



HAL
open science

Modeling Cache Coherence to Expose Interference

Nathanaël Sensfelder, Julien Brunel, Claire Pagetti

► **To cite this version:**

Nathanaël Sensfelder, Julien Brunel, Claire Pagetti. Modeling Cache Coherence to Expose Interference. ECRTS 2019, Jul 2019, Stuttgart, Germany. <10.4230/LIPICs.ECRTS.2019.18>. <hal-02165139>

HAL Id: hal-02165139

<https://hal.science/hal-02165139v1>

Submitted on 25 Jun 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Distributed under a Creative Commons CC BY 4.0 - Attribution - International License

Modeling Cache Coherence to Expose Interference

Nathanaël Sensfelder

ONERA, France

Julien Brunel

ONERA, France

Claire Pagetti

ONERA, 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.2019.26

Supplement Material <https://www.onera.fr/sites/default/files/598/ecrts19.zip>

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.

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



© Nathanaël Sensfelder and Julien Brunel and Claire Pagetti;
licensed under Creative Commons License CC-BY

31st Euromicro Conference on Real-Time Systems (ECRTS 2019).

Editor: Sophie Quinton; Article No. 26; pp. 26:1–26:22

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

26:2 Modeling Cache Coherence to Expose Interference

43 hindrance that we qualify of *implicit* interference, because of how difficult to predict they
44 are. Additionally, it can result in the removal of elements from the cache, which may lead to
45 time consuming communications with the system's memory (*cache misses*).

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

50 **Current Research Practices** Several approaches have been developed in the literature to
51 deal with the interference found in multi-core processors. The main solutions to ensure
52 predictability are 1) preventing any kind of uncontrolled interference (e.g. run-time services
53 [15, 28]); 2) enforcing a unique access to any shared resource at any time, so as to be equivalent
54 to a single core situation (e.g. execution models [18, 5, 12]). Because its interference is difficult
55 to predict, most of the considered hardware do not have or use automatic cache coherency.
56 Instead, the burden of cache management is placed on the developers, forcing an application-
57 specific solution (e.g. *scratchpad memory* [25, 22]). Such solutions prevent the gains in
58 performance that would otherwise be provided by automatic hardware cache management
59 mechanisms.

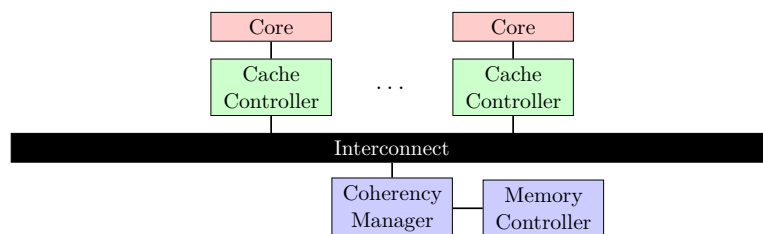
60 **Contributions** We believe that the implicit interference generated by the cache coherence
61 can be exposed and taken into account to achieve predictable programming of a multi-core
62 processor. In this work, we focus on exposing these unexpected delays, the analysis of a
63 formal model of the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

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

71 2 Cache Coherence Protocols

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

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



■ **Figure 1** Components involved in cache coherence

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

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

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

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

108 **Memory Controller** Component that handles the modification or copy of the original memory
 109 elements.

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

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

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

119 Each message transiting through the interconnect, and each cache controller query, is
 120 about a specific memory element. Upon receiving either one of those, cache controllers look

121 up the state they associate with their copy of the memory element for this address, and act
 122 according to the cache coherence protocol.

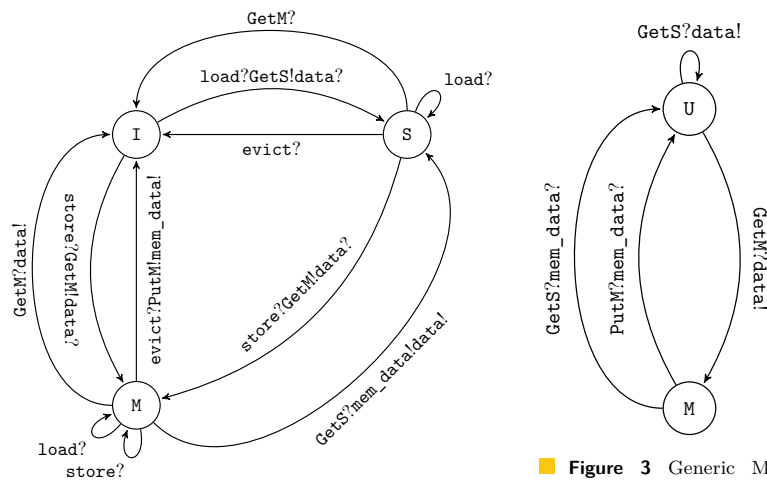
123 **2.1 Protocols**

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

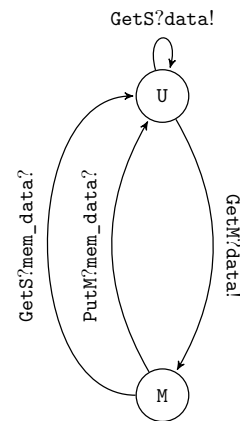
129 MSI-based protocols are all categorized as *Write-Back*, because caches may contain a
 130 more up-to-date value of the memory element than the RAM.

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

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



■ **Figure 2** Generic MSI Cache Controller



■ **Figure 3** Generic MSI Coherency Manager

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

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

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

153 **3 MSI Snooping-Based Protocol**

154 **3.1 A Few Caveats**

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

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

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

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

167 **3.2 From Abstract to Concrete Behaviour**

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

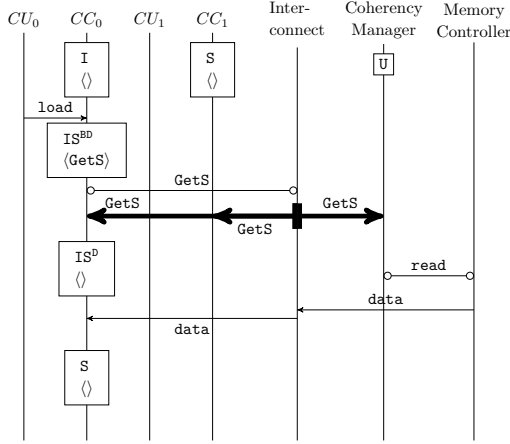
174 In the sequence diagram of Figure 4, we see the behavior of all components involved.
 175 Once the core issues the `load` request, the cache controller generates a `GetS` query to the
 176 interconnect. The latter broadcasts the `GetS` to all cache controllers, including the query's
 177 originator,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

180 In order to expose the interference, we need to model the atomic transitions and interme-
 181 diate states, such as the ones shown in the figure.

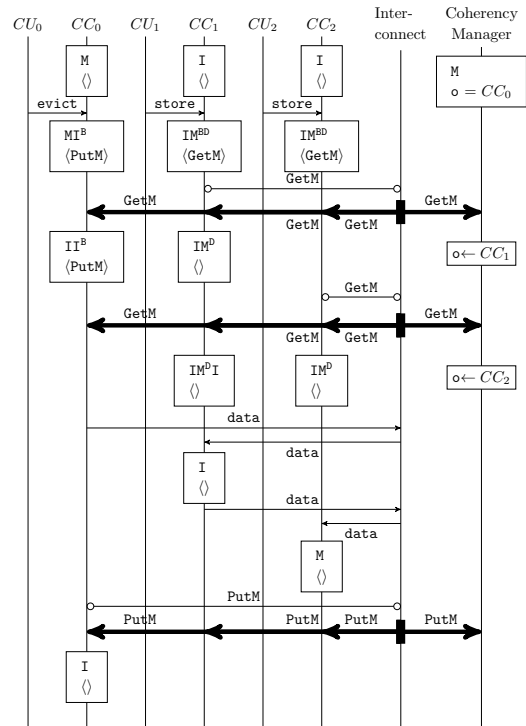
182 **3.3 Detailed Snooping-Based MSI Protocol**

183 Instead of representing the full automaton as a graph, we use a matrix representation (see
 184 Figure 6). The first column details every possible states. As in [20], the naming of each
 185 state is determined by the following reasoning: Invalid (I), Shared (S), and Modified (M) are
 186 the three *stable* states of the MSI protocol. The other states are *transient*. Reception of a
 187 request that requires use of the interconnect will usually lead to a XY^{BD} transient state, which
 188 means that the cache controller is handling a transition between the stable states X and Y,
 189 with $(^B)$ indicating that this transition requires the acquisition of the interconnect and $(^D)$
 190 the reception of a related data reply (whether it comes from an other cache controller or

26:6 Modeling Cache Coherence to Expose Interference



■ Figure 4 load request



■ Figure 5 Double store

191 the memory). This can be followed by XY^D if the cache controller sees its own query before
 192 receiving a reply, or XY^B if a reply is received before the query is processed. This happens
 193 when, despite processing all queries in the same order, not all cache controllers take the same
 194 time to do so. Another possibility is for an external query to be received when in the XY^D
 195 state. Indeed, at that point, the system pretty much considers that the cache controller is in
 196 the Y state and thus has the responsibilities that the Y would require. This makes it possible
 197 for a cache controller to see a query it needs to act upon before being actually ready to do so
 198 (e.g. observing a $GetM$ query while waiting for data). These states have a XY^DA form (which
 199 means that when all is handled, the cache controller ends up in the A state), or XY^DAB (which
 200 ends up leading to the B state). As it may be that the required action is to reply to said
 201 query, it is sometimes necessary to remember the originator of the query. This is marked as
 202 $r \leftarrow s$.

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

210 The third macro-column (named *Interconnect access*) indicates what happens when the
 211 previously queued query is broadcasted on the interconnect. When in the IS^{BD} state, we
 212 know that, at some point, our previously queued $GetS$ query is going to be broadcasted on
 213 the interconnect. This will result in reaching the IS^D state. As a side note, if the core makes
 214 a second load request on the same memory element while the copy is IS^{BD} , that new request

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])

215 is stalled.

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

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

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

225

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

230 that its value is out of date (state M).

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

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

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

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

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

252 CC_2 receives the data and completes CU_2 ’s request.

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

256 **3.4 Coherency Manager**

State	Received Queries				Data Reply data
	GetS	GetM	PutM (Owner)	PutM (Other)	
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])

257 Figure 7 shows how the coherency manager keeps track of whether the RAM has the
 258 most up-to-date value for a memory element (state U) or if a cache controller does (state M).
 259 This is used to determine if the RAM should be the one to reply when either a *GetS* or a
 260 *GetM* query passes through the interconnect. The U state indicates that the RAM currently
 261 has the most up-to-date value. The U^D state indicates that the RAM should be the one to
 262 respond to queries, but it still hasn’t received the latest value. Unlike the cache controller, it
 263 will not switch to a dedicated state but instead force queries from the interconnect to stall

264 until the problematic query can be fulfilled. U^B indicates that the RAM has received the
 265 latest value, but has not yet seen the query that led this data to be sent.

266 The exact cache controller currently in charge of the memory element is kept track of.
 267 Change of ownership are marked as $o \leftarrow s$ (the query originator becomes the new owner) and
 268 $o \leftarrow \emptyset$ (there is no longer an owner, meaning that the RAM is currently responsible for it).

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

277 **4 Interference**

State	Received Queries		
	GetS	GetM	PutM
I	Mi.	Mi.	Mi.
IS ^{BD}	Mi.	Mi.	Mi.
IS ^B	Mi.	Mi.	
IS ^U	Mi.	Ex.	
IS ^{UI}	Mi.	Mi.	
IM ^{BD}	Mi.	Mi.	Mi.
IM ^B	Mi.	Mi.	Mi.
IM ^U	De.	Ex.	
IM ^{UI}	Mi.	Mi.	
IM ^S	Mi.	Ex.	
IM ^{SI}	Mi.	Mi.	
SM ^{BD}	Mi.	Ex.	
SM ^B	Mi.	Mi.	
SM ^U	De.	Ex.	
SM ^{UI}	Mi.	Mi.	
SM ^S	Mi.	Ex.	
SM ^{SI}	Mi.	Mi.	
M	De.	Ex.	
MT ^B	Ex.	Ex.	
IT ^B	Mi.	Mi.	Mi.

Minor

Expelling

Demoting

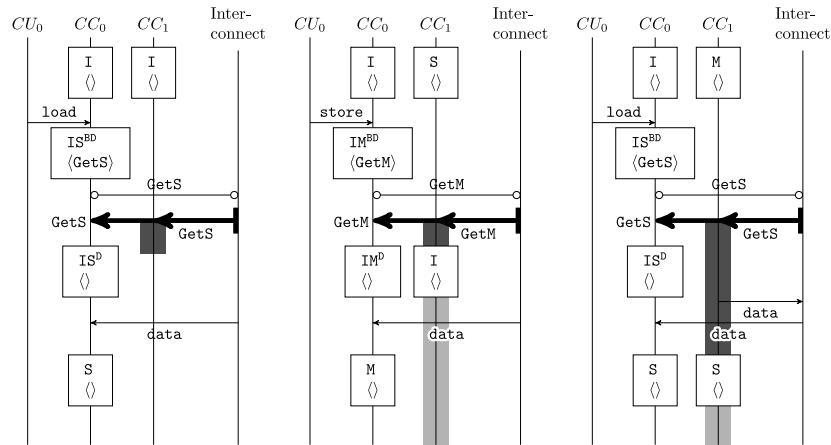
■ **Figure 8** Occurrences of Interference

278 Let us now categorize how a cache controller may be negatively affected by the actions of
 279 another. Figure 8 summarizes the occurrences of each interference category. In the Figures 9,
 280 11, and 10, the dark gray area indicates when the cache controller is unavailable due to
 281 having to handle the incoming query (deciding how to act and, potentially, updating its
 282 internal state), and the light gray area shows when its core's next request for that memory
 283 element may be negatively impacted by the change of state.

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

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

26:10 Modeling Cache Coherence to Expose Interference



■ Figure 9 Minor

■ Figure 10 Expelling

■ Figure 11 Demoting

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

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

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

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

331 5 Formal Modeling of Real-Time Systems with Timed Automata

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

334 A *timed automaton* is an extended automaton with variables and clocks. During the
 335 system's execution, the state of timed automaton is defined as a location, the value of its
 336 integer variables and of its clocks. The evolution of these integer variables is controlled
 337 by the automaton's transitions, whereas all of the system's clocks progress at the same
 338 rate, following the passing of time. To indicate that a location should be left immediately,
 339 UPPAAL [4] offers the following location modifiers:

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

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

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

327 The automata transitions are composed of the following:

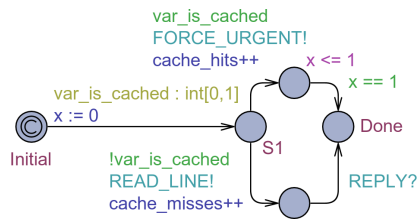
328 **Guard**: Prerequisite (linear constraint) for this transition to be able to *fire*. The condition
 329 uses the automaton’s integer variables, clocks, or both.

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

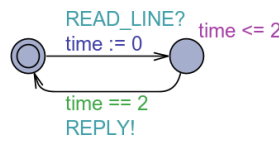
339 **Update**: Sequence of instructions to alter the automaton’s integer variables, or reset its
 340 clocks.

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

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



■ Figure 12 Core and cache



■ Figure 13 RAM



■ Figure 14 Ur-
gence

348 **The Core Automaton (Figure 12)** Its initial location is marked as committed, meaning
 349 that it is left immediately. The exiting transition sets the x clock to 0, and the var_is_cached
 350 variable to a value in the $[0, 1]$ range. The x clock will be used to know how long it took for
 351 the processor to get its variable. Two transitions are fireable from the $S1$ location, depending
 352 on whether the targeted variable is cached or not. If it is indeed cached, the transition
 353 labeled **FORCE_URGENT** is the only one fireable and it synchronizes with the automaton of
 354 Figure 14, forcing it to be taken as soon as possible. Additionally, the transition increases an
 355 integer variable that counts the number of times a variable was found in the cache. Taking
 356 said transition leads to a location in which the only exiting transition requires the x clock to
 357 equal 1 unit of time before arriving in the *Done* location.

358 If the variable was not in the cache, the other transition from $S1$ is active and leads
 359 to a synchronization on the **READ_LINE** which is also to be taken as soon as possible. This
 360 time, however, it is possible for that synchronization to not be immediately available, as
 361 the RAM controller automaton may be handling another query and thus not be ready to

26:12 Modeling Cache Coherence to Expose Interference

362 synchronize as it would not be in its initial location. This also justifies not marking the
363 location as urgent or committed: the automaton may have to wait an unknown amount of
364 time. Once the synchronization does happen, an integer variable counting the number of
365 times the variable was not found in the cache is incremented, then the automaton waits for
366 the RAM automaton to synchronize on the `REPLY` channel before considering it has acquired
367 the variable.

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

376 **6 Model of the Cache Coherence**

377 This sections describes the general ideas behind how we modeled the cache coherence in
378 UPPAAL. We have released the model under an LGPL v3 license at <https://www.onera.fr/sites/default/files/598/ecrts19.zip>.
379

380 **6.1 Modeling Strategy**

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

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

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

400 **6.2 Core**

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

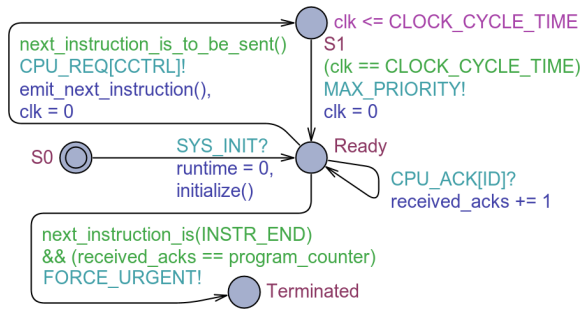


Figure 15 Model of the Core

```

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

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

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

6.3 Coherency Manager and Memory Controller

415

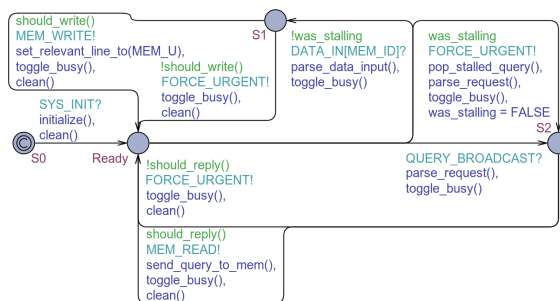


Figure 17 Model of the Coherency Manager

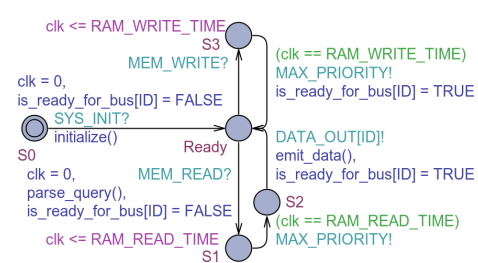


Figure 18 Model of the Memory Controller

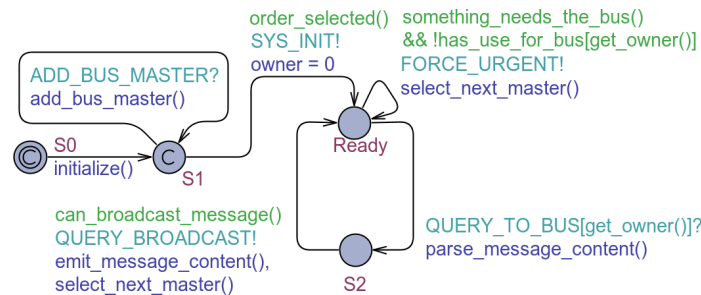
416 The timed automaton modeling the coherency manager can be seen on Figure 17. The
 417 coherency manager has to know for which memory elements the RAM copy is to be considered
 418 as superseded by a cache controller. For this purpose, it maintains an array associating a
 419 state to each memory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

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

26:14 Modeling Cache Coherence to Expose Interference

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

433 6.4 Interconnect



■ **Figure 19** Model of the Interconnect

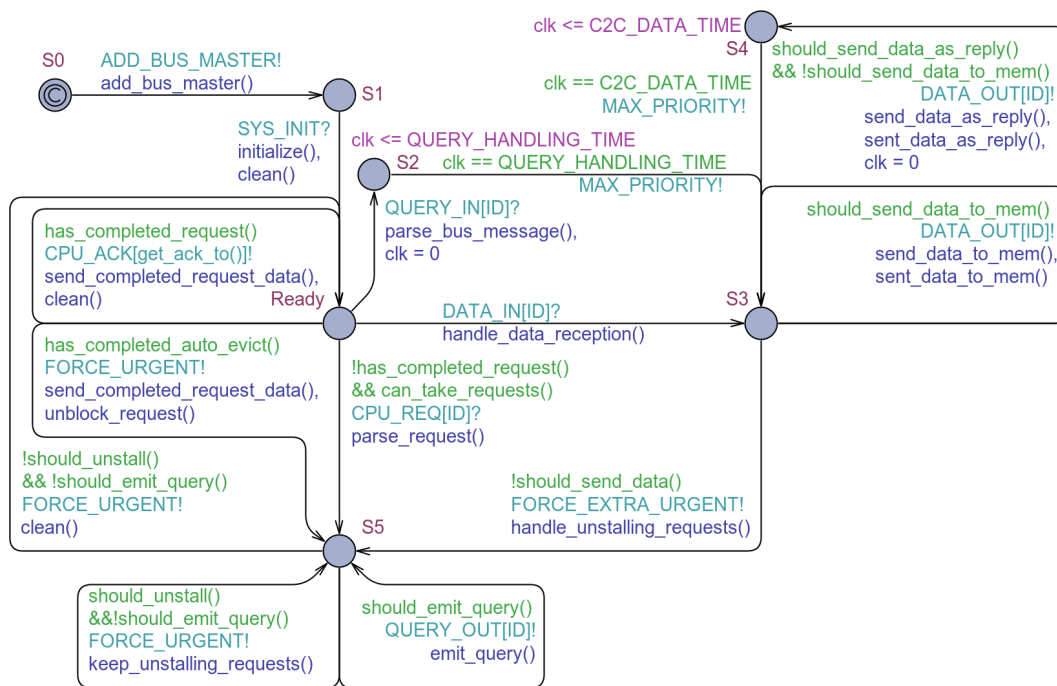
434 Figure 19 shows the timed automaton for the interconnect. It starts (`S1`) by waiting for
 435 cache controllers to synchronize through the `ADD_BUS_MASTER` so that they can be added to
 436 the bus policy. The order in which the cache controllers make that synchronization is not
 437 deterministic.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

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

449 6.5 Cache Controller

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

456 **CPU Communications** Each cache controller has a queue of outstanding requests from its
 457 core, as well as a queue of completed requests to inform the core of. Both queues are first in,



■ **Figure 20** Model of the Cache Controller

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

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

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

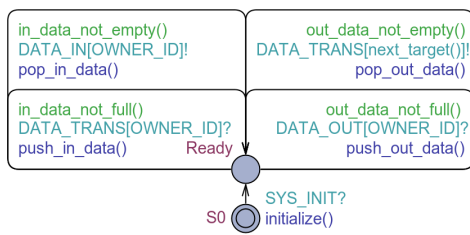
476 **stall:** the request is put in the outstanding requests queue, if it is not already there. The
 477 handling of stalled requests is stopped.

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

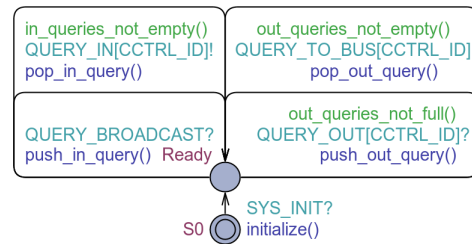
26:16 Modeling Cache Coherence to Expose Interference

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

491 6.6 Message Queues



■ **Figure 21** Model of the Data FIFOs



■ **Figure 22** Model of the Query FIFOs

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

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

504 7 Checking Properties

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

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

```

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 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

26:18 Modeling Cache Coherence to Expose Interference

548 application with accesses to new variables, setting the time impact of minor interferences
549 to nil, and having the framework compute the new maximum execution time so that it
550 can be compared to the one with shared variables left intact.

551 On the dual core example, we first measure the execution time with the system as is,
552 then replace the program running on one of the two cores by Figure 24 and set the cost
553 of minor interferences to zero. Our first analysis indicates a maximum execution time
554 of 1602 time units, the second one indicates 1050 time units. This implies that cache
555 coherence causes a 16 percent increase in execution time.

556 Alternatively, by keeping the time impact of minor interferences to its default value,
557 a WCET of these two programs lets us deduce how much time is lost due to minor
558 interferences. In the dual core example, the result is still 1050 time units, showing a lack
559 of negative effects from minor interference.

560 7.2 Model Validation

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

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

578 8 Related Works

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

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

598 ■ **WCET Analysis for Multi-Core with Private Caches:** Readers can refer to [17] for
599 an overview of Multi-Core WCET Analysis. [16], proposes a UPPAAL-based framework
600 to estimate the WCET of applications running on a multi-core processor. They consider
601 the delays caused by contention on the interconnect and a private instruction cache for
602 each core (data caches are not considered). They perform analysis on the memory blocks
603 pertinent to the instructions of the program. A memory block may contain one or more
604 instruction. For each instruction, they are only interested in whether it: is always found
605 in the cache; is always found except on the first access; is never found in the cache; is
606 undecided. They have defined a timed automaton to model each of these possibilities
607 (modeling the need for interconnect access, time to read the memory blocks, and updates
608 to the cache). They consider programs as control flow graphs in which each node is a
609 memory block. As such, they model each program by a single timed automaton based
610 on the control flow graph, but in which each instruction has been replaced by one of
611 the aforementioned timed automata corresponding to its impact. Their paper presents
612 models for two types of interconnects: TDMA and FCFS, which control the order the
613 bus can be accessed by the timed automata modeling the instructions. Cache coherence
614 is not addressed.

615 ■ **WCET Analysis for Multi-Core with Shared Caches:** The authors of [8] focus
616 on the estimation of WCET on multi-core processors. Their point of interests are the
617 delays caused by hierarchical caches, the use of a shared cache, and the interconnect.
618 They do not use UPPAAL, but instead model the applications as task-dependency graphs
619 and perform computations to estimate the WCET. Their approach starts by analyzing
620 how the L1 caches are accessed, to remove elements that are sure to always be present
621 from further consideration. The other accesses are dependent on both the content of the
622 L2 cache, and access to the bus. The content of the L2 cache depends on which tasks
623 are running, which in turns, depends on bus access time access. To resolve the circular
624 dependency, they propose an iterative approach: starting by considering the worse case
625 scenario in which all tasks interfere, they estimate the running time of the tasks, which
626 lets them remove any interference between two tasks whose running time are disjoint,
627 and start over until a fix point is reached. Data caches are not taken into account and
628 are assumed to have no effect on the calculations. Cache coherence is not addressed.

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

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

639 ■ **Cache Coherence Protocol Comparison:** The authors of [2] compare the efficiency of

640 common snooping-based cache coherence protocols. To do so, they described a multi-core
 641 processor and the cache coherence protocols in *Simula*. Much like ours, the programs
 642 running on this simulation are described as a succession of memory related instructions.
 643 However, they do not use explicit addresses for these instructions. Instead, they have
 644 defined system-wide weights to regulate the probability of an instruction to be applied
 645 on a private memory element (i.e. a memory element the cache is the sole user of) or a
 646 shared block (i.e. a memory element used by multiple blocks). Thus, cache coherence *is*
 647 addressed, but only in a very broad context. Indeed, whereas our work focuses on the
 648 impact of cache coherence on specific applications on a specific architecture, the cache
 649 coherence protocol comparison made by the authors of [2] provides a general idea of
 650 which protocol is more fitted for which type of application.

651 ■ **Predictable Cache Coherence:** An alternative to trying to predict how cache coher-
 652 ence is going to behave is to use a kind of cache coherence designed to be predictable.
 653 [24] lists the cache coherence related latencies that need to be known before predictability
 654 of the protocol can be achieved. Its authors argue that write-through, update-based
 655 protocols (i.e. writes are propagated to other caches and to the memory) can be made to
 656 be predictable.

657 [14] presents PMSI, a variation on the MSI protocol that uses a TDM bus to achieve
 658 predictability. Emission of coherence queries and is restricted to a core's TDM slots.
 659 As a result, a cache does not suffer from interference during its own TDM slots. [21]
 660 expands on this by introducing HourGlass, which allows separate handling of critical
 661 and non-critical cores. HourGlass uses timers to allow cores to hold access to a memory
 662 element for a predefined time duration. The evaluation of queries that would remove
 663 an access currently protected a timer are delayed until its time is up. Both PMSI and
 664 HourGlass require hardware modification, which prevents them from being used in a
 665 context that relies on COTS.

666 9 Conclusion and Future Work

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

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

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

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

686 — **References** —

- 687 **1** Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–
688 235, April 1994. URL: [http://dx.doi.org/10.1016/](http://dx.doi.org/10.1016/0304-3975(94)90010-8)
689 [0304-3975\(94\)90010-8](http://dx.doi.org/10.1016/0304-3975(94)90010-8), doi:10.1016/
0304-3975(94)90010-8.
- 690 **2** James Archibald and Jean-Loup Baer. Cache coherence protocols: Evaluation using a
691 multiprocessor simulation model. *ACM Trans. Comput. Syst.*, 4(4):273–298, September 1986.
692 URL: <http://doi.acm.org/10.1145/6513.6514>, doi:10.1145/6513.6514.
- 693 **3** Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. *Handbook of*
694 *Satisfiability*, chapter Satisfiability Modulo Theories, pages 825–885. IOS Press, 2009.
- 695 **4** Gerd Behrmann, Alexandre David, and Kim G. Larsen. *A Tutorial on Uppaal*, pages 200–
696 236. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. URL: [https://doi.org/10.1007/](https://doi.org/10.1007/978-3-540-30080-9_7)
697 [978-3-540-30080-9_7](https://doi.org/10.1007/978-3-540-30080-9_7), doi:10.1007/978-3-540-30080-9_7.
- 698 **5** Frédéric Boniol, Hugues Cassé, Eric Noulard, and Claire Pagetti. Deterministic execution
699 model on cots hardware. In *Proceedings of the 25th International Conference on Architecture*
700 *of Computing Systems (ARCS'12)*, pages 98–110, 2012.
- 701 **6** Franck Cassez and Jean-Luc Béchenec. Timing analysis of binary programs with UPPAAL.
702 In *13th International Conference on Application of Concurrency to System Design, ACSD*
703 *2013*, pages 41–50. IEEE Computer Society, July 2013. doi:[http://dx.doi.org/10.1109/](http://dx.doi.org/10.1109/ACSD.2013.7)
704 [ACSD.2013.7](http://dx.doi.org/10.1109/ACSD.2013.7).
- 705 **7** Franck Cassez and Pablo González de Aledo Marugán. Timed automata for modelling caches
706 and pipelines. In Rob J. van Glabbeek, Jan Friso Groote, and Peter Höfner, editors, *Proceedings*
707 *Workshop on Models for Formal Analysis of Real Systems, MARS 2015, Suva, Fiji, November*
708 *23, 2015.*, volume 196 of *EPTCS*, pages 37–45, 2015. doi:10.4204/EPTCS.196.4.
- 709 **8** Sudipta Chattopadhyay, Abhik Roychoudhury, and Tulika Mitra. Modeling shared cache and
710 bus in multi-cores for timing analysis. In *Proceedings of the 13th International Workshop*
711 *on Software & Compilers for Embedded Systems, SCOPES '10*, pages 6:1–6:10, New York,
712 NY, USA, 2010. ACM. URL: <http://doi.acm.org/10.1145/1811212.1811220>, doi:10.1145/
713 [1811212.1811220](http://doi.acm.org/10.1145/1811212.1811220).
- 714 **9** Andreas E. Dalsgaard, Mads Chr. Olesen, Martin Toft, René Rydhof Hansen, and Kim Guld-
715 strand Larsen. METAMOC: Modular Execution Time Analysis using Model Checking.
716 In Björn Lisper, editor, *10th International Workshop on Worst-Case Execution Time*
717 *Analysis (WCET 2010)*, volume 15 of *OpenAccess Series in Informatics (OASICS)*, pages
718 113–123, Dagstuhl, Germany, 2010. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.
719 The printed version of the WCET'10 proceedings are published by OCG (www.ocg.at)
720 - ISBN 978-3-85403-268-7. URL: <http://drops.dagstuhl.de/opus/volltexte/2010/2831>,
721 doi:10.4230/OASICS.WCET.2010.113.
- 722 **10** Giorgio Delzanno. Automatic verification of parameterized cache coherence protocols. In
723 *Proceedings of the 12th International Conference on Computer Aided Verification, CAV '00*,
724 pages 53–68, London, UK, UK, 2000. Springer-Verlag. URL: [http://dl.acm.org/citation.](http://dl.acm.org/citation.cfm?id=647769.734088)
725 [cfm?id=647769.734088](http://dl.acm.org/citation.cfm?id=647769.734088).
- 726 **11** Philip Enslow, Jr. Multiprocessor organization - a survey. *ACM Comput. Surv.*, 9(1):103–129,
727 March 1977.
- 728 **12** Sylvain Girbal, Xavier Jean, Jimmy le Rhun, Daniel Gracia Pérez, and Marc Gatti. Determin-
729 istic Platform Software for Hard Real-Time systems using multi-core COTS. In *34th Digital*
730 *Avionics Systems Conference (DASC'15)*, 2015.
- 731 **13** Andreas Gustavsson, Andreas Ermedahl, Björn Lisper, and Paul Pettersson. Towards WCET
732 analysis of multicore architectures using UPPAAL. In *10th International Workshop on Worst-*
733 *Case Execution Time Analysis, WCET 2010, July 6, 2010, Brussels, Belgium*, pages 101–112,
734 2010. URL: <https://doi.org/10.4230/OASICS.WCET.2010.101>, doi:10.4230/OASICS.WCET.
735 [2010.101](https://doi.org/10.4230/OASICS.WCET.2010.101).
- 736 **14** Mohamed Hassan, Anirudh M. Kaushik, and Hiren D. Patel. Predictable cache coherence
737 for multi-core real-time systems. In *2017 IEEE Real-Time and Embedded Technology and*

- 738 *Applications Symposium, RTAS 2017, Pittsburg, PA, USA, April 18-21, 2017*, pages 235–246,
739 2017. URL: <https://doi.org/10.1109/RTAS.2017.13>, doi:10.1109/RTAS.2017.13.
- 740 **15** Xavier Jean, David Faura, Marc Gatti, Laurent Pautet, and Thomas Robert. Ensuring
741 robust partitioning in multicore platforms for ima systems. In *31st Digital Avionics Systems
742 Conference (DASC'16)*, 2012.
- 743 **16** M. Lv, W. Yi, N. Guan, and G. Yu. Combining abstract interpretation with model checking for
744 timing analysis of multicore software. In *2010 31st IEEE Real-Time Systems Symposium*,
745 pages 339–349, Nov 2010. doi:10.1109/RTSS.2010.30.
- 746 **17** Claire Maiza, Hamza Rihani, Juan Maria Rivas, Joël, Godelieve Goossens, Sebastian Altmeyer,
747 and Robert I. Davis. A survey of timing verification techniques for multi-core real-time systems.
748 Technical report, Grenoble INP/Ensimag/Verimag, 2018.
- 749 **18** Rodolfo Pellizzoni, Emiliano Betti, Stanley Bak, Gang Yao, John Criswell, Marco Caccamo,
750 and Russell Kegley. A predictable execution model for cots-based embedded systems. In *17th
751 IEEE Real-Time and Embedded Technology and Applications Symposium RTAS 2011*, pages
752 269–279, 2011.
- 753 **19** Fong Pong and Michel Dubois. A new approach for the verification of cache coherence
754 protocols. *IEEE Trans. Parallel Distrib. Syst.*, 6(8):773–787, August 1995. URL: <http://dx.doi.org/10.1109/71.406955>,
755 doi:10.1109/71.406955.
- 756 **20** Daniel J. Sorin, Mark D. Hill, and David A. Wood. *A Primer on Memory Consistency and
757 Cache Coherence*. Morgan & Claypool Publishers, 1st edition, 2011.
- 758 **21** Nivedita Sritharan, Anirudh M. Kaushik, Mohamed Hassan, and Hiren D. Patel. Hourglass:
759 Predictable time-based cache coherence protocol for dual-critical multi-core systems. *CoRR*,
760 abs/1706.07568, 2017. URL: <http://arxiv.org/abs/1706.07568>, arXiv:1706.07568.
- 761 **22** V. Suhendra, T. Mitra, and A. Roychoudhury and. Wcet centric data allocation to scratchpad
762 memory. In *26th IEEE International Real-Time Systems Symposium (RTSS'05)*, pages 10
763 pp.–232, Dec 2005. doi:10.1109/RTSS.2005.45.
- 764 **23** Texas Instruments. TCI6630K2L Multicore DSP+ARM KeyStone II System-on-Chip. Techni-
765 cal Report SPRS893E, Texas Instruments Incorporated, 2013.
- 766 **24** Sascha Uhrig, Lillian Tadros, and Arthur Pyka. Mesi-based cache coherence for hard real-time
767 multicore systems. In Luís Miguel Pinho Pinho, Wolfgang Karl, Albert Cohen, and Uwe
768 Brinkschulte, editors, *Architecture of Computing Systems – ARCS 2015*, pages 212–223, Cham,
769 2015. Springer International Publishing.
- 770 **25** L. Wehmeyer and P. Marwedel. Influence of memory hierarchies on predictability for time
771 constrained embedded software. In *Design, Automation and Test in Europe*, pages 600–605
772 Vol. 1, March 2005. doi:10.1109/DATE.2005.183.
- 773 **26** Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David
774 Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank
775 Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-
776 case execution-time problem - overview of methods and survey of tools. *ACM Transactions
777 Embedded Computing Systems*, 7(3):36:1–36:53, May 2008.
- 778 **27** Reinhard Wilhelm and Jan Reineke. Embedded systems: Many cores - many problems. In *7th
779 IEEE International Symposium on Industrial Embedded Systems (SIES'12)*, pages 176–180,
780 2012.
- 781 **28** Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. MemGuard: Mem-
782 ory bandwidth reservation system for efficient performance isolation in multi-core platforms.
783 In *19th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'13)*,
784 pages 55–64, 2013.
- 785 **29** Wei Zhang and Jun Yan. Static timing analysis of shared caches for multicore processors.
786 *JCSE*, 6(4):267–278, 2012. URL: <https://doi.org/10.5626/JCSE.2012.6.4.267>, doi:10.
787 5626/JCSE.2012.6.4.267.