On How to Identify Cache Coherence: Case of the NXP QorIQ T4240

Nathanaël Sensfelder
ONERA, Toulouse, France

Julien Brunel
ONERA, Toulouse, France

Claire Pagetti
ONERA, Toulouse, France

Abstract

Architectures used in safety critical systems have to pass certain certification standards, which require sufficient proof that they will behave as expected. Multi-core processors make this challenging by featuring complex interactions between the tasks they run. A lot of these interactions are made without explicit instructions from the program designers. Furthermore, they can have strong negative impacts on performance (and potentially affect correctness). One important such source of interactions is cache coherence, which speeds up operations in most cases, but can also lead to unexpected variations in execution time if not fully understood. Architecture documentations often lack details on the implementation of cache coherence. We thus propose a strategy to ascertain that the platform does indeed implement the cache coherence protocol its user believes it to. We also apply this strategy to the NXP QorIQ T4240, resulting in the identification of a protocol (MESIF) other than the one this architecture's documentation led us to believe it was using (MESI).

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

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2020.13

1 Introduction

The ever increasing complexity of aircraft and the market's depreciation of single-core processors are motivating the introduction of multi-core processors in aeronautical systems. While the performance gains offered by a switch to these more recent architectures are enticing, this process is impeded by their seemingly unpredictable nature [34], which is inherently incompatible with safety critical environments and aeronautical certification [9]. Still, a number of works are focusing on determining the means required for aircraft manufacturers to fulfill certification expectations despite the complex internal behaviors of multi-core processors COTS (Commercial Off-The Shelves) [1, 5, 12, 26, 28].

1.1 Cache Coherence – Case of the NXP T4240

Part of this unpredictability can be imputed to the mechanisms that let caches coordinate with one another in order to maintain data coherence without explicit program instructions. There are multiple competing strategies that can be employed to achieve *cache coherence*, and, while the general ideas behind them are known, the details of their implementation tend to be absent from architecture documentations, leaving programmers with the task of finding possibly problematic corner cases and unexpected behaviors.

In this paper, we focus on the NXP QorIQ T4240 [14], a PowerPC architecture featuring twelve e6500 cores, each of which is capable of running two simultaneous threads. The cores are equally distributed among three clusters, with one 2MB L2 cache per cluster. These

three L2 caches coordinate and access memory through a complex interconnect called the CoreNet Coherency Fabric. According to their processor's documentation, [13], these clusters implement the MESI cache coherence protocol. More details can be seen in Figure 1, which displays all the cores, caches, and memory controllers present on that architecture.

To be allowed to embed this architecture in an aeronautical system, the designer must be in control of any *transaction* occurring on the platform, that is, any low level behaviors caused by either explicit requests made by a program or by implicit mechanisms of the platform. This also holds true for cache coherence: it is up to the designer to quantify and control the effects on the application software of any transaction generated by this mechanism.

1.2 Formal Specification and its Validation

Having to keep implicit mechanisms under control is not an easy task for designers. This is especially true in the case of cache coherence, whose impact is difficult to evaluate even when its rules are made known to the designer.

In this paper, we present our analysis of the NXP QorIQ T4240 cache coherence transactions. This first required us to determine the protocol implemented in the architecture. According to its documentation, the protocol is supposed to be MESI [29] (*Modified*, *Exclusive*, *Shared*, *Invalid*). To guarantee the proper coverage of all that is involved, we argue for a formal definition of the cache coherence protocol to be made by the designer, based on their current understanding. Such formal definitions do not leave room for any ambiguities. Thus, we have looked for preexisting models of MESI protocol for split-based transaction buses. As it happens, we found none, making our first contribution (Section 3) a formal definition for a split-transaction bus MESI cache coherence protocol, which also corresponds to what we believed the NXP QorIQ T4240 to be using.

This formal MESI protocol definition describes all the transactions it is supposed to be performing. Thus, through the application of our proposed strategy (Section 4) we are able to make use of appropriate stress testing to observe the platform's behavior and compare with what we expected, in effect validating that the architecture does indeed implement the protocol we believe it to. While we developed this strategy around the T4240 and its limited means of observation, we tried to keep our strategy generic enough that it could be soundly used for other targets.

When we applied the strategy to validate the NXP QorIQ T4240, it became apparent the protocol was not actually MESI. Indeed, thanks to the validation strategy, we observed that there were five stable states instead of four and that one of them behaves in a way that led us to recognize a MESIF protocol [17]. We thus had to formally define a split-transaction bus MESIF protocol (Section 6), as we could not find any preexisting definition for it either. We then applied the validation strategy with this new protocol as the starting point, and this time we only found slight implementation choice differences between the supposed and the observed behaviors (Section 7).

In the sequel, we start with a reminder of hardware components and their contribution to the cache coherence. We then detail the contributions described above. We compare our approach with the related works before concluding the paper.

2 Cache Coherence

This section provides a reminder of the terminology and of the components involved in the description. Figure 2 provides a visual summary of how all these components interact.

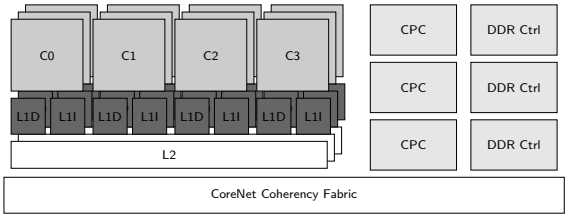


Figure 1 Computation & memory parts of the T4240.

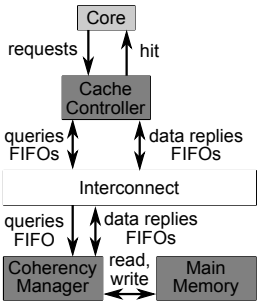


Figure 2 Components involved in coherence.

2.1 The Programs

To keep things simple, we only consider the memory related instructions of programs. Thus, programs are reduced to sequences of `load`, `store`, and `evict` instructions, each being applied to a single given address. Programs do not take into consideration the possibility of either instruction jumping or branching. Addresses are tantamount to *memory element*s (aligned blocks of memory with the size of a cache line), preventing any possible aliasing. Thus, all considered components, including programs, operate on the same memory unit.

2.2 The Caches

Cache controllers keep copies of memory elements to perform core requests. These copies are acquired through queries on the interconnect. Read-only copies are queried using `GetS`, read-and-write copies through `GetM`, and the eviction of a copy can be indicated through `PutM`. A cache controller may reply to the query of another, providing them with data. They are also able to send data to the coherence manager. Each copy of a memory element held in a cache is attributed a state and, optionally, the identifier of a cache controller.

2.3 The Interconnect

The *interconnect* merely arbitrates the order in which queries are broadcasted. Cache controllers do not directly access the interconnect. Instead, interactions between the cache controllers and the interconnect are all done through FIFO queues. There are four in total for each cache controller: incoming and outgoing queries; and incoming and outgoing data messages. The interconnect follows its access policy when choosing which cache controller's outgoing query queue to poll from next, then enqueues that query to every cache controller's incoming query queue (including that of the one from which the query was taken). Thus, all cache controllers and the coherence manager receive all queries and do so in the same order.

We consider interconnects that support split-transactions, meaning that queries and their replies do not block each other, allowing new queries to be sent before the previous ones receive their replies.

2.4 Coherence Manager and Main Memory

Cache controllers do not directly send messages to the system's main memory. Instead, messages meant for the main memory are directed to the *coherence manager*. The coherence manager sees all passing queries. It keeps track of which memory elements are being held by the cache controllers, and has a general idea of their current permissions. In particular, the

coherence manager may consider a cache controller to be the *owner* of a memory element, meaning that this cache controller is tasked with the propagation of the memory element's current value. This lets the coherence manager determine when a query warrants a reply from the main memory.

2.5 Terminology

The term *request* covers all types of communications between a core and its cache controller: *requests* = {`load`, `store`, `evict`}. The term *message* covers both the demands made by cache controllers, and the replies that fullfill them. In other words, *messages* are all communications that pass through the interconnect: *messages* = *queries*∪*data replies*, where *queries* = {`GetM`, `GetS`, `PutM`}, and *data replies* = {`data`, `data-e`, `no-data`}. Note that the actual elements found in *queries* and *data replies* depend on the specified protocol. The values given here being for the procotol described in the very next section.

3 Formal Description of the MESI Protocol

Our first contribution is the formal definition of a split-transaction MESI protocol that relies on a coherence manager. While the general idea behind MESI is available in many existing works, we did not find any that lists all the possible transient states that can be found in a real implementation (i.e. states other than Modified, Exclusive, Shared, and Invalid). These omissions tend to make the protocol much simpler to understand, but they leave ambiguities in the behavior of the protocol. Our description is a conjecture based on [31], which presents a complete definition of the MESI protocol, but that is limited to architectures featuring an atomic bus. Atomic buses only allow a single transaction (query and reply) to occur at any given time, which greatly narrows the number of transient states the system can find itself in.

3.1 Protocol Specification

MESI is based on the MSI protocol, so named because it features three stable states: *Modified*, which indicates read-and-write permissions of a memory element; *Shared*, for read-only permissions; and *Invalid*, the default one, indicating an absence of permissions. Introduced in [29], the MESI protocol adds a fourth stable state, *Exclusive*, which indicates that not only does the cache controller have read-only permissions, but also that no other cache currently holds any permission to access the memory element. This allows the cache controller to upgrade to read-and-write permissions without having to perform a costly communication. Just as it is used to keep track of whether a cache holds a read-and-write copy of a memory element in the MSI protocol, this definition of the MESI protocol uses the coherence manager to detect when a cache can be said to be the sole owner of a memory element.

This version of the MESI protocol uses three types of data replies: `data`, `data-e`, and `no-data`. `data` indicates that the value associated with the memory element is sent. By sending a `no-data` reply, cache controllers can indicate to the coherence manager that the memory element has been discarded (its value is not part of the reply). The coherence manager can send `data-e` replies, which are equivalent to `data`, with the added information that the recipient is its sole owner.

Our description of the MESI protocol can be seen in Table 1. It is split in two tables, one defining the cache controllers' behavior, the other the coherence manager's. In effect, these tables indicate a sequence of actions to be performed when faced with an incoming event (be it a request or a message).

Table 1 Description of the MESI protocol.

Cache Controller									
State	Core Request			Interconnect Access	Data Reply		Received Queries		
	load	store	evict		data	data-e	GetS	GetM	PutM
I	GetS?, IS^{BD}	GetM?, IM^{BD}	hit				-	-	-
IS^{BD}	stall	stall	stall	$IEoS^{D}$	IS^{B}	IE^{B}	-	-	-
IS^{B}	stall	stall	stall	S			-	-	
IS^{D}	stall	stall	stall		r←∅, S	r!data, m!no-data, r←∅, S	-	$IS^{D}I$	
$IEoS^{D}$	stall	stall	stall		S	E	r←s, IS^{D}	r←s, $IS^{D}I$	
$IS^{D}I$	stall	stall	stall		load hit, r←∅, I	load hit, r←∅, r!data, m!no-data, 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		store hit, r!data, r←∅, I		-	-	
$IM^{D}S$	stall	stall	stall		store hit, r!data, m!data, r←∅, S		-	$IM^{D}SI$	
$IM^{D}SI$	stall	stall	stall		store hit, r!data, m!data, r←∅, I		-	-	
S	hit	GetM?, SM^{BD}	hit, 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		store hit, M		r←s, $SM^{D}S$	r←s, $SM^{D}I$	
$SM^{D}I$	hit	stall	stall		store hit, r!data, r←∅, I		-	-	
$SM^{D}S$	hit	stall	stall		store hit, r!data, m!data, r←∅, S		-	$SM^{D}SI$	
$SM^{D}SI$	hit	stall	stall		store hit, r!data, m!data, r←∅, 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			-	-	-
E	hit	hit, M	PutM?, EI^{B}				m!no-data, s!data, S	s!data, I	
IE^{B}	stall	stall	stall	E			-	-	-
EI^{B}	hit	stall	stall	m!no-data, I			m!no-data, s!data, II^{B}	s!data, II^{B}	

Coherence Manager						
State	Received Queries				Data Reply	
	GetS	GetM	PutM (Owner)	PutM (Other)	data	no-data
I	read, s!data-e, r←s, M	s!data, r←s, M		-		
M	r←∅, S^{D}	r←s	r←∅, I^{D}	-	write, IoS^{B}	IoS^{B}
I^{D}	stall	stall	stall	-	write, resume, I	resume, I
S^{D}	stall	stall	stall	-	write, resume, S	resume, S
IoS^{B}	r←∅, S	r←s, M	r←∅, I	-		
S	read, s!data	s!data, r←s, M		-		

In the cache controller's table, columns correspond to the following: *state* refers to the state attributed to the local copy of the memory element by the cache controller. The three *Core request* columns indicate the actions that are performed when receiving a request from the core. *Interconnect access* specifies actions for when the cache controller reads one of its own queries. The *data reply* columns are for when the cache controller receives one of the types of data replies. Lastly, the *received queries* columns are for the reception of queries originating from other cache controllers. The table defining the coherence manager follows the same principles, but does not have columns for core requests, as it cannot receive them, nor for access to the interconnect, as it does not emit queries.

Let us now expand on the semantics of the actions found in these tables. Cache controllers may send queries on the bus (e.g. sending a `GetS` query is noted as `GetS`?). They can also change the state they attribute to a memory element (e.g. moving to the I state, which is noted `I`). If a request coming from their core can be fulfilled without further actions, the table indicates it with `hit`. A similar notation is used to indicate that the oldest request of a given type has just been completed (e.g. `load hit`). As a reaction to an incoming query, cache controllers can mark their copies of memory elements as being associated with the cache controller that sent the query (noted `r`←`s`). This can later be used to send a data message to that cache controller (e.g. `r`!`data`). Data can also be sent as a reply to an incoming query (e.g. `s`!`data`), or to the coherence manager (e.g. `m`!`no-data`). The `stall` action marks that the cache controller is unable to handle the incoming request at the moment. This request is put into a waiting queue until the memory element changes state, at which point it is re-evaluated.

The coherence manager follows a similar syntax, with the exception of the `stall` action, which now blocks *any* incoming query until the next `resume` action (data messages are not blocked, however). The other additions are the `write` and `read` actions, which respectively indicate that the memory controller either writes the received value or reads the current one.

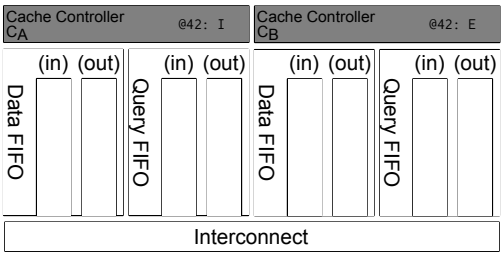
3.2 Examples of Behaviors

Here are some examples of remarkable behaviors exhibited by this definition of the MESI protocol.

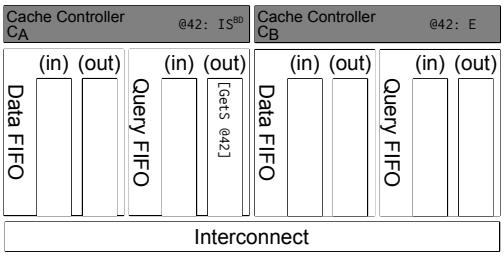
▶ **Example 1 (Reaching `S`).** This example is meant to showcase how exchanges between cache controllers are assumed to take place. To keep things simple, we only consider two cores and a single memory element (whose address is 42). This example is illustrated as a sequence in Figure 3.

▶ **Example 2 (Reaching `E`).** To hold a memory element in the `E` state, a cache must be the only one to have a copy of that memory element. The caches rely on the coherence manager to know when it is the case. The coherence manager uses its `I` state to mark memory elements that are sure to not be in any caches. Thus, if no cache controllers hold the memory element and the coherence manager is in `I`, whenever a core `loads` the data it becomes `E` in its cache. The behavior is similar to Figure 3 except that the main memory will provide the data.

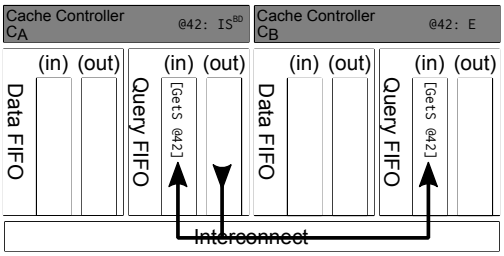
It is important to notice that it is not easy for the coherence manager to detect whether a cache controller is the sole owner. Indeed, the coherence manager is not always able to know that all caches have evicted their copy of a memory element: in Table 1, the cache controller's table indicates that an eviction from `S` does not lead to any message. The only way for the coherence manager to return to the `I` state is for a cache to evict its copy of a memory element in either the `E` or `M` state without another cache asking for a copy.



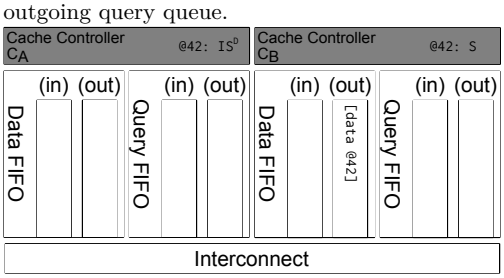
(a) In the initial phase, one cache controller, C_A, holds that memory element in the `I` state, while the other, C_B, holds it in the `E` state.



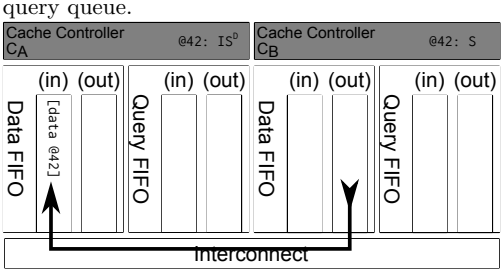
(b) Next, we consider that C_A's core issued a `load` instruction on 42. This leads C_A to move to the IS^{BD} state, and to issue a `GetS` query to its outgoing query queue.



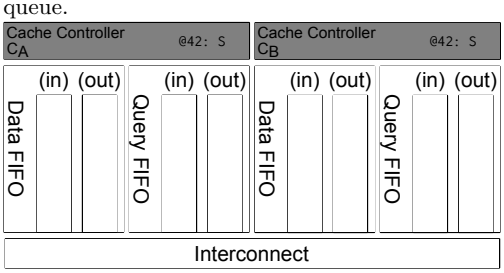
(c) The interconnect broadcasts outgoing queries from caches to all the incoming query queues. As C_A is the only one with an outgoing query, the `GetS` is added to both its own and C_B's incoming query queue.



(d) Consuming the message in their incoming query queues, both C_A and C_B change state, moving to IS^{D} and `S` respectively. In addition, C_B adds its reply for the query to its outgoing data queue.



(e) Data messages are not broadcasted, but instead only added to the incoming data queue of a targeted cache controller (or the coherence manager's). Thus, the reply is moved from C_B's outgoing reply queue to C_A's incoming one.



(f) Finally, C_A consumes the message in its incoming data queue, changing its state to `S` and fulfilling its core's request.

Figure 3 Illustrations for **Reaching `S`**.

▶ **Example 3 (Sharing from `E`).** From the coherence manager's point of view, there is no difference between a cache controller owning a memory element in the `E` state and one in the `M` state. Thus, if there is a cache owning a copy of a memory element in the `E` state, the coherence manager will assume that this cache may have modified the value and that the main memory no longer holds the correct value. As a result, the cache holding the Exclusive copy of the memory element will transfer it to any other cache that asks for it. If this is caused by another cache demanding a read-only copy (`GetS`), the coherence manager will expect an update on the value of the memory element. This update can come in two forms: either the cache that exclusively held the memory element made a modification (in which case it would have moved to the Modified state) and sends a `data` message, or it has not and it sends a `no-data` message.

3.3 System Behavior

The cache coherence protocol is defined for a single address and a single cache controller. However, what we are interested in is a multi-core architecture executing a program. Thus, we need to model the behavior of the overall platform to identify and quantify the transactions generated by the cache coherence. To do so, we use an automaton where each state corresponds to the system state and transitions between states are events produced by one or several components (core, cache controller, interconnect, coherence manager or memory). In this section we formally define such automata.

▶ **Definition 4** (Memory Element State). *Let $\mathcal{S}_s$ (resp. $\mathcal{T}_s$) denote the set of stable (resp. transient) states. From the point of view of a cache controller or a coherence manager, a memory element m can be in any valid stable or transient states, i.e., $m \in \mathcal{S}_s \cup \mathcal{T}_s$. We denote by $\mathcal{G}_s$ the set of states $\mathcal{G}_s = \mathcal{S}_s \cup \mathcal{T}_s$ of a cache controller and $\mathcal{G}_{CM}$ those of the coherence manager.*

▶ **Example 5.** In the MESI protocol defined in Table 1, the states of the cache controllers are $\mathcal{S}_s = \{M, E, S, I\}$ and $\mathcal{T}_s = \{IS^{\mathrm{BD}}, IS^{\mathrm{D}}, \ldots\}$. The states of the coherence manager are $\mathcal{S}_s = \{M, S, I\}$ and $\mathcal{T}_s = \{I^{\mathrm{D}}, S^{\mathrm{D}}, IoS^{\mathrm{B}}\}$.

▶ **Definition 6** (Cache Controller State). *Let $\mathtt{Addr}$ denote the set of all memory element addresses. We define the state of a cache controller CC (resp. of a coherence manager CM) as the function $s_{CC} : \mathtt{Addr} \to \mathcal{G}_s$ (resp. $s_{CM} : \mathtt{Addr} \to \mathcal{G}_{CM}$).*

▶ **Definition 7** (System State). *Let us consider an architecture $\langle CC_1, \ldots, CC_n, CM\rangle$ composed of n cache controllers CC_i and a coherence manager CM. The global state of the architecture consists of the states of all memory elements in all cache controllers and in the coherence manager. Let $\mathtt{Addr}$ be the set of all memory element addresses, the global state $s : \mathtt{Addr} \to \mathcal{G}_s^n \times \mathcal{G}_{CM}$ is defined as $\forall m \in \mathtt{Addr}, s(m) = \langle s_{CC_1}(m), \ldots, s_{CC_n}(m), s_{CM}(m)\rangle$ where s_{CC_i} is the state of the cache controller CC_i as defined in Definition 6. For the sake of simplicity and without loss of the generality, in the sequel, we will only focus on a given address m and define the state of a cache controller as an element in $\mathcal{G}_s$ and of the system as a tuple in $\mathcal{G}_s^n \times \mathcal{G}_{CM}$.*

▶ **Definition 8** (Valid State). *Not all combinations of states are valid, e.g. two cache controllers cannot be in $\mathtt{M}$ for the same address at the same time. We note $\mathcal{V} \subseteq \mathcal{G}_s^n \times \mathcal{G}_{CM}$ the set of valid system states.*

▶ **Definition 9** (Event). *We distinguish between explicit (or controllable) events $E_E = requests = \{\mathtt{load}, \mathtt{store}, \mathtt{evict}\}$, which are made by the user, and the implicit (or uncontrollable) events $E_I = messages \cup \{bus\}$ (where bus corresponds to the cache seeing one of its own query being broadcasted on the interconnect), which are made by the architecture. Thus, on a given cache controller, the possible events are $E_E \cup E_I \cup \{-\}$ where $-$ represents the special event where nothing happens.*

The set of events over the system is denoted by $E \subseteq (E_E \cup E_I \cup \{-\})^n$ (not all combinations of events are possible).

The system event $\langle -, \ldots, -, e, -, \ldots, -\rangle$ consisting of one event e in the cache controller of id i, and nothing in all the other cache controllers, is simply denoted by $\langle e, i\rangle$.

▶ **Definition 10** (Automaton of the system). *The behavior of the system is defined by the automaton $\langle \mathcal{V}, E, s_{init}, Tr\rangle$ where $s_{init} = \langle I, \ldots, I\rangle$ is the initial state and $Tr : \mathcal{V} \times E \to \mathcal{V}$ is the transition function.*

▶ **Example 11.** Using the MESI protocol and 2 cache controllers, we have for instance $\mathrm{Tr}(\langle I, I, I\rangle, \langle load, -\rangle) = \langle IS^{\mathrm{BD}}, I, I\rangle$. As the event is a single component event, we could also use the notation mentioned above $\langle load, 1\rangle = \langle load, -\rangle$ meaning that the cache controller with id 1 does a *load*, whereas all the other do nothing. If we detail all the implicit events leading to the next stable states, we have: $\mathrm{Tr}(\langle IS^{\mathrm{BD}}, I, I\rangle, \langle \mathrm{bus}, GetS, GetS\rangle) = \langle IEoS^{\mathrm{D}}, I, M\rangle$ (the interconnect broadcasts the *GetS*); $\mathrm{Tr}(\langle IEoS^{\mathrm{D}}, I, M\rangle, \langle \text{data-e}, 1\rangle) = \langle E, I, M\rangle$ (the cache coherence triggers the memory which provides the requested data).

▶ **Example 12** (Simultaneous requests)**.** Still using the MESI protocol and 2 cache controllers, there may be several simultaneous requests, e.g. $\langle load, store\rangle$. In such situations, because of the internal dynamics of the interconnect and memory (Round Robin access, delays ...) all combinations of interleaving are envisaged. For instance $\mathrm{Tr}(\langle I, I, I\rangle, \langle load, store\rangle) = \langle IS^{\mathrm{BD}}, IM^{\mathrm{BD}}, I\rangle$ (all local requests are handled). Then, among the possible next steps are both $\mathrm{Tr}(\langle IS^{\mathrm{BD}}, IM^{\mathrm{BD}}, I\rangle, \langle bus, GetS\rangle) = \langle IEoS^{\mathrm{D}}, IM^{\mathrm{BD}}, M\rangle$ (the interconnect chooses the first core first) or $\mathrm{Tr}(\langle IS^{\mathrm{BD}}, IM^{\mathrm{BD}}, I\rangle, \langle GetM, bus\rangle) = \langle IS^{\mathrm{BD}}, IM^{\mathrm{D}}, M\rangle$ (the interconnect chooses the second core first). Thus, several paths leave from $\mathrm{Tr}(\langle I, I, I\rangle, \langle load, store\rangle)$ and they may ultimately lead to separate stable states: $\langle I, M, M\rangle$ if the data reply has reaches core 1 first, or $\langle S, S, S\rangle$ if the data reply reaches core 2 first.

▶ **Definition 13** (Path)**.** *A path in a system automaton corresponds to a succession of transitions, from one state to another, that has been triggered by a controllable event and is followed by a series of adequate implicit events. A path from s to s' triggered by e is denoted by $p : s \rightsquigarrow^{e} s'$.*

▶ **Example 14.** The successive transitions that have been described in Example 11 define the path $\langle I, I, I\rangle \rightsquigarrow^{\langle load,1\rangle} \langle E, I, M\rangle$.

▶ **Definition 15.** *From the transition function Tr, we define the observable transition function $Tr_i^\star$, in the case of single controllable events, as follows: $\forall c \in \mathcal{V}, e \in E_E$ $Tr_i^\star(c, \langle e, i\rangle) = c'$ such that $c \rightsquigarrow^{\langle e,i\rangle} c'$ and no implicit event may be induced by $\langle e, i\rangle$ upon reaching c'.*

For simultaneous explicit events, the observable transition function is defined by composing the observable transition function for each explicit event, taken as a single event. Notice that it is an expected property of the protocol, which we do not study here, that all possible orderings provide the same system state.

▶ **Example 16.** Considering a single event: $\mathrm{Tr}_i^\star(\langle I, I, I\rangle, \langle load, 1\rangle) = \langle E, I, M\rangle$.

Let us now consider multiple events: $\mathrm{Tr}_i^\star(\langle I, I, I\rangle, \langle load, store\rangle) = \{\langle S, S, S\rangle, \langle I, M, M\rangle\}$.

▶ **Definition 17** (Number of Events)**.** *We define the function $NbEvent : \mathcal{P}ath \times 1..n \to \mathbb{N}^3$ (with n the number of cores in the architecture) which associates to each path and core id, the number of accesses to the bus, the number of received queries, and the number of received data replies for the cache controller identified by this id.*

▶ **Example 18.** Let us consider the path $p : \langle I,I,I\rangle \rightsquigarrow^{\langle load,1\rangle} \langle E,I,M\rangle$. Then $NbEvents(p,1) = \langle 1, 1, 1\rangle$ because core 1 has accessed to the bus once, received its own *GetS* and the data reply to its request. $NbEvents(p, 2) = \langle 0, 1, 0\rangle$ because core 2 has simply received the *GetS* generated by core 1.

4 Validation Strategy

This section presents our proposed strategy to assert that a given architecture does indeed implement a given previously defined cache coherence protocol. We have fully defined the

system behavior in Section 3.3 and ideally we would recognize all the automata on the architecture. Unfortunately, we cannot simply observe the states and events as we previously defined them. Instead, we observe flags and performance counters, to which we need to link the notion of states and events. Even worse, our observations are only partial, with some information missing. Thus, in addition to linking the observations to the automaton, we also have to infer the missing elements. We illustrate our ideas on the NXP QorIQ T4240 platform, however, the reasoning could be leveraged for other types of architecture.

Observable States

▶ **Property 1** (T4240 Observable Flags). *We can observe flags with CodeWarrior, the official debugging suite for this architecture. While a lot of information is available, we consider the relevant cache line flags to be: Dirty, Valid, Share, Exclusive and LastReader. Those flags take Boolean value and only provide information on stable states. Indeed, no combination of flags correspond to any transient state. Instead, their value changes upon entering the next stable state following the execution of a request (load, store, or evict) or because of an external query.*

▶ **Definition 19** (Observable Cache Controller State). *Let us consider an architecture with p Boolean flags, an observable state o for a cache controller is a combination of values of the flags $o = \langle f_1, \ldots, f_p \rangle$. Let $\mathcal{R}$ denote the set of cache controller states that can be really observed on a given architecture.*

▶ **Issue 1** (Matching Observable Cache Controller States and Stable States). *For validating that an architecture indeed implements a cache protocol, we need to associate each observed state with a protocol state. More precisely, we need to identify a function* `Decode` $: \mathcal{R} \to \mathcal{G}_s$ *such that* `Decode` *is surjective:* $\forall f \in \mathcal{G}_s,\ \exists r \in \mathcal{R},\ \texttt{Decode}(r) = f$.

Indeed, while having multiple observed states corresponding to the same state is perfectly acceptable at this point (the different states may end up being identical from the cache coherence's point of view), the reverse is not true: if an observed cache coherence state is attributed to multiple formally defined state, the analysis considers that the protocols do not match. This can be caused by missing information (unable to observe the information that would split the observed state into multiple ones). This is the reason why `Decode` has to be surjective.

▶ **Example 20.** For the T4240, the observable state $\langle$*Dirty*=false, *Valid*=false, *Share*=false, *Exclusive*=false,*LastReader*=false$\rangle$ is the initial observable state and corresponds to the I stable state.

The tools at our disposition do not expose anything related to the coherence manager.

▶ **Property 2** (No Observation Available from the Coherence Manager). *We do not have any possibility of observing the coherence manager directly.*

▶ **Definition 21** (Observable System State). *Let us consider an architecture $\langle CC_1, \ldots, CC_n, CM \rangle$ composed of n cache controllers CC_i and a coherence manager CM. The observable system states are the observable states of each CC_i and the CM.*

▶ **Example 22.** On the T4240, the observable system states are the observable states of each cache controller only. Thus, to match the observable state and the real state we have to infer the non observable elements.

▶ **Issue 2** (Matching Observable States and System States). *To validate that an architecture indeed implements a given cache protocol, we need to identify a function* $\texttt{Decode} : \mathcal{R}^n \to \mathcal{V}$ *associating a tuple of observable states with a system state, which is directly defined from the function* `Decode` *of Issue 1 and that is also surjective:* $\forall f \in \mathcal{V},\ \exists r \in \mathcal{R}^n,\ \texttt{Decode}(r) = f$.

Controllable Events and Reachable Observable States

▶ **Property 3** (T4240 controllable events). *On each core, we can execute programs. Thus, to induce implicit cache coherence traffic, we can only trigger some request (load, store or evict) and observe the reached observable states. We have defined a series of benchmarks that can either run a single request on a core or multiple requests on several cores.* $\texttt{Reach}(C, \langle \text{instr}, k\rangle)$ *denotes the observable state after executing the single request* instr *on the core* k *from the observable state* C*. In addition,* $\texttt{Reach}_m(C, \langle \text{instr}_1, \ldots, \text{instr}_n\rangle)$ *denotes the observable states after executing the simultaneous requests* $\text{instr}_i \in \{load, store, evict, -\}$ *on each core from the observable state* C.

▶ **Definition 23** (Reachable System States). *From an architecture in an initial state in which no memory elements are stored in the caches, we can explore the reachable system states by executing benchmarks that trigger requests.*

▶ **Definition 24** (Step 1: Reachability Analysis). *Starting from the initial situation where all cache controllers consider the memory element to be invalid, we compute the reachable observable system states by observing the effect of a single core instruction and the associated transition relation* `Reach`*. The idea is to run a benchmark and observe the reached state. If this state has not been visited, it is added to* $\mathcal{R}^n$*, otherwise it is not. This is a basic reachability algorithm.*

```
R^n ← {init}
Candidates ← {⟨init⟩}
while (Candidates ≠ ∅):
    C ∈ Candidates;
    Candidates ← Candidates/C;
    foreach k ≤ n
        foreach instr ∈ {load, store, evict}
            ObservedState ← benchmark(C, ⟨instr, k⟩)
            Reach (C, ⟨instr, k⟩) ← ObservedState
            if ObservedState ∉ R^n
                R^n ← R^n ∪ {ObservedState}
                Candidates ← Candidates ∪{ ObservedState}
                Events[C, ObservedState] ← PerformanceCounters
```

▶ **Issue 3.** *To be valid, the matching between observable states and system states must be consistent with the transitions of both protocols. That is, the function* `Decode` *has to be a* simulation relation*:* $\forall o \in \mathcal{R}^n,\ \forall c \leq n,\ \forall i \in$ requests, $\texttt{Decode}(\texttt{Reach}(o, \langle i, c\rangle)) = Tr_i^\star(\texttt{Decode}(o), \langle i, c\rangle)$.

▶ **Example 25.** For the T4240, the observable state $f_0 = \langle Dirty$=false, $Valid$=false, $Share$=false, $Exclusive$=false, $LastReader$=false$\rangle$ is the initial observable state and corresponds to the I stable state; whereas $f_1 = \langle Dirty$=false, $Valid$=true, $Share$=false, $Exclusive$=true, $LastReader$=false$\rangle$ seems to be E. When running the benchmark $\langle$`load`$, 1\rangle$ on core 1 from f_0, the reached observable state is f_1, which allows for the possibility of f_1 being E.

Observable Events

The performance registers can count the number of occurrences of predefined events. While the name and identification code for each performance event is indicated in the architecture's documentation, the meaning behind their name is not always obvious.

▶ **Property 4** (T4240 Performance Counters). *Below is a list of the events of interest, as well as their meaning based on our understanding.*

- ***L2 Data Accesses*** *Accesses made to the L2 cache.*
- ***L2 Snoop Hits*** *External queries on a memory element held by this cache.*
- ***L2 Snoop Pushes*** *Replies given to snooped queries.*
- ***External Snoop Requests*** *External queries.*
- ***L2 Reloads From CoreNet*** *Replies received.*
- ***L2 Snoops Causing MINT*** *Replies to a snooped query when holding the memory element in a dirty (modified) state.*
- ***L2 Snoops Causing SINT*** *Replies to a snooped query when holding the memory element in a clean (unmodified) state.*
- ***CPU Cycles***

▶ **Definition 26** (Observable Cache Controller Associated Events). *Let us consider an architecture with p Integer counters, an observable event f for a cache controller is a combination of values of the counters $o = \langle f_1, \ldots, f_p \rangle$. The set of cache controller associated events that can be really observed on a given architecture is denoted by $\mathcal{N}$.*

▶ **Issue 4.** *To be valid, the matching between observable states and system states must be consistent with the events associated with the transitions of both protocols. That is, for single instructions,* $\forall o \in \mathcal{R}^n,\ \forall c \leq n,\ \forall i \in \text{requests},\ \forall j \in 1..n,$
$\text{Events}[o, \texttt{Reach}(o, \langle i, c\rangle), j] = \text{NbEvents}(\texttt{Decode}(o) \rightsquigarrow^{\langle i,c\rangle} \texttt{Decode}(\texttt{Reach}(o, \langle i, c\rangle)), j)$ *where* Events *stores the performance counters, seen from core j, in the benchmark going from o to* $\texttt{Reach}(o, \langle i, c\rangle)$ *(as done in the algorithm of step 1).*

For multiple simultaneous instructions: $\forall o \in \mathcal{R}^n,\ \forall e_1, \ldots, e_n \in \text{requests} \cup \{-\},\ \forall j \in 1..n,\ \forall s \in \texttt{Reach}_m(o, \langle e_1, \ldots, e_n\rangle),\ \text{Events}[o, s, j] = \text{NbEvents}(\texttt{Decode}(o) \rightsquigarrow^{\langle e_1,\ldots,e_n\rangle} \texttt{Decode}(s), j)$.

▶ **Example 27.** Continuing Example 25, when running the benchmark $\langle \texttt{load}, 1\rangle$ on core 1 from f_0, we also collect the observable events and we observe that: on core 1, there are 2 *L2 Data Accesses* (1 for the *data-e*, all such messages are duplicated as explained in the next section) and 1 *L2 Reloads From CoreNet* (1 for the *GetS*); on core 2, there is 1 *External Snoop Requests* (1 for the *GetS*); which still allows f_1 to be E.

▶ **Definition 28** (Step 2: Reachability Analysis with Simultaneous Requests). *In addition to step 1 (see Definition 24), for the multiple simultaneous requests, we need to run additional benchmarks. The idea is similar, except that instead of running benchmark(C, ⟨instr, k⟩), we apply benchmark(C, ⟨instr$_1$, . . . , instr$_n$⟩) for the tuples ⟨instr$_1$, . . . , instr$_n$⟩ in a pre-computed list.*

▶ **Example 29.** Consider that we have run the benchmark $\langle load, store, -\rangle$ from the initial state and we believe that this coincides with the path $\langle I, I, I\rangle \rightsquigarrow^{\langle load,store,-\rangle} \langle I, M, M\rangle$ then we have to count on the core 1: 1 access to the interconnect, 2 queries and 2 data replies.

5 Evaluating Cache Coherence on the T4240

In this section, we illustrate our proposed process by attempting to validate the MESI protocol we defined on the NXP QorIQ T4240 architecture. Much to our surprise, the results quickly conclude that this architecture does not implement MESI.

5.1 The NXP QorIQ T4240 Experimental Setup

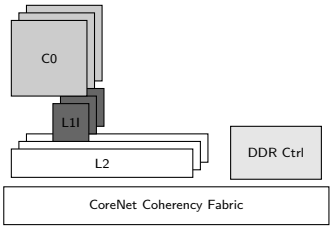


Figure 4 Configuration of the T4240.

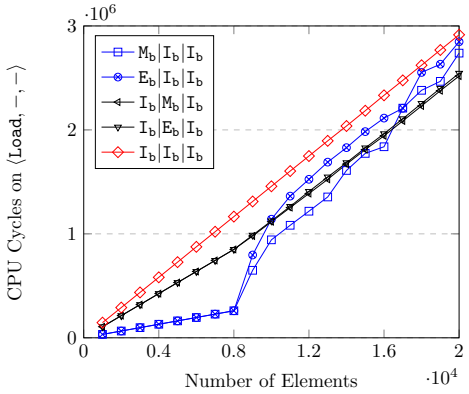


Figure 5 Exposing Cache Eviction.

In order to limit the mechanisms observed to the L2 cache coherence, we chose to disable the architecture's L1 Data caches. Furthermore, we only consider a single core (and execution thread) per cluster, and thus, per L2 cache. In an attempt at reducing the impact of instruction fetching, we keep the L1 Instruction caches enabled. Lastly, our system only uses a single memory controller. Thus, our configuration resembles the one shown in Figure 4, the remaining hardware configuration being left to what it is by default.

The NXP QorIQ T4240 architecture does not feature the `evict` instruction. The closest available instruction (`dcbi`, *Data Cache Block Invalidate*) results in the element being evicted from all the caches, which is significantly different, unless that element has been marked as ignored by cache coherence (which is then pointless for our purposes). Since our benchmarks are very small programs dealing almost exclusively with the set of experimental memory elements, we replaced the application of an `evict` on all of the memory elements with a simple invalidation of the whole local cache, which does still involve cache coherence.

While related to caches and an important factor of their performance, the issue of replacement policy is orthogonal to the cache coherence protocol. Thus, we do not want its effects to be mixed in our benchmarks. Through testing (see Figure 5), we concluded that caches started evicting cache lines when holding somewhere between 8000 and 9000 of them. From the information given in [13], we speculate that this corresponds to the 8192 cache lines held in a bank.

5.2 Partial Matching of States with Step 1

We listed and named every combination we have encountered in Table 2. We made an initial matching that seems coherent with the flags name, but that still needs to be checked by looking at the transitions. We denote by a $_b$ suffix the states observed on the platform.

We check the property required by Issue 3 that applying any request (`load`, `store`, and `evict`) from a matched state leads to the correct matched state. Table 3 shows the original and destination state for a memory element on each of the three clusters according

Table 2 Stable States of the T4240 L2 Caches Protocol.

State	*Dirty*	*Valid*	*Share*	*Exclusive*	*LastReader*
$\mathtt{M_b}$	✓	✓			
$\mathtt{E_b}$		✓		✓	
$\mathtt{I_b}$					
$\varphi_\mathtt{b}$		✓			✓
$\chi_\mathtt{b}$		✓	✓		

Table 3 State Changes.

	⟨Load,-,-⟩		⟨Store,-,-⟩		⟨Evict,-,-⟩	
Origin	**Destination**		**Destination**		**Destination**	
	Observ	**Match**	**Observ**	**Match**	**Observ**	**Match**
$\langle \mathtt{I_b},\mathtt{I_b},\mathtt{I_b}\rangle$	$\langle \mathtt{E_b},\mathtt{I_b},\mathtt{I_b}\rangle$	$\langle \mathtt{E},\mathtt{I},\mathtt{I}\rangle$	$\langle \mathtt{M_b},\mathtt{I_b},\mathtt{I_b}\rangle$	$\langle \mathtt{M},\mathtt{I},\mathtt{I}\rangle$	$\langle \mathtt{I_b},\mathtt{I_b},\mathtt{I_b}\rangle$	$\langle \mathtt{I},\mathtt{I},\mathtt{I}\rangle$
$\langle \mathtt{E_b},\mathtt{I_b},\mathtt{I_b}\rangle$	$\langle \mathtt{E_b},\mathtt{I_b},\mathtt{I_b}\rangle$	$\langle \mathtt{E},\mathtt{I},\mathtt{I}\rangle$				
$\langle \mathtt{M_b},\mathtt{I_b},\mathtt{I_b}\rangle$	$\langle \mathtt{M_b},\mathtt{I_b},\mathtt{I_b}\rangle$	$\langle \mathtt{M},\mathtt{I},\mathtt{I}\rangle$				
$\langle \mathtt{I_b},\mathtt{I_b},\mathtt{M_b}\rangle$	$\langle \varphi_\mathtt{b},\mathtt{I_b},\chi_\mathtt{b}\rangle$	$\langle \mathtt{S},\mathtt{I},\mathtt{S}\rangle$			$\langle \mathtt{I_b},\mathtt{I_b},\mathtt{M_b}\rangle$	$\langle \mathtt{I},\mathtt{I},\mathtt{M}\rangle$
$\langle \mathtt{I_b},\mathtt{I_b},\mathtt{E_b}\rangle$	$\langle \varphi_\mathtt{b},\mathtt{I_b},\chi_\mathtt{b}\rangle$	$\langle \mathtt{S},\mathtt{I},\mathtt{S}\rangle$			$\langle \mathtt{I_b},\mathtt{I_b},\mathtt{E_b}\rangle$	$\langle \mathtt{I},\mathtt{I},\mathtt{E}\rangle$
$\langle \varphi_\mathtt{b},\mathtt{I_b},\mathtt{I_b}\rangle$	$\langle \varphi_\mathtt{b},\mathtt{I_b},\mathtt{I_b}\rangle$	$\langle \mathtt{S},\mathtt{I},\mathtt{I}\rangle$			$\langle \mathtt{I_b},\mathtt{I_b},\mathtt{I_b}\rangle$	$\langle \mathtt{I},\mathtt{I},\mathtt{I}\rangle$
$\langle \chi_\mathtt{b},\varphi_\mathtt{b},\mathtt{I_b}\rangle$	$\langle \chi_\mathtt{b},\varphi_\mathtt{b},\mathtt{I_b}\rangle$	$\langle \mathtt{S},\mathtt{S},\mathtt{I}\rangle$			$\langle \mathtt{I_b},\varphi_\mathtt{b},\mathtt{I_b}\rangle$	$\langle \mathtt{I},\mathtt{S},\mathtt{I}\rangle$
$\langle \chi_\mathtt{b},\chi_\mathtt{b},\varphi_\mathtt{b}\rangle$	$\langle \chi_\mathtt{b},\chi_\mathtt{b},\varphi_\mathtt{b}\rangle$	$\langle \mathtt{S},\mathtt{S},\mathtt{S}\rangle$			$\langle \mathtt{I_b},\chi_\mathtt{b},\varphi_\mathtt{b}\rangle$	$\langle \mathtt{I},\mathtt{S},\mathtt{S}\rangle$
$\langle \varphi_\mathtt{b},\chi_\mathtt{b},\chi_\mathtt{b}\rangle$	$\langle \varphi_\mathtt{b},\chi_\mathtt{b},\chi_\mathtt{b}\rangle$	$\langle \mathtt{S},\mathtt{S},\mathtt{S}\rangle$			$\langle \mathtt{I_b},\chi_\mathtt{b},\chi_\mathtt{b}\rangle$	$\langle \mathtt{I},\mathtt{S},\mathtt{S}\rangle$
$\langle \varphi_\mathtt{b},\chi_\mathtt{b},\mathtt{I_b}\rangle$	$\langle \varphi_\mathtt{b},\chi_\mathtt{b},\mathtt{I_b}\rangle$	$\langle \mathtt{S},\mathtt{S},\mathtt{I}\rangle$			$\langle \mathtt{I_b},\chi_\mathtt{b},\mathtt{I_b}\rangle$	$\langle \mathtt{I},\mathtt{S},\mathtt{I}\rangle$
$\langle \mathtt{I_b},\mathtt{I_b},\varphi_\mathtt{b}\rangle$	$\langle \varphi_\mathtt{b},\mathtt{I_b},\chi_\mathtt{b}\rangle$	$\langle \mathtt{S},\mathtt{I},\mathtt{S}\rangle$			$\langle \mathtt{I_b},\mathtt{I_b},\varphi_\mathtt{b}\rangle$	$\langle \mathtt{I},\mathtt{I},\mathtt{S}\rangle$
$\langle \chi_\mathtt{b},\mathtt{I_b},\mathtt{I_b}\rangle$	$\langle \chi_\mathtt{b},\mathtt{I_b},\mathtt{I_b}\rangle$	$\langle \mathtt{S},\mathtt{I},\mathtt{I}\rangle$			$\langle \mathtt{I_b},\mathtt{I_b},\mathtt{I_b}\rangle$	$\langle \mathtt{I},\mathtt{I},\mathtt{I}\rangle$
$\langle \mathtt{I_b},\mathtt{I_b},\chi_\mathtt{b}\rangle$	$\langle \varphi_\mathtt{b},\mathtt{I_b},\chi_\mathtt{b}\rangle$	$\langle \mathtt{S},\mathtt{I},\mathtt{S}\rangle$			$\langle \mathtt{I_b},\mathtt{I_b},\chi_\mathtt{b}\rangle$	$\langle \mathtt{I},\mathtt{I},\mathtt{S}\rangle$
$\langle \mathtt{I_b},\varphi_\mathtt{b},\chi_\mathtt{b}\rangle$	$\langle \varphi_\mathtt{b},\chi_\mathtt{b},\chi_\mathtt{b}\rangle$	$\langle \mathtt{S},\mathtt{S},\mathtt{S}\rangle$			$\langle \mathtt{I_b},\varphi_\mathtt{b},\chi_\mathtt{b}\rangle$	$\langle \mathtt{I},\mathtt{S},\mathtt{S}\rangle$
$\langle \mathtt{I_b},\chi_\mathtt{b},\chi_\mathtt{b}\rangle$	$\langle \varphi_\mathtt{b},\chi_\mathtt{b},\chi_\mathtt{b}\rangle$	$\langle \mathtt{S},\mathtt{S},\mathtt{S}\rangle$			$\langle \mathtt{I_b},\chi_\mathtt{b},\chi_\mathtt{b}\rangle$	$\langle \mathtt{I},\mathtt{S},\mathtt{S}\rangle$
$\langle \chi_\mathtt{b},\chi_\mathtt{b},\mathtt{I_b}\rangle$	$\langle \chi_\mathtt{b},\chi_\mathtt{b},\mathtt{I_b}\rangle$	$\langle \mathtt{S},\mathtt{S},\mathtt{I}\rangle$			$\langle \mathtt{I_b},\chi_\mathtt{b},\mathtt{I_b}\rangle$	$\langle \mathtt{I},\mathtt{S},\mathtt{I}\rangle$

to what instruction was applied to the first cluster. This figure covers all the possible sets of stable states for the coherence of a single memory element on the system's clusters, since the permutation of two clusters does not impact the cache coherence's mechanisms. The transitions, however, are limited to those relevant when only a single operation is applied across the whole system. Furthermore, this does not account for any state of the coherence manager, since we are unable to observe them.

According to Table 3 we match each observed stable state with one of the formal ones. The $\mathtt{M_b}$, $\mathtt{E_b}$, and $\mathtt{I_b}$ states we observed perfectly match their `M`, `E`, and `I` counterparts from the MESI protocol. The `S` state, however, seems to match our observations of both the $\varphi_\mathtt{b}$ and $\mathtt{I_b}$ states. Indeed, when starting from $\langle \mathtt{I_b},\mathtt{I_b},\mathtt{M_b}\rangle$ and performing a `load` operation on the first cluster, we end up with two different states, $\varphi_\mathtt{b}$ and $\chi_\mathtt{b}$, where we would have expected to see two of the `S` state equivalent. The same occurs when starting from $\langle \mathtt{I_b},\mathtt{I_b},\mathtt{E_b}\rangle$. By itself, this observation is not sufficient to conclude that there is a discrepancy between the protocol we defined and the one observed on the architecture.

As we go through the different transitions from one stable state to another, we observe that performing an evict on either $\varphi_\mathtt{b}$ or $\chi_\mathtt{b}$ does not affect the other caches' state, which means that reaching either $\langle \chi_\mathtt{b},\mathtt{I_b},\mathtt{I_b}\rangle$ or $\langle \varphi_\mathtt{b},\mathtt{I_b},\mathtt{I_b}\rangle$ (or any permutation of these clusters)

is possible. In addition, the previous step showed that there is no way to have a system in which two clusters hold the same memory element in the φ_b state: the first cluster to reach the φ_b moves to the χ_b state upon seeing the other's query. Neither is it possible to have all three clusters in the χ_b state: the last cluster to load from I_b always enters φ_b, and there is no way to reach φ_b other than doing exactly that.

5.3 Consolidated Matching of States with Observable Events

Table 4 Unexpected Behaviors.

	⟨load, -, -⟩	
Origin	**Behavior**	
	Expected	**Observed**
$\langle I_b, I_b, I_b \rangle$	8000 L2D Accesses, 8000 Reloads From CoreNet	16000 L2D Accesses, 8000 Reloads From CoreNet, 1166700 CPU Cycles
$\langle I_b, I_b, \varphi_b \rangle$	8000 L2D Accesses, 8000 Reloads From CoreNet	16000 L2D Accesses, 8000 Reloads From CoreNet, 850600 CPU Cycles
$\langle I_b, I_b, \chi_b \rangle$	8000 L2D Accesses, 8000 Reloads From CoreNet	16000 L2D Accesses, 8000 Reloads From CoreNet, 1172600 CPU Cycles

	⟨-, -, load⟩	
Origin	**Behavior**	
	Expected	**Observed**
$\langle I_b, I_b, I_b \rangle$	8000 External Snoop Requests	8000 External Snoop Requests
$\langle \varphi_b, I_b, I_b \rangle$	8000 L2 Snoop Hits, 8000 External Snoop Requests	8000 L2 Snoop Hits, 8000 L2 Snoop Pushes, 8000 External Snoop Requests, 8000 SINTs
$\langle \chi_b, I_b, I_b \rangle$	8000 L2 Snoop Hits, 8000 External Snoop Requests	8000 L2 Snoop Hits, 8000 External Snoop Requests

While observing the existence of the χ_b and φ_b states may not have been sufficient to contradict a MESI protocol, they definitely did put it into question and so we prioritized furthering their analysis.

Table 4 shows our observations when loading a dataset of 8000 unique memory elements from the I_b state. The upper table indicates what is recorded on the cluster performing the `load` operations and the bottom table corresponds to what is recorded on the farthest cluster, hence the symmetry of origin state and of operation between the two tables. The $\langle I_b, I_b, I_b \rangle$ is given as a reference point. Indeed, the other lines involve either χ_b or φ_b, which we have so far assumed to be equivalent to an `S` state, meaning that the results ought to have been the same in all the lines of this first table.

The first surprising result is that we consistently observed twice the amount of expected L2D accesses. While it is odd, we do not consider it to be a sufficient contradiction of our proposed definition, as it holds true for every single one of our benchmarks.

Much more interesting is the hint of a truly unexpected behavior found in the upper table, where the $\langle I_b, I_b, \varphi_b \rangle$ benchmarks is performed using less CPU cycles than the others. Looking at what happens on the bottom table for the symmetrical line, we can see that the

cache holding the memory elements in the $\varphi_{\mathtt{b}}$ is actually providing them to the demanding cluster. This is in clear contradiction with our understanding of the architecture's protocol. Furthermore, this is not simply a case of having a different behavior for what should be the `S` state: the $\langle \chi_{\mathtt{b}}, \mathtt{I}_{\mathtt{b}}, \mathtt{I}_{\mathtt{b}} \rangle$ line of the bottom table indicates that no such thing is happening for memory elements in the $\chi_{\mathtt{b}}$ state. This allows us conclude that $\varphi_{\mathtt{b}}$ and $\chi_{\mathtt{b}}$ are, in fact, two completely separate stable states. This confirms that the NXP QorIQ T4240 architecture does not use MESI as its coherence protocol.

6 Formal MESIF Description

From the observations we made, we believe the implemented protocol to be MESIF. Table 5 shows our formal definition of the MESIF protocol.

The MESIF protocol [17] adds a *Forward* stable state. This state is equivalent to a *Shared* state with the added constraint of being responsible for the propagation of the memory element's current value. Thus making it possible to avoid reading from the system's main memory even when multiple caches hold the same memory element. Unlike the *Exclusive* state, it does not allow the cache to upgrade to a *Modified* state by itself, since the other caches still have to be informed that their copies are out-of-date.

As with any stable state that gives a cache the responsibility of propagating the memory element's current value, the challenge lies in determining when a cache can enter that state, and making sure that the responsibility is properly transferred when the cache leaves it.

The coherence manager keeps track of which cache holds memory elements in the *Forward* state. As this cache cannot actually make modifications while in this state, informing the coherence manager that it was left does not require sending any kind of `data` message: a simple `PutM` query broadcast is sufficient.

A cache moving from *Forward* to *Modified* still has to broadcast a `GetM` query and process all the queries that preceded before proceeding. We assume that if the cache still is responsible for the propagation of the memory element when it sees its own `GetM` query (meaning that it stayed in the $\mathtt{FM}^{\mathtt{B}}$ state), then it should be able to simply move to the *Modified* state without receiving any `data` reply. However, if the responsibility was lost (because of either an external `GetS` or `GetM` query), then it will need to re-acquire the current value of the memory element as a `data` reply before entering the *Modified* state.

7 Validating MESIF on the T4240

We apply again our validation strategy, this time with the MESIF protocol. First, we match the observable states with the stables: we now identify $\varphi_{\mathtt{b}}$ as corresponding to the `F` state, making the name $\mathtt{F}_{\mathtt{b}}$ more appropriate. Likewise, the $\chi_{\mathtt{b}}$ state is now named $\mathtt{S}_{\mathtt{b}}$, as it does appear to correspond to the `S` state.

Overall, our results confirm a MESIF protocol, albeit differing in some of the implementation choices. For the sake of brevity, we omitted all the results that were exactly as expected.

No `store` Optimization on `F`

Our MESIF protocol formalization considers that performing a `store` on `F` does not require a `data` reply if no other query occurs simultaneously, since that particular cache is the one in charge of distributing the value. However, the performance monitors on the T4240 show that the memory elements were actually received again (CoreNet Reloads) and that the `F`

Table 5 Description of the MESIF protocol.

Cache Controller									
State	Core Request			Interconnect Access	Data Reply		Received Queries		
	load	store	evict		data	data-e	GetS	GetM	PutM
I	GetS?, IF^{BD}	GetM?, IM^{BD}	hit				-	-	-
IF^{BD}	stall	stall	stall	$IEoF^{D}$	IF^{B}	IE^{B}	-	-	-
IF^{B}	stall	stall	stall	F			-	-	
$IEoF^{D}$	stall	stall	stall		F	E	r←s, IS^{D}	r←s, $IS^{D}I$	
IS^{D}	stall	stall	stall		r!data, r← ∅, S	r!data, m!no-data, r← ∅, S	-	$IS^{D}I$	
$IS^{D}I$	stall	stall	stall		load hit, r!data, r← ∅, I	load hit, r!data, r← ∅, m!no-data, 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		store hit, r!data, r← ∅, I		-	-	
$IM^{D}S$	stall	stall	stall		store hit, r!data, m!data, r← ∅, S		-	$IM^{D}SI$	
$IM^{D}SI$	stall	stall	stall		store hit, r!data, m!data, r← ∅, I		-	-	
S	hit	GetM?, SM^{BD}	hit, I				-	I	
F	hit	GetM?, FM^{B}	PutM?, FI^{B}				s!data, S	s!data, I	
SM^{BD}	hit	stall	stall	SM^{D}	SM^{B}		-	IM^{BD}	
FM^{B}	hit	stall	stall	M			s!data, SM^{BD}	s!data, IM^{B}	
SM^{B}	hit	stall	stall	M			-	IM^{B}	
SM^{D}	hit	stall	stall		store hit, M		r←s, $SM^{D}S$	r←s, $SM^{D}I$	
$SM^{D}I$	hit	stall	stall		store hit, r!data, r← ∅, I		-	-	
$SM^{D}S$	hit	stall	stall		store hit, r!data, m!data, r← ∅, S		-	$SM^{D}SI$	
$SM^{D}SI$	hit	stall	stall		store hit, r!data, m!data, r← ∅, 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			-	-	-
E	hit	hit, M	PutM?, EI^{B}				m!no-data, s!data, S	s!data, I	
IE^{B}	stall	stall	stall	E			-	-	-
EI^{B}	hit	stall	stall	m!no-data, I			m!no-data, s!data, II^{B}	s!data, II^{B}	
FI^{B}	hit	stall	stall	I			s!data, II^{B}	s!data, II^{B}	

Coherence Manager						
State	Received Queries				Data Reply	
	GetS	GetM	PutM (Owner)	PutM (Other)	data	no-data
I	read, s!data-e, r←s, M	s!data, r←s, M		-		
M	r←s, F^{D}	r←s	r← ∅, I^{D}	-	write, IoF^{B}	IoF^{B}
I^{D}	stall	stall	stall	-	write, resume, I	resume, I
F^{D}	stall	stall	stall	-	write, resume, F	resume, F
IoF^{B}	r←s, F	r←s, M	r← ∅, I	-	write	-
S	read, s!data, F	s!data, r←s, M		-		
F	r←s	r←s, M	r← ∅, S	-	write, IoF^{B}	IoF^{B}

cache is not sending them to itself (Snoop Pushes). This may be a standard implementation choice for MESIF, and exactly the kind we believe important for the architecture's user to know about.

	⟨store,-,-⟩	
Origin	**Behavior**	
	Expected	**Observed**
⟨E_b,I_b,I_b⟩	8000 L2D Accesses	16000 L2D Accesses, 248532 CPU Cycles
⟨F_b,I_b,I_b⟩	8000 L2D Accesses	16000 L2D Accesses, 8000 CoreNet Reloads, 252900 CPU Cycles

Odd Results with `evict` on `M`

Eviction from `M` yielded surprising results. Indeed, if not for the absence of any External Snoop Requests, these values are what one would expect to see when a cache in the `M` state sees another cache's `GetM` query. The number of L2D Accesses are not significant in this benchmark since, as previously indicated, we do not perform separate `evict` operations on each memory element but rather a general eviction of that particular cache.

	⟨evict, -, -⟩	
Origin	**Behavior**	
	Expected	**Observed**
⟨E_b,I_b,I_b⟩	8000 L2D Accesses	42 L2D Accesses, 22400 CPU Cycles
⟨M_b, I_b, I_b⟩	8000 L2D Accesses, 8000 Snoop Pushes	42 L2D Accesses, 8000 Snoop Hits, 8000 Snoop Pushes, 8000 MINTs, 65700 CPU Cycles

Better Coherence Manager

While we are unable to see the coherence manager, we still tried to expose the issue we mentioned in Example 2, where the coherence manager is unable to grant the Exclusive state when all caches evicted from `S`. As it happens, our benchmark showed that the Exclusive was indeed reached, pointing to either a better coherence manager being used, or some other co-ordination strategy.

Simultaneous Events Behaviors

Considering the limited control and observation points available to us on the platform, performing benchmark to validate the simultaneous events behaviors is particularly difficult. We have observed multiple observable states for a same combination of multiple events as expected but we are unable to detect whether the transactions that fulfilled the request of each core interlaced or if they were simply resolved in a sequence. In the latter case, all the behaviors correspond to single event ones instead. Furthermore, the possibility of an optimization being present on the architecture for certain scenarios is hardly detectable.

8 Related Works

Cache Coherence Error Detection

[11] proposes the detection of design issues in the architecture by automatically generating and performing tests on a simulation of that architecture. Indeed, while the protocol itself may be correct, its implementation and interaction with other components can still be a source of issues. In effect, this also performs a validation of the protocol on the architecture through tests, but it requires a valid model of the architecture to already be available. [20] also proposes a framework for automated test generation, this time focused on validating memory controllers according to models of the behaviors they are supposed to follow.

A number of papers propose the inclusion of hardware to implement redundant coherency mechanisms which are continuously compared with the primary ones. [6] is such a paper: it runs a simplified (stable states only) coherence protocol alongside the real one, reacting to every query and instruction. It detects local errors by comparing the cache line states according to the simplified protocol with their states according to the real protocol. It also detects errors related to the system entering an invalid coherence state by having each cache broadcast its state according to the simplified protocol so that the others can react if it is incompatible with theirs (e.g. a cache in the `M` state seeing the broadcast of another cache signaling they entered `M` as well). [33]'s solution is similar, with the exception that the states are not broadcasted. Instead, a centralized checking unit simply accesses them to check whether the system entered an invalid coherence state. In terms of detected mismatches, this is the equivalent of continuously performing the flags matching step of our strategy.

[22] proposes keeping a short backlog of relevant coherence mechanisms information (state of outgoing/incoming messages, state of cache lines) at each cycle. The information captured at each cycle is first studied in isolation, using invariants to check that the protocol would indeed allow the system to reach such a state. Then, the protocol is applied to the system state that was logged at each cycle to check that the result is compatible with the system state that was logged in the next cycle. [10] presents CoSMa, another solution making use of a backlog, but this time it is stored within the caches themselves instead of a separate component. The system periodically stops its activities to go perform a coherence check and detect if any error have occurred. The authors point out that this is not meant to be ran in production, but instead as a post-silicon validation process, which should be done prior to the product's release. Compared to the approaches from the previous paragraph, these two go further, by including the validation of behaviors.

Cache Coherence Profiling & Modeling

By successfully validating that the formally defined protocol indeed matches what is implemented on the architecture, a model of part of the platform can be created so as to verify properties relevant to the user. An important such property being the WCET, and its computation for multi-core systems is difficult and the subject of many publications. [25] provides a general survey for WCET in multi-core systems, and [24] provides another survey, this time focused on caches.

Much like our own approach, [4] and [16] make use of benchmarks and monitors to learn the characteristics of a given architecture. Their focus is on exposing unexpectedly shared resources by overwhelming them through stress testing.

We ourselves have used timed automata for the modeling of platform and program in [30], with a focus on the identification of interferences (negative impact caused by external queries). Timed automata were also used in models aimed at WCET computation [8, 19].

WCET is also strongly impacted by cache eviction policies, which we did not address in this paper. Solutions to analyze the impact of cache eviction are plenty [15, 18, 23, 32], especially since this problematic predates multicore processors.

To ease WCET computation, some cache coherence protocols are designed to be predictable in their impact on runtime, such as [21]. This does not, however, remove the need for the coherence protocol implementation to be properly identified and validated. Another way to make WCET computation easier is to limit what is being affected by cache coherence. For example, [7] leverages sensible scheduling so that parts of the program that require access to the bus are less likely to be happening simultaneously; [27] also makes use of careful scheduling, this time so that tasks can simply leave their results in cache so that it will be used by the next task without needing to be fetched; [2] suggests making use of the platform capabilities to better control what is kept in what cache, and what should be affected by coherence mechanisms.

Even if not interested in an easy to predict WCET, good understanding of the impact of cache coherence can be used to improve performance and/or reduce wasteful operations. [3] adds hardware that will consider each cache line as being write-through or write-back depending on what is preferable.

9 Conclusion

In this paper we presented a strategy to validate the user's understanding of the cache coherence mechanisms implemented on an architecture. To illustrate our process, we applied it to the NXP QorIQ T4240 architecture, which we understood to be running MESI. We thus proposed a formal definition for a split-transaction bus MESI protocol, which we tried to validate using the aforementioned process. To our surprise, where we expected to only see differences in implementation choices, we learned that the architecture is in fact implementing MESIF. We validated this by proposing a formal definition for that protocol and re-applying the process a second time. This time, the results indicated a match, with the exception of a few implementation choices.

In the future, we will make measure on more temporal behavior from the NXP QorIQ T4240 architecture relative to the cache coherence to quantify the impact induced by cache coherence on software running on the cores. We will also extend our Uppaal model from [30] to integrate several cache protocol, including MESIF, and to use real delay values so as to be able to offer the formal model of a validated system.

References

1 Jyotika Athavale, Riccardo Mariani, and Michael Paulitsch. Flight safety certification implications for complex multi-core processor based avionics systems. In *25th IEEE International Symposium on On-Line Testing and Robust System Design, IOLTS 2019, Rhodes, Greece, July 1-3, 2019*, pages 38–39, 2019.

2 Ayoosh Bansal, Jayati Singh, Yifan Hao, Jen-Yang Wen, Renato Mancuso, and Marco Caccamo. Cache where you want! reconciling predictability and coherent caching, 2019. `arXiv:1909.05349`.

3 Pedro Benedicte, Carles Hernandez, Jaume Abella, and Francisco J. Cazorla. HWP: Hardware Support to Reconcile Cache Energy, Complexity, Performance and WCET Estimates in Multicore Real-Time Systems. In *30th Euromicro Conference on Real-Time Systems (ECRTS 2018)*, volume 106 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 3:1–3:22, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/LIPIcs.ECRTS.2018.3`.

4 Jingyi Bin, Sylvain Girbal, Daniel Gracia Perez, Arnaud Grasset, and Alain Merigot. Studying co-running avionic real-time applications on multi-core cots architectures. In *Embedded Real Time Software and System Conference (ERTS2)*, February 2014.

5 Frédéric Boniol, Youcef Bouchebaba, Julien Brunel, Kevin Delmas, Thomas Loquen, Alfonso Mascarenas Gonzalez, Claire Pagetti, Thomas Polacsek, and Nathanaël Sensfelder. PHYLOG certification methodology: a sane way to embed multi-core processors. In *10th European Congress on Embedded Real Time Software and Systems (ERTS 2020)*, Toulouse, France, January 2020. URL: `https://hal.archives-ouvertes.fr/hal-02441323`.

6 Jason F. Cantin, Mikko H. Lipasti, and James E. Smith. *Dynamic Verification of Cache Coherence Protocols*, pages 25–42. Springer New York, New York, NY, 2004. `doi:10.1007/978-1-4419-8987-1_3`.

7 Thomas Carle and Hugues Cassé. Reducing Timing Interferences in Real-Time Applications Running on Multicore Architectures. In *18th International Workshop on Worst-Case Execution Time Analysis (WCET 2018)*, volume 63 of *OpenAccess Series in Informatics (OASIcs)*, pages 3:1–3:12, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/OASIcs.WCET.2018.3`.

8 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`.

9 Certification Authorities Software Team. Multi-core Processors - Position Paper. Technical Report CAST 32-A, Federal Aviation Administration, November 2016.

10 A. DeOrio, A. Bauserman, and V. Bertacco. Post-silicon verification for cache coherence. In *2008 IEEE International Conference on Computer Design*, pages 348–355, October 2008. `doi:10.1109/ICCD.2008.4751884`.

11 M. Elver and V. Nagarajan. Mcversi: A test generation framework for fast memory consistency verification in simulation. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 618–630, March 2016. `doi:10.1109/HPCA.2016.7446099`.

12 Hakan Forsberg and Andreas Schwierz. Emerging cots-based computing platforms in avionics need a new assurance concept. In *the 38th Digital Avionics Systems Conference (DASC'19)*. IEEE Press, 2019.

13 Freescale. e6500 core reference manual, rev 0, 2014.

14 Freescale. T4240 QorIQ: Integrated multicore communications processor family reference manual, 2014.

15 Michele Garetto, Emilio Leonardi, and Valentina Martina. A unified approach to the performance analysis of caching systems. *ACM Trans. Model. Perform. Eval. Comput. Syst.*, 1(3), May 2016. `doi:10.1145/2896380`.

16 Sylvain Girbal, Jimmy le Rhun, and Hadi Saoud. METrICS: a measurement environment for multi-core time critical systems. In *9th European Congress on Embedded Real Time Software and Systems (ERTS'18)*, 2018.

17 James Goodman and Hhj Hum. Mesif: A two-hop cache coherency protocol for point-to-point interconnects (2004), 2004.

18 Daniel Grund and Jan Reineke. Toward Precise PLRU Cache Analysis. In *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, volume 15 of *OpenAccess Series in Informatics (OASIcs)*, pages 23–35, 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.2010.23`.

19 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`.

20 M. Hassan and H. Patel. Mcxplore: Automating the validation process of dram memory controller designs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(5):1050–1063, May 2018. doi:10.1109/TCAD.2017.2705123.

21 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.

22 B. Kumar, A. K. Bhosale, M. Fujita, and V. Singh. Validating multi-processor cache coherence mechanisms under diminished observability. In *2019 IEEE 28th Asian Test Symposium (ATS)*, pages 99–995, 2019.

23 Benjamin Lesage, David Griffin, Sebastian Altmeyer, Liliana Cucu-Grosjean, and Robert I. Davis. On the analysis of random replacement caches using static probabilistic timing methods for multi-path programs. *Real-Time Syst.*, 54(2):307–388, April 2018. doi:10.1007/s11241-017-9295-2.

24 Mingsong Lv, Nan Guan, Jan Reineke, Reinhard Wilhelm, and Wang Yi. A survey on static cache analysis for real-time systems. *Leibniz Transactions on Embedded Systems*, 3(1):05–1–05:48, 2016. doi:10.4230/LITES-v003-i001-a005.

25 Claire Maiza, Hamza Rihani, Juan M. Rivas, Joël Goossens, Sebastian Altmeyer, and Robert I. Davis. A survey of timing verification techniques for multi-core real-time systems. *ACM Comput. Surv.*, 52(3), June 2019. doi:10.1145/3323212.

26 Laurence Mutuel, Xavier Jean, Vincent Brindejonc, Anthony Roger, Thomas Megel, and E. Alepins. Assurance of Multicore Processors in Airborne Systems, 2017.

27 Viet Anh Nguyen, Damien Hardy, and Isabelle Puaut. Cache-conscious Off-Line Real-Time Scheduling for Multi-Core Platforms: Algorithms and Implementation. *Real-Time Systems*, pages 1–37, 2019. doi:10.4230/LIPIcs.ECRTS.2017.14.

28 Jan Nowotsch and Michael Paulitsch. Leveraging multi-core computing architectures in avionics. In *Proceedings of the 2012 Ninth European Dependable Computing Conference*, EDCC '12, pages 132–143, Washington, DC, USA, 2012. IEEE Computer Society.

29 Mark S. Papamarcos and Janak H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. *SIGARCH Comput. Archit. News*, 12(3):348–354, January 1984. doi:10.1145/773453.808204.

30 Nathanaël Sensfelder, Julien Brunel, and Claire Pagetti. Modeling Cache Coherence to Expose Interference. In *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*, volume 133 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 18:1–18:22, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.ECRTS.2019.18.

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

32 Valentin Touzeau, Claire Maïza, David Monniaux, and Jan Reineke. Fast and exact analysis for lru caches. *Proc. ACM Program. Lang.*, 3(POPL), January 2019. doi:10.1145/3290367.

33 Hui Wang, Sandeep Baldawa, and Rama Sangireddy. Dynamic error detection for dependable cache coherency in multicore architectures. *21st International Conference on VLSI Design (VLSID 2008)*, pages 279–285, 2008.

34 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.