Deterministic Execution Model on COTS Hardware

Frédéric Boniol[1], Hugues Cassé[1], Eric Noulard[1], and Claire Pagetti[1,2]

[1] ONERA, Toulouse, France
[2] IRIT, University of Toulouse, France

Abstract. In order to be able to use multicore COTS hardware in critical systems, we put forward a time-oriented execution model and provide a general framework for programming and analysing a multicore compliant with the execution model.

1 Introduction

The use of multicore brings several benefits in embedded systems such as the weight reduction, the increased performance or a reduced maintenance. Embedding a Commercial-Off-the Shelf (COTS) hardware in a safety critical system must inevitably be accompanied by a formal proof of correctness. In this paper, we propose a way to execute safely a specification on a multicore COTS. Our *system model* consists of a set of concurrent periodic tasks communicating via shared variables and subjected to dependencies. Our *system architecture* is a symmetric multicore processor where each core has private first-level (L1) and second-level (L2) caches with a shared memory bus and controller for accessing the DRAM.

1.1 Challenge for Embedding a Multicore COTS

An execution of the system is correct if all the constraints and in particular the temporal ones are met. It is therefore necessary to compute the worst case execution time (wcet) for any task. This computation takes into account the possible interactions and resource contentions due to the competition generated by the other tasks. Unfortunately, today, there is no solution to compute tight WCETs for multi-threaded code on multicore COTS. Indeed, existing works [23] and tools, such as Absint [13] and OTAWA [1] have severe restrictions. They operate well as long as the code is sequential and the processor is single core. Furthermore, no other device with an unpredictable behaviour should interfere with the processor execution timings. There are additional difficulties for multicore:

1. the *shared internal bus* access. The concurrent accesses to the same resource (for instance the RAM) are serialised in a non predictable way.
2. the *cache coherency*. The update of a variable concurrently stored in several caches is automatically done by the COTS in a hidden and non predictable way (it requires an access to the bus with no user explicit instruction).

3. available documentation including descriptions of shared buses, memory and other devices controllers, does not give enough details on the COTS.

Any critical systems designer has to cope with these problems and has mainly two approaches to safely embed a multicore. The first involves designing a *time-able* processor architecture [22,14] or an internal bus with a time division multiple access (TDMA) [4,20]. It is, thus, possible to strongly improve worst-case analyses. The cost of such specific hardware developments may often prevent their use and may force the designer to rely on a COTS. The second approach is then to apply an *execution model*: the idea is to define some rules that constrain and reduce the number of non predictable behaviours. If the rules are well chosen, the system may be analysed without too much pessimism. The basis is to apply time oriented mechanisms by constraining the behaviours within timing slots.

1.2 Contribution

We defined a generic deterministic execution model allowing the computation of worst case times. This industrial approach had led to a patent which is public since June 2010. The purpose of this article, is to present a process for programming and analysing a multicore, the usage of which is compliant to the execution model.

The generic execution model distinguishes, on each core, times of functional computation and times for accesses to the memory. These two types of computation occur in different *slices* which are statically defined. The *sliced execution model* operates as follows:

1. two kinds of slices alternate indefinitely on each core:
   - *execution slices:* a core in an execution slice executes a functional code without any shared resource access. This means that all the instructions and data are locally stored in the caches;
   - *communication slices:* during such a slice, the core does not make any functional computation. It first flushes from local cache(s) to the RAM the values computed during the previous execution slice, and then fetches into local cache(s) all the codes and data required for the next execution slice.
2. a *static synchronous scheduling* of the slices on each core is defined off line. It describes a repetitive pattern where communication slices are synchronous.

The right side of Figure 1 shows an example of a slice scheduling for a dual core. White blocks represent execution slices. Grey (resp. black) blocks represent flushes (resp. fetches). Assume that each core has a local clock physically derived from a common hardware clock, implementing time slots do not need any specific hardware or software synchronisation mechanism.

From now on, a COTS hardware equipped with a sliced execution model will be called a *sliced architecture*. The main contribution of this paper is to propose an automatic programming framework and a series of analysis techniques for a tight evaluation of a sliced architecture. Figure 1 illustrates the development process for such an architecture:

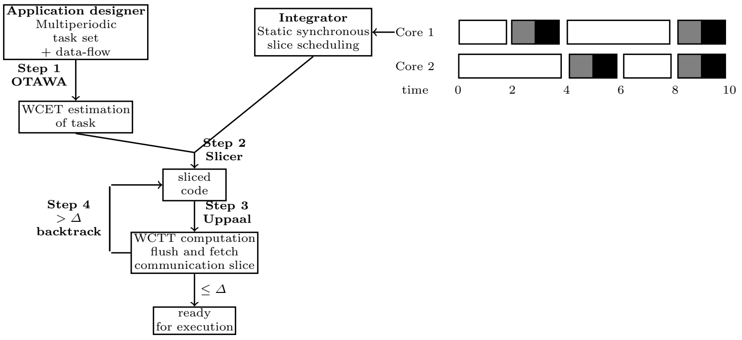


Fig. 1. Development process

- the inputs are the functional specification and the synchronous slice scheduling chosen by the integrator.
- (step 1) involves evaluating the wcet of each execution slice separately. This is reduced to a static wcet evaluation of a non preemptive sequential code without any non predictable behaviour (thanks to the locality of the data and code within the caches) on a uniprocessor, which is a well-known problem.
- (step 2) is the generation of a static allocation of the tasks and the data-flows on the sliced architecture. The algorithm chooses the addresses of the instructions and data, as well as the allocation in the execution slices, such that all the functional (e.g., precedences, periods, deadlines) and capacity constraints (e.g., slice and cache bounds, memory capacities) are met.
- (step 3) involves proving that the *worst case traversal time* (wctt) of each communication slice is less than the temporal length of the slice. The idea is to explore all possible concurrent write (resp. read) combinations from the cores and peripherals involved in a communication slice by making a precise model of the shared bus, the memory controller and the RAM.

We show the feasibility and how we can implement a sliced multicore by developing the tools for a MPC8641D [10] target. It is a dual core processor made up with two Power PC e600 cores[9], a MPX Coherency Module (MCM) which serialises the requests for the memory controllers and a dynamic RAM. We have developed an Otawa model for computing execution slice wcet, we have developed a slicer based on constraint programming and Uppaal [2] models for computing wctt. We have applied them successfully on a avionics application [3].

1.3 Related Work

The use of an execution model is not new. The Bulk Synchronous Parallel (BSP) model [21] had been designed to restrict the behaviour of parallel programs. More recently, the predictable execution model named *PREM* [17] had been proposed for uniprocessor COTS in order to cope with CPU and I/O interferences. More recently, predictable concepts for many-core architecture [16] have been defined and implemented in a cycle accurate simulator.

Our solution goes further in the time-oriented separation of concerns than existing execution models (such as BSP or *PREM*). The cores are synchronised and repeat an off-line scheduling. Fixed intervals are enforced both for execution and communication.

The main novelty is that we propose a development process and an automatic tool set for the designer to program a sliced architecture. For instance, in [17], the designer must annotate its code in order to indicate to the PREM-aware compiler which piece of code executes in a predictable interval. In our approach, the allocation is left to the slicer. If the wcet computation is reduced to a standard calculation, the tight computation of wctt is rather new. This is made possible because we know which data are read (resp. written) within the communication slices but also because we provide a detailed architectural description of the memory accesses. Most papers are dedicated to the definition of predictable memory controllers but very few propose an abstract model of the multicore COTS communication system. [11] proposed a Petri net based formal modelling of uniprocessor memory access. [7] modeled exchanges between a radar and a memory with a Uppaal model.

2 Wcet of an Execution Slice

The WCET computation has been applied on an e600 processor core [9] but it is suitable for any kind of processor core which does not share caches with other parts of the multicore.

2.1 Reminder on Uniprocessor wcet Estimation

Most of the approaches and tools work on three major steps [23].

1. Control flow graph construction The first step involves generating the *control flow graph* (CFG) from the binary of the program. The nodes are the instructions (usually grouped in basic blocks) and the edges the transitions between instructions. Building a CFG is generally automatic and requires to determine the maximum number of iterations in a loop and the number of iterations overall the run of the program.

2. Micro-architectural analysis This analysis computes the timings of the program blocks based on the *execution graph* technique [19]: the idea is to describe precisely the temporal behaviour of an instruction in each stage of the pipeline and dependencies of instructions and pipeline resources. Such a precise hardware model is built from available documentation and measures.

3. Overall wcet is obtained by combining the cost of the basic blocks. The *Implicit Path Enumeration Technique* (IPET) encodes the timing behaviours and variations of the program as an integer linear programming problem (ILP).

In our experimentation, we have used Otawa [1], a framework developed at IRIT since 2004, to compute WCETs of C code programs.

2.2 Application to a Sliced Architecture

Using OTAWA on the sliced architecture requires only the definition of the micro-architectural analysis. Since there is no access to any external component, computing the WCET for an execution slice is as simple as for a uniprocessor. A precise model of the PowerPC e600 [9], which belongs to the family of 32-bit Power Architecture microprocessor cores developed by Freescale, had been defined. Due to the high complexity of the core and even if the behaviour is highly constrained and confined, calculating the WCET for an e600 was a real challenge.

Architecture Modelling. The memory and the pipeline have to be modeled. In order to simplify analyses : (1) all data are pre-loaded in the L1 data cache causing a constant access time (1 cycle), (2) all instructions are pre-loaded in the L2 instruction cache causing also a constant access time (12 cycles), (3) the dynamic branch predictor is deactivated (only static branch prediction, encoded in the instruction code, is supported to achieve a maximum of determinism). From the timing point of view, the memory hierarchy is only made of (1) an L2 cache which can be viewed as a single scratch-pad memory (SPM) bank with constant access time of 12-cycles, (2) the L1I cache is not used, (3) an 8-way associative L1D cache.

The pipeline is composed of usual stages: the *Fetch* stage loads the instruction from the L2 cache (since the L1I is deactivated) and can store up to 12 instructions in the instruction queue *IQ*. The *Dispatch* stage pulls part of these instructions and stores them in the Complete queue *GIQ*. This queue can keep at most 6 instructions and dispatch 3 instructions. The instructions are then distributed to the different functional units (FU). There are 3 parallel integer units $IU1_i$, $i = 1, \ldots, 3$ for operations on registers and immediate, a unit *IU2* for multiplication and division, a load and store unit *LSU* for the memory accesses and a dedicated branch prediction unit *BPU*. During the execution, the instructions are stored in the completion queue *CQ* of size 16. The final stage is the completion unit *CU* which extract instructions from *CQ* and the instruction changes to the registers.

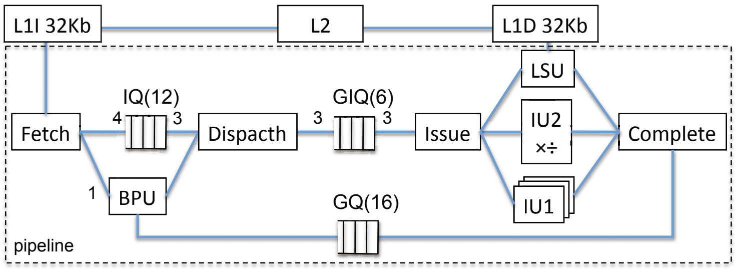


Fig. 2. PowerPC pipeline

We have made some simplifications. First, we assume that the instructions only handle integer (no float and no vector). Second, we did not encode the

GIQ. The purpose of this queue is to reduce contention on the IQ. Without the GIQ, the IQ becomes full faster. So our model is more pessimistic than the real pipeline and generates a small overestimation of the WCET.

Integration in the Development Process. The WCET is computed before the allocation, therefore addresses are not known. It's not a real matter since, whatever will be the final allocation, the upper bound remains the same. We computed the WCET of each task with addresses chosen by the gcc compiler.

Otawa computed the WCET of the avionics application [3] which is composed of 762 tasks in less than 2 hours.

3 Slicer

The mapping problem consists in assigning (allocating) each task to an execution slice such that all functional (precedence and deadline) and non functional (memory and processor capacities) constraints are met. This problem is a variation of the *bin packing problem* which is a NP-complete problem. We have addressed a similar problem in [3] for a distributed architecture. The novelty stands in:

- we consider additional dimensions. Indeed, the slicer generates the memory address allocation for the instructions and a mapping of variables within memory blocks. The blocks must be stored in the cache lines according to the associativity.
- we have developed an ad hoc constraint resolution code since the solvers were not able to treat the problem, the size of which was too huge.

Note that the constraints are architecture dependent since they must encode the cache characteristics such as the properties of associativity. If a task overflows the capacity either of the L2 or the L1D cache, it has to be split into several sub-tasks with the same period and deadline, and with additional precedence constraints.

System model. The system model consists of a set of periodic tasks communicating via shared data and subject to dependencies. Such an applicative may be the result of a data-flow specification associated with primitives on the system design. It is the case, for instance, of a specification in SIMULINK [15] coupled with a multithreaded code generator (such as Real-time workshop-embedded coder from the MathWorks or TargetLink from dSpace). It is also the case of a specification in LUSTRE [12] extended with the operators and the relaxed synchronous hypothesis of [6]. A last example is a specification in PRELUDE [8] which aims at describing multiperiodic data-flow functions.

Definition 1 (System model). *A system $\mathcal{S}$ is a tuple $\langle \mathcal{F}, \mathcal{V}, \mathcal{R} \rangle$ such that:*

1. *$\mathcal{F} = \{f_1, \ldots, f_n\}$ is a finite set of tasks where each task is a terminating sequential program described by:*

   (a) *size_code: (resp. wcet:)* $\mathcal{F} \rightarrow \mathbb{N}$ *associates to each task* f_i *the number of memory lines required to store the code of* f_i *(resp. its wcet, for instance computed by* OTAWA*);*
   (b) T *: (resp.* D *:)* $\mathcal{F} \rightarrow \mathbb{N}$ *provides the period of a task (resp. the relative deadline). We assume* $D(f_i) \leq T(f_i)$*;*
2. $\mathcal{V} = \{v_1, \ldots, v_k\}$ *is a finite set of data consumed and produced by* f_i *with:*
   (a) *size_var:* $\mathcal{V} \rightarrow \mathbb{N}$ *associates to each data its size in bytes;*
   (b) *in:* $\mathcal{F} \rightarrow 2^{\mathcal{V}}$ *(resp. out:* $\mathcal{S} \rightarrow 2^{\mathcal{V}}$*) gives the set of data consumed (resp. produced) by each task. We assume that each data is produced by at most one function:* $\forall i, v_i \in out(f_k) \cap out(f_l) \implies k = l$*;*
   (c) $\mathcal{I} \subseteq \mathcal{V}$ *(resp.* $\mathcal{O} \subseteq \mathcal{V}$*) is the set of inputs from (resp. outputs for) the environment;*
3. $\mathcal{R} \subseteq \mathcal{F} \times \mathcal{F}$ *is a relation of precedence relating tasks with the same period. If* $(f_i, f_j) \in \mathcal{R}$*, this means that* f_i *must always execute before* f_j*. In the following, we only consider causal systems, i.e., such that the precedence relation defines an acyclic graph.*

Sliced code. The *slicer* is an off-line algorithm which computes a mapping (if any exists) of a system compliant with definition 1 onto a sliced architecture. The result, called *sliced code*, statically defines the addresses of the instructions and data in RAM, the slices where tasks execute, the slices where I/O are emitted and a pattern that is repeated infinitely.

Definition 2 (Sliced code). *Let* $\mathcal{S} = \langle \mathcal{F}, \mathcal{V}, \mathcal{R} \rangle$ *be a system and* $\mathcal{P}$ *a slice scheduling, a sliced code is defined by* $\langle H$, *addr_c*, *addr_v*, *alloc*, *alloc*$_{io}\rangle$ *where:*

1. H *is the length of repetition (it is a multiple of the length* L *of* $\mathcal{P}$*),*
2. *addr_v:* $\mathcal{V} \rightarrow \mathbb{N}$ *associates an address to each data,*
3. *addr_c:* $\mathcal{F} \rightarrow 2^{\mathbb{N}}$ *associates to each task* f_i *a set of size_code*(f_i) *addresses,*
4. *alloc:* $\mathcal{F} \rightarrow 2^{[1,n] \times [1,H/L]}$ *indicates on which slices the task* f_i *executes,*
5. *alloc*$_{io}$*:* $\mathcal{I} \cup \mathcal{O} \rightarrow 2^{[1,n] \times [1,H/L]}$ *indicates on which communication slices each input is stored in the RAM and each output is emitted on the peripheral.*

Example 1. Let us consider the system:

$\mathcal{F}$ (name,(code size, wcet))						$\mathcal{V}$ (name,size,producer,consumers)				$\mathcal{R}$	
10 ms		20 ms		40 ms							
f_1^1	(10,1)	f_1^2	(5,0.5)	f_1^3	(15,1.5)	v$_0$	12	f_1^1	(f_1^3,f_2^1,f_4^1)	f_1^1	f_2^1
f_2^1	(10,1)	f_2^2	(10,1.5)	f_2^3	(20,3)	v$_1$	15	f_1^2	(f_1^3,f_1^1)	f_1^2	f_2^2
f_3^1	(10,1)	f_3^2	(3,0.3)			v$_2$	20	f_1^3	f_2^3		
f_4^1	(3,0.3)					i_1	10		(f_1^1,f_1^2)		
f_5^1	(3,0.4)					o_1	12	f_2^2			

A sliced code for the scheduling of Figure 1 is given by $\langle 40$, *addr_c*, *addr_v*, *alloc*, *alloc*$_{io}\rangle$ where:

– *addr_c*$(f_i^j) = \{l_1^{i,j}, \ldots, l_{size\_code(f_i^j)}^{i,j}\}$ where $l_k^{i,j}$ is a line (or a row);

- $addr\_v(v_i) = a_i$ with $a_i \in l_i$ (the address a_i is in line l_i), $addr\_v(i_1) = a$ with $a \in l$ and $addr\_v(o_1) = a'$ with $a' \in l'$. The lines satisfy the constraints $l_0 = l_1 = l$ (v_0, v_1 and i_1 are stored in the same line) and $l_2 = l'$;
- *alloc* and $alloc_{io}$ are described below on [0,20]:

	$slice_{i,1}$	$slice_{i,2}$	$slice_{i,3}$	$slice_{i,4}$
core 1	$f_1^1; f_2^1$	$f_5^1; f_3^2; f_1^3$	$f_1^1; f_2^1$	f_5^1
core 2	$f_3^1; f_4^1; f_1^2$	f_2^2	$f_3^1; f_4^1; f_2^3$	

$com_{1,1}$	$com_{2,1}$	$com_{1,2}$	$com_{1,3}$	$com_{2,3}$	$com_{1,4}$
		i_1, o_1			i_1

We can check for instance on $slice_{1,1}$ that the constraints are fulfilled: $size\_code(f_1^1) + size\_code(f_2^1) = 20$ lines (L2 has a capacity of 1-Mbyte and a line requires 64 bytes), $wcet(f_1^1) + wcet(f_2^1) = 1.5 \leq 2$, $size\_var(v_0) + size\_var(v_1) = 27$ bytes (L1 has a capacity of 32 kbyte). Caches are 8-way set-associative. Let $f : \mathcal{L} \rightarrow [1, 8]$ be the function that gives the set where a line $l \in \mathcal{L}$ is stored. We have $\forall c \in [1, 8], (\Sigma_i(f(l_i^{1,1}) = c) + \Sigma_i(f(l_i^{1,2}) = c)) \leq 8$ (no more than 8 lines belong to the same set).

We have implemented a depth-first search algorithm. It tries to allocate each task in an execution slice e by verifying all the capacity constraints such as $\sum_{i \in e} wcet(f_i) \leq length(e)$. Once a solution is found, it computes an address mapping that respects the cache associativity. Although the mapping problem is NP-complete, the implemented algorithm was successfully applied on the avionics applications. It finds more than 1000 solutions in less than 1s.

4 Wctt of a Communication Slice

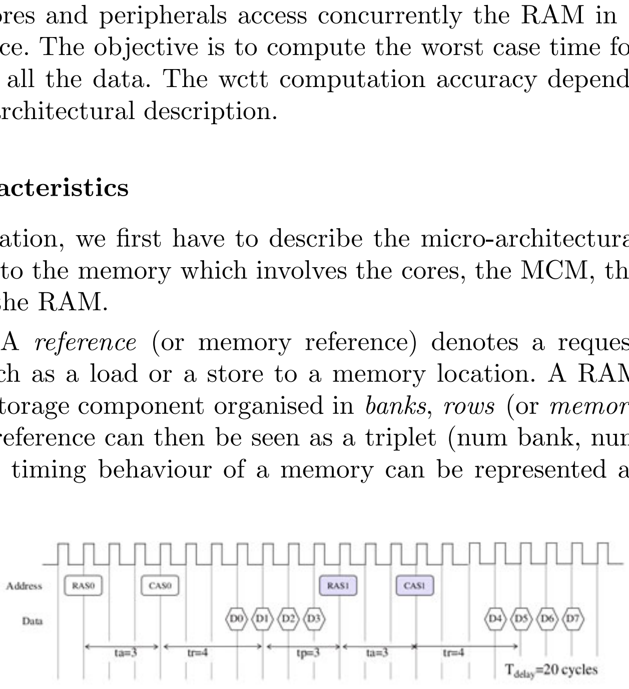
Once the sliced code has been generated, we know exactly which data are flushed or fetched and which cores and peripherals access concurrently the RAM in a given communication slice. The objective is to compute the worst case time for flushing (resp. fetching) all the data. The wctt computation accuracy depends on the accuracy of the architectural description.

4.1 Hardware Characteristics

As for the wcet computation, we first have to describe the micro-architectural behaviour of any access to the memory which involves the cores, the MCM, the memory controller and the RAM.

Memory behaviour. A *reference* (or memory reference) denotes a request generated by a core, such as a load or a store to a memory location. A RAM [18] is a 3-dimensional storage component organised in *banks*, *rows* (or *memory pages*) and *columns*. A reference can then be seen as a triplet (num bank, num row, num column). The timing behaviour of a memory can be represented as follows:

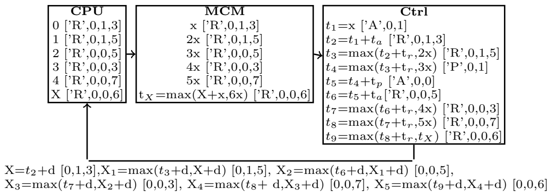
action	timing
row activation 'A'	t_a
bank precharge 'P'	t_p
write 'W'	t_w
read 'R'	t_r
refresh action 'REF'	t_{ref}
minimum time between 'P'	t_i



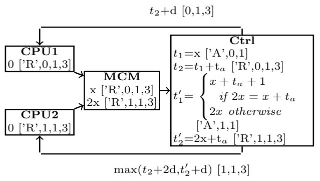
Actions happen in a given order: initially a bank is idle. Then, if there is a reference in the controller FIFO, the controller first asks for an activation A and it takes t_a cycles for the RAM to store the row in the buffer. Then, the controller emits a command R or W which takes t_r or t_w cycles for the RAM to execute. If there is a second reference in the same bank and in the same row, the controller asks directly for the read or write, but if it is on a different row, the controller first asks for a precharge P which takes t_p cycles, then for an activation and finally emits the command. There is a constraint on two successive precharges: they must be spaced of t_i cycles. There is also a price for switching from a read to a write and vice versa but this never occurs since we distinguish the requests. The banks work independently and can store a row in their local buffer.

Refresh actions occur regularly on the DDR for physical reasons. This happens nb_{ref} times during a duration of I_{ref} units of time. When a refresh occurs, all the banks are precharged, refreshed and left in the idle state.

Requests and data paths for the memory. The cores have a FIFO of size 5 (resp. 8) for emitting the read requests (resp. the data to flush). The MCM has two FIFOs of size 8 for storing the requests of each core (which has been represented as a 16-cell FIFO). The MCM has also a 16-cell FIFO for the data exchanged with the RAM. The size of the FIFO of the controller is 4.


CPU
0 ['R',0,1,3]
1 ['R',0,1,5]
2 ['R',0,0,5]
3 ['R',0,0,3]
4 ['R',0,0,7]
X ['R',0,0,6]
MCM
x ['R',0,1,3]
2x ['R',0,1,5]
3x ['R',0,0,5]
4x ['R',0,0,3]
5x ['R',0,0,7]
t_X=max(X+x,6x) ['R',0,0,6]
Ctrl
t_1=x ['A',0,1]
$t_2=t_1+t_a$ ['R',0,1,3]
t_3=max(t_2+t_r,2x) ['R',0,1,5]
t_4=max(t_3+t_r,3x) ['P',0,1]
$t_5=t_4+t_p$ ['A',0,0]
$t_6=t_5+t_a$ ['R',0,0,5]
t_7=max(t_6+t_r,4x) ['R',0,0,3]
t_8=max(t_7+t_r,5x) ['R',0,0,7]
t_9=max(t_8+t_r,t_X) ['R',0,0,6]
X=t_2+d [0,1,3],X$_1$=max(t_3+d,X+d) [0,1,5], X$_2$=max(t_6+d,X$_1$+d) [0,0,5],
X$_3$=max(t_7+d,X$_2$+d) [0,0,3], X$_4$=max(t_8+ d,X$_3$+d) [0,0,7], X$_5$=max(t_9+d,X$_4$+d) [0,0,6]

(a) Same core, same bank


t_2+d [0,1,3]
CPU1
0 ['R',0,1,3]
CPU2
0 ['R',1,1,3]
MCM
x ['R',0,1,3]
2x ['R',1,1,3]
Ctrl
t_1=x ['A',0,1]
$t_2=t_1+t_a$ ['R',0,1,3]
$t'_1= \begin{cases} x+t_a+1 \\ \quad if\ 2x = x+t_a \\ 2x\ otherwise \end{cases}$
['A',1,1]
t'_2=2x+t_a ['R',1,1,3]
max(t_2+2d,t'_2+d) [1,1,3]

(b) 2 cores, 2 banks

Fig. 3. Timing for a series of read

Figure 3 (a) gives the time behaviour of the requests and the data for a series of read. The core emits 6 requests: since the FIFO is of size 5, the first 5 requests are emitted in sequence and the 6th cannot be emitted as long as the first requested data is not received by the core. The timings in the boxes describe the time when the request or data is output. For instance, the CPU emits at 0 the request ['R',0,1,3] (access to 0: num bank, 1: num row, 3: num column). The core can send a request every cycle. The MCM acts like a queue with a rate x of emission. The first request received at 0 by the MCM is emitted to the controller at x, the second received at 1 can start to be treated at x and requires x to be transmitted. The controller receives the first request at x, since the bank is idle, it first asks for an activation 'A' and after t_a for a read 'R'. The second request is available in the controller at $2x$ and the controller can start to treat it at $t_2 + t_r$, this is the reason why the 'R' is launched at $t_3 = max(t_2 + t_r, 2x)$. The value of [0,1,3] is sent to the CPU and arrives at $X = t_2 + d$ where d is the output rate of the data bus which is also a FIFO queue. The value of [0,1,5] is therefore emitted after the previous data was sent at X and after being emitted by the RAM at t_3.

When several components read or write in a same communication slice, there are several possible interleavings. The requests are first serialised in the MCM. When reaching the controller there may exist several concurrent scenarii:

- the RAM allows several rows to be opened at the same time. In that case, it reduces the number of 'P' and 'A',
- the components access different banks as shown in figure 3 (b),
- the component access different controllers (when the system includes several memory controllers).

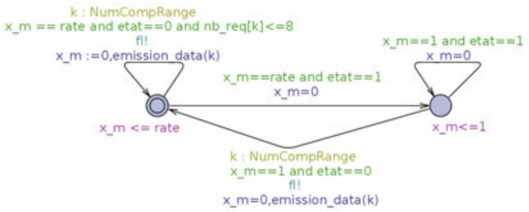
A refresh occurs at any time and closes all the banks. At worst, it is inserted between two reads in the same row and the local timing cost is $t_p + t_a + t_{ref}$. This cost may then be absorbed in the output FIFO and may not impact the emission of the data.

4.2 Wctt Computation

In this section, we present a solution for computing tight wctt. A first over-approximation is to determine the worst case times t_{read}^{max} for a memory read or t_{write}^{max} for a write; and then multiply by the number of reads and writes. According to the previous description, the worst case occurs when the MCM FIFOs are full and the RAM is opened in a bad row. We obtain for instance $t_{read}^{max} = 16x + t_i + t_p + t_a + 16d$. This solution entails a very high over dimensioning.

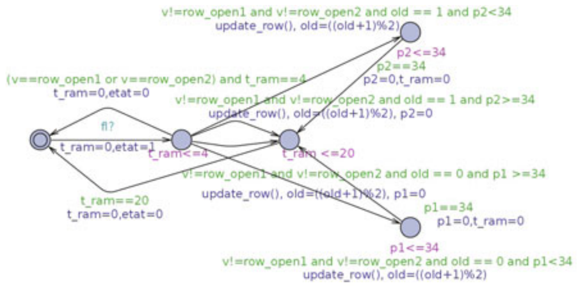
To avoid it, we formally modeled the exact behaviour with a network of timed automata using UPPAAL. For the MPC8641D, we have $t_a = 8$, $t_p = 8$, $t_w = 4$, $t_r = 4$, $t_{ref} = 64$, $t_i = 34$, $x = 6$ and $d = 12$. We know statically which data are flushed (resp. read): we store those data as 2 lists (one for each core and one for a peripheral) with the bank and row numbers. Let us illustrate this idea for a flush slice. There are 2 automata:

1. The first represents the exchanges between the MCM and the controller.



The loop on the initial state is non deterministic and allows all the possible interleaving between the components emission. Data is emitted at exactly 12 if the memory is available or is postponed until the memory becomes free;

2. The second models the memory with 2 simultaneous open pages.


v!=row_open1 and v!=row_open2 and old == 1 and p2<34
update_row(), old=((old+1)%2)
p2<=34
p2==34
p2=0,t_ram=0
(v==row_open1 or v==row_open2) and t_ram==4
t_ram=0,etat=0
v!=row_open1 and v!=row_open2 and old == 1 and p2>=34
update_row(), old=((old+1)%2), p2=0
fl?
t_ram=0,etat=1
t_ram<=4
t_ram <=20
v!=row_open1 and v!=row_open2 and old == 0 and p1 >=34
update_row(), old=((old+1)%2), p1=0
t_ram==20
t_ram=0,etat=0
p1==34
p1=0,t_ram=0
p1<=34
v!=row_open1 and v!=row_open2 and old == 0 and p1<34
update_row(), old=((old+1)%2)

For modelling the data exchange between the controller and the RAM, we use an automaton synchronisation, which syntax in Uppaal is $fl?$ for the receiver and $fl!$ for the emitter. After the synchronisation, writing takes 4 cycles if the page is opened. If the last prefetch occurs more than 34 cycles ago then it takes 20 cycles otherwise it takes 20 cycles plus the difference between 34 cycles and the last prefetch of the row.

Using this model, we can formally verify that the flush is done by simply verifying that the global clock never exceeds the bound which is expressed by $A[](h \leq \mathit{length})$. Since, the computation does not take into account the refreshes, we add a penalty by counting the maximal number of refreshes during a slice. Therefore, we just need to add a constant cost.

We apply the wctt evaluation on the case study. For small communication slices, the model checker succeeds but as soon as the number of data grows, it encounters the combinatorial explosion.

5 Conclusion

The aim of this article was to propose a development cycle for multicore COTS under time triggered execution model. We have presented several tools for studying and programming such an architecture. The results produced by those experiments encourage us to go a step further. We are currently applying the method on an open source avionics application which controls the longitudinal behaviour of an aircraft [5]. Future work mainly concerns two directions. The first perspective is to improve the wctt using an ILP (integer linear programming) approach which seems to be more suitable for this kind of problem. The second one aims at studying the backtrack (step 4). If the wctt exceeds the bounds, the slicer must provide a new solution. Providing a new solution is a difficult problem. Removing a single mapping will not give hints on the intrinsic overflow.

References

1. Ballabriga, C., Cassé, H., Rochange, C., Sainrat, P.: OTAWA: An Open Toolbox for Adaptive WCET Analysis. In: Min, S.L., Pettit, R., Puschner, P., Ungerer, T. (eds.) SEUS 2010. LNCS, vol. 6399, pp. 35–46. Springer, Heidelberg (2010)

2. Behrmann, G., David, A., Larsen, K.G., Håkansson, J., Pettersson, P., Yi, W., Hendriks, M.: UPPAAL 4.0. In: Proceedings of the 3rd International Conference on the Quantitative Evaluation of SysTems (QEST 2006) , pp. 125–126. IEEE Computer Society (2006)
3. Boniol, F., Hladik, P.-E., Pagetti, C., Aspro, F., Jégu, V.: A Framework for Distributing Real-Time Functions. In: Cassez, F., Jard, C. (eds.) FORMATS 2008. LNCS, vol. 5215, pp. 155–169. Springer, Heidelberg (2008)
4. Chattopadhyay, S., Roychoudhury, A., Mitra, T.: Modeling shared cache and bus in multi-cores for timing analysis. In: 13th International Workshop on Software Compilers for Embedded Systems (SCOPES 2010), pp. 1–10. ACM (2010)
5. Chaudron, J.-B., Saussié, D., Siron, P., Adelantado, M.: Real time aircraft simulation using HLA standard - an overview. In: Proceedings of the First Simulation in Aerospace Conference - Toulouse, France (April 2011)
6. Curic, A.: Implementing Lustre Programs on Distributed Platforms with Real-Time Constraints. PhD thesis, Université Joseph Fourier, Grenoble (2005)
7. Ernits, J.: Memory arbiter synthesis and verification for a radar memory interface card. Nordic J. of Computing 12, 68–88 (2005)
8. Forget, J., Boniol, F., Lesens, D., Pagetti, C.: A real-time architecture design language for multi-rate embedded control systems. In: SAC, pp. 527–534. ACM (2010)
9. Freescale. e600 PowerPC - Reference Manual (2006)
10. Freescale. MPC8641D: Integrated host processor family reference manual (2008)
11. Gries, M.: Modeling a memory subsystem with petri nets: A case study. Hardware Design and Petri Nets, 291–310 (2000)
12. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous dataflow programming language lustre. Proceedings of the IEEE 79(9), 1305–1320 (1991)
13. Heckmann, R., Ferdinand, C.: White paper: WCET prediction by static program analysis (2009)
14. Liu, I., Reineke, J., Lee, E.A.: A PRET architecture supporting concurrent programs with composable timing properties. In: 44th Asilomar Conference on Signals, Systems, and Computers (November 2010)
15. T. Mathworks, Simulink: User's Guide
16. Metzlaff, S., Mische, J., Ungerer, T.: A real-time capable many-core model. In: Proceedings of the 32nd IEEE Real-Time Systems Symposium (RTSS 2011), Session Work in Progress (2011)
17. Pellizzoni, R., Betti, E., Bak, S., Yao, G., Criswell, J., Caccamo, M., Kegley, R.: A predictable execution model for COTS-based embedded systems. In: 17th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2011 (2011)
18. Rixner, S., Dally, W.J., Kapasi, U.J., Mattson, P.R., Owens, J.D.: Memory access scheduling. In: 27th International Symposium on Computer Architecture (ISCA 2000), pp. 128–138 (2000)
19. Rochange, C., Sainrat, P.: A Context-Parameterized Model for Static Analysis of Execution Times. Transactions on High-Performance Embedded Architecture and Compilation 2(3), 109–128 (2007)
20. Schranzhofer, A., Chen, J.-J., Thiele, L.: Timing analysis for TDMA arbitration in resource sharing systems. In: 16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2010), Stockholm, Sweden (2010)

21. Skillicorn, D.B., Talia, D.: Models and languages for parallel computation. ACM Computing Surveys 30, 123–169 (1998)
22. Ungerrer, T., Cazorla, F.J., Sainrat, P., Bernat, G., Petrov, Z., Cassé, H., Rochange, C., Quinones, E., Uhrig, S., Gerdes, M., Guliashvili, I., Houston, M., Kluge, F., Metzlaff, S., Mische, J., Paolieri, M., Wolf, J.: MERASA: Multi-core execution of hard real-time applications supporting analysability. IEEE Micro. 30(5), 66–75 (2010)
23. Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., Stenström, P.: The worst-case execution-time problem - overview of methods and survey of tools. ACM Trans. Embed. Comput. Syst. 7, 36:1–36:53 (2008)