Modeling Cache Coherence to Expose Interference

Nathanaël Sensfelder
ONERA, Toulouse, France

Julien Brunel
ONERA, Toulouse, France

Claire Pagetti
ONERA, Toulouse, France

Abstract

To facilitate programming, most multi-core processors feature automated mechanisms maintaining coherence between each core's cache. These mechanisms introduce interference, that is, delays caused by concurrent access to a shared resource. This type of interference is hard to predict, leading to the mechanisms being shunned by real-time system designers, at the cost of potential benefits in both running time and system complexity.

We believe that formal methods can provide the means to ensure that the effects of this interference are properly exposed and mitigated. Consequently, this paper proposes a nascent framework relying on timed automata to model and analyze the interference caused by cache coherence.

2012 ACM Subject Classification Computer systems organization → Multicore architectures; Computer systems organization → Real-time systems

Keywords and phrases Real-time systems, multi-core processor, cache coherence, formal methods

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2019.18

Supplement Material ECRTS 2019 Artifact Evaluation approved artifact available at
`https://dx.doi.org/10.4230/DARTS.5.1.7`
`https://github.com/nsensfel/phylog-cache-coherence/`

Acknowledgements We would like to thank Mamoun Filali-Amine (IRIT-CNRS) for his helpful insights on how to validate our model and related works suggestions.

1 Introduction

The next generation of aircrafts will embed multi-core processors. Indeed, it will be more and more difficult to find mono-core processors on the market and, when correctly programmed, multi-core processors offer huge opportunities to reduce the amount of equipment required to host multiple applications compared to *federated* or single-core IMA (*Integrated Modular Avionics*) architectures. However, multi-core processors come with several drawbacks, among which is the lack of predictability [26, 27], one of the key elements of certification expectations. This lack of predictability is caused by *interference*, a delay inherent to the concurrent access to a shared resource.

Cache Coherence. In most multi-core processors, each core has its own cache memory, of which it is virtually the sole accessor. A *cache coherence* protocol ensures that:

- At any given time, a memory location can either be accessed by a single cache controller, in which case both writing and reading are allowed, or by any number of cache controllers, in which case only reading is allowed.
- Any copy of a memory location held in a cache has the most up-to-date value.

31st Euromicro Conference on Real-Time Systems (ECRTS 2019).
Editor: Sophie Quinton; Article No. 18; pp. 18:1–18:22



Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Maintaining this cache coherence requires exchanges of information between cache memories. These exchanges can be the source of a large amount of additional traffic, a potential hindrance that we qualify of *implicit* interference, because of how difficult to predict they are. Additionally, it can result in the removal of elements from the cache, which may lead to time consuming communications with the system's memory (*cache misses*).

While multi-core processors feature hardware to efficiently and automatically handle cache coherence, the black-box nature of commercial processors leads to a lack of control, visibility, and predictability of the cache coherence protocol and, by extension, of the delays it may create.

Current Research Practices. Several approaches have been developed in the literature to deal with the interference found in multi-core processors. The main solutions to ensure predictability are 1) preventing any kind of uncontrolled interference (e.g. run-time services [15, 28]); 2) enforcing a unique access to any shared resource at any time, so as to be equivalent to a single core situation (e.g. execution models [18, 5, 12]). Because its interference is difficult to predict, most of the considered hardware do not have or use automatic cache coherency. Instead, the burden of cache management is placed on the developers, forcing an application-specific solution (e.g. *scratchpad memory* [25, 22]). Such solutions prevent the gains in performance that would otherwise be provided by automatic hardware cache management mechanisms.

Contributions. We believe that the implicit interference generated by the cache coherence can be exposed and taken into account to achieve predictable programming of a multi-core processor. In this work, we focus on exposing these unexpected delays, the analysis of a formal model of the processor.

We start this paper by going into more details on how cache coherence can be achieved (Section 2), the type of system we are interested in (Section 3), and the categories of interference it can host (Section 4). We then present the tools that we use to model and analyze it (Section 5). Afterwards, we explain our choices in how we modeled the cache coherence in a multi-core processor (Section 6). Finally, we showcase some of the results that can be extracted from our model (Section 7), before listing some related works (Section 8) and concluding (Section 9).

2 Cache Coherence Protocols

We start by introducing archetypal systems on which coherence protocols run. We then present how those protocols behave.

A number of components (see Figure 1) are involved in the coherence.

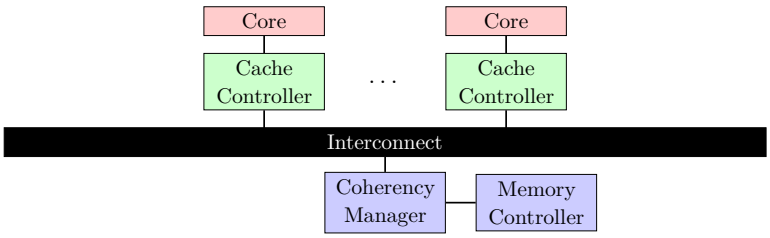

Figure 1 Components involved in cache coherence.

Memory Element: The main memory is composed of chunks (or memory elements) which have a fixed size and contain multiple addressable elements. Reading/writing from/to an address in the main memory actually corresponds to reading or writing a whole memory element. The distinction between an addressable space and a memory element is not relevant to cache coherence, and thus, for simplification purposes, this paper considers that each memory element has a single address.

Core: The component actually using and modifying memory element values. Instead of accessing the original memory elements through the interconnect, each core is linked to its own private cache. The content of this cache is managed by an associated cache controller. The core can ask its cache controller for the value held by the memory element at a given address through a `load` request. It can also send a `store` request to modify this value. Additionally, the core can issue an `evict` request, which tells its cache controller to invalidate a memory element copy. While it is rare for cores to be the initiators of `evict` requests, it remains a possibility (e.g. for micro-optimization). Cores can be made to `stall` by their cache controller, delaying the emission of a request until the cache controller is ready to accept it.

Cache Controller: Component that handles requests from its core, potentially initiating a transaction by making a query on the interconnect. Such queries take the form of a `GetS` when asking for a read-only copy of a memory element and that of a `GetM` when asking for a read-and-write copy. Queries that indicate a new value for the memory element are done through `PutM` messages. Depending on the protocol, variants of these messages may be used. Cache controllers are also able to reply to the query of another cache controller with a data reply (`data`). Additionally, cache controllers may initiate `evict` requests on themselves to make space for new memory element copies. These self-requests are controlled by a *cache replacement policy*, which is most commonly a speed-over-accuracy variation on the *Least Recently Used* policy.

Coherency Manager: Component that stores information on the state of the cache controllers, to help maintain the cache coherence. Using this stored information, it can tell if a query should be answered by the memory controller or not. This component is very much dependent on which protocol is being implemented, and can range from being a simple link between the cache controllers to actually being multiple separate components (e.g. all directory nodes of a directory-based cache coherence protocol). It is usually found inside the interconnect.

Memory Controller: Component that handles the modification or copy of the original memory elements.

Interconnect: Component that regulates and handles the propagation of messages between cache controllers, memory controllers, and the coherency manager.

▶ **Definition 1** (Request, Message, Query, and Data Reply)**.** *To keep things separate, we use the term request when talking about communications between a core and its cache controller, and the term message when talking about communications that use the interconnect. As such, queries (e.g.* `GetM`*,* `GetS`*,* `PutM`*) are messages, and so are data replies (e.g.* `data`*). Thus, messages = queries ∪ data replies.*

▶ **Definition 2** (Transaction)**.** *A transaction is composed of a query and of all the data messages the completion of that query requires.*

Each message transiting through the interconnect, and each cache controller query, is about a specific memory element. Upon receiving either one of those, cache controllers look up the state they associate with their copy of the memory element for this address, and act according to the cache coherence protocol.

2.1 Protocols

Most cache coherence protocols are based on the MSI protocol, named after the states given to copies of the memory elements by the cache controllers. M stands for Modified, the state a cache controller gives its copy of the memory element to indicate that it has both read-and-write access to the original. S stands for Shared, and is the equivalent for read-only access. I stands for Invalid, when a cache controller does not currently have a copy.

MSI-based protocols are all categorized as *Write-Back*, because caches may contain a more up-to-date value of the memory element than the RAM.

The aforementioned protocols are referred to according to their states and general idea, however, the definition of their behavior depends on the system they are implemented on.

There are two main families of cache coherence implementation: snooping-based and directory-based. When using a snooping-based protocol, cache controller queries are broadcasted to all cache controllers and to the coherency manager. The protocol also ensures that only one of the components answers the query. This answer is not broadcasted, but is instead only meant for the query's originator. For such protocols to properly function, all the components have to receive the queries in the same order. In the sequel, we only take into consideration snooping-based protocols.

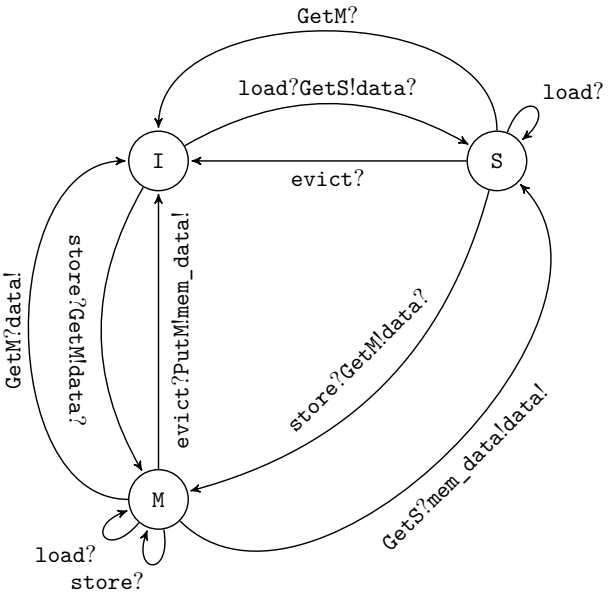

Figure 2 Generic MSI Cache Controller.

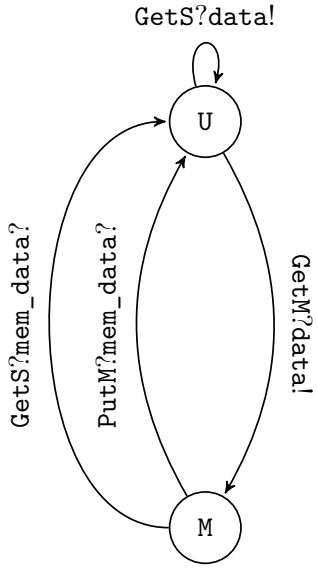

Figure 3 Generic MSI Coherency Manager.

Automata describing a generic snooping-based MSI protocol can be seen in Figure 2 and Figure 3. Figure 2 shows how the state given to a memory element's copy evolves when receiving a request (`store`?, `load`?, or `evict`?), or a query (`GetM`? or `GetS`?). Data exchanges between cache controllers are also represented (`data`! and `data`?). Cache controllers do not differentiate between data sent from another cache controller and data sent from the memory controller (both use `data`?). Sending data *to* the memory controller, however, is marked as `mem_data`!. Figure 3 represents the coherency manager, which keeps track of whether the memory has the most up-to-date value for a memory element (state `U`) or not (state `M`).

This particular protocol considers that cache controllers delay incoming requests until they are able to use the interconnect, and that transactions cannot take place simultaneously.

These automata actually describe a generic snooping-based MSI protocol. They feature *macro-transitions* (a succession of atomic transitions). The next section presents a more detailed protocol.

3 MSI Snooping-Based Protocol

3.1 A Few Caveats

These are the hypotheses made on the targeted hardware. Placing such hypotheses (or lack thereof) is necessary to properly define the targeted cache coherence protocol.

▶ **Hypothesis 1** (Non-Atomic Requests). *Cores are able to issue* `load`*,* `store`*, and* `evict` *requests to their cache controller regardless of whether the cache controller is currently able to initiate a transaction on the interconnect. In this paper, we consider that this is implemented through the use of a FIFO queue between each cache controller and the interconnect.*

▶ **Hypothesis 2** (Unique Interconnect). *The interconnect is unique. As a result, all cache controllers are able to see all transactions, and those transactions are all seen in the same order. Examples of excluded hardware include many-core processors, which feature a Network-On-Chip.*

▶ **Hypothesis 3** (Split-Transaction Interconnect). *The interconnect supports simultaneous transfer of data and queries and allows multiple transactions to take place simultaneously.*

3.2 From Abstract to Concrete Behaviour

In Subsection 2.1, we have seen automata using macro-transitions to describe a generic snooping-based MSI protocol. Let us now look in details at what is composing the transition from I to S with the `load?GetS!data?` label. Let us consider two cache controllers, CC_0 and CC_1, each of which is driven by its own core (CU_0 and CU_1, respectively) and a memory element. Let us assume that, while CC_1 already has read-only access to that memory element, CC_0 does not, and that core CU_0 issues a `load` request to acquire it.

In the sequence diagram of Figure 4, we see the behavior of all components involved. Once the core issues the `load` request, the cache controller generates a `GetS` query to the interconnect. The latter broadcasts the `GetS` to all cache controllers, including the query's originator, and the coherency manager. As the owner of the data is the memory, the coherency manager transmits that query to the memory controller, which, in turn, sends the data to the core CU_0.

In order to expose the interference, we need to model the atomic transitions and intermediate states, such as the ones shown in the figure.

3.3 Detailed Snooping-Based MSI Protocol

Instead of representing the full automaton as a graph, we use a matrix representation (see Figure 6). The first column details every possible states. As in [20], the naming of each state is determined by the following reasoning: Invalid (`I`), Shared (`S`), and Modified (`M`) are the three *stable* states of the `MSI` protocol. The other states are *transient*. Reception of a request that requires use of the interconnect will usually lead to a $\texttt{XY}^{\texttt{BD}}$ transient state, which means that the cache controller is handling a transition between the stable states `X` and `Y`,

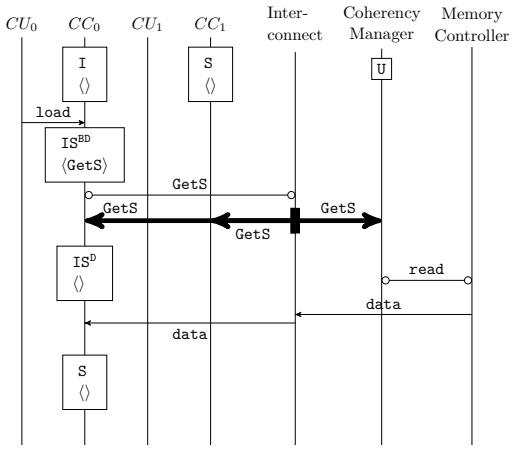

Figure 4 `load` request.

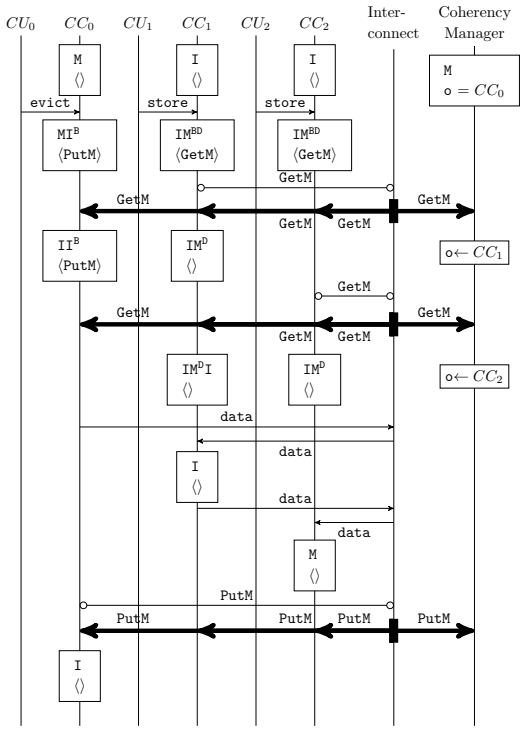

Figure 5 Double `store`.

with ($^{\mathrm{B}}$) indicating that this transition requires the acquisition of the interconnect and ($^{\mathrm{D}}$) the reception of a related data reply (whether it comes from an other cache controller or the memory). This can be followed by `XY`$^{\mathrm{D}}$ if the cache controller sees its own query before receiving a reply, or `XY`$^{\mathrm{B}}$ if a reply is received before the query is processed. This happens when, despite processing all queries in the same order, not all cache controllers take the same time to do so. Another possibility is for an external query to be received when in the `XY`$^{\mathrm{D}}$ state. Indeed, at that point, the system pretty much considers that the cache controller is in the `Y` state and thus has the responsibilities that the `Y` would require. This makes it possible for a cache controller to see a query it needs to act upon before being actually ready to do so (e.g. observing a `GetM` query while waiting for `data`). These states have a `XY`$^{\mathrm{D}}$`A` form (which means that when all is handled, the cache controller ends up in the `A` state), or `XY`$^{\mathrm{D}}$`AB` (which ends up leading to the `B` state). As it may be that the required action is to reply to said query, it is sometimes necessary to remember the originator of the query. This is marked as `r`←`s`.

When the core makes a request (`load`, `store` or `evict`), the second macro-column indicates how the cache controller behaves. The `a`/`b` notation denotes the emission of an `a` message on the interconnect, followed by a transition of the memory element copy's state to `b`. If you look at the `load` from the `I` state, the cell indicates that the `GetS` request will be generated and the reached state is `IS`$^{\mathrm{BD}}$. We recognize the beginning of the sequence diagram described in Figure 4. Grayed out cells indicate situations that cannot occur in the protocol, due to our hypotheses.

The third macro-column (named *Interconnect access*) indicates what happens when the previously queued query is broadcasted on the interconnect. When in the `IS`$^{\mathrm{BD}}$ state, we know that, at some point, our previously queued `GetS` query is going to be broadcasted

State	Core Request			Interconnect Access	Data Reply	Received Queries		
	load	store	evict			GetS	GetM	PutM
I	GetS/IS^{BD}	GetM/IM^{BD}				-	-	-
IS^{BD}	stall	stall	stall	-/IS^{D}	-/IS^{B}	-	-	-
IS^{B}	stall	stall	stall	-/S		-	-	
IS^{D}	stall	stall	stall		-/S	-	-/IS^{D}I	
IS^{D}I	stall	stall	stall		-/I	-	-	
IM^{BD}	stall	stall	stall	-/IM^{D}	-/IM^{B}	-	-	-
IM^{B}	stall	stall	stall	-/M		-	-	-
IM^{D}	stall	stall	stall		-/M	r←s -/IM^{D}S	r←s -/IM^{D}I	
IM^{D}I	stall	stall	stall		r!data -/I	-	-	
IM^{D}S	stall	stall	stall		r!data m!data -/I	-	-/IM^{D}SI	
IM^{D}SI	stall	stall	stall		r!data m!data -/I	-	-	
S	hit	GetM/SM^{BD}	-/I			-	-/I	
SM^{BD}	hit	stall	stall	-/SM^{D}	-/SM^{B}	-	-/IM^{BD}	
SM^{B}	hit	stall	stall	-/M		-	-/IM^{B}	
SM^{D}	hit	stall	stall		-/M	r←s -/SM^{D}S	r←s -/SM^{D}I	
SM^{D}I	hit	stall	stall		r!data -/I	-	-	
SM^{D}S	hit	stall	stall		r!data m!data -/S	-	-/SM^{D}SI	
SM^{D}SI	hit	stall	stall		r!data m!data -/I	-	-	
M	hit	hit	PutM/MI^{B}			m!data s!data -/S	s!data -/I	
MI^{B}	hit	hit	stall	m!data -/I		m!data s!data -/II^{B}	s!data -/II^{B}	
II^{B}	stall	stall	stall	-/I		-	-	-
	Handling Requests					Handling Queries		

Figure 6 Cache Controller Memory Element State Changes (adapted from [20]).

on the interconnect. This will result in reaching the IS^{D} state. As a side note, if the core makes a second `load` request on the same memory element while the copy is IS^{BD}, that new request is stalled.

The fourth macro-column describes the behavior upon reception of a data reply.

The fifth macro-column (named *Received Queries*) defines the behavior of the cache controller when snooping a transiting query that is not its own (which would otherwise pertain to the third macro-column). For instance, from state `S`, when snooping a `GetS`, the cache controller does not do anything, as can be seen with core CU_1 in the sequence diagram of Figure 4.

Replying with a message `d`, meant for `t` (`t` = `m` when sending to the memory controller and the coherency manager, `t` = `s` when sending to the cache controller that initiated the transaction, and `t` = `r` when sending to the initiator of an earlier query) is written as `t!d`.

▶ **Example 3.** *Let us have a look at a more complex behavior: when 2 cores attempt modification of the same memory element. This is illustrated in the sequence diagram of Figure 5.* CC_0 *starts with read-and-write access to the memory element (its copy being in the* `M` *state), neither* CC_1 *nor* CC_2 *have a copy (state* `I`*), and the coherency manager knows that its value is out of date (state* `M`*).*

The sequence starts when CU_0 *issues an* `evict` *request and both* CU_1 *and* CU_2 *issue a* `store` *request.* CC_0 *receives the* `evict` *request, queues a* `PutM` *query and now considers the memory element to be* MI^B *(that is, "was Modified, will be Invalid once access to the interconnect is granted"). On the other hand, the other two caches receive their* `store` *requests, queue a* `GetM` *query, and now consider the memory element to be* IM^{BD} *("was Invalid, will be Modified after access to the interconnect and reception of a* `data` *reply").*

All the cache controllers want to access the interconnect. The internal behavior of the interconnect will drive this choice. Most of the time, the interconnect is based on Fair-RR (Round Robin) [11]. In this scenario, the interconnect first broadcasts the `GetM` *query from* CC_1*'s queue, which is now empty.*

CC_1*, seeing its own query, confirms that it has accessed the interconnect, and switches to the* IM^D *state to await a* `data` *reply. The coherency manager ignores the query. Seeing* CC_1*'s* `GetM` *query passing through the interconnect,* CC_0 *has to reply with a* `data` *message (this corresponds to* `s!data` *in the protocol definition), containing its value for the memory element, and to transition to the* II^B *state.*

CC_2*'s* `GetM` *query is broadcasted. As it is about to receive the data with read-and-write access,* CC_1 *is the component that should reply to* CC_2*'s query. Not having the data yet,* CC_1 *is currently unable to do so. Instead, it transitions to the* IM^DI*, remembering that it should send the data to* CC_2 *as soon as possible.*

Finally receiving the data, CC_1 *applies completes* CU_1*'s request, sends the updated data to* CC_2 *and transitions to the* `I` *state (as* CC_2 *wants read-and-write access).*

CC_2 *receives the data and completes* CU_2*'s request.*

CC_0*'s* `PutM` *is broadcasted, but has been superseded by a previous* `GetM` *and thus causes no reaction in the other cache controllers or the coherency manager.* CC_0 *transitions to* `I`*, completing its core's request.*

3.4 Coherency Manager

State	Received Queries				Data Reply
	GetS	GetM	PutM (Owner)	PutM (Other)	data
U	s!data	s!data o←s -/M		-	
U^D	stall	stall	stall	-	-/U
U^B	o← ∅ -/U	-	o← ∅ -/U	-	
M	o← ∅ -/U^D	o←s	o← ∅ -/U^D	-	-/U^B

Figure 7 Coherency Manager Memory Element State Changes (adapted from [20]).

Figure 7 shows how the coherency manager keeps track of whether the RAM has the most up-to-date value for a memory element (state `U`) or if a cache controller does (state `M`). This is used to determine if the RAM should be the one to reply when either a `GetS` or a `GetM` query passes through the interconnect. The `U` state indicates that the RAM currently

has the most up-to-date value. The U^D state indicates that the RAM should be the one to respond to queries, but it still hasn't received the latest value. Unlike the cache controller, it will not switch to a dedicated state but instead force queries from the interconnect to stall until the problematic query can be fulfilled. U^B indicates that the RAM has received the latest value, but has not yet seen the query that led this data to be sent.

The exact cache controller currently in charge of the memory element is kept track of. Change of ownership are marked as o←s (the query originator becomes the new owner) and o← ∅ (there is no longer an owner, meaning that the RAM is currently responsible for it).

▶ **Example 4.** *Back to the sequence diagram of Figure 5 and to Example 3, let us observe the behavior of the coherency manager. The coherency manager reacts to each* `GetM` *query, updating its internal state to reflect the change of ownership. Thus, the coherency manager starts by considering that CC_0 is the only one to have a valid (i.e. up-to-date) value of the memory element, then, upon seeing the first* `GetM`*, considers CC_1 to be responsible for it (o←s in the table). As a result, at the end of the execution, the coherency manager knows that the* `PutM` *query is originating from a cache controller that is not currently responsible for that memory element and can thus safely ignore it.*

4 Interference

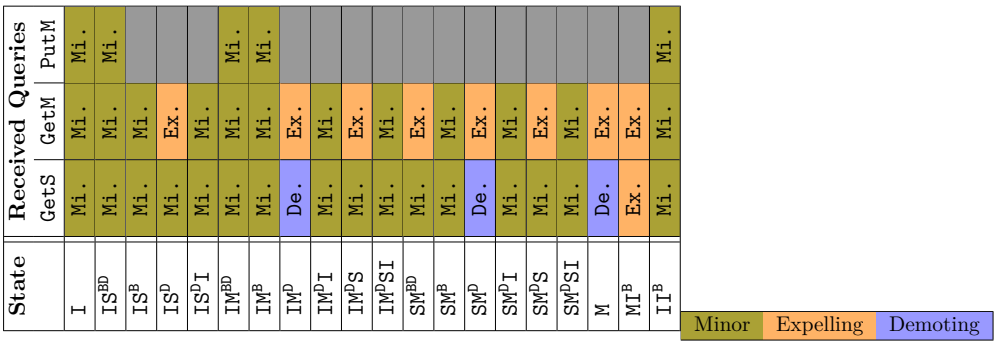

Figure 8 Occurrences of Interference.

Let us now categorize how a cache controller may be negatively affected by the actions of another. Figure 8 summarizes the occurrences of each interference category. In the Figures 9, 11, and 10, the dark gray area indicates when the cache controller is unavailable due to having to handle the incoming query (deciding how to act and, potentially, updating its internal state), and the light gray area shows when its core's next request for that memory element may be negatively impacted by the change of state.

▶ **Definition 5** (Minor Interference). *Cache controllers have actions to perform upon receiving any type of request. Because of this, every time a cache controller has to deal with an incoming query, there is a very small amount of time during which it cannot be used by its core. We call this unavailability period minor interference. And, while the effect of each minor interference is so small as to be considered negligible, their accumulation most definitely is not. Indeed, minor interferences are one of the main motivations behind the use of a directory-based coherency protocol (in which minor interferences are only experienced by cache controllers likely to have a use for that query) over a snooping-based one (in which* all *cache controllers are affected by every query).*

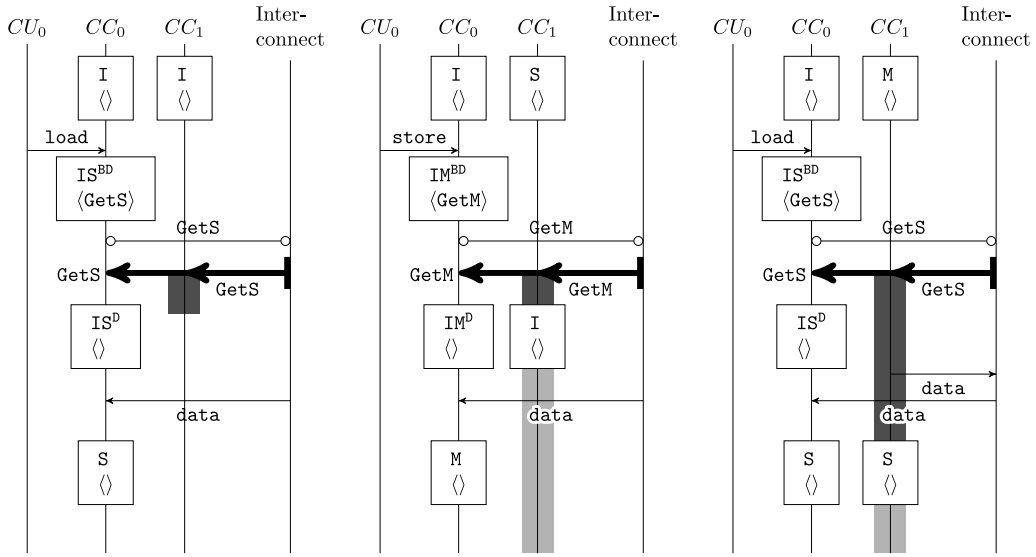

Figure 9 Minor. **Figure 10** Expelling. **Figure 11** Demoting.

Figure 9 shows an example of minor interference: the CC_1 cache controller has to process the `GetS` broadcast, despite that message not requiring any reply or internal state update from CC_1.

▶ **Definition 6** (Expelling Interference). *To maintain the principles of cache coherency, it may be required for a cache controller to dispense of its copy of a memory element, relinquishing its access rights. This is caused by another cache controller demanding read-and-write access to that memory element (a `GetM` query). We have, however, marked the reception of a `GetS` query for an element in the MI^B as being an expelling interference in Figure 8. It could be argued that reaching the MI^B indicates that the cache controller is already in the process of evicting its copy of the memory element. But, as the MI^B state allows immediate (i.e. `hit`) access for both writing and reading that memory element, we still consider this event to have a negative impact.*

Figure 10 shows an example of expelling interference: the CC_1 cache controller, receiving a demand for read-and-write access, is forced to relinquish its read-only copy.

▶ **Definition 7** (Demoting Interference). *Another type of interference is the demoting interference, in which a cache controller has to abandon its writing access rights to a memory element, while retaining its reading access.*

Figure 11 shows an example of demoting interference: the CC_1 cache controller, receiving a demand for read access on that memory element, has to update the value from the main memory and go from read-and-write access to read-only access.

5 Formal Modeling of Real-Time Systems with Timed Automata

To expose the interference presented in the previous section, we chose to use formal methods. More precisely, we are relying on timed automata [1] to model and analyze our system.

A *timed automaton* is an extended automaton with variables and clocks. During the system's execution, the state of timed automaton is defined as a location, the value of its integer variables and of its clocks. The evolution of these integer variables is controlled

by the automaton's transitions, whereas all of the system's clocks progress at the same rate, following the passing of time. To indicate that a location should be left immediately, UPPAAL [4] offers the following location modifiers:

Urgent: The location must be left before any time passes.

Committed: The location must be left before any time passes, and the next transition must originate from a **committed** location.

Invariant ϕ: The location is defined only if a linear constraint ϕ holds true. ϕ may reason over the automaton's integer variables, clocks, or both.

The automata transitions are composed of the following:

Guard: Prerequisite (linear constraint) for this transition to be able to *fire*. The condition uses the automaton's integer variables, clocks, or both.

Synchronization: Allows to have more than one automaton transitioning during a step, by synchronizing multiple transitions over a channel. The channel can be used in either receiver (with a `?` suffix) or sender (with a `!` suffix) mode. On a channel that was declared without modifier, the transition requires exactly two automata to synchronize during this step: the sender, and the receiver. It is also possible for a channel to have been declared as a **broadcast** channel, in which case the sender synchronizes with all available receivers. Furthermore, the channel may have been declared as **urgent**, which prevents waiting in a location if the synchronization can occur. Finally, priorities between channels may be put in place.

Update: Sequence of instructions to alter the automaton's integer variables, or reset its clocks.

Select: The transition selects the given integer variables' next value from a specified range.

Example. This subsection presents an example of UPPAAL model: a processor attempts to read a variable, which may be either in RAM or in its cache. The automaton in Figure 12 corresponds to the core, the one in Figure 13 to the RAM controller, and the remaining one (Figure 14) is used to mark a transition as urgent by having an automaton always ready to synchronize on a dedicated urgent channel (`FORCE_URGENT`). In this model, the `FORCE_URGENT` and `READ_LINE` channels are both declared as urgent.

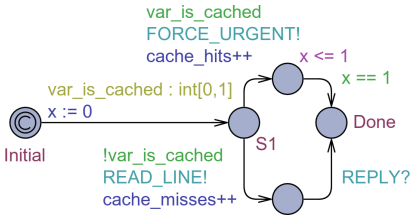

Figure 12 Core and cache.

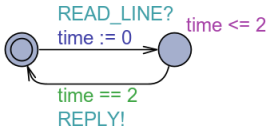

Figure 13 RAM.

Figure 14 Urgence.

The Core Automaton (Figure 12). Its initial location is marked as committed, meaning that it is left immediately. The exiting transition sets the `x` clock to 0, and the *var_is_cached* variable to a value in the $[0, 1]$ range. The `x` clock will be used to know how long it took for the processor to get its variable. Two transitions are fireable from the *S1* location, depending on whether the targeted variable is cached or not. If it is indeed cached, the transition labeled `FORCE_URGENT` is the only one fireable and it synchronizes with the automaton of Figure 14, forcing it to be taken as soon as possible. Additionally, the transition increases an

integer variable that counts the number of times a variable was found in the cache. Taking said transition leads to a location in which the only exiting transition requires the `x` clock to equal 1 unit of time before arriving in the *Done* location.

If the variable was not in the cache, the other transition from *S1* is active and leads to a synchronization on the `READ_LINE` which is also to be taken as soon as possible. This time, however, it is possible for that synchronization to not be immediately available, as the RAM controller automaton may be handling another query and thus not be ready to synchronize as it would not be in its initial location. This also justifies not marking the location as urgent or committed: the automaton may have to wait an unknown amount of time. Once the synchronization does happen, an integer variable counting the number of times the variable was not found in the cache is incremented, then the automaton waits for the RAM automaton to synchronize on the `REPLY` channel before considering it has acquired the variable.

The RAM Controller Automaton (Figure 13). Its initial location awaits synchronization on the `READ_LINE` channel. Since `READ_LINE` is urgent, the transition happens as soon as possible. It resets the automaton's `time` clock back to 0. The synchronization leads to a location which has to be left strictly before more than 2 units of time pass, as defined by the invariant. To ensure that the automaton stays in this location for exactly 2 units of time, the only exiting transition has a guard stating just that. This transition also requires a synchronization on the `REPLY` channel before allowing a return to the automaton's initial location.

6 Model of the Cache Coherence

This sections describes the general ideas behind how we modeled the cache coherence in UPPAAL. We have released the model under an LGPL v3 license at `https://www.onera.fr/sites/default/files/598/ecrts19.zip`.

6.1 Modeling Strategy

The model contains one automaton per component present in Figure 1, an automaton in charge of synchronizing on the `FORCE_URGENT` channel (in an identical manner to the one in Figure 14), as well as message queues for both queries and data (Sub-section 6.6). Each core runs exactly one program. To change the number of cores, one simply has to add or remove cores (and associated cache controllers) and to change the value of a dedicated system-wide constant. Moreover, each component has a unique identifier, which is used both to target a specific automaton on some synchronization, and to indicate the emitter of requests and queries.

The states and transitions seen on the automata do not visibly reflect any program or protocol. This means that the stable states (`M`, `S`, `I`) and the transient states ($\texttt{IS}^{\texttt{BD}}$, $\texttt{IS}^{\texttt{D}}$, ...) will not appear explicitly. Instead, the automata's designs are focused on their synchronizations, with the logic (and state) of the protocols being held in their variables instead. As such, the same automaton can easily be used for any program or protocol (provided the hypotheses from Sub-Section 3.1 remain), only requiring small changes in the definition of the functions found in its transitions.

Priorities on synchronizations are used to reduce the number of redundant system states. For example, any transition that exits a waiting location (i.e. location in which nothing happens until a clock has reached a certain count) has a higher priority than any other type of transition.

6.2 Core

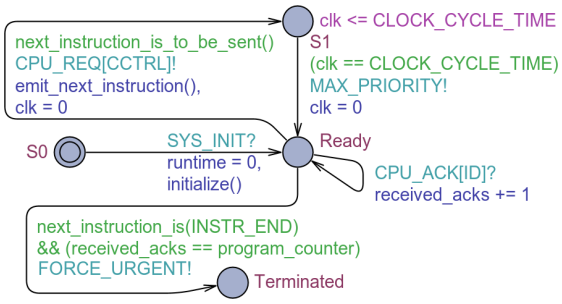

Figure 15 Model of the Core.

```
program_line_t program_0 [7] =
   {
      {INSTR_LOAD,     1},
      {INSTR_LOAD,     2},
      {INSTR_STORE,    3},
      {INSTR_LOAD,     3},
      {INSTR_STORE,    1},
      {INSTR_EVICT,    1},
      {INSTR_END,      0}
   };
```

Figure 16 Model of a Program.

Programs are modeled using arrays of address-targeting instructions, not so dissimilar to their binary executable. These arrays only contain instructions related to memory accesses (`INSTR_LOAD`, `INSTR_STORE`, `INSTR_EVICT`), and one (`INSTR_END`) to indicate that the execution of the program is completed. An example can be seen in Figure 16.

The automaton corresponding to the core is shown in Figure 15. Progress of the program's execution is tracked by the `program_counter`, which is incremented each time an instruction has been started. Another integer variable, `received_acks`, counts how many times the cache controller has confirmed that a request has been fulfilled. The sending of each instruction to the cache controller is separated by at least the time of a clock cycle.

To ensure that synchronization occurs with the right automaton, the request uses the cache controller's identifier to select a sub-channel of `CPU_REQ`. Conversely, acknowledgments are received on the sub-channel of `CPU_ACK` corresponding to the core's identifier. Upon reaching the `INSTR_END` instruction, the automaton has to wait until all of its outstanding requests have been fulfilled before being able to reach the `TERMINATED` state.

6.3 Coherency Manager and Memory Controller

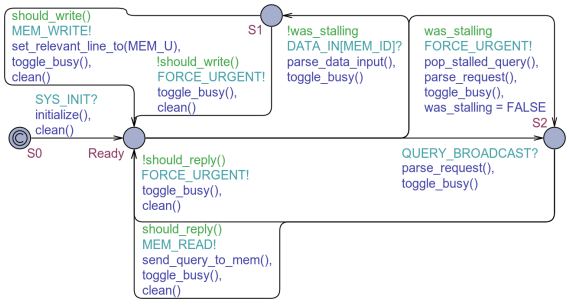

Figure 17 Model of the Coherency Manager.

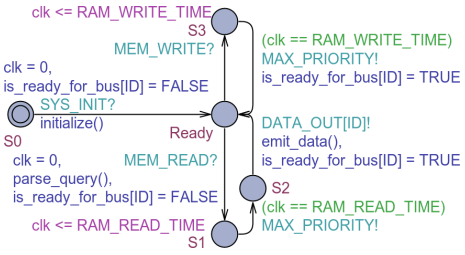

Figure 18 Model of the Memory Controller.

The timed automaton modeling the coherency manager can be seen on Figure 17. The coherency manager has to know for which memory elements the RAM copy is to be considered as superseded by a cache controller. For this purpose, it maintains an array associating a state to each memory element address. The size of this array must be able to accommodate all cache controllers having their caches full of superseding copies of memory elements. In effect, $|mem\_array| = |cache\_array| \times |caches|$.

After initializing its array with default values, the timed automaton waits for either a cache controller query or a `data` message.Receiving any of these leads to an update of the internal state associated with the related memory element, as described by the array in Figure 7.

Upon receiving a cache controller query, the update to the internal state may indicate the need to provide `data` from the RAM, leading the automaton to synchronize with the memory controller to wait for `RAM_READ_TIME` units of time before providing a reply to the query's originator. Alternatively, when receiving `data`, the automaton synchronizes with the memory controller to wait for `RAM_WRITE_TIME` units of time. The memory controller's automaton is shown in Figure 18. It has a local clock, `clk`, which is used to wait either `RAM_WRITE_TIME` or `RAM_READ_TIME`, depending on what the coherency manager demands.

6.4 Interconnect

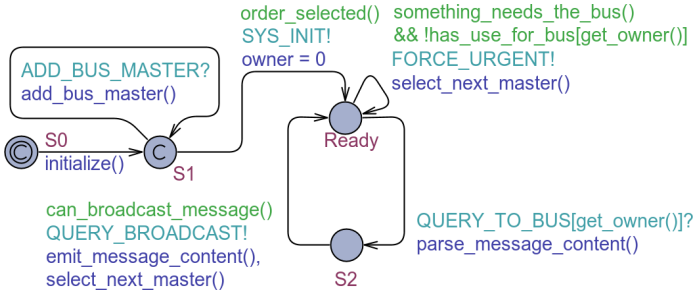

Figure 19 Model of the Interconnect.

Figure 19 shows the timed automaton for the interconnect. It starts (`S1`) by waiting for cache controllers to synchronize through the **`ADD_BUS_MASTER`** so that they can be added to the bus policy. The order in which the cache controllers make that synchronization is not deterministic. This results in all possible orders being explored when analyzing the system. Once all cache controllers have been added, the automaton proceeds and synchronizes with all the other components by broadcasting on the **`SYS_INIT`** channel.

Using a component identifier to select the appropriate sub-channel, the interconnect awaits either an incoming cache controller query, or a notice that the cache controller does not have any to send (`Ready`). If the latter happens, the access policy is followed to determine which cache controller should be made able to send its query (e.g. with a Fair-Round-Robin the next cache controller is chosen). With the former, the query is first received by the interconnect (`Ready`→`S2`), then, in a second transition (`S2`→`Ready`), it is broadcasted to all components that listen for cache controller queries. This broadcast is stalled if any of the components that need to receive it indicate that they are not ready to do so (e.g. because their incoming query queue is full).

6.5 Cache Controller

The automaton used to model a cache controller is rather complex. As previously stated, it does not feature any of the states found in the protocol description (e.g. the ones of the matrix in Figure 6). Instead, this automaton keeps an array that indicates the protocol state associated with a given memory element. The automaton starts by synchronizing with the interconnect so that it is taken into account by the interconnect's access policy (`S0`→`S1`). It then waits for the broadcast on the **`SYS_INIT`** channel (`S1`→`Ready`).

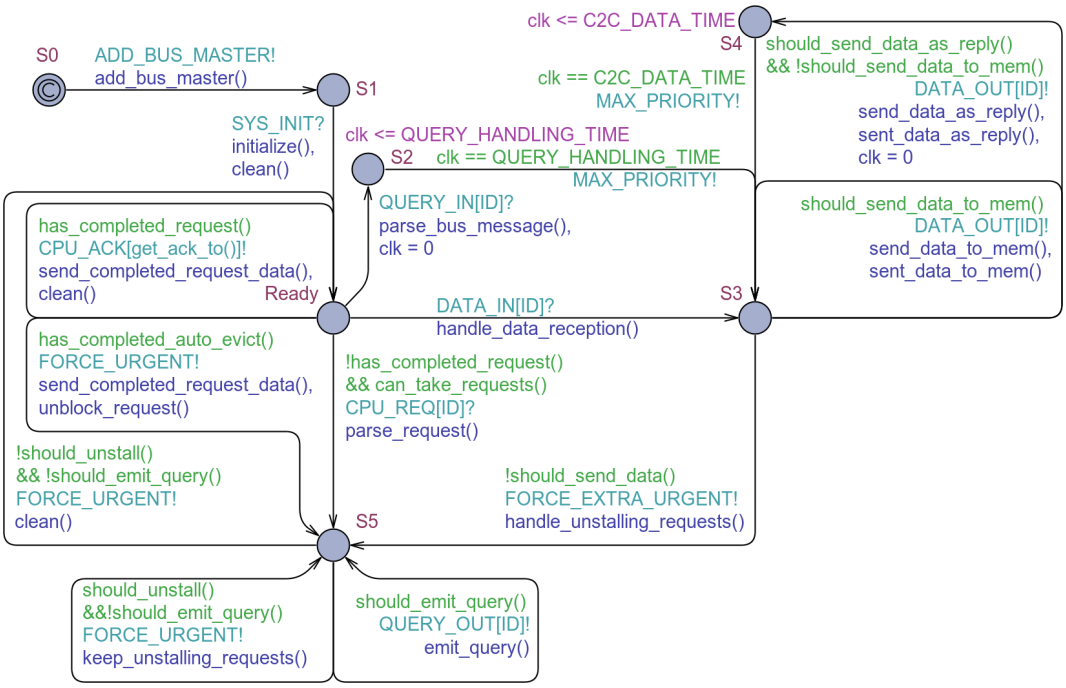

Figure 20 Model of the Cache Controller.

CPU Communications. Each cache controller has a queue of outstanding requests from its core, as well as a queue of completed requests to inform the core of. Both queues are first in, first out. Upon receiving a request from its core (middle `Ready`→`S5` transition), the cache controller attempts to find a line in its array either corresponding to the associated address, or, if none exists, one that is not currently used (*Invalid*). If no such line is found, the request is stalled, meaning that it is simply put in the outstanding requests queue for later. Otherwise, the behavior of the cache controller depends on the cache coherence protocol and the state held by the line, such as indicated in Figure 6. If the eviction policy is applicable and no line can currently accommodate the request, an automated eviction occurs. The cache controller is re-evaluated once the eviction has been completed (leftmost `Ready`→`S5` transition). In our model, we use an accurate LRU eviction policy, meaning that the cache controller keeps track of the order in which its cache lines have been used and will allow an automated eviction to occur if the least recently used line points to a state for which the protocol does not indicate stall in case of `evict` request.

There are two possible reasons for a request to be acted upon: it is an incoming request from a core, or it is a previously stalled request on a memory element which just changed state.

hit: the request is moved to the completed requests queue. The handling of stalled requests continues. This also counts as a use of the line according to the eviction policy, if the request is not an `evict`.

stall: the request is put in the outstanding requests queue, if it is not already there. The handling of stalled requests is stopped.

***msg*/*state*:** the state of the line is set to *state*, the request is put in the outstanding requests queue, if it is not already there. If this is encountered during the un-stalling of requests, the request is re-evaluated. In the latter case, this counts as a use of the line according to the eviction policy, if the request is not an `evict`.

Interconnect Communications. Handling of pending incoming queries is done through the `Ready`→`S2`→`S3` transitions. This updates the internal state of the cache following what was indicated by Figure 6 and has a waiting period that accounts for the simulated query handling time period. Handling of pending incoming data is similar (`Ready`→`S3`). The `S3` location is where data emission is handled. Data can be sent to either memory or another cache controller (the latter introducing yet another delay). This data is actually sent to a FIFO queue and not to the other components directly. When there is no data to send, the `S3`→`S5` transition evaluates the impact the changes had on the currently stalled core requests.

6.6 Message Queues

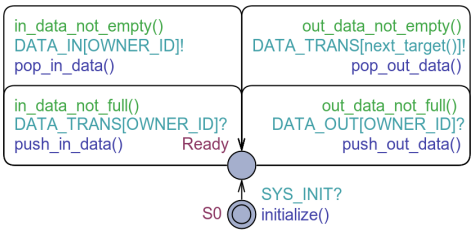

Figure 21 Model of the Data FIFOs.

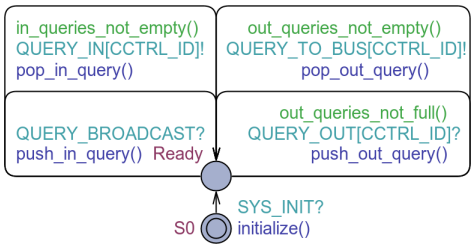

Figure 22 Model of the Query FIFOs.

Access to the bus is done through message queues. We use separate automata for data and query queues to avoid over-encumbering the automata that use them (we would otherwise need to add their transitions to nearly all the locations of the cache controller automaton). These automata actually handle both an incoming and outgoing queue. Each cache controller has a dedicated instance of both automata. The memory controller has an instance of the data queues automaton.

The data and query queues automata are fairly straightforward, having one transition to take and one transition to push items in either direction. However, the actual condition for incoming queries to be allowed in is hidden behind a shared variable. Indeed, the queries come from broadcasts made by the bus and UPPAAL does not allow conditions on transitions receiving from a broadcast channel. Thus, the condition of having all query message queues ready to receive is actually handled on the side of the interconnect.

7 Checking Properties

UPPAAL lets users check if their model verify properties. These properties can be used to know if *at least one* (`E`) or *all* (`A`) execution paths *always* ($\Box$) or *at least once* ($\Diamond$) verify a given formula over the automata's clocks, integer variables, or location. In addition, UPPAAL has an operator that looks for the highest value reachable by an automaton's clock or integer variable.

For example, taking the system from Section 5, with two CPUs ($C0$ and $C1$), we can know if both processors always end up getting their variable (all paths lead to both automata reaching the *Done* location, $A\Diamond(C0.\text{Done} \;\&\&\; C1.\text{Done})$), or the longest time it would take for one of them to do so (what is the maximum value the clock can reach before the automaton reaches its *Done* location, `sup{not` $C0$.Done`}:` $C0$.`x`).

```
program_line_t program_200 [11] =
   {
      {INSTR_STORE,   1},
      {INSTR_STORE,   2},
      {INSTR_LOAD,    1},
      {INSTR_STORE,   1},

      {INSTR_LOAD,    2},
      {INSTR_STORE,   2},
      {INSTR_LOAD,    1},
      {INSTR_STORE,   1},

      {INSTR_LOAD,    2},
      {INSTR_STORE,   2},
      {INSTR_END,     0}
   };
```

Figure 23 Program Model 200.

```
program_line_t program_201 [11] =
   {
      {INSTR_STORE,   3},
      {INSTR_STORE,   4},
      {INSTR_LOAD,    3},
      {INSTR_STORE,   3},

      {INSTR_LOAD,    4},
      {INSTR_STORE,   4},
      {INSTR_LOAD,    3},
      {INSTR_STORE,   3},

      {INSTR_LOAD,    4},
      {INSTR_STORE,   4},
      {INSTR_END,     0}
   };
```

Figure 24 Program Model 201.

7.1 Exposing Interference

Using such properties, we are able to expose the interference in a number of fashions. The example we will take for showcasing them is that of a dual core on which two instances of the program modeled by Figure 23 are running.

- **Counting Hits & Misses:** An easy metric to measure is the number of cache hits and misses for each address. This can be achieved by simply looking at the state of the memory element upon reception of a core's request, and increasing the right integer variable accordingly (much like in Section 5).
  In the dual core example, this shows that each core has 2 cache hits and 3 cache misses for the first address; one core has 2 cache hits and 3 cache misses for the second address, whereas whereas the other has 1 cache hit and 4 cache misses.
- **Counting All Occurrences:** We can expose interference by counting all of its occurrences, without regards for whether it had an impact on the system's execution or not. In effect, this equates to having one integer variable per address and type of interference, and increasing the right one according to what is described in Figure 8.
  When applied to the dual core example, we can see that for the second address, both caches have 4 occurrences of minor interference, 1 occurrence of demoting interference, and 2 occurrences of expelling ones. For the first address, one cache has 4 minors, 1 demoting, and 1 expelling, whereas the other has 3 minors, no demoting, and 3 expelling.
- **Counting Meaningful Occurrences:** Another pertinent information is an account of the interference that actually has an impact on the system. Since we are already able to detect any occurrence of the interference, we simply have to isolate the occurrences which impacted the cache's completion of core's requests. To do so, each cache keeps track, for each address, of whether an interference occurred since that address was last involved in a core request. Thus, if the CPU requests a read on an address for which the expelling flag is active, we consider that a meaningful expelling interference occurred.
  Using this with the dual core example, we can see that, for the second address, both caches are affected by the effects of 1 demoting and 1 expelling interference. For the first address, one cache has the same and the other experiences the effects of 2 expelling interferences.
- **Execution Time Analysis:** A more general metric is the execution time. Indeed, we can measure the impact that cache coherence has on an application's execution time. This can be achieved by simply replacing all accesses to shared variables made by the target

application with accesses to new variables, setting the time impact of minor interferences to nil, and having the framework compute the new maximum execution time so that it can be compared to the one with shared variables left intact.
On the dual core example, we first measure the execution time with the system as is, then replace the program running on one of the two cores by Figure 24 and set the cost of minor interferences to zero. Our first analysis indicates a maximum execution time of 1602 time units, the second one indicates 1050 time units. This implies that cache coherence causes a 16 percent increase in execution time.
Alternatively, by keeping the time impact of minor interferences to its default value, a WCET of these two programs lets us deduce how much time is lost due to minor interferences. In the dual core example, the result is still 1050 time units, showing a lack of negative effects from minor interference.

7.2 Model Validation

In addition, we can assert that the behavior of our model does indeed correspond to what we expect. The successful verification of all these properties gives us a reasonable confidence in the validity of the protocol used in our model. The validation of the chosen timing parameters, however, would still require a few judicious benchmarks.

- **Programs Always Terminate:** By checking that all possible execution paths lead all cores to the `Terminated` location, we ensured that there are no deadlocks in our model.
- **No Incompatible States:** As stated in Section 1, there should never be two cache controllers simultaneously having writing access to the same memory element. Thus, we checked that if a cache controller is in a state where it may write to a memory element, then the others are not in a state where they may read that memory element.
- **Values Are Always Up-To-Date:** Another point stated in Section 1 is that the values in cache should be up-to-date. We verified that it is the case in our model by creating a version in which the exact value of each memory element is taken into account. Using a shared variable to keep track of the expected system-wide value, we tested that every time an action (either read or write) was taken on a memory element, it the local copy of that memory element had a value equal to the system-wide one. This is a standard property to validate coherency protocol [10, 19].

8 Related Works

- **WCET Analysis for Single-Core:** The authors of [9] introduce *METAMOC*, a UPPAAL-based framework for modular WCET analysis of programs running on single-core processors. It transforms program binary executables into timed automata, one for each function of the program. These programs are simplified. For example, a conditional jump may be removed if it would lead to less instructions being executed. This is justified by the assumption that the more instructions there are, the longer the execution time is (the reverse of which is called a *time anomaly*). METAMOC supports instruction pipelines, which are modeled using five timed automata (*fetch*, *decode*, *execute*, *memory*, and *writeback*). These five automata have to be manually made for the targeted architecture. Caching is also supported, and requires a similar attention the architecture's specifics. As it is intended for single-core architectures, METAMOC obviously does not have any concept of cache coherence. We are, however, taking a very similar approach to tackle our problematic.

The work in [6] also shares similarities with [9], as UPPAAL is used to estimate WCET for programs running on single core processors with pipeline and cache, in what is presented as a modular framework. It attempts to improve on the weaknesses of METAMOC by replacing the value analysis based control flow graphs with program slicing. In effect, statements that do not affect dynamic jump addresses are replaced with `nop` (i.e. "do nothing") operations. In [7], they address the state explosion issue.

- **WCET Analysis for Multi-Core with Private Caches:** Readers can refer to [17] for an overview of Multi-Core WCET Analysis. [16], proposes a UPPAAL-based framework to estimate the WCET of applications running on a multi-core processor. They consider the delays caused by contention on the interconnect and a private instruction cache for each core (data caches are not considered). They perform analysis on the memory blocks pertinent to the instructions of the program. A memory block may contain one or more instruction. For each instruction, they are only interested in whether it: is always found in the cache; is always found except on the first access; is never found in the cache; is undecided. They have defined a timed automaton to model each of these possibilities (modeling the need for interconnect access, time to read the memory blocks, and updates to the cache). They consider programs as control flow graphs in which each node is a memory block. As such, they model each program by a single timed automaton based on the control flow graph, but in which each instruction has been replaced by one of the aforementioned timed automata corresponding to its impact. Their paper presents models for two types of interconnects: TDMA and FCFS, which control the order the bus can be accessed by the timed automata modeling the instructions. Cache coherence is not addressed.
- **WCET Analysis for Multi-Core with Shared Caches:** The authors of [8] focus on the estimation of WCET on multi-core processors. Their point of interests are the delays caused by hierarchical caches, the use of a shared cache, and the interconnect. They do not use UPPAAL, but instead model the applications as task-dependency graphs and perform computations to estimate the WCET. Their approach starts by analyzing how the L1 caches are accessed, to remove elements that are sure to always be present from further consideration. The other accesses are dependent on both the content of the L2 cache, and access to the bus. The content of the L2 cache depends on which tasks are running, which in turns, depends on bus access time access. To resolve the circular dependency, they propose an iterative approach: starting by considering the worse case scenario in which all tasks interfere, they estimate the running time of the tasks, which lets them remove any interference between two tasks whose running time are disjoint, and start over until a fix point is reached. Data caches are not taken into account and are assumed to have no effect on the calculations. Cache coherence is not addressed.

  The authors of [29] study the impact of a shared cache (including data caches) on execution time. To do so, they represent each program as an *address flow graph*, in which edges correspond to instruction, and vertices correspond to the state of the cache and its access history. They actually build a *combined cache conflict graph*, which is pretty much the combination of each core's *address flow graph* into a single graph. Cache coherence is not addressed.

  The work done in [13] has similarities with ours, as it uses UPPAAL to calculate WCET of programs running on multi-core processors. Their focus is not on cache coherence, but it does feature some, as `write` requests lead to the invalidation of the memory element in the other caches.

- **Cache Coherence Protocol Comparison:** The authors of [2] compare the efficiency of common snooping-based cache coherence protocols. To do so, they described a multi-core processor and the cache coherence protocols in *Simula*. Much like ours, the programs running on this simulation are described as a succession of memory related instructions. However, they do not use explicit addresses for these instructions. Instead, they have defined system-wide weights to regulate the probability of an instruction to be applied on a private memory element (i.e. a memory element the cache is the sole user of) or a shared block (i.e. a memory element used by multiple blocks). Thus, cache coherence *is* addressed, but only in a very broad context. Indeed, whereas our work focuses on the impact of cache coherence on specific applications on a specific architecture, the cache coherence protocol comparison made by the authors of [2] provides a general idea of which protocol is more fitted for which type of application.
- **Predictable Cache Coherence:** An alternative to trying to predict how cache coherence is going to behave is to use a kind of cache coherence designed to be predictable. [24] lists the cache coherence related latencies that need to be known before predictability of the protocol can be achieved. Its authors argue that write-through, update-based protocols (i.e. writes are propagated to other caches and to the memory) can be made to be predictable.

  [14] presents PMSI, a variation on the MSI protocol that uses a TDM bus to achieve predictability. Emission of coherence queries and is restricted to a core's TDM slots. As a result, a cache does not suffer from interference during its own TDM slots. [21] expands on this by introducing HourGlass, which allows separate handling of critical and non-critical cores. HourGlass uses timers to allow cores to hold access to a memory element for a predefined time duration. The evaluation of queries that would remove an access currently protected a timer are delayed until its time is up. Both PMSI and HourGlass require hardware modification, which prevents them from being used in a context that relies on COTS.

9 Conclusion and Future Work

When using cache coherence, the execution of a program running on a core is affected by the execution of the programs running on the other cores. Because of this, analysis of the execution time becomes much more difficult. In this paper, we categorized the types of interference that cache coherence induces: *minor interference*, caused by the handling of queries irrelevant to the cache controller; *demoting interference*, when an external event forces the loss of writing rights; and *expelling interference*, when an external event forces eviction of a cache line.

We also presented timed automata as a way to model cache coherence so that this interference can be studied and exposed. For this purpose, we also showed and explained our current model for the analysis of cache coherence, as well as the hypotheses made for that model to be applicable.

We are also working on a tool to automatically switch which MSI variant (MESI, MOSI, MOESI, MESIF) is used by the model. We also intend to add another type of instruction to programs soon, adding more non-determinism to the model by having a `INSTR_CALC` instruction that causes the CPU to wait for any amount of time in a given range. Lastly, we have planned to perform a benchmark comparison on the Keystone TCI6630K2L [23] from Texas Instruments to further validate our approach.

Our current model was tested with up to 6 cores. We are working on its scalability issues, and intend to make use of SAT/SMT [3] to tackle this limitation.

References

1 Rajeev Alur and David L. Dill. A Theory of Timed Automata. *Theor. Comput. Sci.*, 126(2):183–235, April 1994. doi:10.1016/0304-3975(94)90010-8.

2 James Archibald and Jean-Loup Baer. Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. *ACM Trans. Comput. Syst.*, 4(4):273–298, September 1986. doi:10.1145/6513.6514.

3 Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. *Handbook of Satisfiability*, chapter Satisfiability Modulo Theories, pages 825–885. IOS Press, 2009.

4 Gerd Behrmann, Alexandre David, and Kim G. Larsen. *A Tutorial on Uppaal*, pages 200–236. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. doi:10.1007/978-3-540-30080-9_7.

5 Frédéric Boniol, Hugues Cassé, Eric Noulard, and Claire Pagetti. Deterministic Execution Model on COTS Hardware. In *Proceedings of the 25th International Conference on Architecture of Computing Systems (ARCS'12)*, pages 98–110, 2012.

6 Franck Cassez and Jean-Luc Béchennec. Timing Analysis of Binary Programs with UPPAAL. In *13th International Conference on Application of Concurrency to System Design, ACSD 2013*, pages 41–50. IEEE Computer Society, July 2013. doi:10.1109/ACSD.2013.7.

7 Franck Cassez and Pablo González de Aledo Marugán. Timed Automata for Modelling Caches and Pipelines. In Rob J. van Glabbeek, Jan Friso Groote, and Peter Höfner, editors, *Proceedings Workshop on Models for Formal Analysis of Real Systems, MARS 2015, Suva, Fiji, November 23, 2015.*, volume 196 of *EPTCS*, pages 37–45, 2015. doi:10.4204/EPTCS.196.4.

8 Sudipta Chattopadhyay, Abhik Roychoudhury, and Tulika Mitra. Modeling Shared Cache and Bus in Multi-cores for Timing Analysis. In *Proceedings of the 13th International Workshop on Software & Compilers for Embedded Systems*, SCOPES '10, pages 6:1–6:10, New York, NY, USA, 2010. ACM. doi:10.1145/1811212.1811220.

9 Andreas E. Dalsgaard, Mads Chr. Olesen, Martin Toft, René Rydhof Hansen, and Kim Guldstrand Larsen. METAMOC: Modular Execution Time Analysis using Model Checking. In Björn Lisper, editor, *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, volume 15 of *OpenAccess Series in Informatics (OASIcs)*, pages 113–123, Dagstuhl, Germany, 2010. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. The printed version of the WCET'10 proceedings are published by OCG (www.ocg.at) - ISBN 978-3-85403-268-7. doi:10.4230/OASIcs.WCET.2010.113.

10 Giorgio Delzanno. Automatic Verification of Parameterized Cache Coherence Protocols. In *Proceedings of the 12th International Conference on Computer Aided Verification*, CAV '00, pages 53–68, London, UK, UK, 2000. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=647769.734088.

11 Philip Enslow, Jr. Multiprocessor Organization - a Survey. *ACM Comput. Surv.*, 9(1):103–129, March 1977.

12 Sylvain Girbal, Xavier Jean, Jimmy le Rhun, Daniel Gracia Pérez, and Marc Gatti. Deterministic Platform Software for Hard Real-Time systems using multi-core COTS. In *34th Digital Avionics Systems Conference (DASC'15)*, 2015.

13 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, WCET 2010, July 6, 2010, Brussels, Belgium*, pages 101–112, 2010. doi:10.4230/OASIcs.WCET.2010.101.

14 Mohamed Hassan, Anirudh M. Kaushik, and Hiren D. Patel. Predictable Cache Coherence for Multi-core Real-Time Systems. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2017, Pittsburg, PA, USA, April 18-21, 2017*, pages 235–246, 2017. doi:10.1109/RTAS.2017.13.

15 Xavier Jean, David Faura, Marc Gatti, Laurent Pautet, and Thomas Robert. Ensuring robust partitioning in multicore platforms for IMA systems. In *31st Digital Avionics Systems Conference (DASC'16)*, 2012.

16 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*, pages 339–349, November 2010. doi:10.1109/RTSS.2010.30.

17 Claire Maiza, Hamza Rihani, Juan Maria Rivas, Joël, Godelieve Goossens, Sebastian Altmeyer, and Robert I. Davis. A Survey of Timing Verification Techniques for Multi-Core Real-Time Systems. Technical report, Grenoble INP/Ensimag/Verimag, 2018.

18 Rodolfo Pellizzoni, Emiliano Betti, Stanley Bak, Gang Yao, John Criswell, Marco Caccamo, and Russell Kegley. A Predictable Execution Model for COTS-Based Embedded Systems. In *17th IEEE Real-Time and Embedded Technology and Applications Symposium RTAS 2011*, pages 269–279, 2011.

19 Fong Pong and Michel Dubois. A New Approach for the Verification of Cache Coherence Protocols. *IEEE Trans. Parallel Distrib. Syst.*, 6(8):773–787, August 1995. doi:10.1109/71.406955.

20 Daniel J. Sorin, Mark D. Hill, and David A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Morgan & Claypool Publishers, 1st edition, 2011.

21 Nivedita Sritharan, Anirudh M. Kaushik, Mohamed Hassan, and Hiren D. Patel. HourGlass: Predictable Time-based Cache Coherence Protocol for Dual-Critical Multi-Core Systems. *CoRR*, abs/1706.07568, 2017. arXiv:1706.07568.

22 V. Suhendra, T. Mitra, and A. Roychoudhury and. WCET centric data allocation to scratchpad memory. In *26th IEEE International Real-Time Systems Symposium (RTSS'05)*, pages 10 pp.–232, December 2005. doi:10.1109/RTSS.2005.45.

23 Texas Instruments. TCI6630K2L Multicore DSP+ARM KeyStone II System-on-Chip. Technical Report SPRS893E, Texas Instruments Incorporated, 2013.

24 Sascha Uhrig, Lillian Tadros, and Arthur Pyka. MESI-Based Cache Coherence for Hard Real-Time Multicore Systems. In Luís Miguel Pinho Pinho, Wolfgang Karl, Albert Cohen, and Uwe Brinkschulte, editors, *Architecture of Computing Systems – ARCS 2015*, pages 212–223, Cham, 2015. Springer International Publishing.

25 L. Wehmeyer and P. Marwedel. Influence of memory hierarchies on predictability for time constrained embedded software. In *Design, Automation and Test in Europe*, pages 600–605 Vol. 1, March 2005. doi:10.1109/DATE.2005.183.

26 Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The Worst-case Execution-time Problem - Overview of Methods and Survey of Tools. *ACM Transactions Embedded Computing Systems*, 7(3):36:1–36:53, May 2008.

27 Reinhard Wilhelm and Jan Reineke. Embedded systems: Many cores - Many problems. In *7th IEEE International Symposium on Industrial Embedded Systems (SIES'12)*, pages 176–180, 2012.

28 Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *19th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'13)*, pages 55–64, 2013.

29 Wei Zhang and Jun Yan. Static Timing Analysis of Shared Caches for Multicore Processors. *JCSE*, 6(4):267–278, 2012. doi:10.5626/JCSE.2012.6.4.267.