An automata-based method for interference analysis in multi-core processors

Thomas Beck, Frédéric Boniol, Jérôme Ermont, Franck Wartel, Luc Maillet

An automata-based method for interference analysis in multi-core processors

Thomas Beck
Airbus Defence and Space
Toulouse, France

Frédéric Boniol
ONERA
Toulouse, France

Jérôme Ermont
IRIT - INP - ENSEEIHT
Toulouse, France

Luc Maillet
Airbus Defence and Space
Toulouse, France

Franck Wartel
Airbus Defence and Space
Toulouse, France

1. INTRODUCTION

Multi-core processors (MCPs) provide huge gains that allows replacing several embedded single-core processors with a smaller number of computing platforms. However, such processors face important challenges to their integration in safety-critical systems. Due to resource sharing, unwanted interferences may exist between the tasks hosted by the different cores that can cause unexpected delays.

Certification authorities have published a set of recommendations [2] for designers who want to certify a MCP hosting a set of safety-critical tasks. Basically, this entails a two steps process: First, the designer must identify all the interferences that can occur between the tasks. Second, he must show that the severity of the interferences is compliant with the real-time requirements of the tasks. Many works have faced this interference challenge in MCPs. The reader can refer to [4] for a large and complete overview of the scientific work on timing verification for MCPs. Among this work, a recent one [1] has shown that it is possible to characterise tasks running on MCPs by a timed sequence of Time Interest Points (TIPS), i.e., a sequence of instructions on which the tasks can suffer from interference.

We focus on the issue of the identification of interferences. We refine the notion of interference with two finer definitions, related to the type of the components causing the interference. And, considering that tasks can be abstracted by timed sequences of TIPS, we develop an the approach automata-based method, inspired from [3], to count the interferences that can occur between the tasks hosted by the processor.

2. METHOD OVERVIEW

The landscape in which the method is involved is shown in Figure 1. Within this landscape, the scope of the paper is highlighted by the gray box. Let us consider a set of tasks τ_i hosted by a MCP (on the left of the Figure). The objective of the method is to compute an upper bound of the extra timing penalty associated with each τ_i due to interference occurring in the architecture (on the right part of the Figure). For that purpose, we compute an upper bound of the maximal interference number (noted IN_i) from which each τ_i can suffer. The approach relies on the analysis method proposed by Carle *et al.* in [1] (on the left part of Figure 1). This method is based on the TIPS notion, which are load and store instructions that generate and suffer from interference. By means of static analysis, Carle *et al.* show that it is possible to extract a temporal segment sequence, as the one shown Figure 2, from the binary code of a task . Each segment is characterised by a duration (noted $d_{i,j}$ for task τ_i and segment j), and the maximal number of memory requests leaving the core and sent to the bus (noted $\mu_{i,j}$). These requests correspond to load and store operations that are not *always hit* in the L1 cache of the core hosting the task. For instance in Figure 2, τ_1 makes zero memory request in segment one, and at most three requests in segment two. Such a sequence defines a *bus access profile* of the task under consideration. It characterises its worst-case memory activity that can lead to interference. As shown in Figure 1 we take these profiles as inputs, and we study an automata-based method to count, by model-checking, the interference that can occur for each τ_i .

3. DEFINITIONS

Let us consider an archetypal MCP architecture depicted Figure 3. This processor is composed of (1) two cores (C_0 and C_1) owning their private cache $L1$, (2) a shared cache $L2$, (3) a DDR memory composed of one bank B, and (4) a shared bus allowing the two cores to address the $L2$ cache and the DDR. Each core C_i hosts a task called τ_i.

Let us suppose that τ_0 and τ_1 are characterised by the profiles shown in Figure 4. They are divided into three segments. τ_0 and τ_1 can compete to access the memory in the first segment (from 0 to 10). In this segment, each task sends at most 2 memory requests. In the next two segments, either τ_0 or τ_1 does not send any request, leaving the memory path available for the other task.

Requests from τ_0 and τ_1 could collide in the bus. If they arrive at the same time, only one of them can pass, the second one must wait. The effect of the interference is a delay caused by a simultaneous collision.

According to the request path of τ_0 and τ_1, a second interference could arrive in $L2$. However, the intrinsic nature of this interference is different. Let us imagine for instance that τ_0 reads a data from the bank B and put it in the $L2$ cache. As long as the data remains in the cache, each time τ_0 accesses it, its request path ends with $L2$. Let us imagine now that τ_1 reads another data from the bank B. According to the cache policy, data of τ_1 could evict data

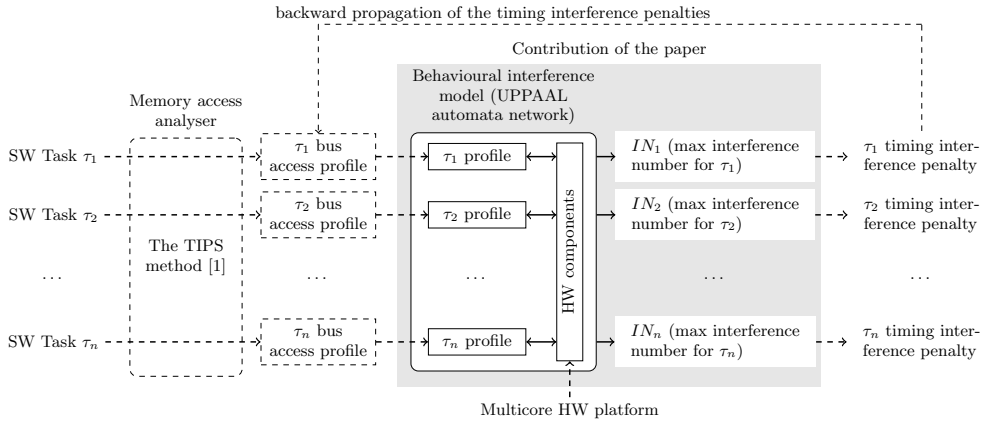

Figure 1: Approach overview (the gray box highlights the scope of the paper, while dashed lines are out of the scope)

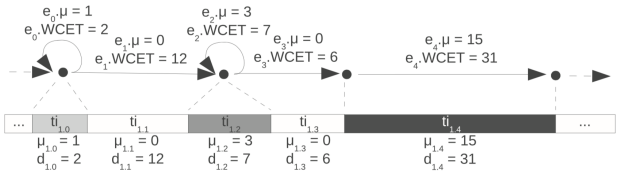

Figure 2: Example of bus access profile (excerpt from [1])

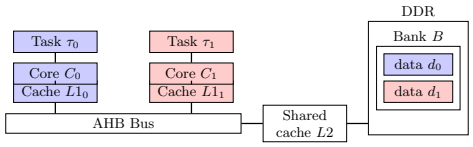

Figure 3: A simplified multi-core platform

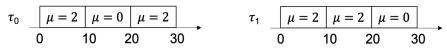

Figure 4: Bus access profile of τ_0 and τ_1

of τ_0 from $L2$. Consequently, the next time τ_0 would try to access its data, it should have to lengthen its request path to the memory. The effect is similar to the bus interference: τ_0 would suffer from a longer delay. However, the scenario of interference is different. There is no simultaneous collision. The two requests can occur at different time. In other words, τ_1 provokes a delayed interference on τ_0.

As shown in this example, in order to analyze the interferences that can occur, it is necessary to identify the types of HW components. We consider two types of components: transport and storage component.

Definition 1. (Transport component) A transport component is a component whose the internal state only depends on the presence or absence of a request using it. If a request is using the component, then it is "occupied". Otherwise, it is "free".

Definition 2. (Storage component) A storage component is a component whose the internal state depends on previous requests (including the current one if any).

AHB Bus of Figure 3 is a transport component. Examples of "storage components" include caches and memory banks. The content of a cache depends on the previous memory requests that used it. It has a direct effect on the length of next memory request paths. Following the distinction, we refine the notion of interference.

Definition 3. An instantaneous interference occurs whenever at least two requests sent by two different tasks collide on a same transport component.

Definition 4. A delayed interference occurs whenever a request r sent by a task τ uses a storage component whose internal state has been made non-compliant with τ by another task τ'.

As explained below instantaneous interferences can occur in *AHB Bus* Figure 3. And a delayed interference can occur in $L2$. In the same way, a second delayed interference can occur in B. Indeed, a memory bank is composed of a row buffer acting as a local cache. It contains the last block of accessed data. Hence, requests to data in the row buffer (one talks about "row hit" requests) are faster than request to data not in the row ("row miss" requests). When a "row miss" occurs, the requested data has to be fetched in the row buffer, making the request time longer. As in cache, the content of the row buffer depends on previous requests. Hence, banks memory are storage components that can cause delayed interferences.

4. MODELING

To model the interferences, we define two class of automata: automata for HW components, and automata for SW tasks.

4.1 HW components

The role of the HW component automata is to model the answers of the component to requests sent by the tasks. It determines if a request creates an interference in the component and notify the software automaton of this result. As said in section 3, we consider two types of HW components: transport and storage components.

4.1.1 Transport component

According to definition 1 a transport component is modeled by the automaton Figure 5. It is composed of three locations (*Free*, *Occupied* and *Check*), and three internal data (*waiting_queue*, *nb_elmt*, and *bus_state*):

- *waiting_queue* is an internal list containing the identifiers of the tasks waiting for the component.
- *nb_elmt* is the number of tasks currently waiting for the component (including the task currently using it).

The three locations of this automaton are:

- *Free*: This location is the initial one. The component is free and waits for a request. When a request arrives the location changes to *Occupied*, and the function *update_bus* is called. This function updates the internal variables of the automaton to let it know which task is calling it.
- *Occupied*: The component is already occupied by a request. Once the request is completed the next location is *Check* if the internal waiting queue is not empty ($nb\_elmt \neq 0$) and *Free* if the queue is empty ($nb\_elmt = 0$). All outgoing transitions of the *Occupied* location are waiting for the synchronization event *end_req*. This event is sent by a task automaton when it releases the component after its request has been completed by the memory.
- *Check*: This location's purpose is to switch the task whose request is handled by the component. It is a transient location reached when the current task occupying the component is releasing it and when another one is waiting for the component. The role of location is to trigger the transition returning to *Occupied* to process the next request. Once again the *update_bus* function is called when returning to *Occupied* to change the internal variables of the component.

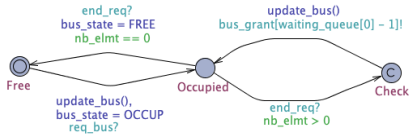

Figure 5: Automaton of a transport component

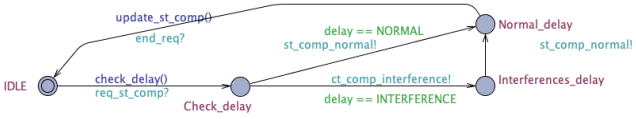

Figure 6: Automaton of a storage component

4.1.2 Storage component automata

Storage component generates delayed interferences, meaning that a task can interfere with another one by modifying the state of component. Storage components are modeled by the automaton Figure 6. The idea is to determine if the component reacts within a favorable delay (the normal case), or conversely within an unfavorable one (the interference case), when receiving a request. This notion of favorable delay is related to the task asking for the component. The automaton is composed of four locations:

- *IDLE* models the state where the component does not handle any request. It is waiting for a memory access request. When receiving it, it reaches *Check_delay* and it calls the function *check_delay* which determines whether there is an interference or not (i.e., whether the component is in an internal favorable state or not). *check_delay* compares the state of the component with the state of the component if the task is alone. Further explanations on this function are presented in the section 5.
- *Check_delay* is a transient location. Its role is to reach *Normal_delay* or *Interference_delay* according to the value of *delay*.
- *Normal_delay* models the normal response delay of request. Once the request is completed the component returns to *IDLE* and waits for another request. Before returning to the *IDLE*, the automaton is waiting for an occurrence of *end_req* sent by the task automaton processing the current request. When taking the transition to *IDLE*, the *update_st_comp* function is called to update the internal state variables of the component.
- *Interferences_delay*: When an interference occurs the response delay is extended. This location represents the added delay induced by the interference. Thereby the next location has to be *Normal_delay* because an interference is represented by an extra time added to the response delay.

4.2 Task automaton

A task automaton models the bus access profile of the task and the path of the requests through the HW components of the processor. Figure 7 gives the automaton of the task τ_0. It is composed of three parts. The first (locations *Request_bus*, *Result_bus*, and *Wainting_bus*) models the answer of the bus to the request. The second part (locations *Request_L2*, *Result_L2*, and *Wainting_L2*) models the answer of the L2 cache. And and the last part models the answer of the DDR memory.

The initial location of the automaton is an *Idle* location. It waits for a start (or a end) event from the scheduler (that is, the automaton implementing the sequence of the segment of the task profile). When receiving a start event, the local counter *nb_req* is set to zero. And the automaton reaches *Request_bus*. The three parts follow then the same pattern composed of three locations:

- *Request_cmp* (where *cmp* is *bus*, *L2*, or *DDR*) models the state in which the task has sent the request to the component *cmp* and is waiting to know if the request generates an interference. If there is an interference the next location is *Waiting_cmp*. If there isn't an interference the next location is *Results_cmp*.
- *Waiting_cmp* models the extra delay the task encounters due to the interference.
- *Results_cmp* models the case in which the request is normaly completed (with or without interference). The

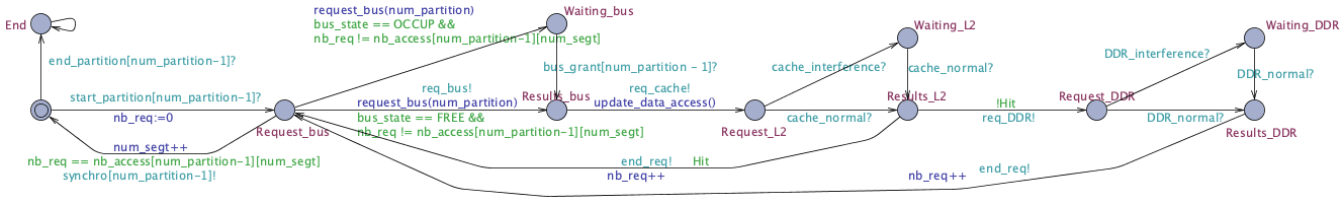

Figure 7: Automaton for task τ_0 of example Figure 3

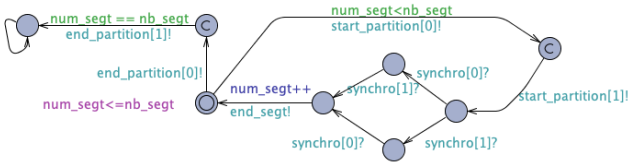

Figure 8: Scheduler automaton

next location is then the *Request_cmp* of the next component or *Idle* if the component is a storage component (L2 or DDR) and if the requested data is present in the component. When the outgoing transition of this location is taken, the task automaton sends an event to the component automaton to notify it of the end of the current request.

4.3 Scheduler automaton

The last automaton models the sequences of segments of the task profiles. For instance, the profiles of τ_0 and τ_1 shown in Figure 4 are modeled by the automaton in Figure 8. From the initial location, the Scheduler starts the first segment of τ_0 and τ_1. Then it waits for an occurence of *synchro* sent by τ_0 and τ_1 at the end of their segment. When receiving them, the scheduler notifies the end of the first segment, and starts the second one. And so on until the last segment (when $num\_segt == nb\_segt$).

5. INTERFERENCE ANALYSIS

To determine when an interference occurs we instrument each component automaton with an algorithm to decide whether or not the task accessing the component is suffering from an interference. These algorithms dependent on the type of the component under consideration.

Transport component. To count interferences in transport components is very simple: there is an interference *iff* the component is occupied and another request is waiting for it.

Storage component. For storage components, the decision algorithm is more complex. To determine if an access from a task X is affected by an access from a task Y, we not only maintain the "actual" state of the component but also the "virtual" state of the component for each task as if the task were alone. Such a "virtual" view of the component is only paying attention to accesses from the task it is associated to. It means that an access from task X to the storage component affects both the "actual" view and its "virtual" view of the component, but not the "virtual" view associated to task Y. Thereby, for each access to a storage component we are able to determine if there is interference or not by comparing the virtual view of the task with the actual view of the storage component. If there is a difference between these two views it means that an interference occurred. This algorithm is implemented by the function *check_delay()* of each storage automaton. In the example of a L2 direct mapped cache with 16.384 lines of 32 bytes, our algorithm works on: (1) an array *cache_array* containing 16 384 lists of 32 bytes representing the current state of the L2 cache ; (2) one similar array *task_i_array* for each task i representing the virtual state of the cache if the task were alone. Let us consider two tasks. Imagine that task 1 requests a data stored in line 250. *cache_array[250]* and *task_1_array[250]* are updated. Then task 2 accesses the same line, *cache_array[250]* and *task_2_array[250]* are modified. When task 1 accesses again line 250, *task_1_array[250]* and *cache_array[250]* are different meaning that task 1 suffers from an interference.

Interference analysis by model-checking. To compute an upper bound of the number of interferences experienced by each task in each component for each segment, we use the UPPAAL model-checker. For instance, let us consider $nb\_interf\_L2[0,\tau_0]$ the number of interference in the first segment of τ_0 in cache L2 . UPPAAL shows that

$$\forall[](nb\_interf\_L2[0,\tau_0] \leq 3)$$

that is, 3 is an upper bound of $nb\_interf\_L2[0,\tau_0]$. This computation takes less than one second (UPPAAL 4.1.24 running on a 3,2 GHz Apple M1 Pro with 32 Go DDR5).

6. NEXT WORK

The next step is to conduct a set of experiments to validate the method. We plan to explore the GR740 processor: a quad-core processor based on SPARC V8 architecture. After this validation, the second work will consider finer placement and replacement policies in our cache model in order to capture more realistic configurations.

7. REFERENCES

[1] Thomas Carle and Hugues Cassé. Reducing timing interferences in real-time applications running on multicore architectures. In *18th International Workshop on Worst-Case Execution Time Analysis*, 2018.

[2] Certification Authorities Software Team. Multi-core Processors - Position Paper. Technical Report CAST 32-A, November 2016.

[3] Andreas Gustavsson, Andreas Ermedahl, Björn Lisper, and Paul Pettersson. Towards WCET analysis of multicore architectures using UPPAAL. In *10th International Workshop on Worst-Case Execution Time Analysis*, 2010.

[4] Claire Maiza, Hamza Rihani, Juan Maria Rivas, Joël Goossens, Sebastian Altmeyer, and Robert I. Davis. A survey of timing verification techniques for multi-core real-time systems. *ACM Comput. Surv.*, 52(3):56:1–56:38, 2019.