Using Model Checking to Identify Timing Interferences on Multicore Processors

Viet Anh Nguyen[1], Eric Jenn[1], Wendelin Serwe[2], Frédéric Lang[2], and Radu Mateescu[2]
[1] *IRT Saint Exupéry, Toulouse, France*
[2] *Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP*[*]*, 38000 Grenoble, France*

Abstract

Multicore platforms provide the computing capabilities and the power efficiency required by the complex applications embedded in aeronautical, spatial, and automotive systems. Some of the hardware resources provided by the platform—including buses, caches, IPs—are shared between tasks executing concurrently and in parallel on different cores. This sharing may lead tasks to *interfere* with each other. Therefore, crucial design activities are to identify interferences, and bound the penalty induced by those interferences, as part of the demonstration of compliance of applications to their temporal requirements.

A first and conservative approach is to consider that every access to a shared resource leads to an interference. This safe approach is usually too pessimistic to be useful. We propose a less pessimistic approach, which takes into account the actual behavior of the application and the hardware to filter out situations where interferences cannot occur.

Our method relies on (i) the behavioral modeling of the applications and their execution platform using the LNT formal language, (ii) the definition of interferences using temporal properties, and (iii) the exploitation of the behavioral model and the temporal properties using the CADP formal verification toolbox. This method is applied to the Infineon AURIX TC275 system-on-chip. Experimental results indicate that our approach is not only safe but also prevents reporting spurious interferences compared to a purely structural analysis.

Index Terms

Multicore, Timing interference, Model checking, CADP, AURIX TriCore

I. Introduction

Multi-core platforms have been considered for the implementation of critical real-time systems for several years now. Those platforms provide the ever increasing computational power required by embedded applications while maintaining an acceptable energy consumption level. However, they also raise difficult challenges, in particular when it comes to demonstrating the temporal properties of the implemented system. Indeed, the correctness of hard real-time systems depends both on the correctness of the computations and on the compliance with various temporal constraints (sampling time, worst-case response times, data aging, etc.). In order to guarantee that all the tasks composing a real-time application meet their deadlines, designers must estimate their Worst-Case Execution Times (WCET). Estimating the WCET of tasks deployed on multiple cores is quite challenging [1]. Because hardware resources are shared between cores, the execution of a task on a core may impact the timing behavior of the tasks executing in parallel on other cores. A reasonable WCET estimation requires thus to identify the sources of such interferences and take their effects into account in the WCET estimation. In this context, we propose a method to automate—at least partially—the detection of interferences.

Our approach uses model checking to identify interferences. The proposed solution relies on three main components: (i) a formal model that captures the behavior of the software application and the execution platform, which are both expressed in LNT [2], (ii) a formal property capturing the concept of interference expressed in MCL [3], and (iii) a verification toolbox—CADP [4]—to exploit the behavioral models and verify whether the property holds on the model.

In this paper, we consider *temporal interferences*, i.e., situations where the execution time of a task changes due to the presence of other tasks running on other cores. In order to capture the changes, we build two classes of formal models: *isolated* and *non-isolated*. In each *isolated* model, only one core is enabled. The tasks executing on the core have thus an exclusive access to the shared resources and are ensured to be free from interferences. In the *non-isolated* model, all cores of the platform are enabled. The tasks executing on the cores share the resources and interfere with each other. The interferences manifest themselves by differences between the execution traces generated by the two behavioral models.

In order to point out these differences, we propose two solutions:

- The first solution, called *PATCHECK*, is applicable when interferences can be defined by specific trace patterns. We use model checking to detect the presence or absence of those patterns on traces generated by the *non-isolated* model.
- The second solution, called *SYNCHECK*, "compares" the traces obtained from the *isolated* and *non-isolated* models using a specific synchronous product. We use model checking to detect differences between the traces obtained for the same task from the *isolated* model and from the *non-isolated* model. The second solution is more complex to deploy, but is more generic as compared to the first solution.

[*]Institute of Engineering Univ. Grenoble Alpes

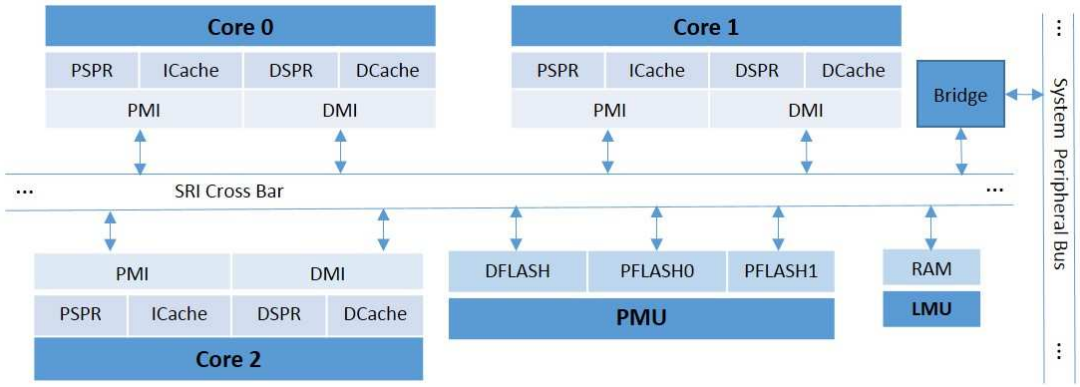


Fig. 1: Block diagram of the Aurix TC275

The rest of this paper is organized as follows. Section II gives the motivation of our work and presents some related works. Section III gives a brief overview of the AURIX TC275 platform and the CADP verification toolbox. Section IV presents the concept of the proposed interference analysis for the execution of applications on the AURIX TC275 platform. Section V presents evaluation results of our method. Finally, we conclude the paper and provide directions for future work in Section VI.

II. Related work

The prevention, the control, and the estimation of interferences in multicore hardware architectures has been an active industrial and academic research subject for more than ten years [5], [6], [7]. However, most studies consider sources of interference to be either known (e.g., caches), identified by a manual analysis of the documentation, or detected by measurements [8]. Some automated analyses [9], [10] are based on a purely structural analysis.

We are not aware of any model-checking based approach that focuses on identifying interferences. Usually, model checking is used to perform timing analysis, as for instance in [11], [12]. More generally, a significant effort has been spent on analyzing the impact of contentions on execution time. A taxonomy of these techniques [13] includes: system-centric techniques, WCET-centric techniques, and architecture-centric techniques. Most of those studies aim to reduce or limit interferences, to tighten the estimated WCETs, and enhance the system's schedulability. In this paper, we focus on identifying the sources of contentions using the behavioral models of the applications and the underneath execution platform.

The work presented in this paper is the continuation of a previous work carried out as part of the CAPHCA project[1], which focused on the identification of interferences on the basis of a structural model of the hardware and software system [10]. That static analysis is completely untimed and cannot capture situations where multiple accesses to a shared resource are guaranteed to be exclusive simply because they occur at different points in time. In order to prevent reporting such spurious interferences, the analysis in this paper takes into account not only the *structure* of the software and hardware elements of the system, but also their *behavior*. Experimental results show that our proposed approach is not only safe but also able to prevent reporting spurious interferences.

III. Overview of the AURIX TC275 platform and the CADP verification toolbox

A. The AURIX TC275 Platform

Our approach has been applied to a part of the Infineon's AURIX TC275 microcontroller [14]. A simplified view of its architecture is presented on Fig. 1. The AURIX TC275 system on chip (SoC) contains three in-order execution cores, each of which owns private local memories for program and data (PSPR and DSPR, respectively, and caches). Each core has two memory interfaces: a Program Memory Interface (PMI) and a Data Memory Interface (DMI). The SoC's components (cores, memories, IPs) are interconnected via the Shared Resource Interconnection Crossbar Interconnect (XBar_SRI) and the System Peripheral Bus (SPB).

The XBar_SRI handles, arbitrates, and forwards transactions between all connected devices. Devices connect to the XBar_SRI via Master Connector Interfaces (MCIx) or Slave Connector Interfaces (SCIx). An SRI master (e.g., PMI or DMI) can initiate transactions on the SRI, whereas an SRI slave (e.g., shared memory units) can only contribute to transactions. SRI masters and SRI slaves are connected point-to-point (as shown in Fig. 2), such that transactions between different pairs of SRI masters and SRI slaves can be done in parallel.

[1]CAPHCA (Critical Applications on Predictable High-Performance Computing Architectures) is a research project conducted at the Institut de Recherche Technologique (IRT) Saint Exupéry in Toulouse. It addresses several issues related to the usage of high-performance processors and SoCs in safety critical systems, including deterministic execution platforms, interferences and WCET analysis, etc.

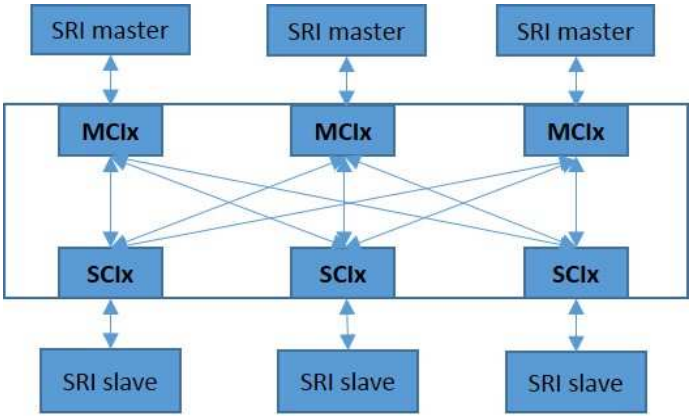


Fig. 2: Point-to-point connections in XBar_SRI

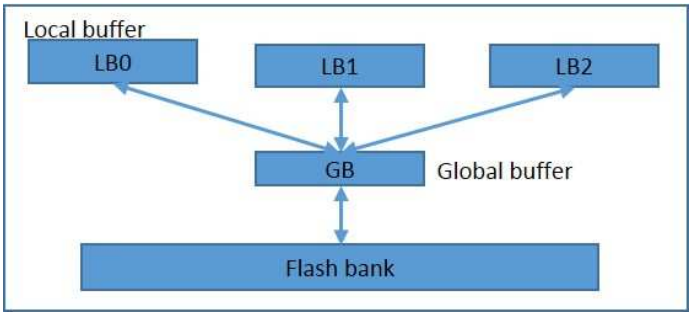


Fig. 3: Block diagram of PFLASH0 and PFLASH1

Simultaneous requests coming to an SCIx are arbitrated at two levels: priority driven arbitration and round robin group arbitration [14]. A priority ranging from 0 to 7 (highest priority is 0) is assigned to each SRI master. By default, MCIs associated with PMIs and DMIs of the three cores are mapped to the same round robin group with priority 5 [14]. Additionally, the platform supports a starvation prevention scheme.

The shared memory system comprises 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 (e.g., PFLASH0 and PFLASH1) and one for data (e.g., DFLASH). PFLASH0 and PFLASH1 are equipped with three local read buffers (LB) and a global read buffer (GB), which have the same size (as shown in Fig. 3). Each LB is assigned to one SRI master [14]. If a SRI master is associated with one of the three LBs and its request misses at both the associated LB and the GB, the requested data are fetched from the flash bank to the GB. The fetched data are then stored in the corresponding LB and forwarded to the SRI master. At the end of the transaction the data at the consecutive address are prefetched into the GB. If a SRI master is associated with none of these LBs, its request goes directly to the flash bank.

From a structural point of view, interferences might occur at arbiter and global buffer levels because these components are shared between cores. In the following sections, we present our analysis that identifies the presence of contentions at these two levels. This analysis uses the CADP toolbox presented hereafter.

B. Construction and Analysis of Distributed Processes (CADP)

CADP [4][2] is a toolbox for protocols and distributed systems engineering. Safely grounded in concurrency theory, CADP provides more than 50 tools and libraries, supporting various qualitative and quantitative analysis techniques, such as model-checking, equivalence checking, performance evaluation, test-case extraction, and rapid prototype generation. The recommended modeling language of CADP is LNT [2], [15], a modern language providing the salient features of process calculi in a user-friendly syntax. An LNT model can be compiled into a labeled transition system (LTS), for which CADP provides the compact BCG format. The EXP.OPEN [16] tool supports a wealth of operators for LTS composition, including those of process calculi, but also so-called synchronization vectors, which we use in SYNCHECK to encode a particular synchronous product. For expressing temporal-logic properties, CADP supports MCL (*Model Checking Language*) [3], which extends the alternation-free modal μ-calculus with regular expressions and data-handling constructs. CADP comes with the SVL scripting language [17], which facilitates the automation of complex analysis scenarios, including compositional ones [18].

[2]http://cadp.inria.fr

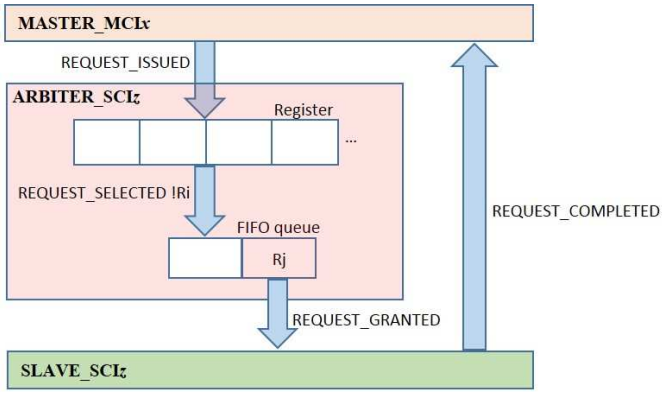


Fig. 4: The interaction between LNT processes that model memory transactions between a pair of MCIx and SCIz

IV. Interference analysis for the execution of applications on the AURIX TC275 platform

Our interference analysis requires the following information:

- The mapping of the tasks on cores
- The schedule (e.g., time-trigger or event-trigger) of the tasks. This schedule can be generated by the ASTERIOS toolchain [19] or by the S1[3] tool given an execution, as well as temporal constraints of tasks.
- The tasks' memory accesses, including the type (e.g., code or data) as well as the source and destination addresses of accesses. This information can be obtained by using the static timing analysis tool OTAWA [20].

In the following sections, we present in detail the behavioral models of the application and the AURIX TC275 platform, the formal property that captures the concept of interferences, as well as the strategies to identify interferences.

A. Behavioral models

1) Structure of the behavioral models: In order to identify interferences at shared memories between tasks executing on different cores, we focus on modeling memory accesses (i.e., code/data) initiated by the software towards the hardware. The formal models are expressed using the formal modeling language LNT [15]. Each transition in the LNT models represents a required step of a memory transaction. The LNT models are organized in three main modules:

- The MASTER module contains a set of MASTER_MCIx processes, each of which models the execution behavior of its associated SRI master, MCIx.
- The ARBITER module contains a set of ARBITER_SCIz processes, each of which models the execution behavior of an arbiter associated with SCIz.
- The SLAVE module contains a set of SLAVE_SCIz processes, each of which models the execution behavior of its associated SRI slave, SCIz.

A memory transaction (corresponding to either a load or a store operation) between a SRI master and a SRI slave is modeled by a sequence of synchronizations of the corresponding processes on the LNT gates (i.e., used for input/output communication/synchronization [15]), including REQUEST_ISSUED, REQUEST_GRANTED, REQUEST_COMPLETED, as shown in Fig. 4. The MASTER_MCIx process synchronizes with the ARBITER_SCIz process on the gate REQUEST_ISSUED when the request is issued. Arrived requests at an arbiter are stored in a register. By default, a round robin algorithm is used to arbitrate these requests. The winner of a round robin group is indicated by a REQUEST_SELECTED transition and then put into a FIFO queue waiting until the SRI slave is free. When the winner is granted access to the SRI slave, the ARBITER_SCIz process synchronizes with the SLAVE_SCIz process on the gate REQUEST_GRANTED. When the request is completed, the SLAVE_SCIz process synchronizes with the MASTER_MCIx process on the gate REQUEST_COMPLETED.

2) Behavioral model of SRI masters: A SRI master issues only one memory request at a time. When the request (either read or write) is issued, the master is blocked until the request is completed. The information about a memory request is stored in the REQUEST record (as shown in Listing 1) which has four attributes: the id of the MCI associated with the master that issues the request; the id of the SCI associated with the slave to which the request is sent; the request's unique id; and the memory address of the request.

```
type REQUEST is
   REQUEST(MCIx: Nat, SCIz: Nat, request_id: Nat, request_addr: Nat)
   with "get", "set"
end type
```

Listing 1: Information contained in a memory request

[3]This tool is part of the SCADE multicore toolset provided by ANSYS.

```
process MASTER_MCIx[REQUEST_ISSUED, REQUEST_COMPLETED: REQUEST_CHANNEL] is
    var request: REQUEST in
      request := REQUEST(0,0,0,0);
      request := gen_request(x, 7, 1, 4); -- generates a request from MCIx to SCI7
      REQUEST_ISSUED(request);
      REQUEST_COMPLETED(?request) where request.MCIx == x;
      request := gen_request(x, 7, 2, 8);
      REQUEST_ISSUED(request);
      REQUEST_COMPLETED(?request) where request.MCIx == x
    end var
end process
```

Listing 2: An example of MASTER_MCIx process

Listing 2 shows the LNT process that models the SRI master associated with MCIx. In this example, we assume that the SRI master sends two requests to PFLASH0 associated with SCI7 [14]. The "REQUEST_ISSUED !Ri" transition signifies that the request "Ri" has been sent to the SCI associated with PFLASH0. Before issuing the next request, the MASTER_MCIx process has to wait until it is notified that the previous request has been completed, i.e., the value exchanged through "REQUEST_COMPLETED" synchronization gate matches with the id of its associated MCI. Note that the number of memory requests issued by MCIx and their order depends on the execution of tasks running on the core to which the master belongs, as well as on the synchronizations between tasks.

3) Behavioral model of arbiters: There are three main branches in the ARBITER_SCIz process (as shown in Listing 3): one for enqueuing the received requests, one for arbitrating those requests, and one for forwarding the requests of the winning master to the destination slave.

Listing 4 shows the enqueuing requests process. When a SCI receives a new request, i.e., the value exchanged through "REQUEST_ISSUED" synchronization gate matches with its id, the request is stored in a register. The register is modeled as an array of RegSlotInfo records (as shown in Listing 5). The number of slots in the register is equal to the number of SRI masters. Each slot of the array is dedicated to store one request issued by a SRI master. SRI masters (or MCIs) are mapped to the array in the ascending order of their id, i.e., MCI0 is mapped to the first slot of the array, MCI1 is mapped to the second slot of the array, and so on.

```
process ARBITER_SCIz[REQUEST_ISSUED, REQUEST_SELECTED, REQUEST_FORWARDED: REQUEST_CHANNEL] is
  var register: RegSlotInfo_Array, previous_winning_MCIx: Nat, fifo_queue : QUEUE in
    register := RegSlotInfo_Array(RegSlotInfo(REQUEST(0,0,0,0),false));
    fifo_queue := EMPTY_QUEUE;
    loop L in
      select
          ... (* Enqueue requests *)
      []  ... (* Arbitrate request *)
      []  ... (* Forward requests to the associated slave *)
      []  break L
      end select
    end loop
  end var
end process
```

Listing 3: Code skeleton of behavioral model of an arbiter

```
var request: REQUEST, MCIx : Nat, slotInfo : RegSlotInfo in
  request := REQUEST(0,0,0,0);
  slotInfo := RegSlotInfo(REQUEST(0,0,0,0), false);
  REQUEST_ISSUED(?request) where request.SCIz == z; -- the SCI receives a new request
  MCIx := request.MCIx;
  slotInfo := slotInfo.{request => request, is_occupied => true};
  register[MCIx] := slotInfo
end var
```

Listing 4: Enqueuing requests process

```
type RegSlotInfo is
  RegSlotInfo (request: REQUEST, is_occupied: Bool)
  with "get", "set"
end type
```

Listing 5: Information stored in a slot of the register

We consider default mapping of SRI masters such that the MCIs associated with the PMIs and the DMIs of three cores are mapped to the same round robin group. Listing 6 shows the model of arbitrating requests process. The requests of these masters

```
var winning_MCIx: Nat, selected_request: REQUEST, slotInfo:RegSlotInfo, ... in
  selected_request := REQUEST(0,0,0,0);
  slotInfo := RegSlotInfo(REQUEST(0,0,0,0), false);
  winning_MCIx := 0;
  is_register_occupied := false;
  is_register_occupied := check_register(register);
  if is_register_occupied == true then
    winning_MCIx := arbitrate_requests(register, previous_winning_MCIx);
    slotInfo := register[winning_MCIx];
    selected_request := slotInfo.request;
    previous_winning_MCIx := winning_MCIx;
    REQUEST_SELECTED(selected_request);
    fifo_queue := ENQUEUE(selected_request, fifo_queue);
    register[winning_MCIx] := RegSlotInfo(REQUEST(0,0,0,0), false)
  end if
end var
```

Listing 6: Arbitrating requests process

are handled by the round robin algorithm. The winner of a round robin group is revealed by "REQUEST_SELECTED !Ri" transition. After the winner is found, the round robin group starts a new arbitration round in which the next highest MCI's id will win. If there is no requesting MCI with a higher id in the round robin group, the algorithm will start with the MCI with the lowest MCI's id that is mapped to the group, going to the next higher MCI's id and so on. The request of the winning master is put into a FIFO queue. When the slave SCIz is free, ARBITER_SCIz process synchronizes with SLAVE_SCIz process at the gate REQUEST_GRANTED (shown in Listing 7). Then the request at the head of the FIFO queue will be granted to access the slave.

```
var forwarded_request: REQUEST in
  forwarded_request := REQUEST(0,0,0,0);
  if SIZE(fifo_queue) >= 1 then
    forwarded_request := FIRST(fifo_queue); -- get the first element of the fifo queue
    REQUEST_GRANTED(forwarded_request);
    fifo_queue := DEQUEUE(fifo_queue)
  end if
end var
```

Listing 7: Forwarding requests process

4) Behavioral model of SRI slaves: Listing 8 shows the behavior of a SRI slave. A SRI slave allows only one request to be served at a time. When a slave receives a new request, i.e., the value exchanged through the synchronization gate named "REQUEST_GRANTED" matches with the id of its associated SCI, there are two possible cases (with the assumption that the slave is PFLASH0):

- The request is issued by a SRI master not associated with any of the three local read buffers. In this case, the request goes to the flash bank (indicated by "FLASH_BANK_ACCESS" transition)
- The request is issued by a SRI master associated with one of three local read buffers. In this case, the request goes to the local buffer associated with the requesting SRI master. The request either hits in the local read buffer (indicated by "HIT_LB !Ri" transition) or goes to the global read buffer. Then, the request either hits in the global read buffer (indicated by "HIT_GB !Ri" transition) or goes to the flash bank. At the end of this transaction, the data at the consecutive address (i.e., the next four bytes) is prefetched into the global read buffer. Note that if the request hits at either the local read buffer or the global read buffer, the prefetching mechanism will not be triggered. The addresses of data contained in the local read buffers and the global read buffer are stored in lb_addr and currGBAddr variables, respectively.

When the memory transaction is completed, SLAVE_SCIz and MASTER_MCIx processes synchronize at gate "REQUEST_COMPLETED". Note that the previous behavioral model only applies to slave memories PFLASH0 and PFLASH1.

B. Interference detection

In the AURIX TC275 platform, interferences between tasks executing on different cores potentially occur at arbiters and global buffers (in PFLASH0/1). In this section, we present our strategies for identifying these interferences given the behavioral models of the software and the hardware presented earlier.

1) Interference detection using PATCHECK: Interferences at an arbiter happen when tasks executing on different cores have simultaneous access to the same shared memory unit. Such type of interference leads the tasks to wait at the arbiter associated with the shared memory unit. The delay induced by interferences is reflected by a specific behavioural pattern: when a request arrives to the arbiter, instead of being selected, and then being granted access immediately to the shared memory unit (as shown in Fig. 5a), it waits for another request to be processed (as shown in Fig. 5b). "REQUEST_ISSUED !R1" and

```
process SLAVE_SCIz[REQUEST_FORWARDED, ...:REQUEST_CHANNEL] is
    var request: REQUEST, MCIx, currGBAddr, reqAddr, LBId: Nat, ... in
      (* Initiate variables*)
      -- Assign local buffers to MCI
      -- By default local buffers are assigned to DMI of each core
      -- Local buffers'id ranges from 0 to 2
      LB_2_MCI[0] := 8;
      LB_2_MCI[1] := 10;
      LB_2_MCI[2] := 12;
      loop L in
        select
          REQUEST_GRANTED(?request) where request.SCIz == z;
          MCIx := request.MCIx;
          reqAddr := request.request_addr;
          LBId := get_mapped_LBId(LB_2_MCI, MCIx);
          if LBId < 3 -- the master is associated to one of read buffers
          then
            if reqAddr == lb_addr[LBId]
            then
              HIT_LB(request)
            else
              MISS_LB(request);
              if reqAddr == currGBAddr
              then
                HIT_GB(request);
                lb_addr[LBId] := currGBAddr
              else
                MISS_GB(request);
                FLASH_BANK_ACCESS(request);
                lb_addr[LBId] := reqAddr;
                currGBAddr := reqAddr + 4
              end if
            end if
          else -- the master is not associated to any one of read buffers
            FLASH_BANK_ACCESS(request)
          end if;
              REQUEST_COMPLETED(request)
        []
          break L
        end select
      end loop
    end var
end process
```

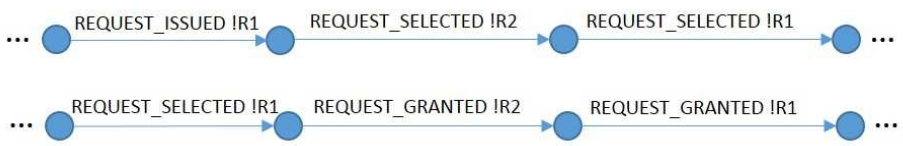
Listing 8: Behavioral model of a SRI slave

"REQUEST_SELECTED !R1" transitions of the request "R1" are interrupted by "REQUEST_SELECTED !R2" transition of the request "R2". Additionally, "REQUEST_SELECTED !R1" and "REQUEST_GRANTED !R1" transitions of the request "R1" are interrupted by "REQUEST_GRANTED !R2" of the request "R2".

We apply *PATCHECK* to identify such type of interferences. We formally express the concept of interference free in MCL [3] (as shown in Listing 9). Two conditions must be verified to ensure the absence of interferences:

- There shall be no "REQUEST_SELECTED" transition of another request between "REQUEST_ISSUED" and "REQUEST_SELECTED" transitions of the same request

(a) Sequence of transitions without interferences

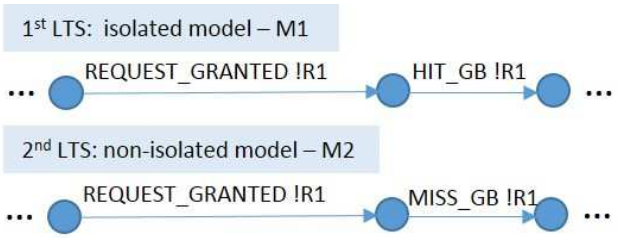


(b) Sequence of transitions when interferences occur

Fig. 5: Effects of interferences on the sequence of transitions

```
property CHECKING_ARBITER_INTERFERENCE(LNT_MODEL, RESULT, Ri)
"Memory request $Ri potentially suffers interferences at the arbiter"
is
  "a.bcg" = generation of "$LNT_MODEL.lnt";
  "a.bcg" |=with evaluator4
  < true*. "REQUEST_ISSUED !$Ri". (not "REQUEST_SELECTED !$Ri")*
  . {REQUEST_SELECTED ?Rj:String where $Ri<>Rj} >true
  or
  < true*. "REQUEST_SELECTED !$Ri". (not "REQUEST_GRANTED !$Ri")*
  . {REQUEST_GRANTED ?Rj:String where $Ri<>Rj} >true;
  expected "$RESULT"
end property
```

Listing 9: MCL formula used for checking interferences at arbiters



(a) Examples of LTSs of *isolated* and *non-isolated* models

(b) Synchronous product of the two LTSs

Fig. 6: LTSs of *isolated* and *non-isolated* models and their synchronous product

- There shall be no "REQUEST_GRANTED" transition of another request between "REQUEST_SELECTED" and "REQUEST_GRANTED" transitions of the same request.

For identifying interferences, we verify the property on the *non-isolated* model. Interferences are reported if one of the two conditions does not hold on the model.

2) Interference detection using SYNCHECK: Interferences at global buffers occur when tasks executing on different cores alternatively send requests to the same global buffer. These alternated accesses "pollute" the content of the global buffer in such a way that a task cannot reuse the prefetched data as it would do if it were executed in isolation. That effect is reflected by changes in the transition of tasks, from "HIT_GB !Ri" to "MISS_GB !Ri" (as shown in Fig. 6a), in which "HIT_GB !Ri" transition indicates that the request hits at the global buffer, whereas "MISS_GB !Ri" transition indicates that the request misses at the global buffer.

We apply *SYNCHECK* to identify such type of interferences. Our strategy reports interferences by tracking the changes in the trace obtained from the *isolated* model, when following them in the trace obtained from the *non-isolated* model. This is done by computing a synchronous product of the LTSs of the two models, using the EXP.OPEN tool [16]. The tool allows us to define parallel compositions of LTSs by synchronization vectors which are described as follows:

$$\{t1 * _ \rightarrow t1 \quad | \quad t1 \in M1\} \qquad \cup$$
$$\{t1 * t2 \rightarrow INTF \quad | \quad t1 \in M1 \text{ and } t2 \in M2 \text{ and } t1 \neq t2\} \qquad \cup$$
$$\{_ * t2 \rightarrow t2 \quad | \quad t2 \in M2\}$$

In these vectors, t1 and t2 denote the transitions contained in M1 and M2, respectively. The first and the third sets of the synchronization vectors trace the sequence of the transitions of the two models. The second set of the synchronization vectors captures the differences (induced by interferences) between the two models. The occurrence of interferences is signified by a transition labeled by "INTF !Ri" (as shown in Fig. 6b). We formulate a MCL formula (shown in Listing 10) to verify the existence of the transition in the output of the synchronous product.

SYNCHECK is a generic approach. Contrary to *PATCHECK*, which is based on an a priori knowledge about what has been observed (e.g., the insertion of the specific transition "REQUEST_SELECTED !R2" in the middle of a sequence of transitions

```
property CHECKING_ARBITER_INTERFERENCE(LNT_MODEL, RESULT, Ri)
"Memory request $Ri potentially suffers interferences at the global buffer"
is
    "$BCG.bcg" |=with evaluator4
  <true*. "INTF !$Ri"> true;
   expected "$RESULT"
end property
```

Listing 10: MCL formula used for checking the existence of the transition labelled by "INTF"

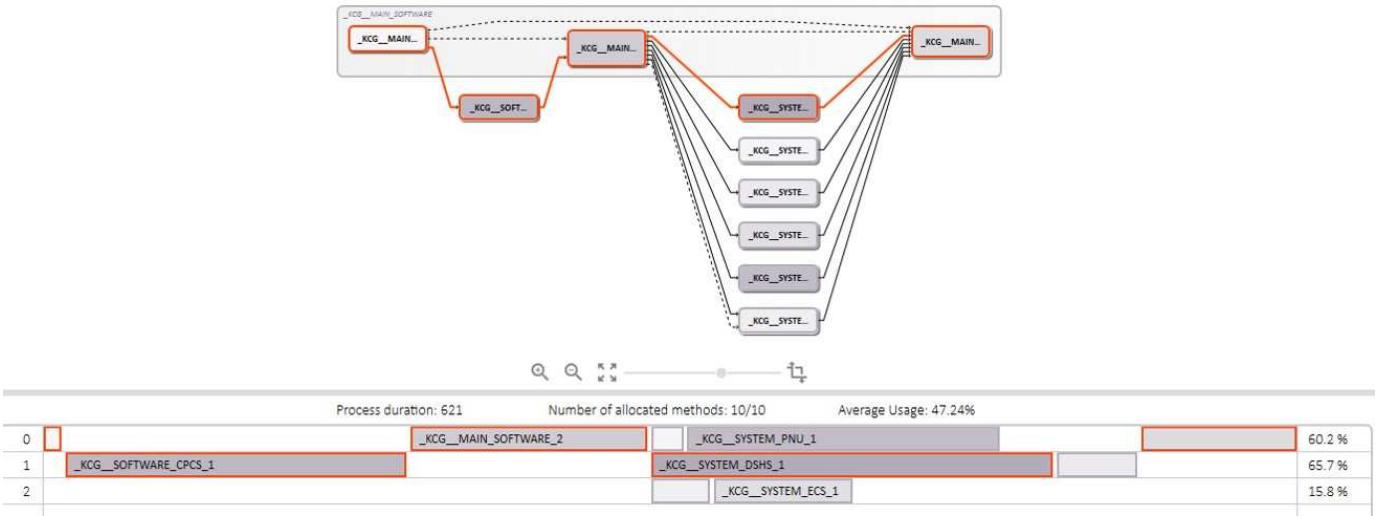


Fig. 7: Task graph of the IASC application and a partial view of its event-trigger schedule on a three-core hardware

"REQUEST_ISSUED !R1" and "REQUEST_SELECTED !R1"), *SYNCHECK* detects any modification due to the insertion of any temporal event, i.e., any event representing the passing of time.

V. Experimental results

A. Evaluation on synthetic applications

Initially, we have evaluated our approach on a very simple system composed of three synthetic tasks executing on the AURIX TC275 platform. The time-triggered schedule of those tasks has been computed, in which tasks T1 and T2 execute on *core 1* and task T3 executes on *core 2*. Task T2 executes after the completions of tasks T1 and T3. Task T1 executes in parallel with task T3. We assume that each task sends a data request to PFLASH0. Tasks T1 and T2 request for data located at consecutive addresses, while task T3 requests for data located at a distant address.

According to the interference analysis carried out using the STRANGE tool [10], all memory accesses of the three tasks suffer from contentions at the arbiter associated with PFLASH0. The reason is that they execute on two different cores which have crossing paths connected to PFLASH0. On the contrary, given the behavioral models (presented in Section IV-A), the model checker reports interferences at the arbiter suffered by tasks T1 and T3 and verifies that task T2 is free from contentions at the arbiter. That is true because the memory requests of task T2 and task T3 are temporally exclusive.

Moreover, the model checker reports that task T2 suffers from a contention at the global buffer. That is also true because task T1 and task T3 alternatively send requests to PFLASH0, which causes pollution of the global buffer. As a result, task T2 cannot reuse the prefetched data to the global buffer after the execution of task T1 as in the case when *core 1* executes in isolation.

B. Evaluation on an industrial use-case

We have also applied our interference analysis on an Integrated Air System Control (IASC) application[4]. The application, written in SCADE, is deployed on 6 parallel tasks as shown in the upper part of Fig. 7. Deployment is done using the SCADE toolsuite[5]. An event-trigger schedule (illustrated in the lower part of Fig. 7) of the application on a three-core platform is generated by using the S1[6] tool.

[4]This application is provided by LIEBHERR.
[5]We use ANSYS' SCADE KCG multicore suite [21].
[6]This tool is part of the SCADE multicore toolset provided by ANSYS.

The application is executed on the platform with the following constraints:

- Code and constant data of tasks mapped on different cores are allocated in different program flash memories;
- Data exchanged between different tasks happen only at the beginning (for reading shared data) and at the end (for writing shared data) of their execution;
- Tasks communicate with each other through data scratch-pad memories (DSPRs). The sender, at the end of its execution, stores exchanged data to the DSPR of the core on which the receiver is mapped. We assume that data written by different tasks to the same DSPR are located at different memory cells;
- Event-trigger is done by using semaphore variables which are located in local DSPR of each core.

With these execution constraints, tasks executing on different cores are ensured to be free from contentions when accessing to program flashes, but may interfere with each other when storing their output to the same DSPR. That warning is reported by our interference analysis. In particular, the accesses to the remote DSPR of the following tasks: _KCG__SYSTEM_WAIT_1, _KCG__SYSTEM_PNU_1, _KCG__SYSTEM_DSHS_1, _KCG__SYSTEM_AVS_1, _KCG__SYSTEM_OPS_1, and _KCG__SYSTEM_ECS_1 are reported to suffer from interferences.

VI. Conclusion and future work

In this paper, we have presented an approach for identifying interferences on multicore systems based on the behavioral modeling of the software and the hardware. We have proposed two strategies to identify interferences: *PATCHECK*, applicable when interferences can be defined by a prior known pattern, and *SYNCHECK*, based on the comparison of traces obtained from the *isolated* model and the *non-isolated* model. Both strategies are implemented using the LNT and MCL formal languages, and the CADP toolbox. We have applied our analysis to a part of the AURIX TC275 platform, and have shown that our analysis is not only safe but also able to prevent reporting spurious interferences.

At the current state of our behavioral models of the software and the hardware, we are not able to capture the case where tasks running in parallel issue memory requests at different points in time, which may result in pessimistic analysis. In order to enhance the precision of the analysis, in the future, we plan to enrich the current behavioral models by adding the information about the duration that state transitions of tasks will take. First experiments have been done using Time Petri Nets (TPN) design and analyzed using the Tina toolchain developed at LAAS[7].

References

[1] D. Kästner, M. Schlickling, M. Pister, C. Cullmann, G. Gebhard, R. Heckmann, and C. Ferdinand, "Meeting Real-Time Requirements with Multi-core Processors," in *Computer Safety, Reliability, and Security*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, vol. 7613, pp. 117–131.

[2] H. Garavel, F. Lang, and W. Serwe, "From LOTOS to LNT," in *ModelEd, TestEd, TrustEd – Essays Dedicated to Ed Brinksma on the Occasion of His 60th Birthday*, ser. LNCS, J.-P. Katoen, R. Langerak, and A. Rensink, Eds., vol. 10500. Springer, Oct. 2017, pp. 3–26.

[3] R. Mateescu and D. Thivolle, "A Model Checking Language for Concurrent Value-Passing Systems," in *FM 2008: Formal Methods*, J. Cuellar, T. Maibaum, and K. Sere, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, vol. 5014, pp. 148–164.

[4] H. Garavel, F. Lang, R. Mateescu, and W. Serwe, "CADP 2011: A Toolbox for the Construction and Analysis of Distributed Processes..." *Springer International Journal on Software Tools for Technology Transfer (STTT)*, vol. 15, no. 2, pp. 89–107, 2013.

[5] X. Jean, "Maîtrise de la couche hyperviseur sur les architectures multi-coeurs COTS dans un contexte avionique," PhD Thesis, 2015.

[6] "Certification Authorities Software Team (CAST)," 2016.

[7] THALES AVIONICS, "The use of multicore processors in airbone systems," 2011.

[8] S. Girbal, J. Le Rhun, and H. Saoud, "Metrics: A Measurement Environment For Multi-Core Time Critical Systems," *Zenodo*, Jul. 2018.

[9] P. Bieber, F. Boniol, Y. Bouchebaba, J. Brunel, C. Pagetti, T. Polacesk, and L. Santinelli, "Phylog - etude de la certificabilité des architectures logiciel-matériel reposant sur des calculateurs multi/many-core," ONERA, Tech. Rep., 2017.

[10] W-T. Sun, E. Jenn, and T. Carle, "Automatic Identification of Timing Interferences on Multi-Core Processors in a Model-Based Approach."

[11] M. Lv, W. Yi, N. Guan, and G. Yu, "Combining Abstract Interpretation with Model Checking for Timing Analysis of Multicore Software," in *2010 31st IEEE Real-Time Systems Symposium*. San Diego, CA, USA: IEEE, Nov. 2010, pp. 339–349.

[12] A. Gustavsson, A. Ermedahl, B. Lisper, and P. Pettersson, "Towards WCET analysis of multicore architectures using UPPAAL," in *10th international workshop on worst-case execution time analysis (WCET 2010)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2010.

[13] G. Fernandez, J. Abella, E. Quiñones, C. Rochange, T. Vardanega, and F. J. Cazorla, "Contention in multicore hardware shared resources: Understanding of the state of the art," in *14th International Workshop on Worst-Case Execution Time Analysis*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2014.

[14] "AURIX TC27x D-Step 32-Bit Single-Chip Microcontroller User's Manual V2.2 2014-12."

[15] D. Champelovier, X. Clere, H. Garavel, Y. Guerte, F. Lang, C. McKinty, V. Powazny, W. Serwe, and G. Smeding, "Reference manual of the lnt to lotos translator," 2018.

[16] F. Lang, "Exp.Open 2.0: A Flexible Tool Integrating Partial Order, Compositional, and On-The-Fly Verification Methods," in *Integrated Formal Methods*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, vol. 3771, pp. 70–88.

[17] H. Garavel and F. Lang, "SVL: a Scripting Language for Compositional Verification," in *Proceedings of the 21st IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE'01), Cheju Island, Korea*, M. Kim, B. Chin, S. Kang, and D. Lee, Eds. Kluwer Academic Publishers, Aug. 2001, pp. 377–392.

[18] H. Garavel, F. Lang, and R. Mateescu, "Compositional verification of asynchronous concurrent systems using CADP," *Acta Informatica*, vol. 52, no. 4-5, pp. 337–392, 2015.

[19] D. Chabrol, V. David, P. Oudin, G. Zeppa, and M. Jan, "Freedom from interference among time-triggered and angle-triggered tasks: a powertrain case study," 2014.

[20] H. Cassé and P. Sainrat, "OTAWA, a framework for experimenting WCET computations," in *3rd European Congress on Embedded Real-Time Software*, vol. 1, 2006.

[21] B. Pagano, C. Pasteur, G. Siegel, and R. Knizek, "A Model Based Safety Critical Flow for the AURIX(tm) Multi-core Platform," Toulouse, France, 2018.

[7] http://projects.laas.fr/tina/