided a virtualized environment, such as those based on GMV's AIR for the space and avionics domains [27]. Such an approach does not only allow developing and testing the functional behavior of applications in a fast manner, but also allows each supplier to develop several applications in parallel without the need to purchase a physical board for each of them. This approach is followed by several OEMs including the European Space Agency for several projects [15].

Contribution. While virtualized environments allow functional testing, they fail to provide timing estimates of the executed applications. In this paper we propose an approach for providing, during the EDP, fast and accurate timing estimates of tasks' execution time when the target (virtual) hardware comprises multicores. Our proposal extends virtualized environments with a light-weight timing model that i) provides high accuracy and low overhead; and ii) does not require code or binaries to be shared among software suppliers, which helps keeping the confidentiality on their developed software. Hence, our proposal simplifies and speeds up the process of getting timing estimates during the EDP when the target (virtualized) computing platform is a multicore integrating software from different providers.

We realize our model for the Cobham Gaisler NGMP processor [4][7], acknowledged as a potential on-board multicore platform for future European Space Agency's missions. We show how our model achieves high-accuracy in terms of predicting multicore contention on execution time, while requiring much shorter time to execute than full-fledged timing models. For EEMBC Automotive benchmarks and several benchmarks from the European Space Agency the average inaccuracy is only 19% with an average time to compute contention of less than 0.2 seconds.

II. Background and Problem Statement

In single-core integrated-architectures (such as IMA [1] in avionics and AUTOSAR [6] in automotive) early in the design the OEM assigns a CPU quota (budget) to each software supplier – together with the functionality to perform. In terms of timing, OEMs usually implement budgets via time partitioning: time is split into windows each of which is assigned to a different application, and hence to its corresponding supplier. From the supplier point of view, other than some overheads due to context switches, time analysis of its applications can be done in isolation. Interestingly, the interaction in I/O resources can be handled via forcing that the I/O operations of an application occur during its assigned window or during a specific period designated for that purpose (e.g. at the end of each time window in the context of cyclic-executive scheduling). Hence, single-core CPUs allow each supplier to easily design applications to fit in its assigned quota or negotiate with the OEM a larger quota. This can occur during the EDP, which reduces the cost of any change that is required on the timing or functional behavior of the system.

Multicores complicate this approach because the timing behavior of an application depends on its co-runners. Conceptually, the execution time of an application in a multicore (et^{muc}) can be broken down into two components as follows:

$$et^{muc} = et^{solo} + \Delta t \tag{1}$$

where et^{solo} is the execution time of the application in isolation and Δt is the execution time increase the application suffers due to contention in the access to multicore shared resources. While suppliers have confidence on the estimates derived for et^{solo}, the same cannot be said about Δt since it depends on co-runners the supplier does not know, and might not be allowed to know due to IP restrictions. Several studies show that Δt can be as high as et^{solo} [19][20], so it can have a great impact on the scheduling plan defined by the OEM to determine the budget and the specific time windows given to each application. If violations to assigned budgets are discovered during the LDP this may require costly application re-coding, changing the scheduling plan or even changing or adding more multicore CPUs if there are not enough computation capabilities to guarantee the execution of all the required functionalities. This, of course, may significantly increase the overall product (system) cost and time-to-market. Therefore, obtaining early and tight estimates of Δt is of great help to reduce the risk of LDP changes. There is a general consensus in the literature [11][14] that during the EDP accuracy of the timing estimates is not the only metric to consider, with tight upper-bounding estimates being rather required for LDP. Instead, the speed to obtaining those estimates plays a key role to allow engineers to explore a vast set of design choices in a timely manner. However, no particular figure is reported for the required accuracy in timing predictions during the EDP, which in our view is end-user dependent. In the context of multicores, it has been reported that the impact of contention in execution time can be as high as 20x for some kernels and as high as 5.5x for some EEMBC Automotive benchmarks [13]. In this respect, we deem the accuracy results obtained by our approach, which ranges between 0.6x and 1.4x, as sufficiently high to verify the scheduling plan during the EDP.

In the context defined in this paper, each software supplier is provided with a virtual machine (VM) that mimics the functional behavior of the target hardware, the NGMP in our case. This allows the supplier to develop and validate the functionality of its software. Each VM can be attached a timing simulator of the underlying multicore processor to derive timing estimates including the impact of contention. The main problem of this approach is that timing simulators incur a high

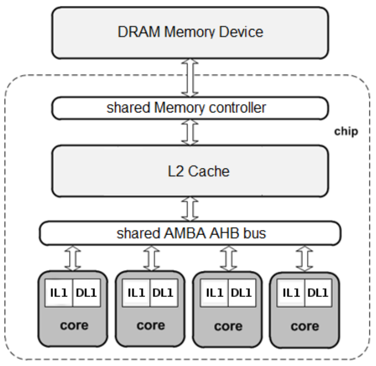


Fig. 1. Block Diagram of the 4-core NGMP architecture.

timing overhead: virtualization incurs performance penalties that are as low as few percentage points and can range up to 1x-2x slowdown depending on the virtualization technology and whether the host's ISA is the same as the simulated ISA. Instead, full-fledged timing simulators can be much slower because for each simulated instruction the timing simulator executes hundreds or even thousands of native instructions to model the delays incurred by the simulated instruction on the simulated CPU, cache, interconnection network, etc. This may lead to slowdowns in the 100x-1000x range. This is undesirable, for software suppliers who, despite willing to obtain timing estimates for their applications, cannot pay this overhead in the speed of the VM. Our approach i) controls time overhead by performing a characterization of each application in isolation, that despite being a slow process it is performed only once per application; and ii) speeds up the much more frequent computation of contention. Furthermore, detailed timing simulators require information about co-runner tasks that is unlikely to be available due to IP restrictions.

We focus on measurement-based timing analysis techniques. While a discussion of when static or measurement-based timing analysis approaches are convenient is out of the scope of this paper, it is a fact that different industries for different systems (functions) use both [28][3]. Hence, research on both techniques is needed so both are able to support multicore timing analysis in early and late phases of the system design. Our reference architecture is the Cobham Gailser Next-Generation Multipurpose Processor (NGMP) [7], sketched in Figure 1. The NGMP comprises four LEON4 cores, each core having a private instruction cache (iL1) and data cache (dL1), and a global (shared) unified second level cache (uL2). Cores and caches are connected with a bus. A memory controller acts as interface between the processor cores and memory.

III. Our Approach

In this section we present how the overall approach works, while in subsequent sections we detail its main steps. We build on an example with cyclic executive scheduling, widely used in many domains such as avionics and automotive – though our proposal is valid for other scheduling approaches. Cyclic executives divide time into major cycles (*mac*), which are further divided into minor cycles (*mic*). In each *mic*, several jobs are executed in a non-preemtable manner. Each job is required to finish in a *mic* (also called frame). Usually, *mic*s have the same duration to simplify implementation. While the jobs executed in each *mic* may vary making those *mics*

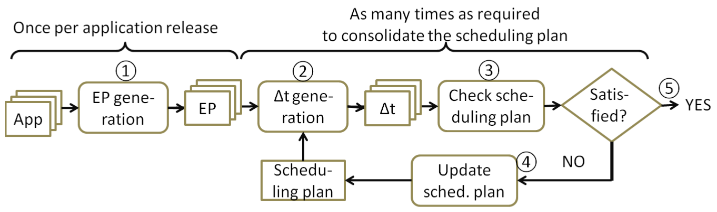


Fig. 2. Application steps of our approach.

different, all *macs* are identical. That is, each *mac* has exactly the same sequence of *mics* and set of jobs called in each *mic*. Despite its static nature, due to its simplicity a cyclic executive is often the preferred scheduling solution in real-time systems in domains such automotive and avionics in which the ARINC 653 standard recommends its use for partitions [2].

A. Application Process

Our approach builds upon the concept of an execution profile (EP) which encapsulates for each task information about its resource usage. The process to apply our approach involves the steps of generating the EP for each task and then combining several EPs – in accordance with a scheduling plan – by means of a contention model to derive Δt, see Figure 2.

EP generation ①. The EP is generated once per new release of each application, for which it is considered that the application usage of resources can significantly vary. Since in every new release of the application it performs its required functionalities more precisely, the EP generated for every new release better represents the actual use of resources of the final version of the application.

EP generation requires, as a first step, adding instrumentation code in the VM to extract information from the application execution. In particular, for every executed instruction information such as its opcode, program counter, etc. is extracted. This information is then processed to produce an EP that summarizes the execution information of the application, and reveals no functional information of the application, keeping its functionality confidential. While this process can be slow, it is performed just once for every application release (more details in Section IV).

Contention modelling via EP mixing. At the core of our approach we find the *Contention Model* (or *CM*), which combines (mixes) the EP of those applications that co-run in the multicore to predict Δt, see ② in Figure 2.

The OEM distributes the scheduling plan to every supplier together with the EP of all applications. This allows each supplier to determine those applications that are co-runners of its own ones. Each supplier uses the CM to estimate Δt (②) for each of its applications. Δt for each application along its corresponding et^{solo} is sent back to the OEM. If there is no violation of the budgets (③), the scheduling plan is deemed as valid (⑤). On the contrary, the OEM can increase the budget given to a supplier – if some slack is available – or change the scheduling plan. On its side, the supplier can also try to reduce the CPU requirements of its application (④).

As an illustrative example, Figure 3 shows a cyclic-executive based scheduling plan provided by the OEM from where each supplier can determine the co-runners of its application in each minor cycle (*mic*). For instance, in mic_1 applications A and B interact with C in the multicore so it

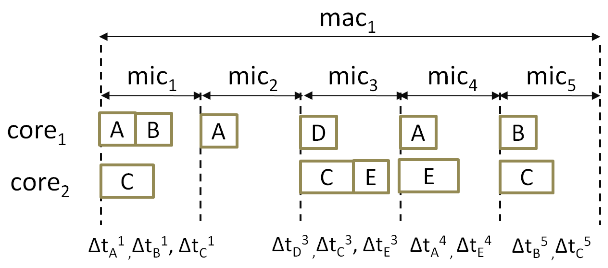


Fig. 3. EP usage to derive Δt

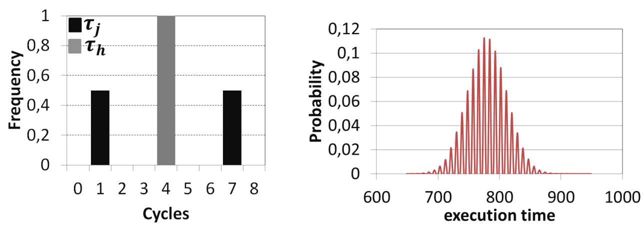
is required to derive Δt_A^1, Δt_B^1 and Δt_C^1, where Δt_A^1 corresponds to Δt of application A in mic_1. If both $et_A^{muc} + et_B^{muc}$ and et_C^{muc} fit in a *mic*, no change to the schedule (for this first *mic*) is required. The same process is repeated for all *mics*. Interestingly, in mic_3 if $et_C^{muc} > et_D^{muc}$ then E suffers no contention from D, i.e. $\Delta t_E^3 = 0$. It is worth noting that in the first iteration of this process between the OEM and the suppliers, the OEM creates a scheduling plan assuming no contention or a nominal contention based on its previous experience. This initial value is refined through the different iterations of our approach.

B. Histogram-based approach for contention modelling

Standard processor simulators keep the state of the modeled hardware in software data structures. The most clear example is cache memories that are usually modelled with bidimensional tables. Each cell, usually modelled with a `struct`, keeps information of a cache line including the LRU bits, valid bits, the data, etc. On the event of an access, data structures are searched and their internal state is appropriately updated, which is a time-consuming process.

In order to reduce simulation speed, we instead keep no information about the execution history and model each instruction in isolation. We use the information in the EPs of the task under analysis and the contender tasks to predict their timing behavior. In particular, we use distributions (histograms) to build a representative scenario so that the timing behavior of the instruction approximates that of the real execution. In the EP, the application's profile data such as frequency of access to the caches and hit rates are stored as distributions (histograms) rather than average values. Histograms are easy to capture and require relatively low space in the EP. Further, histograms help improving our model's accuracy, which we illustrate with the following example.

Let us assume that we want to model the impact on tasks τ_j and τ_h execution time when they share a single-entry cache. Further, assume that all of τ_j's accesses go to a given address and τ_h's accesses go to a different address. Cache hits take 1 cycle and misses 10 cycles. In this scenario, consecutive accesses from the same task result in hits and interleaved accesses among tasks result in misses since tasks evict each other's data from cache. For the purpose of this example further assume as input data for the model τ_j's and τ_h's distribution of time between consecutive accesses (i.e. the time since an accesses is performed in cache until the following one is sent). These histograms, which are respectively called T_j and T_h, are as shown in Figure 4(a): every τ_h access is sent to cache 4 cycles after the last one completed, while 50% of



(a) T_j and T_h for τ_j and τ_h (b) τ_h's execution time prediction

Fig. 4. Example of use of the histogram-based modelling approach

τ_j requests are sent one cycle after the previous one and the other 50% every 7 cycles after the previous one completes.

Average. If we use information about average both τ_j and τ_h are assumed to send one request to the cache at the same rate of every 4 cycles. As a result all accesses would interleave and be misses. If τ_j performs 100 accesses, its predicted execution time would be 1,000 cycles. This fails to capture the fact that τ_j has a bimodal distribution, which means that the number of requests from τ_j among requests of τ_h varies: it can be 0, 1, 2 or 3 and hence some accesses can be hits.

Histogram. With the approach based on histograms, for every τ_j access we derive the time since the last access according to its histogram. To that end we define a random variable X modelling frequencies in the histogram as probabilities. For instance, T_j is the random variable capturing the distribution of cycles between consecutive τ_j's accesses.

We refer to a realization of the random variable (distribution) X as x, i.e. the name of the random variable but in lower case. Hence, t_j is one particular value obtained from the distribution T_j. For instance, to obtain t_j we generate a random number (r) between 0 and 1, so $r \in [0, 1)$. Given that the time between accesses for τ_j is 1 or 7 cycles with 50% probability each, t_j is 1 or 7 as follows:

$$t_j = \begin{cases} 1\ cycle & if \quad (r < 0.5) \\ 7\ cycles & if \quad (r \geq 0.5) \end{cases}$$

This process for obtaining one value from a random variable, which can be performed for histograms with any number of points and density, is called *realization*. We represent it as $x = rand(X)$, that for the case of the time between accesses is $t_j = rand(T_j)$.

Coming back to the example in Figure 4, the histogram based approach results in τ_j and τ_h experiencing hits and misses – as it would be expected based on their frequency of access. To obtain the predicted execution time for τ_j, we perform several *runs of the model*. In each run, the access delay of each access is obtained by performing one realization of T_j (e.g., $t_j = 7$) and as many as needed of T_h to determine how many accesses of τ_h occurred since the previous access of τ_j. The estimate obtained for the execution time distribution for τ_j is as shown in Figure 4(b). We observe that the resulting distribution captures the fact that the alignment between tasks' accesses impacts each task's execution time. The average execution time is 779 cycles instead of 1,000 as with the average-based model.

Our results in Section VI show for real benchmarks that taking averages instead of considering the histogram leads to high inaccuracies since accesses interleave systematically in the same way despite the fact that, in reality, they interleave

TABLE I
BASIC NOTATION.

		Symbol	Description	Comments
per task (τ_j)	cache	TS_j	Time between τ_j's access to the same Set	Apply to each cache: iL1(i), dL1(d) and uL2(u) for the NGMP, e.g., Ki_j and Ku_j are the stack distance of the access to iL1 uL2 respectiv.
		K_j	StacK distance of τ_j's access to cache	
		E_j	sEt distance of τ_j's access to cache	
		H_j	Hit Rate of τ_j's access to cache	
		d_j	Set dispersion of τ_j's accesses to cache	-
		Δm_j^{uL2}	Increment in miss count in uL2 that τ_j suffers due to its contender tasks	-
		et_j^{solo}	Execution Time estimate for τ_j in isolation	-
	Time	et_j^{uL2}	Exec. Time estimate for τ_j factoring in Δm_j^{uL2}	-
		et_j^{muc}	Execution Time estimate for τ_j in multicore	All increments are caused by τ_j's contender tasks accesses to the different hardware shared resources
		Δt_j	Time increment τ_j suffers in multicore	
		Δt_j^{BUS}	Time increment τ_j suffers due to bus sharing	
		Δt_j^{MEM}	Time increment τ_j suffers due to mem. sharing	
		Δt_j^{uL2}	Time increment τ_j suffers due to L2 sharing	
		$etbus_j^{uL2}$	Time task τ_j uses the bus factoring in Δm_j^{l2}	-
		$ubus_j^{uL2}$	τ_j's utilization of the bus factoring in Δm_j^{l2}	-
		$ubus_{c(j)}$	Utilization of the bus of τ_j's contender tasks	-
		$abus_j$	Availability (percentage of cycles) of the bus for τ_j when running with its contenders	-
	instr-uction	n_{total}	Total instruction count of τ_j	-
		I_{mix}	Percentage of instructions of each type for τ_j	-
		n_y	Number of instructions of type *y*	-
per access (@A)		$t_{@Al}$	Time between accesses $@A_l$ and $@A_{l-1}$	-
		$k_{@Al}^{solo}$	Stack distance of access $@A_l$ in singlecore	-
		$k_{@Al}^{muc}$	Stack distance of access $@A_l$ in multicore	-
		$\Delta t_{@}^{bus}$	Time increment an access suffers in the bus	-
		$a_{@Al}^{h}$	Number of intermediate accesses τ_h generates between $@A_l$ and $@A_{l-1}$	-
		$\Delta k_{@Al}^{h}$	Increment in $@A_l$'s set distance caused by the intermediate accessed generated by τ_h	-
		$\Delta t_{@}^{mem}$	Time increment an access suffers in memory	-
instruction		l_y	Latency of the instructions of type *y*	-
cache		s	Number of sets in cache	Apply to each cache level, e.g. s_d is the number of dL1 sets
		w	Number of ways in cache	

in many different ways. Instead, histogram-based interleaving captures tasks' access interleaving much more accurately.

C. Restricted Access Interleaving Information

A key element in our approach is that we assume no information about the distribution (over time) of the accesses of a given program to hardware resources. For instance, for τ_j in Figure 4(a) we know 50% of the accesses are sent 1 cycle after the previous completes and the other 50% 7 cycles. Our model does not record, for instance, information about whether those accesses concentrate on the initial phase of the execution of the program or at the end, that is, how they interleave with other task instructions. If we assume that τ_j and τ_h run in parallel it could be the case that all τ_j accesses occur before those of τ_h, so in reality they are not going to suffer inter-task contention in cache. Likewise, we do not record information about whether τ_j accesses of the different types interleave.

There are several reasons behind this choice. (i) Keeping time-dependent distribution information would increase the size of EP, since we could not summarize it in a histogram but we would need to keep the exact sequence of accesses and how they interleave over time. This would also result in more complex and time-consuming modelling. (ii) This approach would also affect time composability [23][22] since provided contention bounds would be specific to how requests interleave, which changes when tasks suffer any type of shift.

Since our model aims at predicting the worst-case contention among tasks, not the average, whenever two tasks partially overlap in the scheduling plan we pessimistically increase their et^{muc} assuming they fully overlap and hence, suffer continuously high contention. Despite this adds some pessimism, it simplifies EP and makes the CM lighter – which is critical since the CM is used in an iterative manner to adjust the scheduling plan, so it has to incur a small slowdown.

D. Notation and Parameters

The main parameters used for our model are those in Table I. We introduce some of them in this section, while others are presented as they are used. In Table I, starting bottom up, we observe that parameters provide cache, per-instruction, per-access information and per-task information. The latter is further broken down into cache, time and instruction information of the task. For cache information of the task we use the convention $metric - cache - task$: $metric$ are the initials of the metric described (in capital letters to mean it is a random variable, i.e. histogram); $cache$ is cache initial, that for the NGMP is i for the instruction cache, d for the data cache and u for the unified L2 cache. Finally $task$ is the task id that is added as subindex. For instance, Kd_j is the stack distance of the accesses to dL1 of task τ_j. When we talk about a cache in general we omit the cache initial, e.g K_j.

We introduce some of Table I's parameters via the example sequence in Figure 5. For each access we report its address, accessed cache set and the time in which it happens.

- *Time between accesses to the same set* (TS) captures the time between accesses targeting the same set. For instance, accesses #2 and #8 to addresses $@D$ and $@E$ respectively, are consecutive accesses to set 1 (set_1). They occur in cycles 4 and 25 respectively, which results in a time between accesses to the same set of 21 cycles.
- *Set distance* (E) for a given set set_i is the number of other sets (different than set_i but not necessarily unique) accessed since the last time set_i was accessed. For instance, access #2 in *Seq1* accesses set_1, which is accessed again by access #8. In between there are 5 accesses to sets different than set_1, hence set distance equals 5 for #8.
- *Stack distance* (K). The stack distance of an access $@A_l$ is defined as the number of unique (i.e. non-repeated) addresses mapped to the same set where $@A_l$ is mapped and that are accessed between $@A_l$ and the previous access to it, i.e. $@A_{l-1}$. Note that stack distance is similar to the concept of reuse distance, though the latter does not break down accesses per set. For instance, in *Seq1* the accesses to set_0 are ($AABCBAAABC$) with stack distances ($\infty 0 \infty \infty 120012$) respectively. The stack distance of a task τ_j (K_j) is built from the stack distances of its accesses.

It is worth noting that common eviction cache policies such as LRU have the stack property [18], which determines whether a given address is still in cache. For LRU, the focus of this paper, each set in a cache can be seen as an LRU stack with lines sorted based on their last access cycle. The first line of the LRU stack is the Most Recently Used (MRU) and the last is the LRU. Interestingly, i) the position of a line in the LRU stack defines its stack distance; ii) those accesses with a stack distance smaller than or equal to the number of cache ways (w) result in a hit and vice versa; and iii) the sensitivity of the access of τ_j to be evicted from cache by accesses of a contending task τ_h depends on its stack distance: the higher it is, the higher its sensitivity.

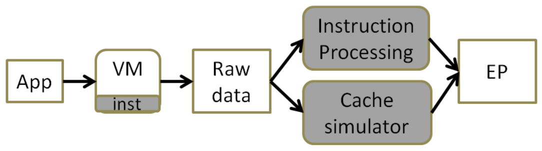


Fig. 6. View of the software modules required to derive EP.

The process followed to collect these parameters from the execution of a given task is described in the next Section.

IV. EP generation and Format

The time modeling process starts with the collection of relevant data that represent the main traits of an application – when running the application on an instrumented VM – and their post-processing. The result of the processing is an EP per application, which is used by the contention model to predict Δt for each application under a given schedule.

We use several pieces of information to model multicore contention. They are based on the appreciation that multicore designs for real-time systems comprise a relatively simple – usually in-order – execution pipeline, with some local caches and a global shared cache, as it is the case for the NGMP.

The process to generate the EP starts by instrumenting the VM so it outputs for every emulated instruction its opcode, program counter and data address (for memory operations). Most VM provide functions (APIs) to access the information of each executed instruction including its Program Counter, destination address (in case it is a memory operation), accessed registers including their values, operation code, etc. These functions are invoked right after the *execute()* function of the VM that emulates its behavior in the simulated machine, i.e. LEON4 in our case. The VM instrumentation is activated so that for each instruction of the application – or selected parts of it (e.g. excluding the initialization code) – this information is extracted. While the overall process is slow, it is done just once for every new release of the application.

From these 'raw data', a cache simulator and an instruction processing module are used to generate the EP, see Figure 6.

In our case we use a state-of-the-art multi-level cache simulator that is highly parametrizable including cache line size, number of ways, number of sets, placement and replacement policy, inclusion policy, etc. Below we list the fields in the EP of each application and explain how they are derived.

1) *Instruction count*: For each task we keep the total number of instructions it executed n_{total}.
2) *Per-type instruction count*: We keep the number of instructions of each type (n_y) in the task. This can be obtained from the opcode of each instruction. It follows that $\sum_{y \in \mathcal{Y}} n_y = n_{total}$, where $\mathcal{Y}$ is the set of all instruction types.
3) *Instruction mix*. I_{mix} provides the distribution of instructions across types, i.e. n_y/n_{total} for each type y.
4) *Per-type instruction latencies*. It provides information about the latency of each different type of instruction (l_y). This information can be derived by benchmarking [12] or can be found in the user manuals provided by the chip vendor. The information provided covers the core latency of operations and the latency of the local caches, global caches and the main memory for load/store operations. We differentiate the

Access number	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
address	@A	@D	@A	@B	@F	@C	@B	@E	@A	@A	@F	@A	@B	@E	@C	@A	@F
accessed set	s0	s1	s0	s0	s2	s0	s0	s1	s0	s0	s2	s0	s0	s1	s0	s0	s2
time	1	4	10	14	16	20	22	25	32	36	40	41	43	50	56	58	60
TS_j	0	0	9	4	0	6	2	21	10	4	24	5	2	25	13	2	20
E_j	∞	∞	1	0	∞	1	0	5	1	0	5	1	0	5	1	0	5
K_j	∞	∞	0	∞	∞	∞	1	∞	2	0	0	0	1	0	2	2	0

Fig. 5. Access Sequence *Seq1* generated by a task τ_j

TABLE II
OPERATION TYPES IN THE NGMP AND THEIR ASSUMED LATENCIES

operation type	jitter	min-max latency	Assumed latency
int. short latency	NO	1	1
int. long latency	YES	1-35	35
control	NO	1	1
fp. short latency	NO	4	4
fp. long latency	YES	16-25	25

following instruction types ($\mathcal{Y}$) since they are common in several RISC architectures: integer short latency (e.g., `add`, `cmp`), integer long latency (e.g., `idiv`, `imult`), control (e.g., `bne`), floating point short latency (e.g., `fpadd`, `fpmult`), floating point long latency (e.g., `fpdiv` and `fpsqrt`) and memory operations (e.g., `ld` and `st`). Some of these types can be further divided. For instance, the floating point long latency type can be split into divisions (`fpdiv`) and square roots (`fpsqrt`) since the execution time of these two instruction types can be quite different.

It is noted that each instruction instance may suffer variability in its execution time due to two factors. First, *input-data dependence* that occurs when instructions such as floating-point division take variable latency depending on the particular values (input-data) operated. And second, *pipeline state* dependence: in this case, a given instruction may have variable latency depending on its predecessor instructions. In our aim to model the worst-case we handle these sources of jitter by assuming as the latency for every type an upperbound to those latencies. This provides an upperbound to the execution of the instruction. This incurs relatively low inaccuracy while keeping the model simple and fast. For the NGMP, which has mostly a stall-free pipeline, so removing pipeline-state dependences, Table II shows the latency we assume in our model for every instruction type.

5) *Local caches information*: For the local caches, our cache simulator provides the hit rate. In particular for each τ_j we keep Hi_j and Hd_j.

6) *Global caches information*: As for the local caches, we record information on the hit rate of the application for the global caches, i.e. Hu_j for the NGMP.

7) *Inter-access latency*: For every core instruction executed we have its latency l_y. For instruction and cache accesses the cache simulator is used to determine whether they hit/miss in the different cache levels, for which we have an associated latency. With this information, for every two accesses to uL2 we can predict the execution time of the instructions between them, and hence we can derive the time between consecutive accesses. The histogram (TSu_j) is derived by counting how many times each latency occurs between two consecutive uL2 accesses to the same set.

8) *Ku_j and Eu_j*: From the cache simulator it is straightforward to derive *stack distance* and *set distance* since we have the memory operations accessing uL2 and the set they access.

TABLE III
SPECIFIC INFORMATION IN THE EXECUTION PROFILE

Hi_j, Hd_j, Hu_j	cache hit rates (miss rates derived as $(1 - Hx_j)$)
TSu_j	Time between accesses going to the same set in uL2
Ku_j	uL2 stack distance of τ_i (including all data accesses)
Eu_j	uL2 set distance of τ_i (including all data accesses)
I_{mix}	Percentage of instructions of each type
n_{total},n_y	Total and per-type instruction count
l_y	Nominal back-end latency per instruction type
et_j^{solo}	τ_j's execution time in isolation

9) *Solo performance* . In our environment, software suppliers are provided with a virtualized environment (e.g. for the SPARC based NGMP in our case) that runs on a host platform (e.g. x86) where et^{solo} – that is the first addend in Equation 5 – cannot be derived. We derive et^{solo} by applying a simple approach in which for each instruction we add its front-end (of the pipeline) latency and its back-end latency.

$$et_j^{solo} = \sum_{y \in \mathcal{Y}} [n_y \times (fend(y) + bend(y))] \tag{2}$$

Instruction's front-end latency depend on whether they hit or miss in iL1 and uL2. For each of these scenarios their associated latency is different. Whether an instruction hits/misses in cache is provided by the cache simulator. For core operations, such as `add` or `mult`, l_y gives an estimate of their execution time in the back-end. For memory operations such as `load` or `store`, their back-end latency also depends on whether they hit or miss in dL1 and uL2.

Overall, the EP generation deploys a high-level cache simulator and some extra modules that process the information coming from the instrumentation of the VM. As a result we obtain an EP that is the input for contention modelling. The information in the EP is summarized in Table III.

V. CONTENTION MODELING

In this section we explain the main elements of our contention model as well as the assumptions on which it holds.

The main shared resources we consider in our target architecture (see Figure 1) are the uL2 cache, the bus and the memory bandwidth. For the former we explain our contention model in Section V-A and for the latter two in Section V-B. Our model keeps no state information about executed instructions, i.e. it models each instruction in isolation.

First, the cache contention model predicts the increment in number of misses that τ_j suffers due to the contention created by its co-runners, Δm_j^{uL2}. Then it derives the execution time increment (Δt_j^{uL2}) caused by those Δm_j^{uL2} hits in the execution in isolation that become misses due to its co-runners.

In a second step, we account for the impact on the bus and the memory controller that each access suffers when accessing those resources. The access latency of each uL2 hit is increased by the contention on the bus:

$$l_{uL2h}^{muc} = l_{uL2h}^{solo} + \Delta t_{@}^{bus} \tag{3}$$

Likewise, the time it takes the memory to serve a uL2 miss increases due to contention on the bus and memory:

$$l_{uL2m}^{muc} = l_{uL2m}^{solo} + \Delta t_{@}^{bus} + \Delta t_{@}^{mem} \quad (4)$$

From the number of hits and misses in the private caches; the increment in misses in the uL2; and hit and miss delays in the access to the cache/memory (as presented in Equation 3 and Equation 4) we derive the overall execution time increase due to the bus and the memory, called Δt_j^{BUS} and Δt_j^{MEM}, respectively. With this, the overall execution time on multicore is predicted as:

$$et_j^{muc} = et_j^{solo} + \Delta t_j^{uL2} + \Delta t_j^{BUS} + \Delta t_j^{MEM} \quad (5)$$

A. Cache Contention Model

The whole modeling process starts by iterating on a loop several times[2], with each loop iteration modelling the duration of one instruction. At the beginning of the iteration we define the type of the instruction by performing a realization of I_{mix}. For that instruction, based on its type y, we obtain its backend latency l_y. Then we perform one realization of Hi_j to determine whether the instruction hits in the iL1. If the instruction is a memory operation, we determine whether it hits in dL1 performing a realization of Hd_j. If it is determined that the instruction misses in iL1 or dL1, we determine whether it hits in uL2 by performing one realization of Hu_j for each iL1 and dL1 miss.

Hits in iL1 and dL1 are not affected by cache contention since first level caches are private and non-inclusive. Interestingly, uL2 misses are not affected either. Only uL2 hits can become misses due to evictions of contender tasks. Our model does not keep execution history information across instructions. Instead, the contention model for each uL2 access $@A_l$ of the task under analysis τ_j determines the increase in stack distance it suffers due to the accesses that contender tasks injected since the last access to address $@A$, i.e. $@A_{l-1}$. Potentially, each such intermediate access can increase the stack distance of $@A_l$. Overall, if $@A_l$'s stack distance in isolation ($k_{@A_l}^{solo}$) is smaller or equal to the number of uL2 ways (w_{uL2}) and its multicore stack distance ($k_{@A_l}^{muc}$) becomes larger than the number of ways, see Expression 6, then in single-core $@A_l$ is a hit and in multicore it is a miss, increasing Δm_j^{l2}.

$$(k_{@A_l}^{solo} \leq w_{uL2}) \quad \text{and} \quad (k_{@A_l}^{muc} > w_{uL2}) \quad (6)$$

The challenge lies on deriving $k_{@A_l}^{muc}$. To do so we estimate the time between two consecutive accesses to the same address $@A$ and how many lines contenders fetch to that cache set during that interval as follows: the accesses going to a given set are spread over the different lines in that set. For a given access $@A_l$ from τ_j hitting in uL2 its stack distance – computed as $k_{@A_l}^{solo} = rand(Ku_j)$ – provides the number of accesses from τ_j (to that set) before $@A$ is accessed again, i.e. the number of accesses between $@A_{l-1}$ and $@A_l$. By multiplying $k_{@A_l}^{solo}$ and the time between accesses to the same set $tsu_j = rand(TSu_j)$ we obtain $t_{@A_l}^{solo}$, the time between $@A_l$ and the previous access to the same address, i.e. $@A_{l-1}$.

$$t_{@A_l}^{solo} = tsu_j \times k_{@A_l}^{solo} \quad (7)$$

As a second step, we compute the number of accesses τ_j's contender tasks (i.e. $\tau_h \in c(\tau_j)$) introduce between $@A_l$ and $@A_{l-1}$ in the same uL2 set. To that end we use TSu_h that provides the time it takes τ_h to send accesses to the same set. By dividing $t_{@A_l}^{solo}$ and $tsu_h = rand(TSu_h)$ we obtain the number of intermediate accesses injected by each contender task τ_h between A_l and A_{l-1}, see first addend in Equation 8. When the $t_{@A_l}^{solo}$ and tsu_h are not multipliers, τ_h can generate one extra access. This is determined by generating a random number between 0 and tsu_h. If this number is lower than the remaining time until the next τ_h access, then we assume that the contender was able to introduce one extra access.

$$a_{@Al}^{h} = \left\lfloor \frac{t_{@Al}^{solo}}{tsu_h} \right\rfloor + min\left(\left\lfloor \frac{t_{@Al}^{solo} \mod tsu_h}{rand(1, tsu_h)} \right\rfloor, 1\right) \quad (8)$$

For instance, if $t_{@A_l}^{solo} = 30$ and $tsu_h = 9$, then $\left\lfloor \frac{t_{@Al}^{solo}}{tsu_h} \right\rfloor = 3$. Depending on how accesses align in time, one extra access could occur during those $t_{@A_l}^{solo} = 30$ cycles. If $@A_{l-1}$ occurs at cycle 0 and $@A_l$ at cycle 30, τ_h accesses could occur for instance at cycles 5, 14 and 23 (so $a_{@Al}^{h} = 3$) or at cycles 2, 11, 20, 29 (so $a_{@Al}^{h} = 4$). This is taken into account with the second addend in Equation 8.

Set collision distribution. We also consider whether contender accesses can really interfere τ_j accesses because they span across several sets and thus, have low chances of interfering τ_j accesses. For instance, let us assume that τ_h accesses a single set, while τ_j accesses a high number of cache sets. For every τ_j access, i.e. $@A_l$, our previous model (Equation 8) assumes that its stack distance is affected by τ_h intermediate accesses. In reality, however, τ_h affects at most those accesses mapped to one particular set.

To capture this effect we define *set dispersion* as the probability that contenders' intermediate accesses go to the same set where $@A_l$ is mapped. To compute set dispersion we average set distance from Eu_h. This gives us an indication of the number of sets used by τ_h. For instance, if it is 2, then on average τ_h accesses two different sets. By dividing this by the total number of sets su in cache we approximate the τ_h's utilization of sets (i.e its dispersion), du_h. Set dispersion is used to decide whether all τ_h's intermediate accesses among $@A_l$ and $@A_{l-1}$, i.e $a_{@Al}^{h}$, actually contend with $@A_l$.

The probability that those $a_{@Al}^{h}$ accesses are mapped $@A_l$'s set in the uL2 is du_j. Hence, there is probability $P = 1 - du_h$ they are not, in which case they do not contend with $@A_l$. Thus, $a_{@Al}^{h}$ is redefined as follows:

$$a_{@Al}^{h} = \left\{ \begin{array}{rcl} a_{@Al}^{h}(as\ in\ Eq.\ 8) & : & rand(0,1) < du_h \\ 0 & : & rand(0,1) \geq du_h \end{array} \right. \quad (9)$$

Increment in stack distance. All $a_{@Al}^{h}$ intermediate accesses injected by each contender task τ_h can increase $@A_l$'s stack distance, i.e $k_{@A_l}^{muc}$. In reality the increase produced

[2]Since a number of random variable realizations are performed for each instruction to determine whether for each realization it hits or misses in each cache level, whether it is interfered by co-runners, etc., latency may change for each instruction realization, and so for the full application. In fact, the execution time of the application is a Monte-Carlo process built upon many Monte-Carlo processes (i.e. all realizations of histogram-based events for each instruction). Thus, if applications execute tens of thousands of instructions, iterating 100 times for each instruction already triggers millions of random events for the whole application so that Monte-Carlo inaccuracy falls orders of magnitude below the inaccuracy due to the simplifications of the model.

depends on the number of accesses generated by τ_h and their stack distance. In one extreme of the spectrum, when all intermediate accesses have stack distance equal to zero they go to the same cache block (line), so they increase $k^{muc}_{@A_l}$ just by one. In the other extreme, when all accesses go to different addresses (blocks) their impact on the stack distance is as high as the number of accesses. For instance, let us assume that τ_h generates 4 accesses between $@A_l$ and $@A_{l-1}$. If their stack distance is 0, they all access the same address (e.g., $BBBB$). If their stack distance is 1, then they access 2 different addresses (e.g., $BCBC$). In particular, the number of different addresses accessed by τ_h is determined by the stack distance (plus one), but never exceeding the total number of $a^h_{@Al}$ intermediate accesses. Note that for each τ_h access we obtain its stack distance as $rand(Ku_h)$.

We take into account the impact of the accesses generated by τ_h on $@A_l$'s stack distance ($\Delta k^h_{@Al}$) as shown in Equation 10. This equation assumes that all accesses have the same stack distance. This simplification still builds upon stack distance histograms, but reduces the cost of performing a realization of Ku_h for each $a^h_{@Al}$ intermediate access.

$$\Delta k^h_{@Al} = min(a^h_{@Al}, rand(Ku_h) + 1) \qquad (10)$$

By accounting for the increase in the stack distance of $k^{solo}_{@A_l}$ caused by each τ_h we derive its $k^{muc}_{@A_l}$:

$$k^{muc}_{@A_l} = k^{solo}_{@A_l} + \sum_{\tau_h \in c(\tau_j)} \Delta k^h_{@Al} \qquad (11)$$

Outcome. The addition of the stack distance of A_l in isolation ($k^{solo}_{@A_l}$) and $\Delta a^h_{@Al}$ for each contender provides $k^{muc}_{@A_l}$. As expressed in Equation 6, if $k^{solo}_{@A_l}$ is smaller or equal to the number of uL2 cache ways and $k^{muc}_{@A_l}$ is larger than the number of ways, that particular access is predicted to be a hit when τ_j runs in isolation and becomes a miss when it runs with its contenders, increasing Δt^{uL2}_j.

B. Bus and memory (BM) and (MM)

The bus and memory contention models for the NGMP take into account that bus accesses are not split[3], so that the bus contention model also captures contention tasks suffer in memory. If bus transactions were split, allowing requests from different tasks contend on the bus and in memory in parallel, then the model should be duplicated for both resources.

Our model estimates the contention experienced by task τ_j on the bus (and memory) that we refer to as Δt^{BUS}_j and is measured in cycles. For that purpose we derive (1) how long τ_j spends using the bus, $etbus^{uL2}_j$, which already factors in the impact of contention misses derived with the cache contention model; (2) the bus utilization of its contenders $ubus_{c(j)}$; (3) the probability than on an access to the bus τ_j finds it available $abus_j$; to finally (4) obtain Δt^{BUS}_j.

The time τ_j spends on the bus, $etbus^{uL2}_j$, is obtained by adding the time spent serving uL2 cache hits and misses in isolation and the time to serve the uL2 contention misses.

[3]In the NGMP the bus is not relinquished until the access is served either by the uL2 cache or memory.

$$\begin{aligned} etbus^{uL2}_j &= ((n^{solo}_{uL2h} - \Delta m^{uL2}_j) \times l_{uL2hit}) + \\ &\quad ((n^{solo}_{uL2m} + \Delta m^{uL2}_j) \times l_{uL2miss}) \\ &= (n^{solo}_{uL2h} \times l_{uL2hit}) + \\ &\quad (n^{solo}_{uL2m} \times l_{uL2miss}) + \\ &\quad \big(\Delta m^{uL2}_j \times (l_{uL2miss} - l_{uL2hit})\big) \qquad (12) \end{aligned}$$

where n^{solo}_{uL2h} and n^{solo}_{uL2m} correspond to the number of uL2 hits and misses in isolation, respectively; and Δm^{uL2}_j of the uL2 hits in isolation become miss due to contention. Therefore, for the bus utilization, $etbus^{uL2}_j$ factors in uL2 hits and misses applying the cache contention model, but not their impact on the bus contention. We obtain the total solo execution time plus the effect of cache contention analogously, by adding the impact of the extra misses due to contention:

$$et^{uL2}_j = et^{solo}_j + \Delta m^{uL2}_j \times (l_{uL2miss} - l_{uL2hit}) \qquad (13)$$

By dividing the time spent using the bus by $et^{uL2}_j = et^{solo}_j$ for a given task, we obtain its bus occupancy in isolation factoring in cache contention, $ubus^{uL2}_j$.

$$ubus^{uL2}_j = \frac{etbus^{uL2}_j}{et^{uL2}_j} \qquad (14)$$

Then, we add the utilization of all contenders of τ_j to obtain the accumulate utilization they cause.

$$ubus_{c(j)} = \sum_{\tau_h \in c(\tau_j)} ubus^{uL2}_h \qquad (15)$$

Note, however, that $ubus_{c(j)}$ can be higher than 100%. For instance, let assume τ_j has two contenders, τ_h and τ_m, each with bus utilization of 60%. This results in a bus utilization of $ubus_{c(j)} = 1.2$ that will be even higher when accounting for τ_j utilization in isolation. It is obvious that contention will increase the time window since total bus utilization of τ_j and its contenders cannot exceed 100%. The time window increases by the true contention that the other tasks cause on τ_j. However, the actual impact of contention on the bus is the result of the model. For instance, recalling the previous example, the impact of τ_h and τ_m bus accesses will increase τ_j execution time, thus increasing bus availability (same number of accesses over a larger time window). However, such increased bus availability is already needed to compute the time window, thus creating a circular dependence.

To break this dependence, we upper-bound contention impact in the time window with the total utilization of the other tasks. See the right addend of Equation 16, where the time window available is 1 (available utilization in isolation) plus the time that other tasks access the bus ($ubus_{c(j)}$). Hence, the actual bus utilization is approximated by dividing the utilization of the other tasks by the total time window. τ_j finds the bus available ($abus_j$) when it is not being used by others.

$$abus_j = 1 - \frac{ubus_{c(j)}}{1 + ubus_{c(j)}} \qquad (16)$$

In the example before, if $ubus_{c(j)} = 1.2$ then $abus_j = 0.45$, so this is the probability assumed for τ_j accesses to find the bus available. Note that in the equation above, if the other tasks did not use the bus, $abus_j = 1$. Conversely, if the utilization of the other tasks was huge, then $abus_j \approx 0$.

Finally, by dividing 1 by the probability of success we obtain how often a bus access from τ_j tries to access the bus to get it. Then we multiply this probability by the time spent on the bus in isolation $etbus_j^{uL2}$ to obtain the total time spent on the bus due to contention, Δt_j^{BUS}, as shown below.

$$\Delta t_j^{BUS} = \left(\frac{1}{abus_j}\right) \times etbus_j^{uL2} \tag{17}$$

C. Simplifications of the Model

Our model balances accuracy in its prediction and the execution time overhead to run it. Results presented in Section VI show that our model achieves a good balance between both. In this section we list the main simplifications in our model to speed it up and how they impact accuracy.

General approach. The most remarkable assumption in our models is that we take frequencies observed when characterizing applications as probabilities. However, events such as cache hits/misses do not have a randomized nature as it would be required to attach probabilities to their occurrence. For the sake of simplicity we make this assumption for any process to limit the complexity of the cache model.

Core model. In general our model does not work with the temporal distribution of events. For instance, we assume that instructions of each type are equidistant in the code. Despite we have histograms we do not consider how instructions are distributed over time. For computing processor core time, we over-approximate the execution time of each instruction in the core instead of tracking pipeline and data dependences, that is, we assume the longest latency of a jittery instruction, e.g. for the NGMP we assume that `fplong`, i.e. fp long-latency instruction we assume, always takes 25 cycles when in reality it can take either 16 or 25 cycles depending on the input values operated. This heavily simplifies the processor core time model with low impact on accuracy.

Cache model. For the cache model, stack distances and set distances are not maintained per set, but we have one stack distance histogram and one set distance histogram for the whole cache. Alternatively, to increase accuracy we could keep information in a per-set basis, but this would add some non-negligible complexity to the model, would increase the amount of information recorded by a factor of s (the number of sets in cache), and would make results dependent on the actual sets (and so memory locations) of the different tasks. Another source of innacuracy is that we average set distance to approximate set dispersion.

Bus model. In the bus model there are two main sources of inaccuracy. First, when determining the bus availability for a task the time window assumed is upper-bounded. Second and more important, our bus model assumes that bus accesses are blocking, i.e. they stall the processor pipeline. However, the LEON4 cores in the NGMP have a store buffer able to manage a pending store without stalling the pipeline. Since dL1 caches are write-through in the LEON4 core, write operations to uL2 are abundant. Hence, the bus contention effect will not be as linear as assumed, resulting in under-estimations of the model.

As shown in the result section, the overall accuracy of the model is acceptable as execution time estimates to be used during the EDP, while it incurs low execution time overhead.

Addressable unit. For sake of simplicity we have assumed in our explanations that each access corresponds to a cache line. When the addressable unit is smaller than a cache line, accesses to different addresses can be mapped to the same cache line. This has no impact on our previous formulation. For instance, let us assume the sequence (A, A, B, C, B, A), in which B and C go to the same line. We can simply abstract this sequence as (A, A, B, B, B, A), hence considering that the access to C corresponds to another access to B, so cache stack distances would be (∞ 0 ∞ 0 0 0 1). This allows us applying the formulation presented.

VI. Realization and evaluation on the NGMP

In this section we show how we realize our models for the NGMP [7], whose general block diagram is shown in Figure 1.

A. Experimental Framework

Basics on the NGMP. In the space domain the NGMP [7], whose latest implementation is the GR740 [8], is being considered by the European Space Agency (ESA) for its future missions. Each LEON4 core comprises a seven-stage scalar in-order pipeline. The NGMP has four cores, each comprising instruction (iL1) and data (dL1) first-level caches. Each core accesses to the L2 cache through an AHB AMBA bus.

Floating point. The FPU has two possible data paths for FP operations: one for FP divisions and square roots, and one for the rest of FP operations. This last data path is fully pipelined, and has a latency of 4 cycles and, under ideal conditions, a throughput of one cycle. The data path for divisions and square roots is not pipelined, so the throughput of these operations is equal to their latency.

Bus: The interconnection bus is a 128-bit AHB AMBA bus arbitrated with round-robin policy. It has 5 masters (4 cores, and I/O master) and 2 slaves (L2 cache and I/O slave). As explained before, the bus-split feature is not implemented in the GR-CPCI-LEON4-N2X [5], meaning that once a core has its access to the bus granted, no other device can access the bus until the access is performed. Our model has been fit to this architecture where bus-split is not available.

Memory model: Every core has 16KB 4-way 128-set iL1 and dL1 caches, implementing write-through, write no allocate policies. The unified L2 cache is shared by all the cores in the processor. It is a 256KB 4-way 2048-set cache. It uses write policy is copy-back/write allocate. Cores also include some store buffers capabilities.

Simulator. We have modeled this architecture on an enhanced version of the SoCLib simulation framework [25] and validated that execution time measurements obtained in the simulator are on average within 3% of those obtained in the real board for the N2X implementation of the NGMP [5] for the EEMBC benchmark suite [21] as well as some space applications. We use the execution times obtained from the simulator as reference to assess the accuracy of our model.

Benchmarks. We use a solid set of benchmarks to assess our contention model.

EEMBC automotive. The EEMBC automotive benchmark suite [21] is a well-known benchmark suite in the real-time domain, and it is particularly useful for evaluating the

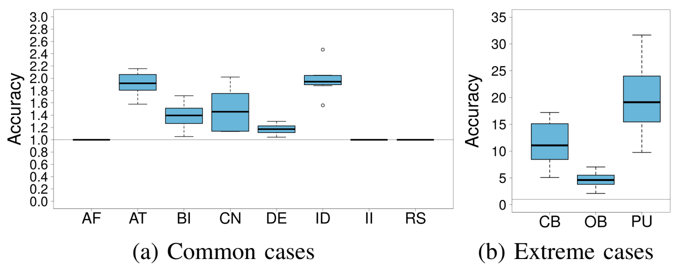


Fig. 7. Accuracy of the cache contention model

TABLE IV
WORKLOADS USED FOR EVALUATION

BENCHMARK	*8 WORKLOADS IN WHICH IT RUNS*
AF	(LLL), (LLH), (LHL), (HLL), (HLH), (HHL), (LHH), (ULL)
AT	(UUU), (UMU), (MUU), (UUM), (MMU), (MUM), (UMM), (MMM)
BI	(UUU), (UMU), (MUU), (UUM), (MMU), (UMM), (MUM), (MMM)
CB	(LUU), (UUU), (ULU), (UUL), (HUU), (MUU), (UHU), (UMU)
CN	(ULU), (UUL), (LUU), (HUU), (UHU), (UUH), (MUL), (ULM)
DE	(MUU), (UMU), (UUU), (UUM), (MMU), (MUM), (UMM), (MMM)
ID	(UUU), (UMU), (MUU), (UUM), (MMU), (UMM), (MUM), (MMM)
II	(LLL), (LLH), (LHL), (HLL), (LHH), (LEL), (HLH), (LLE)
OB	(UUU), (UMU), (MUU), (UUM), (MMU), (MUM), (UMM), (MMM)
PU	(UUU), (UMU), (MUU), (UUM), (LUU), (ULU), (UUL), (UMM)
RS	(LLL), (LLH), (LHL), (HLL), (LEL), (LLE), (ELL), (HLH)

capabilities of embedded microprocessors, compilers, and the associated embedded system implementations. The diversity of this suite ensures that designers can use combinations of EEMBC workloads to make effective design choices. In particular we use these benchmarks: `aifirf` (AF), `aiifft` (AT), `bitmnp` (BI),`cacheb` (CB), `canrdr` (CN), `idctrn` (ID), `iirflt` (II), `puwmod` (PU), `rspeed` (RS).

European Space Agency benchmarks. We use representative benchmarks of ESA on-board software. In particular, the On-board Data Processing (OB) and DEBIE (DE) benchmarks.

- On-board Data Processing contains the algorithms used to process raw frames coming from the state-of-the-art near infrared (NIR) HAWAII-2RG detector [16], already used in real projects, like the Hubble Space Telescope to detect cosmic rays.
- DEBIE is the software that controls an instrument, which was carried on PROBA-1 satellite, to observe micro-meteoroids and small space debris by detecting impacts on its sensors, both mechanically and electrically.

Stressing Kernels. In order to also stress our model in high contention situations, we have implemented five resource stressing kernels [13][24] that create high contention in shared resources. Those benchmarks include `l2full` (U) and `l2half` (H), which traverse continuously an array occupying the whole L2 and half of it, respectively; `l2miss` (M) and `l1miss` (L), which continuously miss on their respective caches; and `mixed-8-12-80` (E), which executes a specific mixture of instructions (8% stores, 12% loads and 80% adds).

Workloads. We run four-task workloads. For the first task we use our model to estimate its execution time bound. This first task is either a EEMBC AutoBench benchmark or an ESA benchmark. The other three (contending) tasks in the workload are `l2full`, `l2half`, `l2miss`, `l1miss` or `mixed-8-12-80`. This creates a stressful scenario in which we can fairly assess the accuracy of the predictions our model. Table IV summarizes the workloads used in this paper.

Metrics. We evaluate the accuracy provided by our model in terms of the increment in the number of L2 misses (Δm_i^{uL2}), bus time ($\Delta t_i^{BUS} + \Delta t_i^{MEM}$) and the overall execution time in multicore (et^{muc}). For each of these three metrics we measure accuracy as $\frac{PredictedValue}{ActualValue}$. Hence the closer to one the better, with values above one showing that the model over-approximates and values below one that the model under-approximates. For each four-task workload, we measure the accuracy in estimating the contention of its first task. As shown in Table IV we create eight workloads for each EEMBC Automotive and the ESA benchmarks. We show the distribution of the accuracy across the eight workloads with a boxplot, thus showing the median, the quantiles, maximum and minimum values and outliers.

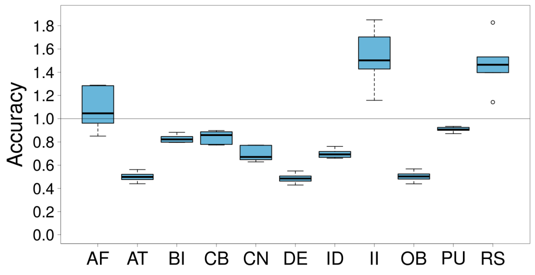


Fig. 8. Accuracy of the bus contention model

B. Experimental Results

Cache Contention Model. As we can see in Figure 7(a) Δm_i^{uL2} is accurately estimated for several benchmarks and somehow overestimated for others. The largest deviation are due to the fact that our cache contention model assumes that stack and set distances are homogeneous across sets for the sake of simplicity. However, this is not always the case and, in fact, our `l2full` and `l2half` synthetic kernels are specifically designed to stress all cache sets and half of them, respectively, so that heterogeneous behavior across sets produces large inaccuracies in our model.

The case of the three benchmarks in Figure 7(b) is completely different since Δm_i^{uL2} is in the range $[10, 100]$. In this case, benchmarks suffer in the order of dozens extra misses in L2 due to contention, which is negligible for those benchmarks executing millions of instructions. Hence, although our model overestimates Δm_i^{uL2} by a factor of 10-20x, this only represents accounting for around of 1,000 extra L2 misses whose impact in the total execution time is negligible.

It can also be observed that the cache contention model leads to an overestimation of Δm_i^{uL2}. The main reason is the fact that our model assumes that cache accesses are homogeneously distributed in time. However, in reality they typically occur in bursts. Thus, if bursts of all different tasks occur simultaneously, the model is accurate but if, instead, not all of them overlap completely (the common case) real interference is lower than estimated because the number of interfering accesses from other tasks during a burst of the task under analysis is lower than predicted.

Average-Based Model. For comparison purposes we obtained results for the cache contention model using average values rather than histograms. Our results show that for all the workloads listed in Table IV the average-based model detects

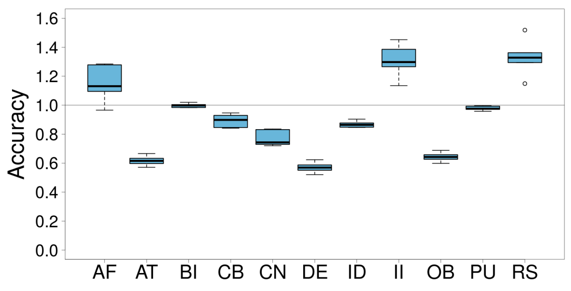


Fig. 9. Performance accuracy of the complete model. Results have been normalized w.r.t. the actual measurements.

almost no contention. This occurs because dividing average values by the number of sets du, as done in Equation 9, leads to very low predicted interferences among tasks. This produces contention predictions as low as 0.0034, that is with an inaccuracy of almost 100% $(1 - 0.0034 = 0.9966)$.

Bus Contention Model. For bus contention, shown in Figure 8, we observe a variety of behaviors across benchmarks. Although such contention is somehow overestimated for few benchmarks, it is typically underestimated for most of them. This effect is particularly noticeable for `aiifft`, `debie` and `obdp`. As explained before, the NGMP processor has store buffers that are able to hide part of the latency of stores. However, since stores occur in bursts, whenever they occur in a short period of time they can produce performance degradations much higher than linear. For instance, it has been proven that execution time may grow by a factor of 20x in the NGMP with just 4 cores due to the (bad) interaction of store operations in the different cores [13]. How to better capture this effect in our bus contention model without incurring high overhead is still part of our future work.

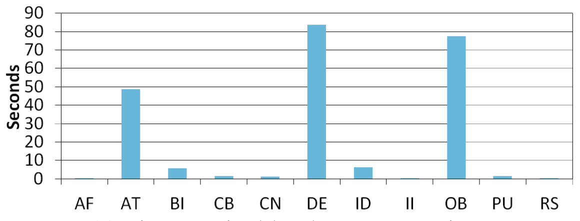
Putting it all together: Multicore Execution Time. We have also analyzed the accuracy of our performance estimates when considering all contention models and assumptions together. The accuracy in determining et^{solo} is high with predictions in the range [1.02, 1.23] for all benchmarks.

Results are shown in Figure 9. We observe that the accuracy of the estimates is mostly dominated by bus contention in the NGMP. The main reason is the fact that the difference between L2 hit and L2 miss latencies is relatively low (9 versus 23 cycles) and only affects Δm_i^{uL2}, whereas the impact of bus contention affects all load misses in L1 and *all store instructions* given that DL1 is write-through. Therefore, inaccuracies due to the effect of the store buffer in terms of bus contention dominate the results.

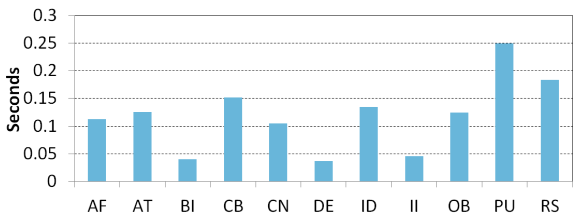
Overall, our simple analytical model is able to keep performance estimates in the range $[0.6, 1.4]$ *w.r.t. the real performance in the NGMP. These are accurate results for execution time estimations for early-design phases. The accuracy of our model allows system designers to really take into account multicore contention in the design choices made (e.g. deciding the scheduling plan).*

Model Execution Time Overhead. To be usable in EDP, the execution time overhead of our model should be low.

To produce the 'raw data' from the VM as presented in Figure 2, we introduce instrumentation instructions to read information about the particular instruction under execution. The number of instructions to add is relatively low: three



(a) Time required by the EP generation



(b) Time to perform one estimation of the multicore contention

Fig. 10. Average overheads of our contention prediction model.

instructions in our case to read opcode, program counter and data address. From the raw data we run the cache simulator and process instructions to generate the EP. Figure 10(a) shows the execution time of this EP generation step, which is run just once per application. If the application has several releases whose resource-usage profile is expected to change, this step is repeated once for each of those releases. The duration of this step depends on the length of the program. For the EEMBC and the real ESA benchmarks, whose execution is in the order of dozens millions of instructions, in the worst case this step takes around 80 seconds. On average EP generation requires around 20 seconds across all benchmarks, which is reasonably short given the low frequency with which this step executes.

Figure 10(b) shows the execution time overhead of contention models, once EP information has been produced. The duration of this step is the most important for the feasibility of our approach. This is so because, once the EPs are generated, the system designer needs short turn-around time of the models to be able to evaluate different design choices. We observe that predicting the multicore performance of an application in a workload requires as low as 120ms on average, which enables a vast design space exploration of various system parameters by system designers.

VII. Related Works

The focus of timing analysis techniques in the literature for EDP has been on single-core architectures. To our knowledge, the work presented in this paper is the first addressing the challenge of providing timing estimates in EDP for multicores.

Some approaches for single-core architectures work on the assumption that the target processor and/or the corresponding compilation toolchain is available, while others do not. When the target processor is not available, several techniques exist to derive timing estimates that help deciding the hardware platform that best satisfies system requirements [11][26] as well as sizing it. The approach consists in compiling the source code for a given set of potential target ISAs. For each of the ISA there is a parameterizable processor simulator (model) from which timing information is gathered. The model allows changing parameters with high impact on timing such as cache configuration [26]. Then program information (e.g. paths) obtained from the executable and timing from the generic processor model are combined to approximate programs execution time. In our case the target ISA and processor are fixed, so such an approach would not be necessary.

Other approaches do not work at the binary level but at the source code level, or an intermediate representation level, which are available earlier in the design cycle of the system. In some cases, timing is integrated in high level modelling environments such as Matlab/Simulink [9]. The ultimate goal is providing the developer knowledge of the worst-case "as the code is written" [10]. In all cases the focus is on single-core architectures, while our focus is on multicore contention.

In many of the approaches above one of the main challenges lies in deriving a light, yet accurate, timing (cost) model for individual instructions or sequences of them. Some papers [10] assume a WCET-friendly processor design, such as the Java Optimized Processor (JOP). This simplifies the timing model since the processor is predictability aware. Other papers propose methods to derive a timing model from measurements of representative code extracts on the target processor. For instance, authors in [17] work on the concept of C-source-level abstract machine which is calibrated based on measurements to match a target real hardware. In this line, [14] proposes the *timing model code level* that combines measurements and a regression model to perform timing estimates of source code. In this latter work, the timing (cost) model is built, i.e. it is not assumed as an input. In both cases the focus is on constructs that frequently appear on the target programs.

While previous works focus on single-core processors, our focus is on multicore specific aspects. In particular the contention in the access to hardware shared resources. For multicore, it could be possible to run the program under analysis against a set of resource stressing kernels (*rsk*s) which put high load on the shared resources [13], [24]. This would provide a good estimation of the program execution time under heavy (extreme) load conditions. However, it has been shown that this approach leads to inflated execution time estimates, up to 20x bigger than programs' execution time in isolation, which makes it impractical to obtain accurate execution time approximations during EDP.

Previous works show that, while exact bounds are required in LDP, during EDP, instead, approximations to those bound are needed [11][14]. Some accuracy is traded to speed up the estimation process so that engineers can make design space exploration taking into account timing. To our knowledge, no particular figure is reported on accuracy required in EDP. For multicores several works show that the impact of contention can up to 20x for some kernels and up to 5.5x for some EEMBC benchmarks [13]. In this context, we deem the accuracy results obtained by our approach (between 0.6x and 1.4x) as sufficiently precise.

VIII. Conclusions

In this paper we present a new timing model for early design stages able to predict in a fast manner the performance of applications when the target (virtual) platform is multicore based. They key idea behind our model is the generation of an execution profile (EP) for each application that software suppliers can share without revealing IP, yet allowing performance analysis of multicore contention. This paper describes how EP are generated and how they are combined to predict the performance of applications under particular schedule. Our results show that useful estimates can be obtained extremely fast since generating an EP takes few seconds and evaluating a schedule less than 0.2 seconds for the NGMP architecture.

Acknowledgement

This work has received funding from the European Space Agency under Project Reference $AO/1-7722/13/NL/LvH$, and has also been supported by the Spanish Ministry of Science and Innovation grant TIN2015-65316-P. Jaume Abella has been partially supported by the MINECO under Ramon y Cajal postdoctoral fellowship number RYC-2013-14717.

References

[1] Guidelines and methods for conducting the safety assessment process on civil airborne systems and equipment. *ARP4761*, 2001.
[2] *ARINC Specification 653: Avionics Application Software Standard Standard Interface, Part 1 and 4*, 2012.
[3] J. Abella et al. WCET analysis methods: Pitfalls and challenges on their trustworthiness. In *SIES*, 2015.
[4] Aeroflex Gaisler. *Quad Core LEON4 SPARC V8 Processor - LEON4-NGMP-DRAFT - Data Sheet and Users Manual*, 2011.
[5] Aeroflex Gaisler. *LEON4-N2X Data Sheet*, 2013.
[6] AUTOSAR. *Technical Overview V2.0.1*, 2006.
[7] Cobham Gaisler. *NGMP Preliminary Datasheet Version 2.1, May 2013.*
[8] Cobham Gaisler. *Quad Core LEON4 SPARC V8 Processor - GR740-UM-DS-D1 - Data Sheet and Users Manual, 2015.*
[9] Raimund Kirner et al. Fully automatic worst-case execution time analysis for matlab/simulink models. In *ECRTS*, 2002.
[10] Trevor Harmon et al. Fast, interactive worst-case execution time analysis with back-annotation. *IEEE Trans. Industrial Informatics*, 8(2), 2012.
[11] C. Ferdinand et al. Integration of code-level and system-level timing analysis for early architecture exploration and reliable timing verification. In *ERTS2*, 2010.
[12] G. Fernandez et al. Increasing confidence on measurement-based contention bounds for real-time round-robin buses. In *DAC*, 2015.
[13] Mikel Fernández et al. Assessing the suitability of the ngmp multi-core processor in the space domain. In *EMSOFT*, 2012.
[14] J. Gustafsson et al. Approximate worst-case execution time analysis for early stage embedded systems development. In *SEUS*, 2009.
[15] M. M. Irvine and A. Dartnell. The use of emulator-based simulators for on-board software maintenance. In *Data Systems in Aerospace (DASIA)*, volume 509 of *ESA Special Publication*, 2002.
[16] Andreas Jung and Pierre-Elie Crouzet. The H2RG infrared detector: introduction and results of data processing on different platforms. Technical report, European Space Agency, 2012.
[17] R. Kirner and P. Puschner. A simple and efficient fully automatic worst-case execution time analysis for model-based application development. In *Workshop on Intelligent Solutions in Embedded Systems*, 2003.
[18] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Syst. J.*, 9(2), June 1970.
[19] M. Paolieri et al. *An Analyzable Memory Controller for Hard Real-Time CMPs* . Embedded System Letters (ESL), 2009.
[20] Zheng Pei Wu et al. Worst case analysis of DRAM latency in multi-requestor systems. In *RTSS*, 2013.
[21] Jason Poovey. *Characterization of the EEMBC Benchmark Suite*. North Carolina State University, 2007.
[22] Peter Puschner and Martin Schoeberl. On Composable System Timing, Task Timing, and WCET Analysis. In *WCET Analysis Workshop*, 2008.
[23] P. Puschner et al. Towards Composable Timing for Real-Time Software. In *Workshop on Software Technologies for Future Dependable Distributed Systems*, 2009.
[24] Petar Radojković et al. On the evaluation of the impact of shared resources in multithreaded cots processors in time-critical environments. *ACM TACO*, 2012.
[25] SoCLib. -, 2003-2012. http://www.soclib.fr/trac/dev.
[26] http://www.absint.com/timingprofiler. *Timing Profiler*. AbsInt.
[27] http://www.gmv.com/en/aeronautics/products/air/. *Robust Partition Safety-Critical Real-Time Operating System.* GMV.
[28] Wilhelm R. et al. The worst-case execution-time problem overview of methods and survey of tools. *ACM TECS*, 7:1–53, May 2008.