Université Fédérale

Toulouse Midi-Pyrénées

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par :

Institut Supérieur de l'Aéronautique et de l'Espace

Présentée et soutenue par :
Nathanaël SENSFELDER
le mercredi 31 mars 2021

Titre :

Analyse et contrôle des interférences liées à la cohérence de cache dans les multi-coeurs COTS

École doctorale et discipline ou spécialité :
ED MITT : Réseaux, télécom, système et architecture

Unité de recherche :
Équipe d'accueil ISAE-ONERA MOIS

Directeur(s) de Thèse :

Mme Claire PAGETTI (directrice de thèse)
M. Julien BRUNEL (co-directeur de thèse)

Jury :
Mme Christine ROCHANGE Professeure des Universités Université Toulouse III - Présidente
M. Björn BRANDENBURG Professeur des Universités Université de la Sarre - Examinateur
M. Julien BRUNEL Chargé de recherche ONERA - Co-directeur de thèse
Mme Janette CARDOSO Professeure ISAE-SUPAERO - Examinatrice
MI. Sylvain CONCHON Professeur des Universités Université Paris-Sud XI - Examinateuré
Mme Claire PAGETTI Maître de Recherche ONERA - Directrice de thèse
Mme Isabelle PUAUT Professeure des Universités Université de Rennes I - Rapporteure
M. Olivier H. ROUX Professeur des Universités École Centrale de Nantes - Rapporteur

Abstract

This thesis proposes tools to help in the certification of multi-core processors for use in aeronautical systems. While the parallel nature of multi-core processors can greatly improve computation speeds, it also makes them difficult to predict, preventing their use in critical environments. Indeed, in such processors, the cores share access to nearly all resources and this causes conflicts, or interference, which lead to seemingly random variation in the execution time. Among the complex mechanisms prone to interference is cache coherence, which ensures that cores that use a same atomic memory block cannot blindly override the modifications made by another core and that all cores are made aware of all modifications. To achieve cache coherence, the processor automatically follows a predetermined protocol which defines messages to be generated according to the actions of a core, as well as the actions to be performed when another core's message is received.

The focus of this thesis is to identify the interference generated by the cache coherence mechanisms and provide a way to predict their effects on the applications, as a first step toward their mitigation. The first contribution made is to address the ambiguities in the understanding applicants have of the coherence protocol implemented on their chosen architecture. Indeed, architecture documentation does not generally offer sufficient details on their cache coherence protocol. This thesis proposes a formalization of some standard cache protocols and a strategy relying on micro-benchmarks in order to clarify the implementation details of the architecture's protocol. This approach is applied to the NXP QorIQ T4240 architecture. Once the protocol has been correctly identified, the second contribution consists in the making of a low-level description of the architecture using timed automata in order to adequately represent the micro-behaviors and understand clearly how the cache coherence protocol acts. In effect, a generic model generation framework has been developed, capable of handling cache coherence protocols as described by the applicant, and to support architectures with different configurations in order to better fit the applicant's chosen architecture. The third contribution explains how to make use of the timed automata low-level representation of the architecture to expose interference. It proposes a strategy to detail the causes and effects of cache coherence interference on the given programs. Starting from a simple analysis of execution time, the results go down to instruction level, indicating how each instruction generates and suffers from interference. This is intended to provide sufficient information on cache coherence interference to the applicant, both for the purposes of certification and to form the base upon which a mitigation strategy can be started.

In effect, this thesis provides the applicant with the means to understand the cache coherence mechanisms used by their chosen architecture and to expose the interference they generate.

Résumé Français

L'objectif de cette thèse est d'offrir des outils d'aide à la certification aéronautique de processeurs COTS multi-cœurs. Ces architectures sont par nature parallèles et peuvent de ce fait largement améliorer les performances de calcul. Cependant elles souffrent d'un grand manque de prédictibilité, au sens où calculer les pires d'exécution même pour des programmes simples est un problème complexe, voire impossible dans le cas général. En effet, les cœurs partagent l'accès à presque toutes les ressources ce qui provoque des conflits (qualifiés d'interférences) entrainant des variations non maîtrisées des temps d'exécutions. Parmi les mécanismes complexes d'un processeur multi-coeur se trouve la cohérence de caches. Celle-ci assure que tous les cœurs lisant ou écrivant dans un même bloc mémoire ne peuvent pas aveuglement ignorer les modifications appliquées par les autres. Afin de maintenir la cohérence de caches, le processeur suit un protocole pré-déterminé qui définit les messages à envoyer en fonction des actions d'un cœur ainsi que les actions à effectuer lors de la réception du message d'un autre cœur.

Cette thèse porte sur l'identification des interférences générées par les mécanismes de cohérence de caches ainsi que sur les moyens de prédiction de leurs effets sur les applications en vue de réduire les effets négatifs temporels. La première contribution adresse les ambiguïtés dans la compréhension que les applicants ont de la cohérence de cache réellement présente dans l'architecture. En effet, la documentation des architectures ne fournit généralement pas suffisamment de détails sur les protocoles. Cette thèse propose une formalisation des protocoles standards, ainsi qu'une stratégie, reposant sur les micro-benchmarks, pour clarifier les choix d'implémentation du protocole de cohérence présent sur l'architecture. Cette stratégie a notamment été appliquée sur le NXP QorIQ T4240. Une fois le protocole correctement identifié, la seconde contribution consiste à réaliser une description bas-niveau de l'architecture en utilisant des automates temporisés afin de représenter convenablement les micro-comportements et comprendre clairement comment le protocole de cohérence de cache agit. Ainsi, un framework de génération de modèles génériques a été développé, capable de supporter plusieurs protocoles de cohérence de cache et de représenter différents agencements d'architectures afin de mieux correspondre à l'architecture choisie par le postulant. La troisième contribution explique comment utiliser cette représentation de l'architecture pour exhiber les interférences. Elle propose une stratégie pour détailler les causes et effets de chaque interférence liée à la cohérence de caches sur les programmes. Commençant par une simple analyse de temps d'exécution, les résultats descendent jusqu'au niveau des instructions pour indiquer comment chaque instruction génère et souffre des interférences. L'objectif étant alors de fournir suffisamment d'information à l'appliquant à la fois pour la certification, mais aussi pour définir une stratégie d'atténuation et de maîtrise des effets temporels.

Ainsi, cette thèse fournit l'appliquant des outils pour comprendre les mécanismes de cohérence de cache présent sur une architecture donnée et pour exhiber les interférences associées.

Contents

Part I

Context

Chapter 1

Introduction

1.1 Context

1.1.1 Aeronautical Embedded Systems

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.

The operation of a safety critical system requires its certification by the relevant authorities. The entity applying to obtain this certification is refered to as the *applicant* in thesis. Indeed, this certification is obtained through a process in which an applicant argues for the compliance of that system with regulation. The introduction of a new category of hardware in such a system renders this process particularly difficult, as it implies a lack of preexisting process for the generation of a proof of compliance. Furthermore, the link between this new hardware and the high level certification objectives may not be obvious.

1.1.2 Multi-core Based Systems Certification

This thesis is part of the Phylog project. The objective of the Phylog project is to provide tools that will help building a strong case for applicants attempting to pass the certification process of an aeronautical computer system. These systems are assumed to be commercial off-the-shelf (*COTS*) products, meaning processors not manufacted solely for this specific use. The requirements that the applicants must prove this computer system passes include those listed in the CAST-32A ([17]). This document focuses on the particularities of multi-core processors and the way these particularities complicate the demonstration of both safety and performance standard objectives fulfillment. Among the requirements listed in the CAST-32A figures *Resource Usage 3*, which requires the complete identification of all interference and its effects with the chosen configuration: *The applicant has identified the interference channels that could permit interference to affect the software applications hosted on the MCP cores, and has verified the applicant's chosen means of mitigation of the interference.* The Phylog project translated this requirement into an assurance case, which can be seen in Figure 1.1. To help applicants fulfill this objective, this thesis focuses on interference generated by a prevalent feature of multi-core processors: cache coherence.

Definition 1 (Interference) *An interference is the unwarranted modification of the execution time of an application because of the actions of another.*

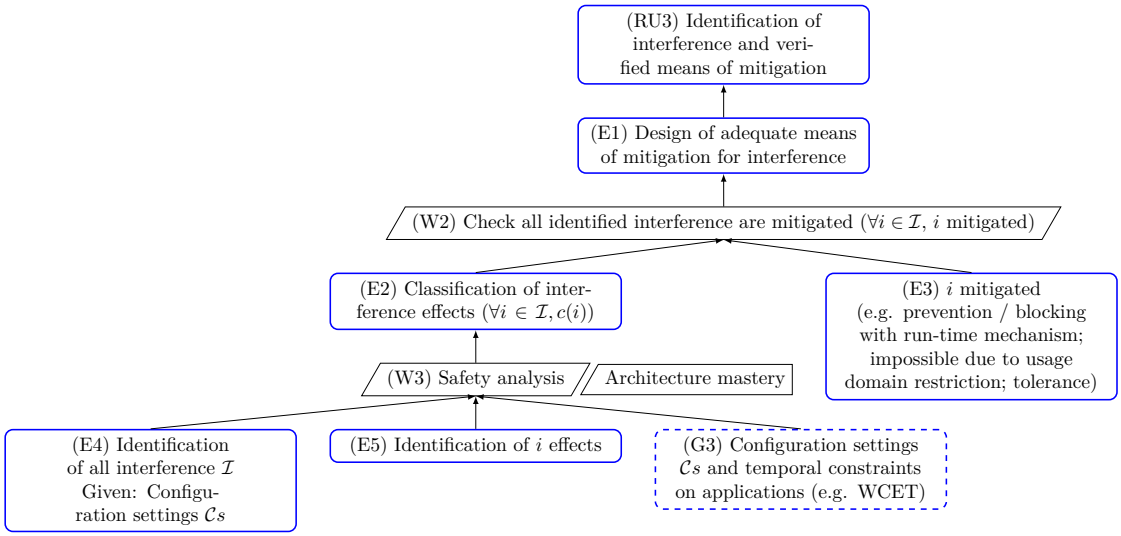


Figure 1.1 – Assurance Case Corresponding to RU3

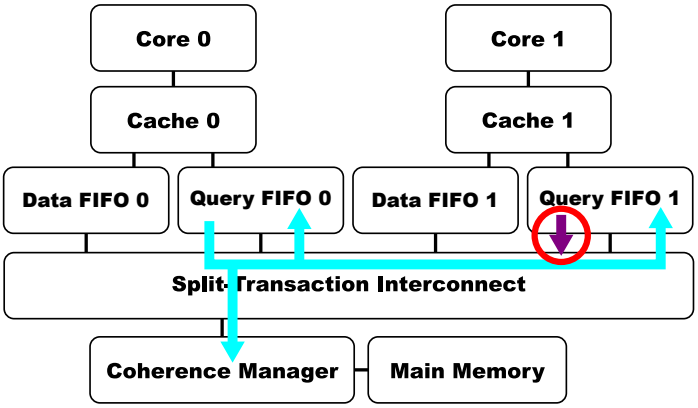


Figure 1.2 – Example of Interference

Example 1 (Example of Interference) *In a system in which two cores, with one cache each, both attempting to send a query to load data simultaneously, the interconnect will end up having to choose one of the two queries to send first, and will put the other query in waiting. This waiting lengthens the execution time of the associated instruction and would not occur if there was no concurrent query, thus making it an interference. Figure 1.2 illustrates this example, by having Cache 1's query be prevented access to the interconnect during the other query's propagation, meaning that the application running on Core 0 is interfering with the one running on Core 1.*

1.2 The Issue of Cache Coherence

When multiple cores make use of the same memory elements, separate copies of these memory elements find themselves in different caches. As these copies are separate, changes made to a copy are not reflected on the other copies. This makes parallel computing difficult: a core might not be using the most up-to-date value of the shared memory elements and the modifications it performs might be blindly overridden by another core.

Cache coherence refers to mechanisms that will ensure all these separate copies stay consistent. In effect, it will ensure that there is never an ambiguity on the current value of memory elements, and that any core accessing a memory element is using this most up-to-date value.

Achieving cache coherence requires caches to coordinate with each other. This is done through shared buses, on which caches send queries to communicate needs and receive data messages in reply. These buses in themselves are thus a heavily used shared resource, making it a source of interference. However, the main cause of cache coherence interference is that, to maintain coherence, these queries can force caches to lose access to some of their content. As a result, the actions of another core will determine whether a core can find the memory element it wants through a quick cache access or if this will require a time consuming fetch.

These cache coherence mechanisms are generally fully automated, meaning that the application developers do not directly control when cache queries are made. This makes predicting the emission and effects of these queries difficult. Indeed, the emission of a query is determined by both program instruction and the content of the cache, and the latter is subject to uncontrolled modifications by queries emitted from other caches. This makes cache coherence a source of important execution time variations and a challenge to certification.

1.3 Overview of the Thesis

This thesis starts by introducing prerequisites: timed automata (Chapter 2) and cache coherence (Chapter 3). Once these have been presented, the focus of this thesis can be explained in full (Chapter 4). Indeed, The purpose of the thesis is to develop a framework to ensure the applicant is made aware of the interference generated by cache coherence in their chosen COTS multi-core processor.

To determine the state of the art and what specifically needs to be developed, a whole part is dedicated to the relevant existing literature. First is architecture profiling, for which existing solutions relying on benchmarks are presented in Chapter 5, including works with a focus on caches. The current practices with regards to the use of caches in multi-core employed in critical environments are detailed in Chapter 6. Since the solution proposed in this thesis relies on formal methods, a number of existing works that have a similar approach to the study of architectures are presented in Chapter 7.

After clarifying what is left to be done to achieve a full framework that will help with the cache coherence part of the certification, this thesis proposes three contributions: a strategy to properly identify an architecture's cache coherence protocol, a model template for multi-core architectures with cache coherence support, and analyses to be performed on instantiated models in order to expose the interference.

- The first contribution, presented in Chapter 8, is meant to ensure the applicant is fully aware of all the peculiarities of the cache coherence protocol implemented by the multi-core architecture of their choice. To achieve this, the applicant is asked to start by formalizing what they believe the cache coherence protocol to be, in a fashion that leaves no possible ambiguity.

This hypothetical cache coherence protocol is then validated against the architecture through observations made with micro-benchmarks.

- Using the previous contribution, the applicant obtained an ambiguity-free cache coherence protocol corresponding to the one used by the targeted multi-core architecture. However, the analysis of the protocol by itself does not reveal much. The second contribution, presented in Chapter 9, proposes the model of a multi-core architecture to the applicant. This model, made of networked timed automata, is meant to be instantiated to fit the applicant's chosen architecture, and can automatically be made to use the aforementioned ambiguity-free cache coherence protocol.
- The instantiated model created using the previous contribution can be used to perform an analysis of interference occurring in the system. This third contribution, presented in Chapter 10, shows how model checking can be employed to expose the causes and effects of cache coherence interference in the system. The analyzes include worst-case execution time estimation, as well as the identification of how each program instruction is affected by and/or generates interference.

Chapter 2

(Timed) Automata

This chapter presents the concept of timed automata, which is the formal model used in this thesis, and an associated formal method: reachability analysis through model checking. In the first section, the main concepts behind classical automata are introduced, with some fairly common additions: the use of variables, having conditions and actions in transitions, and synchronization. A definition of the temporal logic operators used in this thesis is also given in the classical automata section. Timed automata are introduced in a second section. As this thesis does not dwell into the theory of automata, but simply uses them as a modeling tool, many details (such as their precise semantics, the language theory or the details of how model checking is achieved) are omitted. Indeed, the objective of the chapter is for the reader to have an understanding of the models of cache coherence presented in later chapters, as well as the operators being used to query on them. Since the aforementioned details are not directly related to the work presented in this thesis, they are considered to be outside of the scope of this chapter.

2.1 Classical Automata

This section is meant as a reminder on classical automata. Readers looking for in-depth information on the subject are encouraged to read [3], [33], or [8].

2.1.1 System Definition

Definition 2 (Syntax of Constraints and Actions) *Given a set of variables* $\texttt{Var}$*, the grammar used when writing constraints and actions in transitions is as follows, with* ***ident*** *standing for a variable in* $\texttt{Var}$*:*
$\boldsymbol{lop} ::= \wedge \mid \vee$
$\boldsymbol{cop} ::= < \mid \leq \mid = \mid \geq \mid >$
$\boldsymbol{mop} ::= + \mid - \mid * \mid /$
$\boldsymbol{mexpr} ::= \boldsymbol{mexpr}\ \boldsymbol{mop}\ \boldsymbol{mexpr} \mid \boldsymbol{ident} \mid \mathbb{Z}$
$\boldsymbol{abexpr} ::= \boldsymbol{mexpr}\ \boldsymbol{cop}\ \boldsymbol{mexpr} \mid \texttt{true} \mid \texttt{false}$
$\boldsymbol{bexpr} ::= \neg \boldsymbol{bexpr} \mid \boldsymbol{bexpr}\ \boldsymbol{lop}\ \boldsymbol{bexpr} \mid \boldsymbol{abexpr}$
$\boldsymbol{assign} ::= \boldsymbol{assign};\ \boldsymbol{assign} \mid \boldsymbol{ident} := \boldsymbol{mexpr} \mid if\ (\boldsymbol{bexpr})\ \{\boldsymbol{assign}\} \mid \texttt{nop}$

Definition 3 (Classical Automata System) *A classical automata system* $\mathcal{A}$ *is a tuple* $\langle Q, s_{init}, \mathcal{B}, \mathcal{E}, \texttt{Var}, \texttt{Act}, \rightsquigarrow \rangle$ *where:*

- Q *is a finite set of states.*
- s_{init} *is the initial state* ($s_{init} \in Q$).
- $\mathtt{Var}$ *is a finite set of integer variables, taking their value on a finite subset* $\mathtt{D_{Var}}$ *of integers.*
- $\mathcal{E} = \mathcal{E}^{\alpha} \cup \mathcal{E}^{\mathtt{sync}}$ *is a finite set of labels, with* $\mathcal{E}^{\mathtt{sync}}$ *corresponding to labels meant for synchronization and* $\mathcal{E}^{\alpha}$ *being regular labels. The labels in* $\mathcal{E}^{\mathtt{sync}}$ *affixed by either '?' or '!', with '?' denoting a reception on a "channel", and '!' an emission.*
- $\mathcal{B} = \boldsymbol{bexpr}(\mathtt{Var})$, *as defined in Definition 2.*
- $\mathtt{Act} = \boldsymbol{assign}(\mathtt{Var})$, *as defined in Definition 2.*
- $\rightsquigarrow \subseteq Q \times \mathcal{B} \times \mathcal{E} \times \mathtt{Act} \times Q$ *is the transition relation.*

Note that we use both a set Q *of states and a set* $\mathtt{Var}$ *of integer variables, in order to be close to the framework of UPPAAL, which we are using in the rest of the thesis.*

The semantics of $\mathcal{A}$ *is given via its set of valid transitions (see Definition 14) and its execution traces (see Definition 6).*

Definition 4 (Valuation) *Valuations map variables to their value:* $v : \mathtt{Var} \to \mathtt{D_{Var}}$. *Given a valuation* v, *and a guard* $c \in \mathcal{B}$, *we note* $v \models_{PL} c$ *to indicate that* c *is true under the valuation* v.

Similarly, given $a \in \mathtt{Act}$, $v[a]$ *denotes the valuation obtained from* v *by the application of the action* a, *were all variables updated by* a *have their new value and all other variables keep their previous value.*

Definition 5 (Transition) *Given an automaton* $\mathcal{A} = \langle Q, s_{init}, \mathcal{B}, \mathcal{E}, \mathtt{Var}, \mathtt{Act}, \rightsquigarrow \rangle$, *we define* $\mathtt{Step}$, *which indicates all valid transitions that can be performed from* $\langle s, v \rangle$, *with* $s \in Q$ *and* v *a valuation:* $\mathtt{Step}(\langle s, v \rangle) \triangleq \{\langle s', v' \rangle | \exists \langle s, c, l, a, s' \rangle \in \rightsquigarrow \text{ s.t. } ((v \models_{PL} c) \wedge v' = v[a])\}$

Definition 6 (Path & Trace) *We consider a path to be a* maximal *sequence of states/transitions* $\langle s_1, v_1 \rangle \to \langle s_2, v_2 \rangle \to \cdots$ *such that for each* i, $\langle s_{i+1}, v_{i+1} \rangle \in \mathtt{Step}(\langle s_i, v_i \rangle)$. *The sequence is maximal in the sense that it is either infinite or of length* N *and such that* $\mathtt{Step}(\langle s_N, v_N \rangle)$ *is empty. A path starting from* $\langle s_{init}, v_0 \rangle$ *(where* v_0 *is the initial valuation) is called a trace.*

Example 2 (Classical Automata) *Figure 2.1 shows two automata modeling a client (on the left) that fetches a number of files from a server (on the right). In this scenario, the system loops infinitely, with the client initiating a request for files (**request_files**), and counting (fetched) their arrival (**new_file**) until the server indicates that all were transfered (**done**). On each request, the server sends exactly 386 files (as counted by sent).*

Example 3 (Traces) *Here are some examples of traces for the client automaton from Example 2:*

- $\langle S_0, \{\langle fetched, 0 \rangle\} \rangle \xrightarrow{\boldsymbol{err}} \langle S_E, \{\langle fetched, 0 \rangle\} \rangle$
- $\langle S_0, \{\langle fetched, 0 \rangle\} \rangle \xrightarrow[fetched:=0]{\boldsymbol{request\_files!}} \langle S_1, \{\langle fetched, 0 \rangle\} \rangle \xrightarrow{\boldsymbol{err}} \langle S_E, \{\langle fetched, 0 \rangle\} \rangle$

And for the server automaton:

- $\langle S_0, \{\langle sent, 0 \rangle\} \rangle \xrightarrow{\boldsymbol{err}} \langle S_E, \{\langle sent, 0 \rangle\} \rangle$

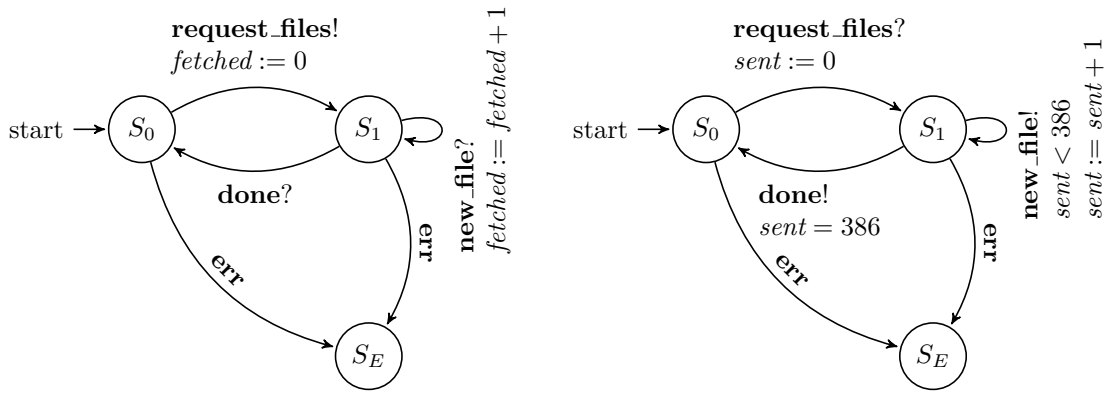


Figure 2.1 – Example of two classical automata

- $\langle S_0, \{\langle sent, 0\rangle\}\rangle \xrightarrow[sent:=0]{\textbf{\textit{request\_files?}}} \langle S_1, \{\langle sent, 0\rangle\}\rangle \xrightarrow[sent:=sent+1]{\textbf{\textit{new\_file!}}\ sent<386} \langle S_1, \{\langle sent, 1\rangle\}\rangle \xrightarrow[sent:=sent+1]{\textbf{\textit{new\_file!}}\ sent<386}$
  $\cdots \xrightarrow[sent:=sent+1]{\textbf{\textit{new\_file!}}\ sent<386} \langle S_1, \{\langle sent, 386\rangle\}\rangle \xrightarrow{\textbf{\textit{done!}}\ sent=386} \langle S_0, \{\langle sent, 386\rangle\}\rangle \xrightarrow{\textbf{\textit{err}}} \langle S_E, \{\langle sent, 386\rangle\}\rangle$

Definition 7 (Synchronized automata) *Given* n *automata* $\mathcal{A}_i = \langle Q_i, s_{initi}, \mathcal{B}_i, \mathcal{E}_i, \mathtt{Var}_i, \mathtt{Act}_i, \rightsquigarrow_i \rangle$*, and a synchronization constraint* $\mathtt{Sync}_{con} \subseteq (\mathcal{E}_1 \cup \{-\}) \times \cdots \times (\mathcal{E}_n \cup \{-\})$*, we can define a new automaton* $\mathcal{A}_s = \langle Q_s, s_{inits}, \mathcal{B}_s, \mathcal{E}_s, \mathtt{Var}_s, \mathtt{Act}_s, \rightsquigarrow \rangle$*, corresponding to the synchronized product of the* $\mathcal{A}_i$ *automata according to* $\mathtt{Sync}_{con}$*, with the following rules:*

- $Q_s = Q_1 \times \cdots \times Q_n$
- $s_{inits} = \langle s_{init1}, \cdots, s_{initn} \rangle$
- $\mathcal{B}_s = \mathcal{B}_1 \times \cdots \times \mathcal{B}_n$
- $\mathcal{E}_s = (\mathcal{E}_1 \cup \{-\}) \times \cdots \times (\mathcal{E}_n \cup \{-\})$*. We extend the labels with* $-$ *to mark that a sub-automaton does not perform any transition.*
- $\mathtt{Var}_s = \mathtt{Var}_1 \cup \cdots \cup \mathtt{Var}_n$*, with* $\forall i, j \in 1..n,\ (i \neq j) \implies (\mathtt{Var}_i \cap \mathtt{Var}_j = \emptyset)$*.*
- $\mathtt{Act}_s = \mathtt{Act}_1 \times \cdots \times \mathtt{Act}_n$
- $\rightsquigarrow_s \subseteq Q_s \times \mathcal{B}_s \times \mathcal{E}_s \times \mathtt{Act}_s \times Q_s$*, with*

$$\langle\langle o_1, \cdots, o_n\rangle, \langle c_1, \cdots, c_n\rangle, \langle l_1, \cdots, l_n\rangle, \langle a_1, \cdots, a_n\rangle, \langle d_1, \cdots, d_n\rangle\rangle \in \rightsquigarrow_s$$

$$\iff \begin{cases} \langle l_1, \cdots, l_n\rangle \in \mathtt{Sync}_{con} \\ \langle \forall i \in 1..n : o_i, c_i, l_i, a_i, d_i\rangle \in \rightsquigarrow_i \vee (o_i = d_i \wedge l_i = - \wedge c_i = \mathtt{true} \wedge a_i = nop) \end{cases}$$

$\mathtt{Sync}_{con}$ *is implicitely defined by the labels in* $\mathcal{E}_{1..n}$*: for any transition with a label in* $\mathcal{E}^{\alpha}$*, there is an entry in* $\mathtt{Sync}_{con}$ *with no other simultaneous transition allowed (indicated by* $-$*, which means the particular sub-automaton does not perform a transition). For any transition chan in* $\mathcal{E}^{sync}$*,* $\mathtt{Sync}_{con}$ *has an entry for each possible chan!, chan?,* $-$ *label combination such that there is a single chan! label and a single chan? label. This convention was introduced in CCS ([37]).*

Definition 8 (Synchronized Automata Semantics) *As the product of synchronized automata is itself a classical automaton, its semantics is the same as those from Definition 6.*

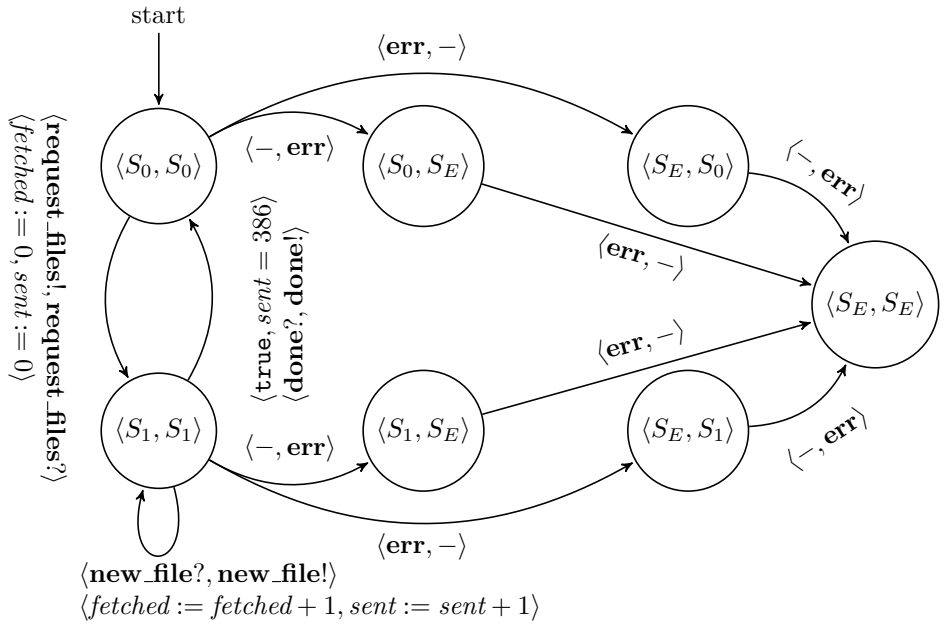


Figure 2.2 – Example of synchronized automaton

Example 4 *Figure 2.1 is in fact a network of automata, which is one way of representing synchronization between automata. Figure 2.2 shows another representation, with a single automaton resulting from the synchronized product of the automata from Figure 2.1. In this case,* $\texttt{Sync}_{con}$ *is defined as:* $\{\langle \textbf{\textit{request\_files}}!, \textbf{\textit{request\_files}}?\rangle, \langle \textbf{\textit{done}}?, \textbf{\textit{done}}!\rangle, \langle \textbf{\textit{new\_file}}?, \textbf{\textit{new\_file}}!\rangle, \langle \textbf{\textit{err}}, -\rangle, \langle -, \textbf{\textit{err}}\rangle\}$.

2.1.2 Query Logic Operators and Semantics

Given an automaton $\mathcal{A}$ and an initial valuation v_0, we can define the satisfaction relation for a property ϕ. We assume ϕ to be a formula in **bexpr**(`Var`) written in the subset of CTL ([19]) temporal operators described below. Readers interested in the details of how these are actually verified are encouraged to read on model-checking (for example, [20]). The satisfaction of $\langle s, v\rangle \models \phi$ is defined using the following decomposition:

$\langle s, v\rangle \models \psi \triangleq\ v \models_{PL} \psi$, where ψ is an expression in **abexpr**(`Var`).

$\langle s, v\rangle \models \neg\phi \triangleq\ \langle s, v\rangle \not\models \phi$

$\langle s, v\rangle \models \phi \wedge \psi \triangleq\ (\langle s, v\rangle \models \phi)$ and $(\langle s, v\rangle \models \psi)$

$\langle s, v\rangle \models$ **AF** $\phi \triangleq$
For all paths starting from $\langle s, v\rangle$, there is, within the path, a $\langle s', v'\rangle$, such that $\langle s', v'\rangle \models\ \phi$

$\langle s, v\rangle \models$ `EF` $\phi \triangleq$
There is a path starting from $\langle s, v\rangle$ in which is found a $\langle s', v'\rangle$, such that $\langle s', v'\rangle \models \phi$

$\langle s, v\rangle \models$ `AG` $\phi \triangleq$
For all paths starting from $\langle s, v\rangle$, all $\langle s', v'\rangle$ of the path verify $\langle s', v'\rangle \models \phi$

$\langle s, v\rangle \models$ `EG` $\phi \triangleq$
There is a path starting from $\langle s, v\rangle$, such that all $\langle s', v'\rangle$ verify $\langle s', v'\rangle \models \phi$

$\langle s, v\rangle \models \phi$ `-->` $\psi \triangleq$
For all paths starting from $\langle s, v\rangle$, any sub-path starting from a $\langle s', v'\rangle$ such that $v' \models_{PL} \phi$ also contains at least one $\langle s'', v''\rangle$ such that $v'' \models_{PL} \psi$.

Example 5 *Examples of reachability analysis that would be relevant for the system used in Example 2 include:*

- *Ensuring the count of transferred files is always within what we expect:* `AG` $fetched \geq 0 \wedge fetched \leq 386 \wedge sent \geq 0 \wedge sent \leq 386$
- *Checking consistency between the number of sent and fetched files:* `AG` $fetched = sent$
- *Verifying that all files always end up being received:* `AF` $fetched = 386$

2.2 UPPAAL and Networks of Timed Automata

This section summarizes the differences brought by timed automata and networks of timed automata to the classical automata described previously. The features presented here are those found within UPPAAL, a modeling tool for networks of timed automata introduced in [7]. The addition of time leads to states in which a new type of variables, clocks, evolve even when no transition is activated. To account for this, *states* are now referred to as *locations* instead. Transitions are all instantaneous. Readers looking for further information on timed automata are encouraged to check out the papers in which they were first described ([1] and [2]), or a more in-depth introductory course on the subject ([14] and [45]).

2.2.1 System Definition

Definition 9 (Clocks) *Timed automata feature a special type of variable, called clocks, which model the passing of time. Transitions are instantaneous, can reset clocks (but not set them to a specific value), and have guards referring to the clock's current value. Within a location, however, time passes at the same rate for all clocks in the system.*

Definition 10 (Syntax of Constraints and Actions) *Given a set of variables* `Var`*, and a set of clocks* `Clocks`*, the grammar used when writing constraints and actions in transitions is as follows, with* ***ident*** *standing for a variable in* `Var`*, and* ***clk*** *standing for a clock in* `Clocks`*:*
lop ::= $\wedge \mid \vee$
cop ::= $< \mid \leq \mid = \mid \geq \mid >$
mop ::= $+ \mid - \mid * \mid /$
val ::= ***ident*** $\mid \mathbb{Z}$
mexpr ::= ***mexpr mop mexpr*** $\mid$ ***val***
bexpr ::= $\neg$***bexpr*** $\mid$ ***bexpr lop bexpr*** $\mid$ ***mexpr cop mexpr*** $\mid$ ***clk cop val*** $\mid$ ***clk*** $-$ ***clk cop val*** $\mid$ `true` $\mid$ `false`

$\textbf{\textit{iexpr}} ::= \textbf{\textit{iexpr}} \wedge \textbf{\textit{iexpr}} \mid \textbf{\textit{clk}}\ \textbf{\textit{cop}}\ \textbf{\textit{val}} \mid \textbf{\textit{clk}} - \textbf{\textit{clk}}\ \textbf{\textit{cop}}\ \textbf{\textit{val}} \mid \texttt{true}$
$\textbf{\textit{assign}} ::= \textbf{\textit{assign}};\ \textbf{\textit{assign}} \mid \textbf{\textit{ident}} := \textbf{\textit{mexpr}} \mid if\ (\textbf{\textit{bexpr}})\ \{\textbf{\textit{assign}}\} \mid \textbf{\textit{clk}} := 0 \mid \texttt{nop}$

Definition 11 (Locations) *Unlike in classic automata, states are referred to as locations, since a state is now defined by a location and a value for each clock (and a value for each integer variable in our framework). Time related attributes can be applied to locations:*

`urgent`: *This location must be left before any time passes.*

`committed`: *This location must be left before any time passes, and only transition leaving a* `committed` *location are enabled.*

Invariant in iexpr: *Locations can feature an invariant on clocks, which must be verified in order for the location to exist.*

The difference between an `urgent` and a `committed` location is only meaningful if there are multiple automata. Example 7 uses a network of automata to illustrate the difference between these two attributes.

Definition 12 (Timed Automata System) *A timed automata system $\mathcal{A}$ is a $\langle Q, \texttt{Inv}_Q, s_{init}, \mathcal{B}, \mathcal{E}, \texttt{Prio}_\mathcal{E}, \texttt{Var}, \texttt{Clocks}, \texttt{Act}, \rightsquigarrow \rangle$ tuple, where:*

- *Q is a finite set of locations. $Q^{\texttt{ugt}} \subseteq Q$ denotes the* `urgent` *locations, and $Q^{\texttt{cmt}} \subseteq Q$ the* `committed` *ones. $Q^{\texttt{ugt}} \cap Q^{\texttt{cmt}} = \emptyset$.*
- *$\texttt{Inv}_Q : Q \to \textbf{\textit{iexpr}}$ indicates the invariant of each location.*
- *s_{init} is the initial location ($s_{init} \in Q$).*
- *`Var` is a finite set of variables.*
- *`Clocks` is a finite set of clocks.*
- *$\mathcal{E} = \mathcal{E}^\alpha \cup \mathcal{E}^{\texttt{sync}}$ is a finite set of labels, with $\mathcal{E}^{\texttt{sync}}$ corresponding to labels meant for synchronization and $\mathcal{E}^\alpha$ being regular labels, with $\mathcal{E}^{\texttt{sync}} \cap \mathcal{E}^\alpha = \emptyset$. The labels in $\mathcal{E}^{\texttt{sync}}$ affixed by either '?' or '!', with '?' denoting a reception on a "channel", and '!' an emission. In addition, synchronization labels can be further categorized into $\mathcal{E}^{\texttt{ugt}} \subseteq \mathcal{E}^{\texttt{sync}}$ corresponding to the* `urgent` *channels, and $\mathcal{E}^{\texttt{brd}} \subseteq \mathcal{E}^{\texttt{sync}}$ corresponding to the* `broadcast` *ones.*
- *$\texttt{Prio}_\mathcal{E} : \mathcal{E}^{\texttt{sync}} \to set(\mathcal{E}^{\texttt{sync}})$ indicates, for any label, which labels have a strictly lower priority. It satisfies the following properties: $\forall l_1, l_2 \in \mathcal{E}^{\texttt{sync}},\ (l_1 \notin \texttt{Prio}_\mathcal{E}(l_1)) \wedge (l_2 \in \texttt{Prio}_\mathcal{E}(l_1)) \implies (l_1 \notin \texttt{Prio}_\mathcal{E}(l_2) \wedge \texttt{Prio}_\mathcal{E}(l_2) \subset \texttt{Prio}_\mathcal{E}(l_1))$.*
- *$\mathcal{B} =$ `bexpr`(`Var`, `Clocks`), as defined in Definition 10.*
- *`Act` =* `assign`*(`Var`, `Clocks`), as defined in Definition 10.*
- *$\rightsquigarrow \subseteq Q \times \mathcal{B} \times \mathcal{E} \times \texttt{Act} \times Q$ is the transition relation.*

The semantics of $\mathcal{A}$ is given via its valid transitions (see Definition 14) and its execution traces (see Definition 15).

Definition 13 (Clock Valuation) *Clocks are kept separate from standard variables, including in the definition of the valuation. $h : \texttt{Clocks} \to \mathbb{R}^+$ is the function mapping each clock to its valuation. As a shorthand, the increment of the value of all clocks in h by t units of time is written $(h + t)$.*

Definition 14 (Transition) *Let* $\mathcal{A} = \langle Q, \mathit{Inv}_Q, s_{init}, \mathcal{B}, \mathcal{E}, \mathit{Prio}_\mathcal{E}, \mathit{Var}, \mathit{Clocks}, \mathit{Act}, \rightsquigarrow \rangle$ *be an automaton. Given a location* $s \in Q$*, a valuation* v*, a clock valuation* h*, a duration* t *and a transition* $\langle s, c, l, a, s' \rangle \in \rightsquigarrow$*, we define the set of the reachable states (without considering priorities)* $\mathit{Reach}(\langle s, v, h \rangle, \langle s, c, l, a, s' \rangle, t) \triangleq \{\langle s', v', h' \rangle | \langle v, (h+t) \rangle \models_{PL} c \wedge v' = v[a] \wedge h' = (h+t)[a] \wedge \langle v, h \rangle \models_{PL} \mathit{Inv}_Q(s) \wedge \langle v', h' \rangle \models_{PL} \mathit{Inv}_Q(s')\}$.

We now define `Step`*, which indicates all valid transitions that can be performed, and takes into account that no transition with higher priority is doable.*

$\mathit{Step}(\langle s, v, h \rangle, t) \triangleq \{\langle s', v', h' \rangle | \exists \langle s, c, l, a, s' \rangle \rangle \in \rightsquigarrow$ *s.t.* $(\langle s', v', h' \rangle \in \mathit{Reach}(\langle \{s, v, h\rangle, t) \wedge \forall \langle s_b, c_b, l_b, a_b, s'_b \rangle \in \rightsquigarrow,$ *if* $l \in \mathit{Prio}_\mathcal{E}(l_b)$ *then* $\mathit{Reach}(\langle \{s, v, h\rangle, \langle s_b, c_b, l_b, a_b, s'_b \rangle, t)$ *is empty.)*$\}$

Definition 15 (Path & Trace) *We consider a path to be a* maximal *sequence of states/transitions* $\langle s_1, v_1, h_1 \rangle \rightarrow^{t_1} \langle s_2, v_2, h_2 \rangle \rightarrow^{t_2} \cdots$ *such that* $\forall i \langle s_{i+1}, v_{i+1}, h_{i+1} \rangle \in \mathit{Step}(\langle s_i, v_i, h_i \rangle, t)$*. The sequence is maximal in the sense that it is either infinite or of length* N *and such that* $\mathit{Step}()$ *is empty for any* $t \in \mathbb{R}$*. A path starting from* $\langle s_{init}, v_0, h_0 \rangle$ *(where* v_0 *is the initial valuation and* h_0 *associates each clock with 0) is called a trace.*

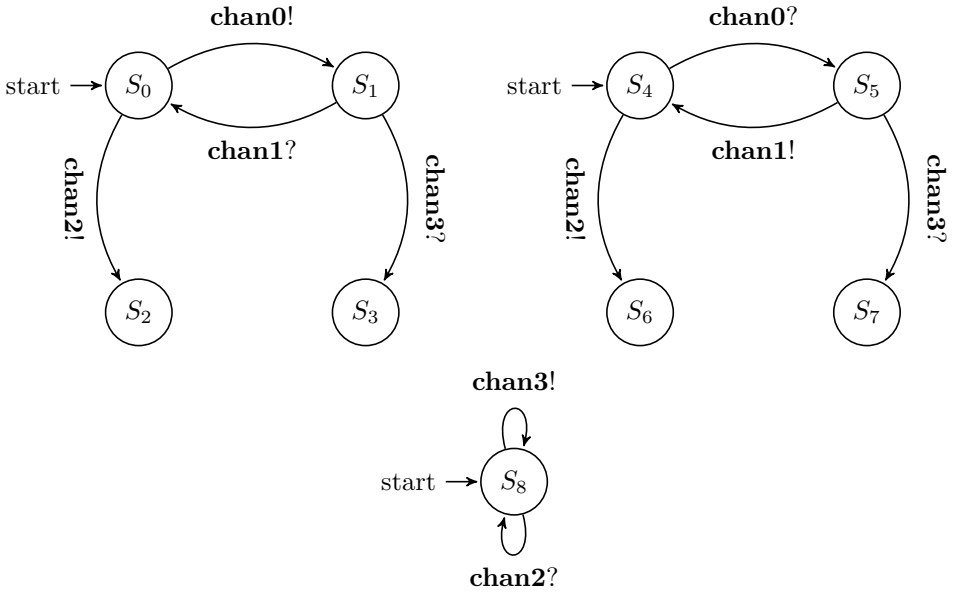


Figure 2.3 – Example of Network of Timed Automata

Example 6 (Urgent and Committed Locations) *Consider the network of timed automata shown in Figure 2.3. Without any attributes on locations, the following path is legal:* $\langle\langle S_0, S_4, S_8\rangle, \{\}, \{\langle C_0, 0\rangle\}\rangle \xrightarrow{\langle \mathit{chan0!}, \mathit{chan0?}, - \rangle}{}^{75} \langle\langle S_1, S_5, S_8\rangle, \{\}, \{\langle C_0, 75\rangle\}\rangle \xrightarrow{\langle \mathit{chan3?}, -, \mathit{chan3!} \rangle}{}^{39} \langle\langle S_3, S_5, S_8\rangle, \{\}, \{\langle C_0, 114\rangle\}\rangle$ *Let us now consider* S_1 *as* `urgent`*. The previous* $\langle\langle S_1, S_5, S_8\rangle, \{\}, \{\langle C_0, 75\rangle\}\rangle \xrightarrow{\langle \mathit{chan3?}, -, \mathit{chan3!} \rangle}{}^{39}$ *transition is no longer legal, as a maximum of* 0 *units of time is now allowed to be stayed at* S_1*. If we instead considered* S_5 *to be* `committed`*, that transition would still not be allowed to occur after more than* 0 *units of time, but in addition, it would also be illegal because it does not involve a transition from a* `committed` *location (neither* S_1 *nor* S_8*, the two locations involved in the transition,*

are `committed`*). Performing* $\langle\langle S_1, S_5, S_8\rangle, \{\}, \{\langle C_0, 75\rangle\}\rangle \xrightarrow{\langle -, \textbf{chan3?}, \textbf{chan3!}\rangle}{}^{0}$ *instead would be legal (although the resulting state is different).*

Definition 16 (Channel Attributes) *In UPPAAL, all synchronizations have a single emitter transition, but the number of receivers depends on the type of channel: standard channels have exactly one receiver, and* `broadcast` *channels activate any automaton able to perform a reception on that channel (even allowing the emitter to transition alone if no receiver is available). It should be noted that available receivers for the channel are forced to transition.*

Communication channels can have attributes related to time. Indeed, the `urgent` *attribute indicates that, if a transition featuring this channel is able to occur, it must do so before any time passes.*

In a network of automata, the $\texttt{Prio}_{\mathcal{E}}$ *function is shared by all automata. Thus, the priority of channels is the same accross all automata.*

Example 7 (Urgent and Broadcast Channels) *Consider the network of timed automata shown in Figure 2.3. Without any attributes on channels, the following paths are legal:*

- $\langle\langle S_0, S_4, S_8\rangle, \{\}, \{\}\rangle \xrightarrow{\langle \textbf{chan0!}, \textbf{chan0?}, -\rangle}{}^{75} \langle\langle S_1, S_5, S_8\rangle, \{\}, \{\}\rangle \xrightarrow{\langle \textbf{chan3?}, -, \textbf{chan3!}\rangle}{}^{39} \langle\langle S_3, S_5, S_8\rangle, \{\}, \{\}\rangle \xrightarrow{\langle -, \textbf{chan3?}, \textbf{chan3!}\rangle}{}^{42} \langle\langle S_3, S_7, S_8\rangle, \{\}, \{\}\rangle$
- $\langle\langle S_0, S_4, S_8\rangle, \{\}, \{\}\rangle \xrightarrow{\langle \textbf{chan2!}, -, \textbf{chan2?}\rangle}{}^{23} \langle\langle S_2, S_4, S_8\rangle, \{\}, \{\}\rangle$
- $\langle\langle S_0, S_4, S_8\rangle, \{\}, \{\}\rangle \xrightarrow{\langle -, \textbf{chan2!}, \textbf{chan2?}\rangle}{}^{0} \langle\langle S_0, S_6, S_8\rangle, \{\}, \{\}\rangle \xrightarrow{\langle \textbf{chan2!}, -, \textbf{chan2?}\rangle}{}^{78} \langle\langle S_2, S_6, S_8\rangle, \{\}, \{\}\rangle$

Let us now consider **chan0** *as* `urgent`*.* $\langle\langle S_0, S_4, S_8\rangle, \{\}, \{\}\rangle \xrightarrow{\langle \textbf{chan0!}, \textbf{chan0?}, -\rangle}{}^{75}$ *is no longer a legal transition: as the synchronization on* **chan0** *is able to occur, it must do so before any time passes. The transition would be legal if* 75 *was* 0 *instead. Furthermore, the transition* $\langle\langle S_0, S_4, S_8\rangle, \{\}, \{\}\rangle \xrightarrow{\langle \textbf{chan2!}, -, \textbf{chan2?}\rangle}{}^{23}$ *is also not legal, for the same reason: despite* **chan0** *not being synchronized on, it was still an available synchronization and so no time must pass while the* **chan0** *synchronization is available. Replacing* 23 *by* 0 *makes this a legal transition. The third proposed path is still allowed when* **chan0** *is marked as urgent: on the first transition,* **chan0** *can be synchronized on, but no time passes prior to the transition occuring; on the second transition,* **chan0** *can no longer be synchronized on, and so its* `urgent` *attribute is not taken into account.*

Let us now consider **chan3** *as a broadcast channel.* $\langle\langle S_1, S_5, S_8\rangle, \{\}, \{\}\rangle \xrightarrow{\langle \textbf{chan3?}, -, \textbf{chan3!}\rangle}{}^{39}$ *is not longer a legal transition:* S_5 *is able to synchronize on* **chan3** *as well, and thus must do so. On the other hand, a previously illegal transition is now possible:*$\langle\langle S_1, S_5, S_8\rangle, \{\}, \{\}\rangle \xrightarrow{\langle \textbf{chan3?}, \textbf{chan3?}, \textbf{chan3!}\rangle}{}^{39}$*. Perhaps more surprisingly, the following path is legal:* $\langle\langle S_0, S_4, S_8\rangle, \{\}, \{\}\rangle \xrightarrow{\langle -, -, \textbf{chan3!}\rangle}{}^{3} \langle\langle S_0, S_4, S_8\rangle, \{\}, \{\}\rangle \xrightarrow{\langle -, -, \textbf{chan3!}\rangle}{}^{89}$*. Indeed, all automata able to synchronize (none) do so. Note that only the receiving label can have any number of occurrences in the transition, there is always a single emitting label. Thus, setting* **chan2** *as a broadcast channel instead would not cause any change in legal paths.*

Example 8 *In an evolution from Example 2, Figure 2.4 shows a network of automata modeling the exact same scenario, but with the addition temporal behaviors. The server now takes between 32*

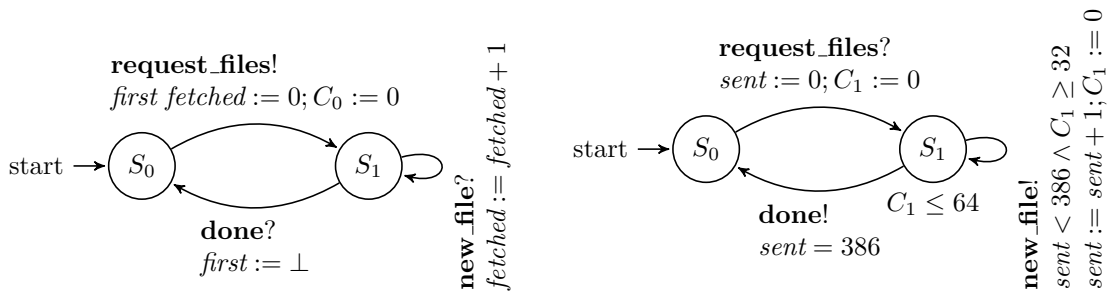


Figure 2.4 – Another example of timed automata

and 64 time units before providing each file. Transfer times are thus able to be modeled. To avoid having both automata wait forever in their S_0 location, we consider the ***request_file*** *channel as being* `urgent`*. Similarly, the* ***done*** *channel needs to be made urgent for the S_1 locations to be left as soon as all files were transfered. Instead of having the* ***request_file*** *channel be* `urgent`*, another solution would be to mark the client's S_0 location as* `urgent`*. This would also lead to a deadlock after all transfers have been done.*

2.2.2 Query Logic Operators and Semantics

UPPAAL uses the temporal operators described in Section 2.1.2. In this case however, ϕ is a **bexpr** ::= ¬**bexpr** | **bexpr lop bexpr** | **mexpr cop mexpr** | **clk cop val** | **clk − clk cop val** | `true` | `false` | `deadlock` | **sub-automaton.location**, with `deadlock` being true if and only if no transition is able to occur and time is not allowed to pass, and **sub-automaton.location** being true if and only if said sub-automaton is currently in the given location.

In addition to these temporal operators, UPPAAL features operators that seek the extremum of either a clock or a variable, with the possibility of an invariant delimiting the system locations in which the value is considered. Given either a variable or a clock t, $\texttt{sup}\{\phi\} : t \triangleq$ maximum value for t across all traces, such that this value has been reached in a state satisfying ϕ.
The `inf` operator is the equivalent for finding the minimum value.

Example 9 *In Example 8, computation of the WCET for file fetching can now be achieved through model checking:* `sup`$\{client.S_1\} : C_0$*. This query looks for the maximum value reached by C_0 when the client automaton is in the S_1 location. Considering max to be the returned value, a trace leading to that value could be obtained by using UPPAAL to query for* `AG` $\neg((C_0 = max) \wedge client.S1)$

2.3 Conclusion

Timed automata can be used to model complex systems featuring real-time constraints. Through the use of queries verified using formal methods such as model checking, these models are then used to validate properties for the systems, such as ascerting its correct behavior or computing running time for some of its components.

In the next chapter, cache coherence mechanics are described. The behavior of cache coherence protocols is defined using (classical) automata. Because of the large size of such automata, a matrix representation is used instead. In this representation, lines correspond to locations, and columns

correspond to labels and guards. For example, the client automaton described in Figure 2.1 would have the matrix representation shown in Figure 2.5.

Location	**request_files!**	**done?**	**err**	**new_file?**
S_0	$fetched := 0$, S_1		S_E	
S_1		S_0	S_E	$fetched := fetched + 1$
S_E				

Figure 2.5 – Example of matrix automaton representation

Chapter 3

Fundamentals of Cache Coherence

This chapter presents all the notions related to cache coherence required for the understanding of the thesis. The content described is based on [49].

Notations Seq(A) indicates a finite sequence, potentially empty, composed of type A elements. Sequences are thus defined as

$$\text{Seq}(A) = \begin{cases} [] \\ A :: \text{Seq}(A) \end{cases}$$

The addition of an element e at the head of a sequence S is written push(e,S), standing for $e :: S$. The retrieval and removal of the head of a sequence S is written pop(S), returning $head(S)$ prior to applying $S \leftarrow tail(S)$. Lastly, isEmpty(S) indicates whether S is an empty sequence, and is equivalent to testing if $S = []$.

3.1 Components

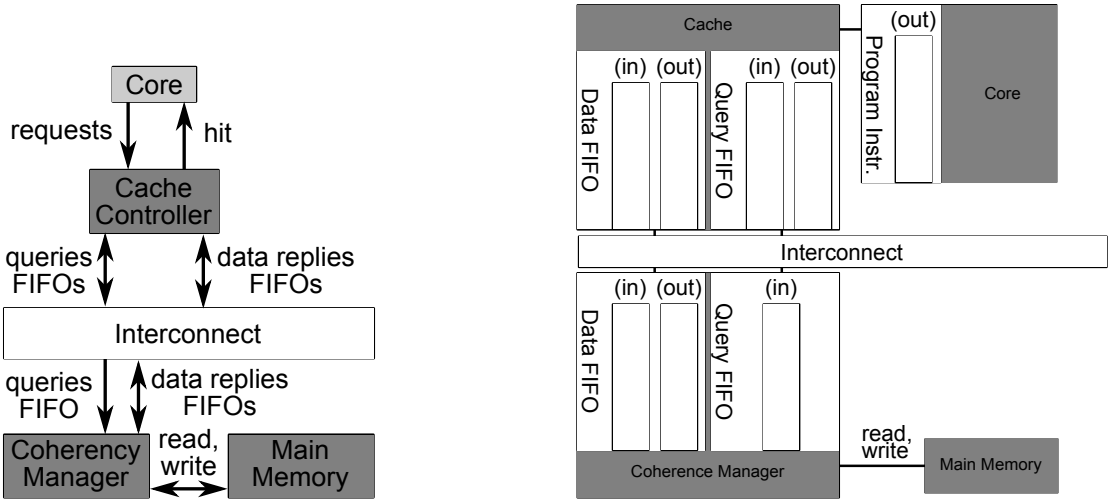
In this section are presented the various components involved in maintaining cache coherence in a multi-core architecture. Figure 3.1 shows how the components presented in this section are interconnected, with Figure 3.1a providing an overview and Figure 3.1b making the involved FIFO queues more apparent.

Architectures generally feature instruction, data, as well as caches holding both. Instruction caches are not affected by cache coherence and thus not considered in this chapter.

3.1.1 Memory Elements

The memory of a system is partitioned into addressable blocks. That is, there are blocks of a certain size for which performing any access operation on its content requires performing an operation on the block in its entirety.

Cache coherence is maintained over a cache line. Cache lines form another partitioning of the system's main memory, such that all cache lines contain the same amount of addressable blocks. Thus, even without accounting for the possibility of multiple addresses pointing to the same atomic block, there may be multiple addresses pointing to different parts of a same cache line. The content of each cache line is made of contiguous memory blocks. Thus, if a cache line of 16 elements starts with a block of address 42, it will also have the blocks of addresses 43 through 57. If ignored,



(a) Overview

(b) Showcasing the FIFOs

Figure 3.1 – Components involved in cache coherence

this mismatch between blocks used for cache coherence and for program instructions can lead to unexpected results, such as false sharing (see Appendix A).

To avoid unwarranted complications, this thesis merges the two partitioning types into one: memory elements. In effect, we consider that memory elements are cache lines with the size of a single addressable block, but change the term so that this simplification remains explicit. Furthermore, since there is only a single address per memory element, a memory element and its address can be used interchangeably to refer to one another.

Definition 17 (Memory Element) *The set of all memory elements is defined as* $\texttt{Addr} \subseteq \mathbb{N}$.

3.1.2 Core: Programs & Instructions

Cache coherence is only affected by programs accessing the memory. As a result, only the instructions related to the writing (`store`), reading (`load`), and the eviction (`evict`) of memory elements are of any considered while examining cache coherence. All other instructions are assimilated into the no operation instruction (`nop`). As a result, when observed through the lens of cache coherence, cores are executing programs by adding instructions into a FIFO queue, without any possibility for branching or jumping. Each of these instructions being applied to a single memory element.

The resolution of each instruction by its cache is signaled instantly to the core, bypassing the need for a dedicated queue.

Definition 18 (Instruction) *The set of all operators is defined as* $\texttt{OPs} = \{\texttt{load}, \texttt{store}, \texttt{evict}, \texttt{nop}\}$, *and instructions are defined as* $\texttt{Instr} = \texttt{OPs} \times \texttt{Addr}$

Definition 19 (Program) *Programs are sequences of instructions, thus* $\texttt{InstrQueue} = \mathit{Seq}(\texttt{Instr})$.

3.1.3 Caches

Definition 20 (Cache) *The set of all caches in the system is defined as* $\mathtt{Ccs}$*. Some components are present in both the caches and the coherence manager. We define* $\mathtt{Ccs}^+ = \mathtt{Ccs} \cup \{\mathtt{mgr}\}$ *to add the coherence manager (*$\mathtt{mgr}$*). Furthermore, we define* $\mathtt{nc}$*, not included in* $\mathtt{Ccs}$*, to indicate the absence of a cache where one could be. The caches can also be referred to using naturals from* 1 *to* $\mathtt{cc}$*, with* $\mathtt{cc} = |\mathtt{Ccs}| - 1$*.*

Caches are tasked with obtaining copies of memory elements able to fulfill the memory accesses that are requested of them by their core. This requires keeping track of permission for each copy of memory element they own, as well as keeping track of operations (steps of the resolution of an instruction) that are in progress. To do so, all caches assign a single state to each of their memory element copies. These states are split into two categories: *stable* states, solely denoting that the cache has a certain permission for that memory element, and *transient* states, which indicate that the resolution of a core's request is in progress and still awaiting either the broadcast of a previously prepared query, or the reception of a data message (or both).

Definition 21 (States in Caches) $\mathcal{S}_c{}^s$ *is the set of all stable states defined by the cache coherence protocol, and* $\mathcal{S}_c{}^t$ *is the set of the transient ones. To shorten some notations, we also define* $\mathcal{S}_c = \mathcal{S}_c{}^s \cup \mathcal{S}_c{}^t$*, the set all cache states. States cannot be both transient and stable, thus* $\mathcal{S}_c{}^s \cap \mathcal{S}_c{}^t = \emptyset$*.*

To acquire new permissions, caches send queries on the interconnect. Each of these queries pertains to a single memory element and leads to at most a single reply being received by the emitter. For most types of data replies, a copy of the memory element is part of the message. The types of queries that can be sent are dependent on the actual cache coherence protocol. However, the protocols used in this thesis all rely on the same ones: demand a read-only copy of a memory element (`GetS`, likely following a `load` instruction), demand a read-and-write copy (`GetM`, likely following a `store` instruction), and signal the eviction of a potentially modified copy (`PutM`, likely following an `evict`). Similarly, the types of replies that can be exchanged are also dependent on the coherence protocol. For those described in this thesis, only three types are required: a message containing the memory element (`data`), one also indicating that no other cache currently has access to that memory element (`data-e`), and a message indicating that no copy of the memory element is going to be sent (`no-data`).

Definition 22 (Query) *The categories of queries are defined as* $\mathtt{Query} = \{\mathtt{GetS}, \mathtt{GetM}, \mathtt{PutM}\}$*. A query message is an element of* $\mathtt{MSG}_{\mathtt{query}} : \mathtt{Query} \times \mathtt{Addr} \times \mathtt{Ccs}$*, which indicates the type of query, the memory element being targeted, and the query's emitter.*

Definition 23 (Reply) *The categories of reply messages are defined as* $\mathtt{Reply} = \{\mathtt{data}, \mathtt{data\text{-}e}, \mathtt{no\text{-}data}\}$*. A reply message is an element of* $\mathtt{MSG}_{\mathtt{data}} : \mathtt{Reply} \times \mathtt{Addr} \times \mathtt{Ccs}^+$*, which indicates its category, relevant memory element, and targeted cache (or* $\mathtt{mgr}$*, when targeting the coherence manager).*

Depending on what the cache coherence protocol allows, it is possible for caches to receive queries they are in charge of replying to, despite not yet having the data to do so. To handle such cases, caches can associate the identifier of another cache with each address. By doing so, they are able to send the data reply at a later date. As there is always at most one cache in charge of providing a reply for any query, this can lead to caches waiting on a reply, from a cache itself waiting on a reply, also receiving a query to which they are supposed to be providing an answer. As the protocols ensure that there is always exactly one component charged with replying for each memory element, this cannot lead to a deadlock.

Definition 24 (Information in Caches) *Each cache associates a state to each memory element, and can also associate with it the identifier of another cache. We define* $\mathtt{C_s} : \mathtt{Ccs} \to \mathtt{Addr} \to \mathtt{S_c}$ *as the function indicating the state associated with a given memory element by a given cache, and* $\mathtt{r} : \mathtt{Ccs} \to \mathtt{Addr} \to (\mathtt{Ccs} \cup \{\mathtt{nc}\})$ *is the function indicating the cache (or lack thereof) expecting a data reply in the future for a given memory element in a given cache.*

Each cache has four FIFO queues, each one handling either incoming or outgoing queries or data messages, as seen in Figure 3.1b.

Definition 25 (Cache FIFOs) *The four FIFOs of each cache are defined as:* $\mathtt{D_{in}} : \mathtt{Ccs}^+ \to Seq(\mathtt{MSG_{data}})$ *and* $\mathtt{D_{out}} : \mathtt{Ccs}^+ \to Seq(\mathtt{MSG_{data}})$ *for the incoming and outgoing data message queues;* $\mathtt{Q_{in}} : \mathtt{Ccs}^+ \to Seq(\mathtt{MSG_{query}})$ *and* $\mathtt{Q_{out}} : \mathtt{Ccs} \to Seq(\mathtt{MSG_{query}})$ *for the incoming and outgoing query message queues.*

The behavior of a cache is described through a transition system focused on a single memory element E and reacting to events (Definition 27). The available actions within a transition for a cache C are as follows:

- **Stalling:** stalling, written as `stall`, is a special action indicating that the event cannot be processed while the memory element is in this state. In effect, the event that led to this action being taken (be it an instruction from the core, an incoming query, or an incoming data message) is not removed from its queue and stays ready to be re-evaluated in any state where `stall` is no longer an action it leads to.
- **Completing a core's request:** the `hit` indicates the cache has fulfilled one of its core's ongoing request for E. In some cases, multiple requests of different types (`load`, `store`, `evict`) may have be ongoing in parallel, leading the operator being fulfilled by the `hit` to be made explicit (e.g. `store hit`).
- **Preparing a query:** a cache C performing $Q?$ is pushing a query of type Q for the memory element E in its outgoing query queue. $\mathtt{Q_{out}}(C) \leftarrow \text{push}(\langle Q, E, C\rangle, \mathtt{Q_{out}}(C))$.
- **Changing state:** The state attributed to E is changed by writting N, with N being the new state. This corresponds to performing $\mathtt{C_s}(C, E) \leftarrow N$.
- **Preparing a data reply:** when sending a data message, there are three possible targets: the emitter of an incoming query (`s`), the coherence manager (`m`), and a previously memorized cache (`r`). $T!D$ indicates that a data message of type D is added to the outgoing data queue, with T being its target. $\mathtt{D_{out}}(C) \leftarrow \text{push}(\langle D, E, T\rangle, \mathtt{D_{out}}(C))$
- **Memorizing a cache:** if an incoming query from C_2 is expected to be replied to by C, yet C is not currently able to provide data, the C_2 is memorized so that C knows to send data to C_2 when able. This is indicated by $\mathtt{r} \leftarrow \mathtt{s}$, which is equivalent to $\mathtt{r}(C, E) \leftarrow C_2$. Although it does not impact the workings of cache coherence, the protocols shown in this thesis also indicate whenever the memorized cache can be cleared ($\mathtt{r} \leftarrow \mathtt{nc}$, which is the same as $\mathtt{r}(C, E) \leftarrow \mathtt{nc}$.

3.1.4 Coherence Manager

Maintaining cache coherence requires the caches to coordinate with each other. Depending on the protocol, this may require some information to be accessible to all caches. The coherence manager is a representation of this information. This may not match any single one physical component of

the system (for example if there is no shared last level cache), as with certain protocols, its behavior may be implemented through direct communication channels between the caches. The role of the coherence manager is to act when none of the caches is able to provide an answer to a cache's query. In order to detect such queries, the coherence manager also assigns a state to each memory element. In effect, the coherence manager acts as an intermediary between the caches and the system's main memory. Furthermore, the coherence manager keeps track of which cache, if any, is in charge of replying to queries.

Definition 26 (Coherence Manager) *Let $\mathcal{S}_m$ the set of states that can be attributed by the coherence manager, $M_s : \mathit{Addr} \to \mathcal{S}_m$ is the function indicating which state is attributed to each memory element. The cache (or lack thereof) associated with each memory element is defined as $o : \mathit{Addr} \to (\mathit{Ccs} \cup \{\mathit{nc}\})$.*

Much like the caches, the coherence manager uses FIFO queues to handle incoming and outgoing messages. It has one less queue however, as the coherence manager never sends any query. This can be seen in Definition 25, where all but one of the types of FIFO queues are shared with the caches.

For a memory element E, the coherence manager's behavior can be defined through the following actions:

- **Stalling:** the coherence manager may sometimes not be able to properly react to an incoming event. Just like caches, it can then use the special `stall` action to indicate that the event should not be processed while E is in this state, leaving the event (either a query or a data message) untouched in its queue.
- **Changing state:** as with the caches, the state attributed to E by the coherence manager is changed by writing N, with N being the new state. This corresponds to $\mathtt{M_s}(E) \leftarrow N$.
- **Preparing a data reply:** the coherence manager replies to incoming cache queries by sending data. With the cache that emitted the query being referred to as `s`, the coherence manager can send a data message of type D by performing $T!D$. This is equivalent to $\mathtt{D_{out}}(\mathtt{mgr}) \leftarrow \text{push}(\langle D, E, T\rangle, \mathtt{D_{out}}(\mathtt{mgr}))$
- **Memorizing the current owner:** the coherence manager keeps track of which cache is currently in charge of distributing data for each memory element, if there is one. Similarly to the memorizing of a cache by another, having a cache C_2 be considered to have this role is done by $\mathtt{o} \leftarrow \mathtt{s}$, which corresponds to $\mathtt{o}(E) \leftarrow C_2$. This value can also be cleared by having $C_2 = \mathtt{nc}$.

3.1.5 Interconnect

The interconnect links all caches together, as well as the coherence manager. It broadcasts queries from the caches to all the components it is linked to, including the original query emitter. The replies, however, are targeted, and thus only received by a single component.

Transfers between components are not instantaneous. Instead, outgoing message queues are used to wait for access to the interconnect to become available. Conversely, the interconnect deposits messages in incoming message queues so that it does not have to synchronize with each component. Depending on the system, some restrictions on how the interconnect behave can be present. Indeed, not only is there generally an access policy controlling the order in which outgoing messages are being sent (Round-Robin being commonly used), but it is also possible for the interconnect to not allow a new query to be broadcasted until the previous one has been resolved (such interconnects are

called atomic), whereas other interconnects allow the interleaving of queries (split-transaction, where data and query use separate channels). This thesis focuses on split-transaction buses architectures, as atomic buses are not found in contemporary architectures.

In effect, the actions performed by the transitions of an automaton corresponding to the interconnect are as follows:

- **Query Broadcast:** Given a cache C_0 such that $\neg\text{isEmpty}(\texttt{Q}_{\texttt{out}}(C_0))$, a query broadcast takes a query Q out of C_0's outgoing query queue ($Q = \text{pop}(\texttt{Q}_{\texttt{out}}(C_0))$) and adds it to every incoming query queues, including C_0's ($\texttt{Q}_{\texttt{out}}(C_1) \leftarrow \text{push}(Q, \texttt{Q}_{\texttt{out}}(C_1))$).

- **Data Transfer:** Given C_0, either a cache or the coherence manager, such that $\neg\text{isEmpty}(\texttt{D}_{\texttt{out}}(C_0))$, a data transfer for $\langle D, E, T\rangle = \text{pop}(\texttt{D}_{\texttt{out}}(C_0))$ adds the corresponding data message to T's incoming data messages queue ($\texttt{D}_{\texttt{in}}(T) \leftarrow \text{push}(\langle D, E, T\rangle, \texttt{D}_{\texttt{in}}(T))$).

3.2 Coherence Protocols

When definiting the protocols, the behavior of the caches and the coherence manager is given in the form of a classical automata (see Chapter 2), and limited to a single memory component.

3.2.1 Introduction to the MSI Protocol

The archetypal cache coherence protocols are those belonging to the MSI protocol family. In their basic version, these protocols feature three stable states from which the MSI name is derived:

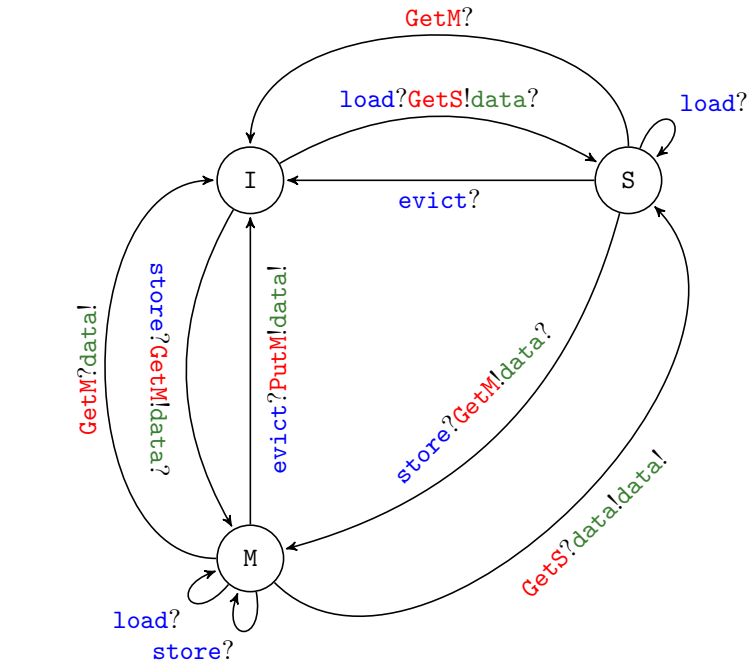
- **Modified:** When a cache assigns the Modified state to a memory element, it indicates that this cache performed at least one write to that memory element. While in this state, it can freely read and write to that memory element. This corresponds to being the *Single Writer* from Property 2.

- **Shared:** When a cache assigns the Shared state to a memory element, it indicates that this cache is able to freely read that memory element but not write to it. This corresponds to being one of the *Multiple Readers* (and potentially the only one) of Property 2.

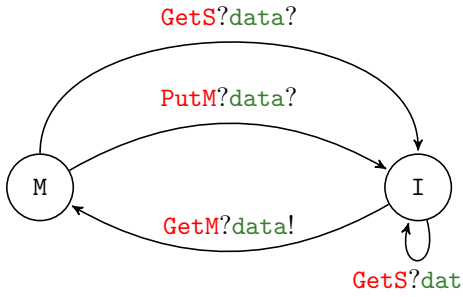
- **Invalid:** When a cache assigns the Invalid state to a memory element, it is no longer considered to be holding a copy of that memory element, thus absolving this cache from verifying Property 1. Not having a copy, the cache can neither read nor write to the memory element.

To keep it simple, the protocol presented in this section does not use any FIFO queue. Instead, all exchanges are done synchronously. The only elements taken into account are the state of the memory elements and the points of synchronization (i.e. instruction and message exchanges). Thus, the notations on the automata of this section are not the notations presented in the previous section, but are instead based on CCS (Calculus of Communicating Systems, [37]): a `?` marks the reception of a signal, and `!` marks its sending. Queries are still sent to all components, and data only to a single one. Furthermore, only a single transaction (that is, the resolution of an instruction) can occur at any moment. This is reflected by transitions featuring multiple synchronizations, as no other actions would be permitted to be interleaved.

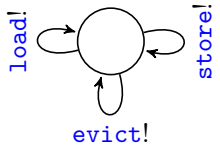
Figure 3.2 shows the two automata corresponding to this simplified protocol. Each of these automata pertains to a single memory element, with states corresponding to what the component attributes to that memory element.



(a) Cache



(b) Coherence Manager



(c) Core

Figure 3.2 – Overview of the MSI Protocol

In the case of the coherence manager (Figure 3.2b), the two states simply correspond to whether or not the coherence manager is in charge of providing the current value for that memory element: `M` indicates that one of the caches is currently in the `M` state, meaning that the value stored in the system's main memory may be out of date; `I` indicates that the main memory's value for that element is up-to-date, leaving (in the MSI protocol) the coherence manager in charge of propagating copies as the caches demand them.

Example 10 (Simple MSI Transition) *Consider a system with two caches, C_A and C_B, and a single memory element E, such that, initially, C_A holds no copy of E (`I` state) and C_B holds one in the `S` state, the coherence manager has the `I` state assigned to E. Were C_A to receive a `store` request, it would broadcast a `GetM` to itself, C_B, and the coherence manager. Upon receiving the `GetM`, C_B would have to abandon the `S` state for `I` in order to maintain Property 2. The coherence manager would react to receiving the `GetM` by sending a copy of E from the system's main memory to C_A (`data`!) before moving to `M` so as to memorize that the most up-to-date value for E is now held by C_A. This `data` reply allows C_A to move to the `M` state, thus completing its transition.*

3.2.2 Properties to be Verified

A system maintaining cache coherence is a system in which the application of each read and write instruction on the same memory elements across multiple caches holds the same results as if these caches shared a permanent single copy of each of those memory elements.

One possible strategy to achieve cache coherence is to enforce Properties 1, 2, and 3. Other approaches may also be possible, but this is the one considered in this thesis.

Property 1 (Caches Have the System-Wide Value) *At any point, for each memory element, all copies of that memory element being held in a cache have the value that was last written to that memory element, regardless of which cache performed the writing.*

Property 2 (Single Writer or Multiple Readers) *At any point, for each memory element, there is either a single cache being able to write to that memory element while the others can neither read nor write to it, or there is any number of caches being able to read the memory element and none able to write to it.*

Property 3 (Forget Me Not) *If a memory element has no copy held in cache, then the system's main memory has the value that was last written to that memory element.*

Cache coherence protocols can be split into categories according to whether they are directory-based or snooping-based, and write-back or write-through.

As this thesis only considers snooping-based protocols, the component descriptions provided in Section 3.1 are unlikely to fit a directory-based protocol. Indeed, in directory-based protocols, cache queries are not broadcasted to every cache but instead addressed to a component that, much like the coherence manager, keeps track of which component may need to see this query and sends a copy of that query to them. In snooping-based protocols, the caches and the coherence manager all receive every query, and do so in the same order.

Write-through protocols ensure that any modification made to a memory element's copy in a cache is also applied to the original in the system's main memory. Conversely, write-back protocols delay the modification of the original until the cache loses the write permissions on its own copy. As it happens, the protocols studied during this thesis are all write-back ones.

3.2.3 Protocol Definition

Protocols are customarily defined by the behavior they require of caches (and the coherence manager, if it is used) for a single memory element according to its currently attributed state in the component and the type of incoming event.

Definition 27 (Event) *Events for a single memory element are defined as* $\texttt{Evts} = \texttt{MSG}_{query} \cup \texttt{Instr} \cup \texttt{MSG}_{data}$

Definition 28 (Protocol) *A protocol is thus defined as two functions:* $\texttt{Pro}_{CC} = \texttt{S}_c \rightarrow \texttt{Evts} \rightarrow set(\texttt{Actions})$*, which defines the behavior of cache controllers, given the state of memory element and an incoming event; and* $\texttt{Pro}_{CMGR} = \texttt{S}_m \rightarrow \texttt{Evts} \rightarrow set(\texttt{Actions})$*, the coherence manager equivalent.* `Actions` *corresponds to the actions described in each component's sub-section.*

Definition 29 (System State) *For the definition of a cache coherence protocol, only a single memory element needs to be considered. Indeed, the behavior is the same for any address, and thus defining it for one is defining it for all. This can be exploited to shorthand the notation of memory element states across the system: Given E the memory element arbitrarily chosen for the protocol's definition, a system* $\langle CC_1, \ldots, CC_n, CM \rangle$ *denotes one in which* $\texttt{M}_s(E) = CM \wedge \forall c \in 0..\texttt{cc},\ \texttt{C}_s(c, E) = CC_c$*. The set of all possible systems states is denoted* `System`*.*

Definition 30 (Stable System Transitions) *To allow reasoning more simply over state changes, another shorthand notation is introduced:* $\texttt{reach} : \texttt{System} \rightarrow \texttt{Instr}^{\texttt{cc}} \rightarrow set(\texttt{System})$ *which, given a system state and instructions to apply on each cache, returns the set of all possible next system states composed solely of stable states that can be reached.*

3.3 Split-Transaction Bus, case of the MSI Protocol

Section 3.2.1 provides an overview of the MSI protocol without considerations for the possibility of interleaving or even asynchronous communications. This section presents an MSI protocol featuring all of those. This leads a large number of additional states and transitions, so instead of using graphs to represent the automata, we use matrices (see Figure 3.3 for the cache controllers, and Figure 3.4 for the coherence manager). The columns in these matrices correspond to the events from Definition 27. Bear in mind that this corresponds to how the caches and the coherence manager act for each memory element, according to the state currently assigned to that state (first column). The **Own Query** column in the cache's table corresponds to when it processes one of its own queries (from its incoming query queue) for that memory element. In the coherence manager's table, the behavior upon reception of a query may differ according to whether the emitter is the owner (as defined by the `o` function) or not.

3.3.1 State Naming

Invalid (`I`), Shared (`S`), and Modified (`M`) are the three stable states of the `MSI` protocol, the other states are transient.

Reception of a request that requires use of the interconnect will usually lead to a $\texttt{XY}^{\texttt{BD}}$ transient state, which means that the cache controller is handling a transition between the stable states `X` and `Y`, with $^{\texttt{B}}$ indicating that this transition requires the acquisition of the interconnect and $^{\texttt{D}}$ the reception of a related data reply (regardless of whether it comes from an other cache controller or the memory).

$\texttt{XY}^{\texttt{D}}$ usually follows $\texttt{XY}^{\texttt{BD}}$ if the cache controller sees its own query before receiving a reply. If the reply is seen first, then the memory element goes into the $\texttt{XY}^{\texttt{B}}$ state instead. While the former is more likely, the latter may occur because, despite processing all queries in the same order, not all cache controllers take the same time to do so.

It is also possible an external query to lead to a change of state, especially when in the $\texttt{XY}^{\texttt{D}}$ state. Indeed, at that point, the system pretty much considers that the cache controller is in the `Y` state, and thus has the responsibilities that the `Y` state would imply. This makes it possible for a cache controller to see a query it needs to act upon before being actually ready to do so (e.g. observing a `GetM` query while waiting for `data`). The states thus reached have either a $\texttt{XY}^{\texttt{D}}\texttt{V}$ pattern, with `V` being a stable state, indicating that the external query would have required this memory element to be assigned the `V` state if it were currently in the `Y` state. Thus, this state is used to memorize that, after sending the data, the cache is expected to perform the appropriate change of state. It is possible for the `Y` state to also be susceptible to external queries. Thus, there are also $\texttt{XY}^{\texttt{D}}\texttt{YZ}$ states, indicating that the final state is now `Z`.

State	Core Request			Own Query	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!data $SM^{D}S$	r!data $SM^{D}I$	
$SM^{D}I$	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		

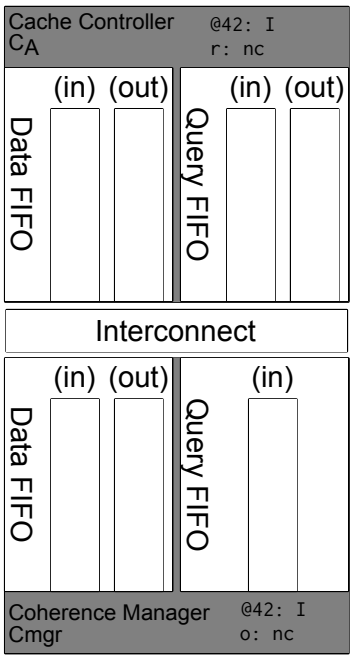
Figure 3.3 – Split-Transaction MSI Automaton for Cache Controllers

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

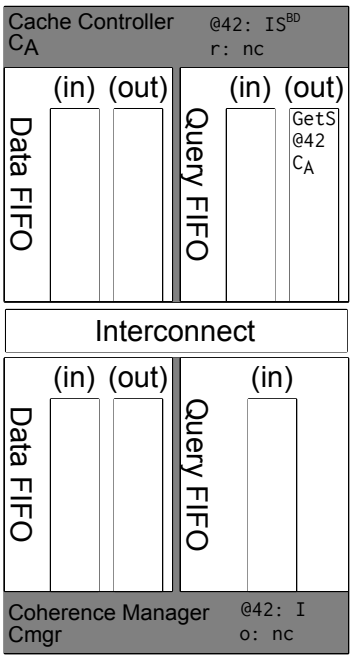
Figure 3.4 – Split-Transaction MSI Automaton for Coherence Manager

3.3.2 Examples

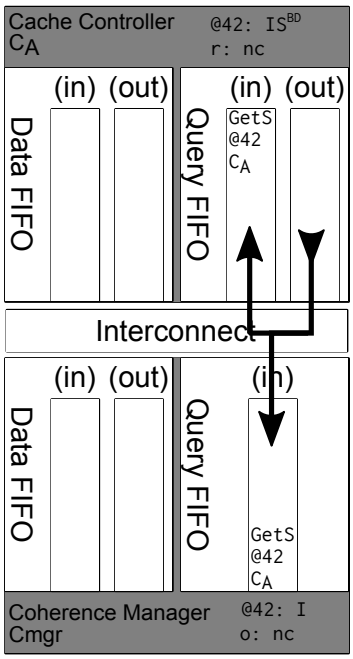
Example 11 (Simple Read) *In this example a single cache is moving from not having any access to a memory element to having a read-only copy. This is one the simplest example of transaction and is meant to show how messages are exchanged. This example is illustrated as a sequence in Figure 3.5.*



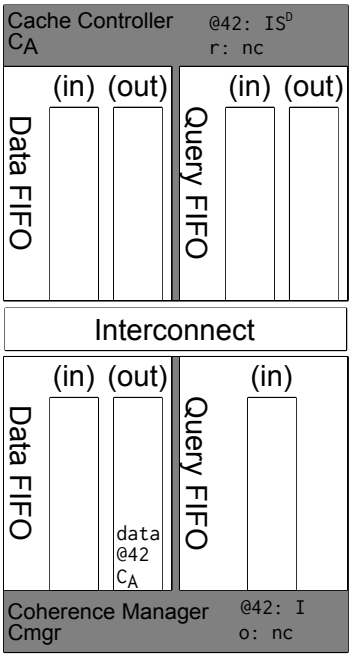
(a) C_A holds the memory element in the `I` state. As no cache holds any copy, its value must be provided by the main memory. Thus, C_{mgr} considers the memory element to be in the `I` state.



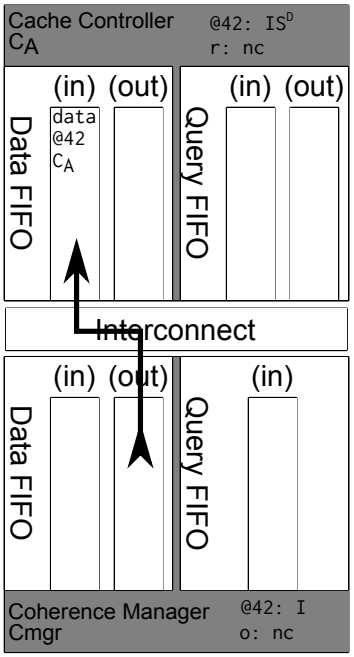
(b) C_A's core now issues a `load` instruction on 42. This leads C_A to move to the IS^{BD} state, and to prepare a `GetS` query in its outgoing query queue.



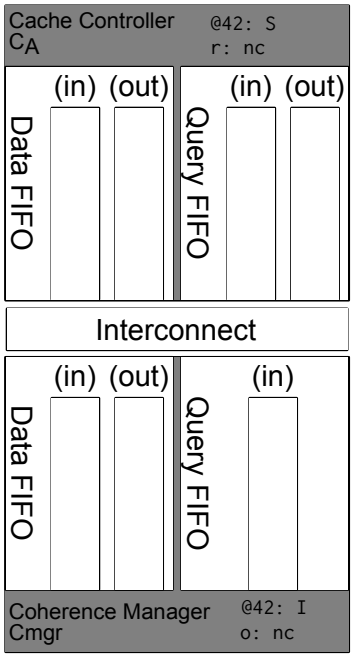
(c) The interconnect broadcasts outgoing queries from caches to all the incoming query queues: C_A's `GetS` is added to both its own and C_{mgr}'s incoming query queue.



(d) Consuming the message in its incoming query queue, C_A confirms it has processed all other prior queries and now associates the IS^D state to 42, which indicates that a data reply is now expected. Similarly, C_{mgr} consumes the GetS query, and, since the main memory is in charge of replying prepares a data message for C_A.



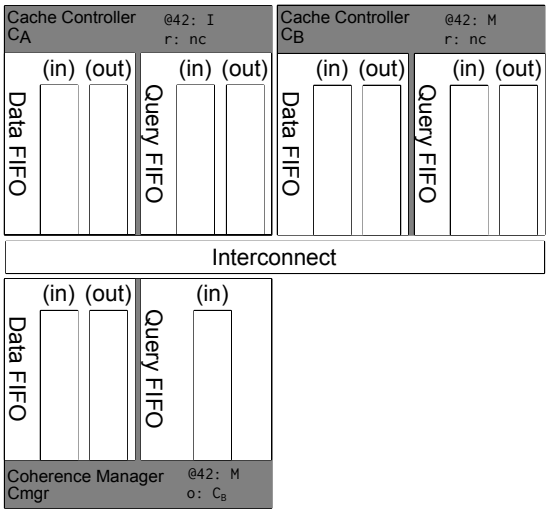
(e) The interconnect transfers the data message from C_{mgr} to C_A.



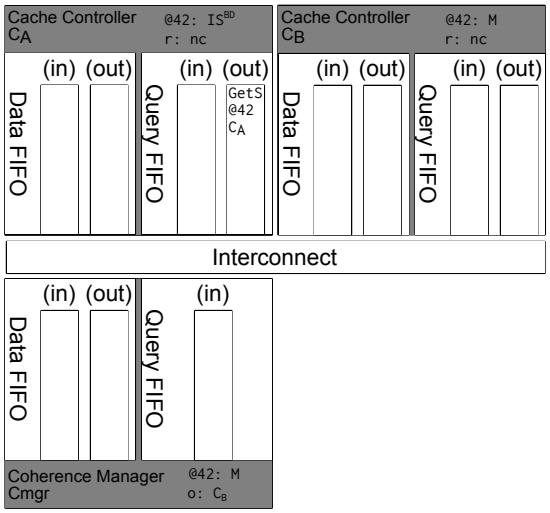
(f) By receiving the data message, C_A has obtained a read-only copy of 42, making it switch to the S state and completing its core's request.

Figure 3.5 – Illustrations for **Simple Read**

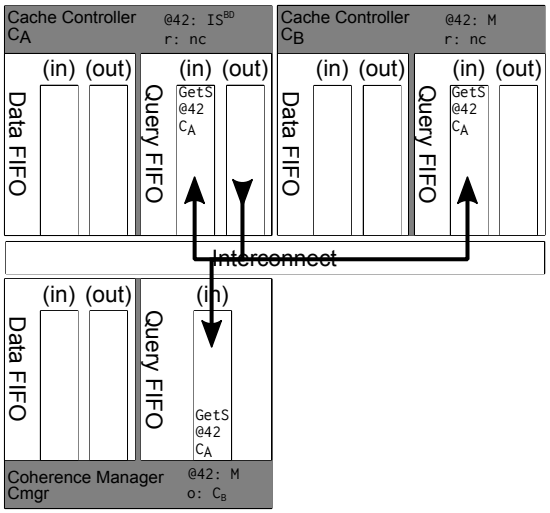
Example 12 (Reaching `S`) *This example is meant to showcase how exchanges between cache controllers are assumed to take place. To keep things simple, we only consider two cores and a single memory element (whose address is* 42*). This example is illustrated as a sequence in Figure 3.6.*



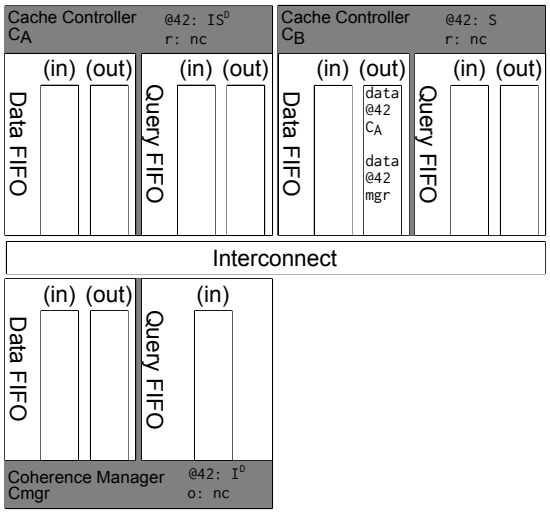
(a) C_A holds that memory element in the `I` state, while the other, C_B, holds it in the `M` state. As C_B may have modified the memory element, C_{mgr} considers the memory element to be in the `M` state. Furthermore, C_{mgr} has memorized that C_B is currently in charge of that particular memory element (`o:` C_B).



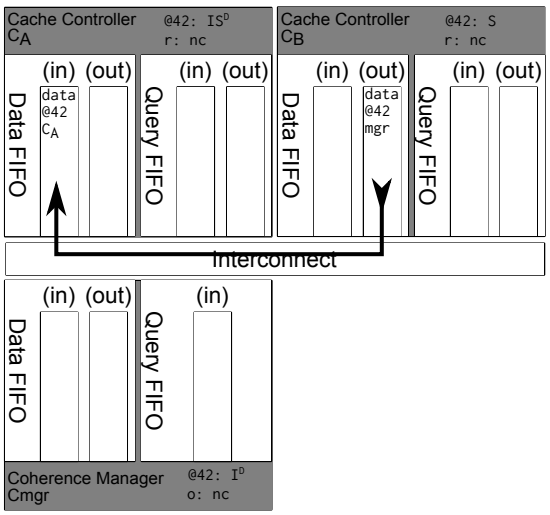
(b) Next, we consider that C_A's core issued a `load` instruction on 42. This leads C_A to move to the $\texttt{IS}^{\texttt{BD}}$ state, and to issue a `GetS` query to its outgoing query queue.



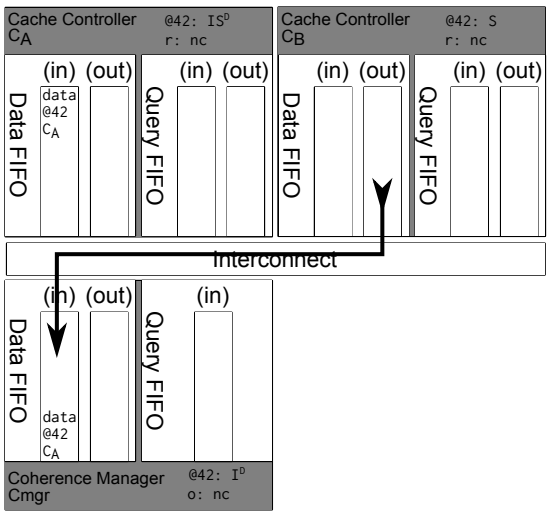
(c) The interconnect broadcasts outgoing queries from caches to all the incoming query queues. As C_A is the only one with an outgoing query, the `GetS` is added to both its own and C_B's incoming query queue, as well as the coherency manager's.



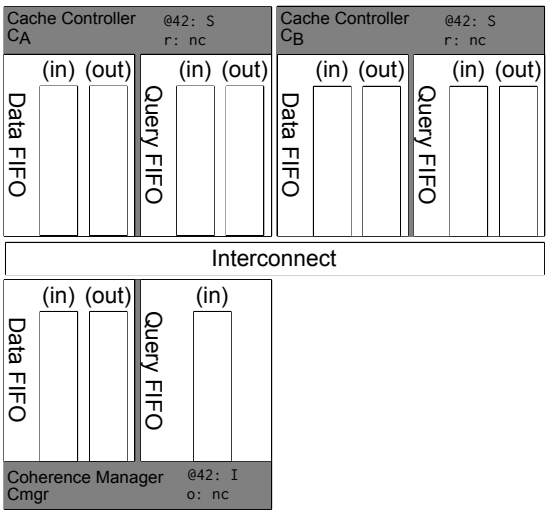
(d) Consuming the message in their incoming query queues, C_A, C_B and C_{mgr} change state, moving to `ISD`, `S`, and `ID` respectively. In addition, C_B adds a reply for the query to its outgoing data queue, as well as a `data` message to inform the coherence manager and the main memory of @42's new value. As it is about to receive the updated value, C_{mgr} no longer considers C_B to be responsible for that memory element.



(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 first reply is moved from C_B's outgoing reply queue to C_A's incoming one.



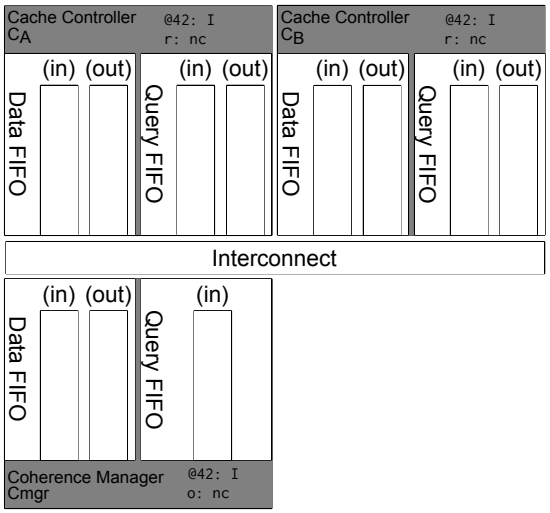
(f) The `data` message meant for the coherence manager follows, being added to C_{mgr}'s incoming data queue.



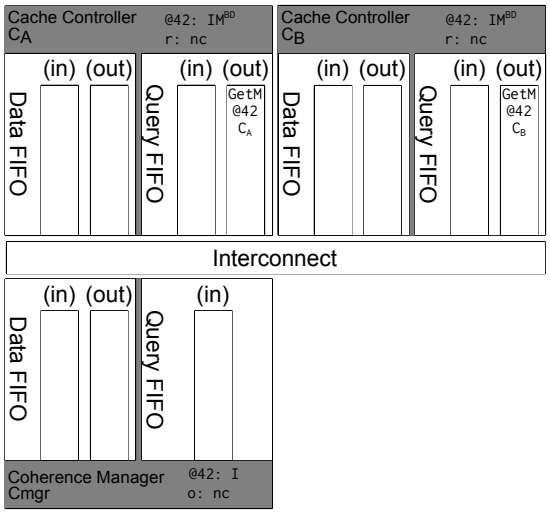
(g) Finally, C_A and C_{mgr} consume the message in their incoming data queue. Thus, C_A changes its state for to `S` and fulfilling its core's request, and C_{mgr} assigns the `I` state to @42, denoting that it is responsible for providing its value in future queries.

Figure 3.6 – Illustrations for **Reaching `S`**

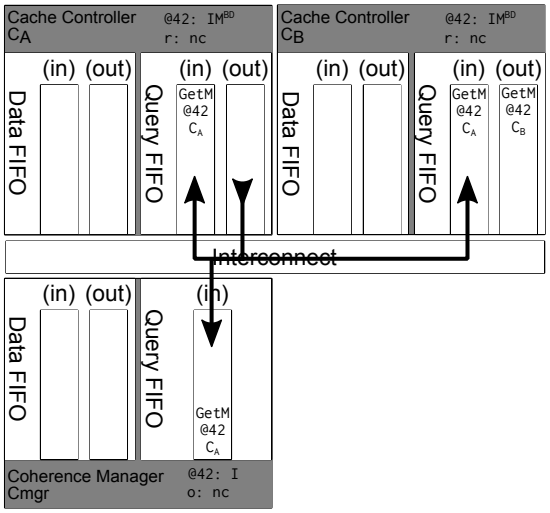
Example 13 (Parallel Stores) *Figure 3.7 illustrates what happens when a cache processes a query it must reply to yet cannot at the moment.*



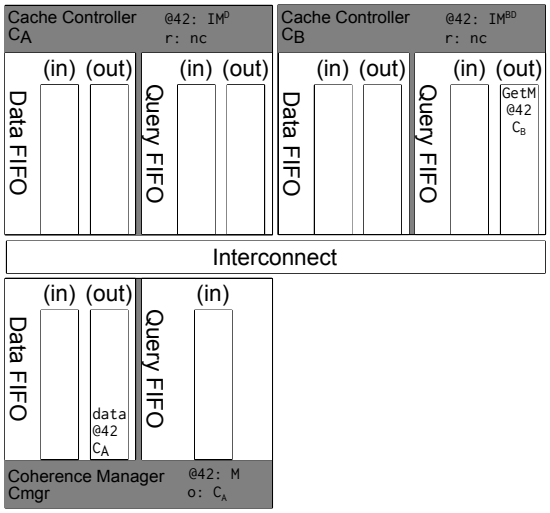
(a) In this initial phase, no cache holds a copy of the memory element (i.e. both are in the `I` state). The coherence manager reflects that state by also having the `I` state assigned to that memory element.



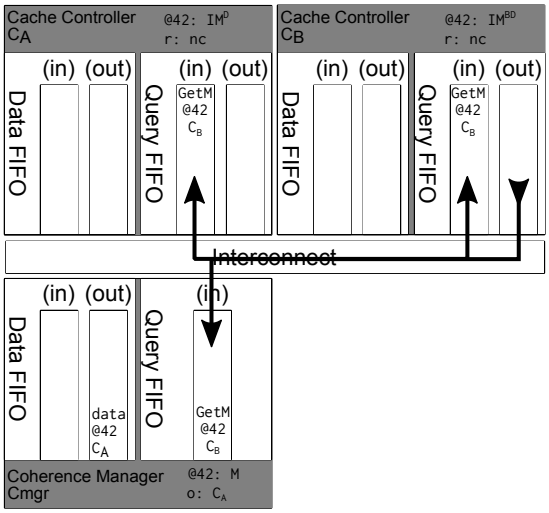
(b) Next, we consider that both cores issue simultaneous `store` instructions on their respective cache. This leads the caches to move to the $\texttt{IM}^{\texttt{BD}}$ state, and to prepare a `GetM` query in their outgoing query queue.



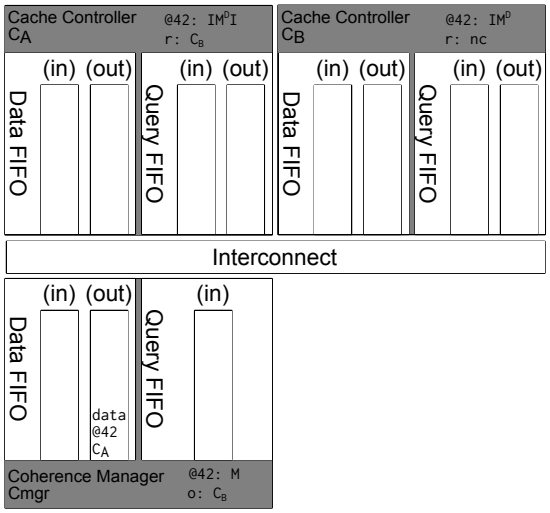
(c) The interconnect's access policy dictates which query is broadcasted first. In this example, C_A's `GetM` is broadcasted to all incoming query queues (includings C_A's).



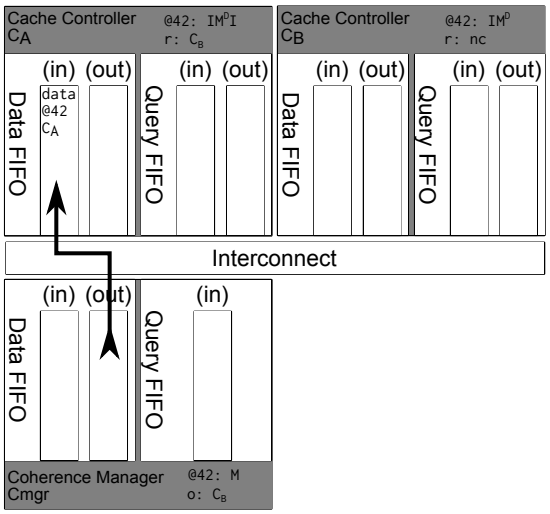
(d) C_A, C_B, and the coherence manager consume C_A's query next. C_A simply updates its associated state accordingly. C_B does not change anything as, in its current state, it behaves to external queries as if it was in the `I` state. The coherence manager reacts by preparing a `data` reply for C_A, memorizing that C_A is now responsible for the memory element's propagation, and updating its associated state.



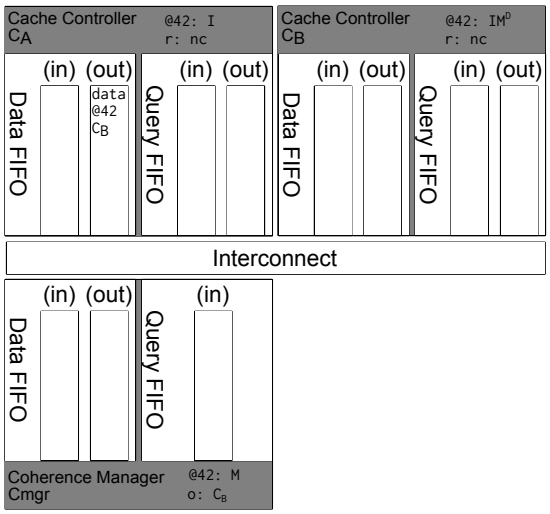
(e) The coherence manager's data may take a while to be read off the main memory, so we'll see what happens if C_B's `GetM` is broadcasted next.



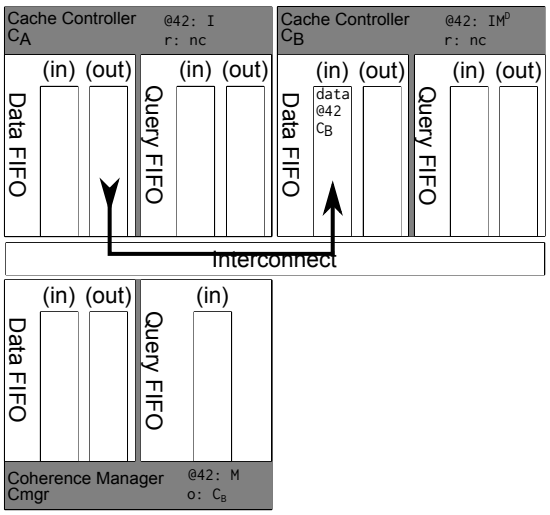
(f) C_B reacts to consuming its own query exactly as C_A did in 3.7d. The coherence manager simply updates which cache is now assumed to be able to perform modifications on the memory element. It does not prepare data for C_B. C_A must act as if it already had modified the memory element and is responsible for providing C_B with data. As it is currently unable to do so, it memorizes C_B needs the data and sets its associated state to reflect the effects of receiving a `GetM` when in `M`.



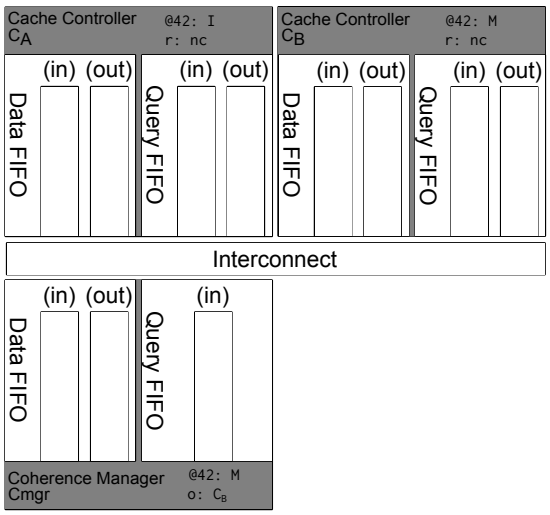
(g) The data message meant for C_A ends up in C_A's incoming data queue.



(h) Upon receiving data, C_A completes its core's request, prepares a data message with the new value meant for C_B, and moves to the `I` state, as it cannot hold a copy of the memory element while C_B may modify it.



(i) The data message moves from one core's outgoing data queue to the other's incoming one.



(j) Upon reception of data, C_B simply completes its core's request.

Figure 3.7 – Illustrations for **Parallel Stores**

3.4 Variants

This section presents the most common variants of the MSI. In effect, the main factor of a cache coherence protocol's efficiency is how frequently it needs to fetch or write data to the main memory. As a result, it is unsurprising to see that all of these optimizations are focused on removing that necessity in certain common scenarios. Each of them adds a new stable state, corresponding to a new permission or new responsibility, and each one can be combined with the others to form a new variant (forming for example, the MESI, MOESI, and MESIF protocols).

Exclusive State The Exclusive state (`E`) is equivalent to the `S` state, with the added information that no other cache currently holds any copy of the memory element. As a result, the cache modifying the value of its copy of the memory element without broadcasting for new permission does not violate Properties 1 and 2. It does, however, mean that the coherence manager must be able to detect that no caches hold any copy of the memory element, which implies some change in how caches evict from the `S` state, as they otherwise need not do inform any other component.

Owned State The Owned state (`O`) allows a cache to keep its modified copy of a memory element without having to perform a write-back when another cache asks for a read-only copy of that memory element.

Forward State The Forward state (`F`) is equivalent to an `S` state where the cache is also in charge of providing a copy of the memory element to any cache that demand it.

3.5 Cache Line Organization

Caches hold a finite amount of cache lines. Thus, at some point, the need for a new cache line when none are available may arise. Deciding which cache line to evict in order to store the new one has a very important impact on performance. This is typically decided by two factors: its placement policy and its replacement policy. These two policies interact: the placement policy determines which group of cache lines are considered by the replacement policy.

3.5.1 Replacement Policies

The cache eviction policy is the algorithm used to determine which line to evict when there is no more room. The optimal strategy would ensure that the line that is evicted is the one that would not be used for the longest time. However, being able to determine which line this is would require explicit management of the caches.

Since it is commonly assumed that data recently accessed is likely to be accessed again soon (a principle known as *locality of reference*), the most common replacement strategy is to choose the least recently used cache line as the one to evict. This is referred to as the *LRU* policy. However, keeping accurate track of which line was accessed last is costly in both space and time, and so, in practice, a less precise approach, the Pseudo-LRU (or *PLRU*) is implemented instead. The principle behind *PLRU* is similar to having a binary tree, with the leaves corresponding to cache lines. Each node of the tree has a single bit. This bit indicates which of the two children to follow to find a cache line that was not recently used. When using any of the cache lines, all nodes in the path leading to it are set so that they point to a node not also in the path.

Other replacement policies include *FIFO*, in which the cache line that was the least recently *allocated* is evicted, and cache line uses have no impact; using frequency of use instead of date, in something similar to the LRU policy (called *LFU*, for Least-Frequently Used); and random replacement (*RR*).

These policies are applied to a set of cache lines. Exactly what lines are part of this set is determined by the placement policy.

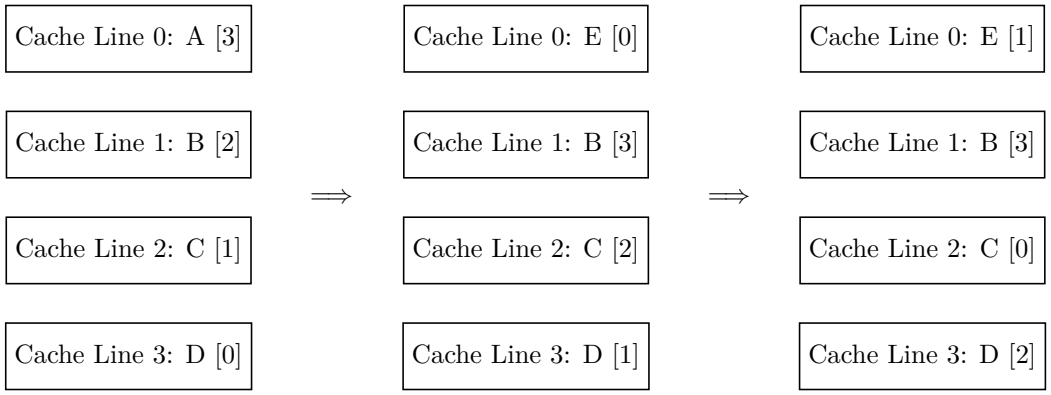


Figure 3.8 – Example of LRU Replacement Policy

Figure 3.8 is an example of LRU replacement policy in action: in a cache of four lines, the memory elements A, B, C, and D were initially loaded, in that order. In the figure, the rightmost number in each cache line indicates its index for the LRU replacement policy. The figure shows what happens when a memory element E is loaded, then the memory element C is accessed. Because A was the least-recently used element, it is the one that was evicted upon insertion of E. As it was just added, the element E is considered as having been accessed, and all other indices are updated accordingly. The figure then shows what happens when C is loaded: all indices strictly below C's are increased by one, then the index of C is set to 0, marking it as the most recently used cache line.

3.5.2 Placement Policies

Placement policies determine where a memory element can be stored within the cache, according to the memory element's address. There are three common strategies: allow it to go anywhere; allow it to go only at one location; and allow it to go anywhere within a group of cache lines;

A cache in which memory elements can be placed anywhere is called a *fully associative* cache. This makes the replacement policy able to perform at its best, since it applies to the whole cache at once. However, locating a cache line is costly, since it can effectively be anywhere in the cache. An example of fully associative cache organization is shown in Figure 3.9.

A cache in which each address maps to a single cache line is called a *direct-mapped* cache. This effectively negates the need for a replacement policy. Access to a cache line is very fast. Depending on the mapping algorithm and the accesses made, this can end up being very efficient. However, even a very frequently accessed memory element is at risk of being evicted if another memory element that maps to same cache line is also accessed. Furthermore, this may lead to cache lines being evicted even when other cache lines remain unused. An example of directly mapped cache organization is shown in Figure 3.10.

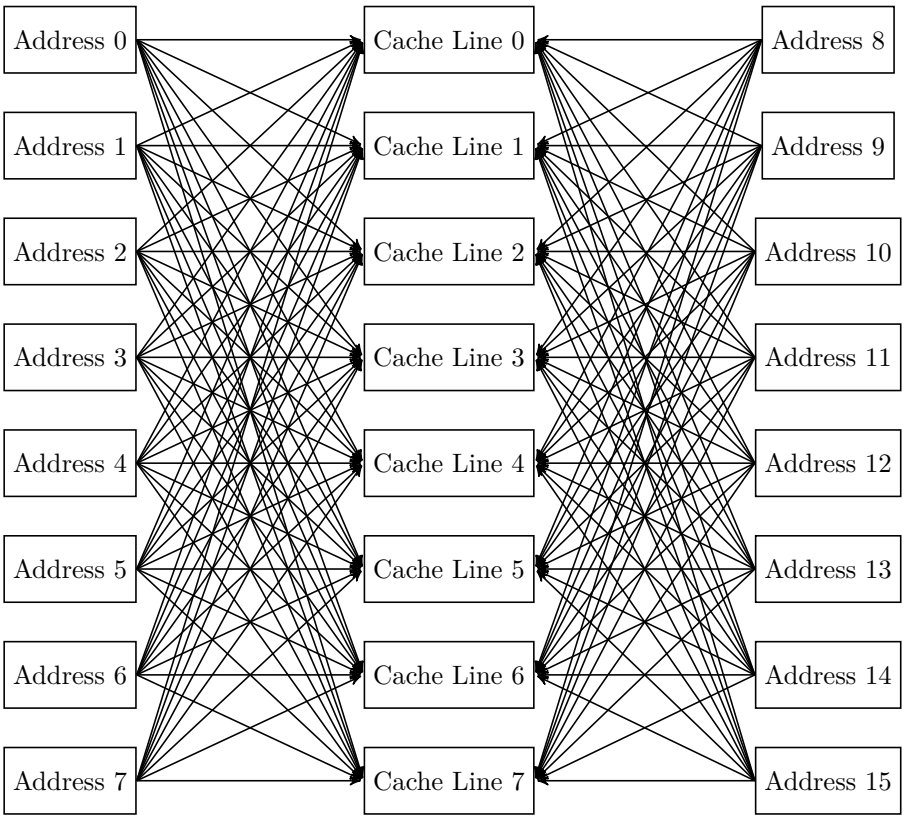


Figure 3.9 – Example of Fully Associative Cache Organization

A cache in which a mix of both strategies is used is called a *set-associative* cache. This is equivalent to having the cache be split into equally sized groups of cache lines, and applying the direct mapping to groups instead of single cache lines: each address is mapped to a group, but the memory element can end up anywhere within that group. The replacement policy is then only applied within the relevant group, which is much cheaper than a cache-wide implementation. Additionally, more frequently used memory elements sharing a set with other memory elements are less likely to get evicted. The issue of cache lines being evicted despite having potentially unused cache lines is still there, but only if these unused cache lines are not part of the group to which the newly allocated cache line belongs. An example of set associative cache organization where each set has two lines is shown in Figure 3.11.

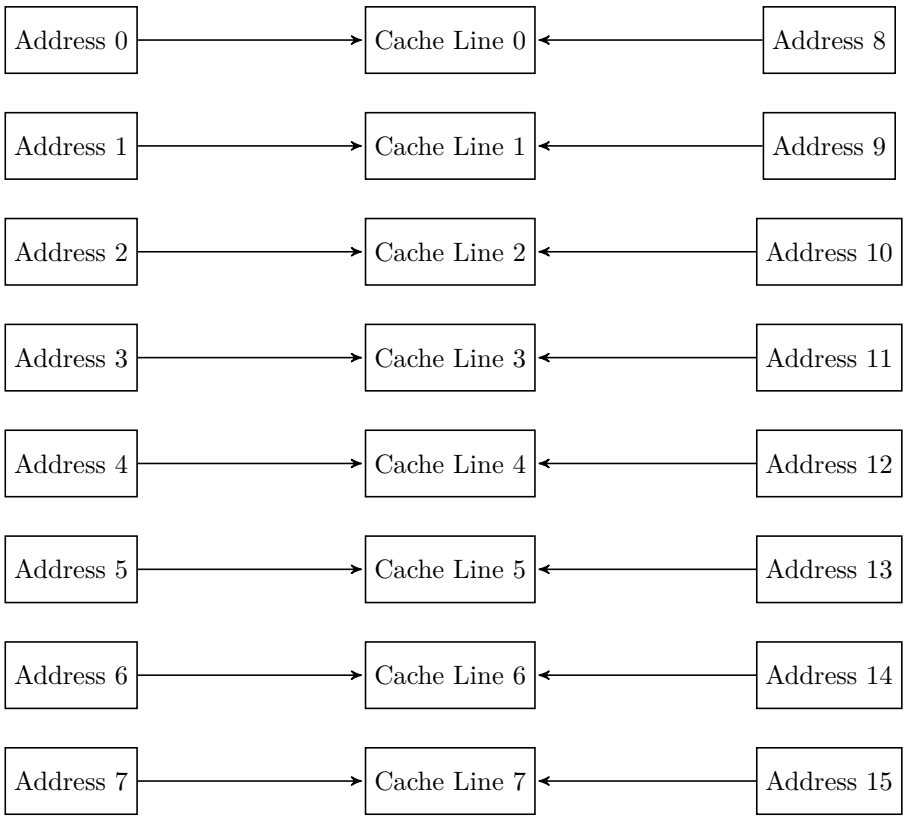


Figure 3.10 – Example of Directly Mapped Cache Organization

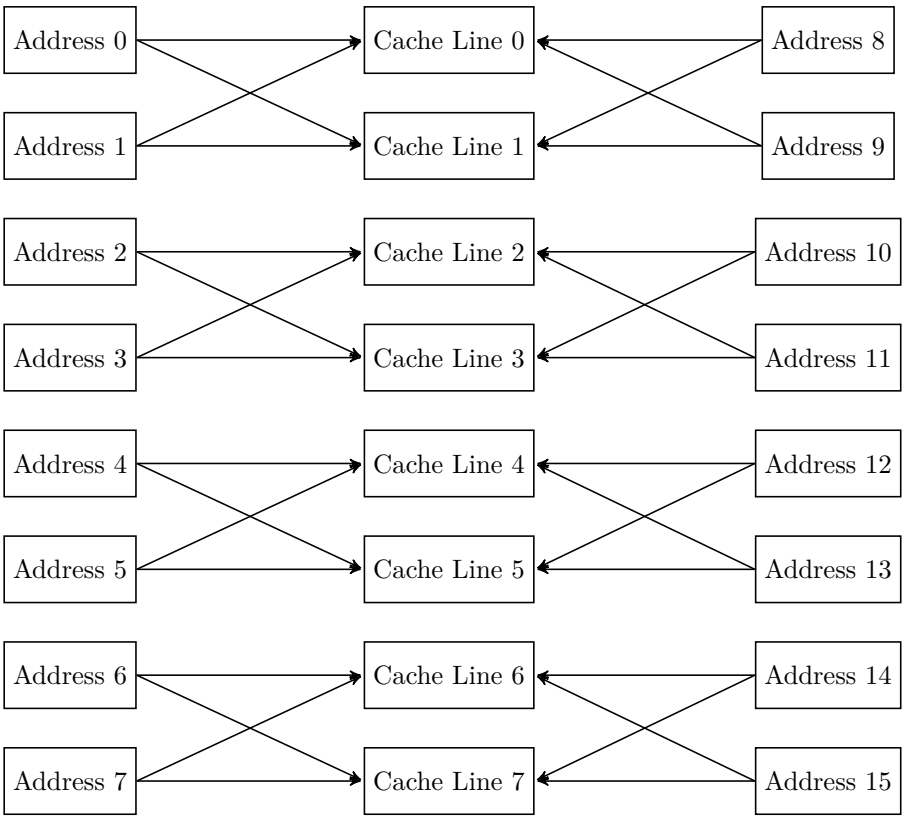


Figure 3.11 – Example of Set Associative Cache Organization

3.6 Conclusion

This chapter ensured readers have sufficient knowledge of cache coherence mechanisms to understand the concepts shown in this thesis. Indeed, it provided a formalization and examples in order to allow the readers to familiarize themselves with the MSI protocol.

It should be noted that the MSI protocol presented here is a very basic cache coherence protocol, even in its split-transaction bus adaptation. Despite its primitive nature, the complex behaviors it allows to result from seemingly simple sequences of instructions make the analysis of its impact on software execution difficult to estimate. Especially when considering that, outside of the content of these sequences of instructions, all the behaviors are done automatically and outside of the user's control.

While its predictability is an issue, the benefits of using cache coherence on an architecture to exchange data between concurrent software are too important to simply go without. The next part presents existing solutions proposed to tackle this problem.

Chapter 4

Objective

The general objective of this thesis is to provide tools that will expose cache coherence interference to an applicant seeking the certification of system with multi-core processor, thereby contributing to the resolution of the *Resource Usage 3* requirement. Indeed, that requirement asks that the applicant identifies all interference and its impact on the applications.

As the necessary notions have now been introduced, the problem statement from Section 1.2 and the proposed solution can be presented in more details. Thus, this chapter clarifies the scope of this thesis and provides an explanation of how its contributions form a framework that exposes cache coherence interference.

4.1 Tasks Required of the Applicant

To fulfill the *RU3* objective, the applicant has to catalog all interference channels, as well as all interference, and to mitigate them so that they do not prevent the applications from otherwise fulfilling other requirements. To determine where the interference is and how much of an impact they could potentially have, an understanding of the architecture's mechanisms is required. When the mechanisms that generate them are not removed all together, the applicant must be able to quantify their impact on the running software.

4.1.1 Coherence Protocol Identification

Ideally, the solution would simply be to consult the architecture's documentation, where the mechanisms would be described in details and, hopefully, so would all interference inducing behaviors. However, actual documentation for COTS processors do not generally include details on cache coherence. This is the first issue addressed in this thesis: how can one be sure of their understanding of the architecture's cache coherence mechanisms? On this topic, the contribution this thesis makes is to point out a missing preliminary step, and propose a strategy for it. Indeed, instead of focusing on the already resolved issue of measuring the time required to perform an access, or the bandwidth such accesses would have, the issue of incorrect assumptions on the architecture's cache coherence mechanisms is addressed. In effect, because the documentation of the implemented cache coherence mechanisms is not precise, there is a strong risk of mismatch between what the architecture actually performs and what is being tested, for why would one test the performance of something they understand not to be present?

For example, if a *MESI* architecture was benchmarked as if it were a *MSI* one, the interference generated by having two caches each perform an initial read on a memory would, for one of the two caches, not be the same as with a true *MSI* architecture. Not realizing this would lead the applicant to wrongly characterize the architecture. They could incorrectly underestimate the cost of simultaneous reading, if the measures happened to be done on the second cache to complete reading, as the data would then be provided by the first cache. Since this is happens only in special cases with the *MESI* protocol, to consider this measure representative of the general case of parallel reading would lead to a completely incorrect analysis of the effects of interference. Since these special cases are not found within the *MSI* cache protocol, the applicant would not know to be wary of them.

Ensuring that the architecture's cache coherence mechanisms have properly been identified and their characteristics measured is a necessary step for the analysis of the architecture. It is however not sufficient to infer what interference the architecture's mechanisms will generate, nor how it will affect the execution of the applications.

4.1.2 Measuring the Impact of Interference of the Software

Definition 31 (Impact of Interference) *The impact of an interference is a quantification of the temporal effects of interference on an application's execution. More precisely, it is the amount of additional CPU cycles required to execute a given fragment of the application compared its execution in isolation.*

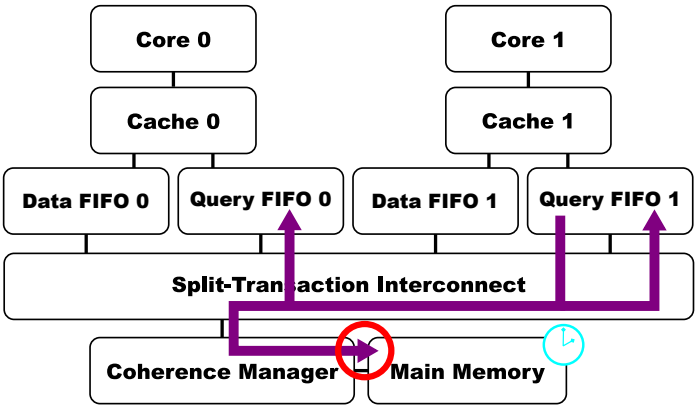


Figure 4.1 – Second Example of Interference

Example 14 (Impact of Interference) *The interference from Example 1, by itself, may not be noticeable by Core 1 if it lasts less than a cycle. However, an impact could be measured in a follow-up interference: once Cache 0's query reaches the main memory, it stays busy for an appreciable amount of time before becoming available once again (see Figure 4.1). Here also, Cache 1's query ends up delayed. In effect, if the main memory stays busy for 12 CPU cycles before Cache 1's query is considered, then the impact of that interference is the 12 CPU cycles delay.*

Even when the architecture's mechanisms have been fully identified, predicting the impact of interference on the execution of programs is not an issue currently resolved by the existing literature. Restricting this issue to caches in which coherence has been disabled still does not lead to a perfect

solution. Indeed, this was already problematic in single core systems in which multiple applications share a same cache. The issue being that some aspects of the architecture will still remain slightly ambiguous, and so the exact order in which events occur cannot be determined. Depending on the placement and replacement policies of the cache, this makes it difficult to know the content of the cache at any given time. Cache coherence adds to the number of events that can occur, and make the content of other caches decide whether events occur, thus tremendously complexifying the issue. Ignoring the effects of such mechanisms would quickly lead to either incorrect or unusable execution time estimations, as any memory access would have a seemingly random temporal cost as the then invisible coherence states would influence it. The existing literature offers compromises: either analyzing strategies that will yield overly pessimistic WCET, or restrictions to eliminate sources of interference so that their effects need not be taken into account. The available analysis strategies do not address cache coherence. Some of the approaches using restrictions do take cache coherence into account, but only through hardware modifications, which is not deemed acceptable in an aeronautical context. This thesis proposes an approach to analyze the effects of cache coherence. The idea is to have an accurate representation of all mechanisms that are known and use formal methods to ensure all possible behaviors of the mechanisms for which ambiguities remain are explored.

4.2 Proposed Solution

In effect, this thesis proposes to tackle the issue of interference generated by cache coherence by first ensuring complete understanding of the architecture's cache coherence mechanisms through the proposed identification strategy, then using existing approaches to measure the timing characteristics of these mechanisms, to finally use formal methods on a model based on what is presented in this thesis in order to infer the effects of the interference on the running software.

4.2.1 Hypotheses and Limitations

This section presents the hypotheses made on the context in order to limit the scope of this thesis, as well as the reason behind these limitations and, when applicable, how these could be removed in future works.

The first scope restrictions stem from the aeronautical context. Indeed:

- Only COTS architectures are considered. The use of self-made hardware is not deemed acceptable in critical systems such as avionics.
- Uses cache coherence. Using multi-core processors without cache coherence prevents any gain they would otherwise provide over single-core processors when running application with parallel processing.
- Snooping-based cache coherence. This is the more common approach to cache coherence in COTS, and thus makes for a good starting point. Extending this thesis to also handle directory-based cache coherence would require the creation of a different architecture model.

Other restrictions are more related to what the thesis' time constraints permitted. The focus stayed on cache coherence itself, since this is what the existing literature consistently avoids. Thus, elements that are already explored in other numerous works have not been expended upon past the bare minimum. An example of architecture complying with these restrictions can be seen in Figure 4.2.

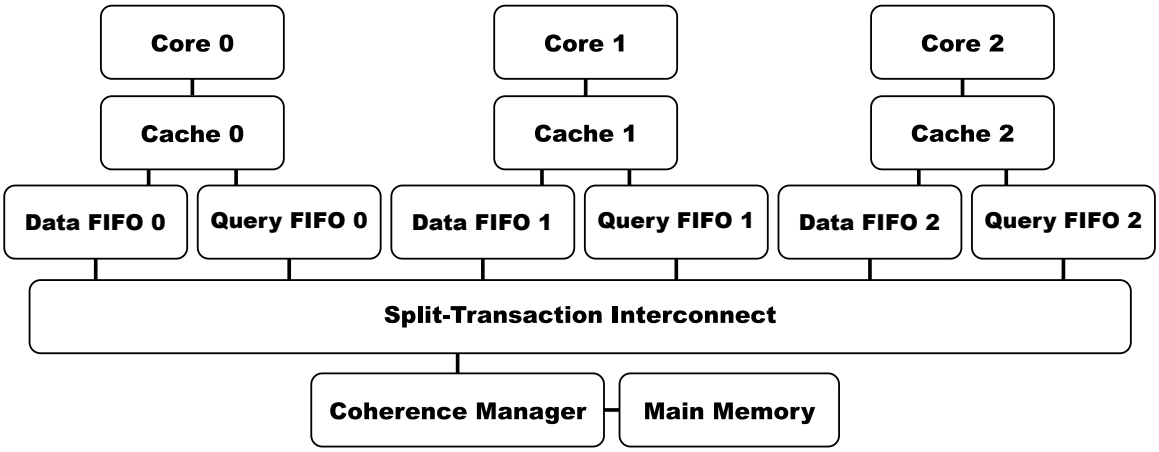


Figure 4.2 – Typical profile of the targeted architecture

- No cache hierarchy. The coherence is studied within a single level of cache, there is no issue of memory element propagation across L1, L2, or L3 caches. Adding proper support for cache hierarchy to the tools presented in this thesis would require a large number of modifications.
- One core per cache. The solution proposed in this thesis does not properly handle multiple cores per cache, as the model fails to account for the originator of each request in at least one of its functionalities. Unless as of yet unknown issues arise, this limitation should be fairly easily removed.
- Fully associative cache placement policy. Supporting the segregation of cache lines would add a layer of complexity without being a fundamental change in what the tools prove to be possible. Allowing configuration by the user so that other placement policies may be used should be straight forward, and is only absent because of its low priority.
- LRU cache replacement policy. While a replacement policy had to be chosen, the most popular one, PLRU, was not the one implemented. LRU is easier to debug, which is why it was chosen. Here also, adding support for other policies would not constitute a major change in the model, but neither was it considered a high priority feature.
- Simplified program representation. The focus is really put on cache coherence. No matter how complex, programs only interact with cache coherence through memory accesses. Thus, only sequences of memory accesses are represented and, by default, instruction jumping and branching is not supported. Technically, this can already be bypassed by creating a new automaton for each of the more complex program, however, automation of the automaton creation is required before such a solution can be considered reasonable.

Lastly, the tools proposed in this thesis only constitute part of the solution needed to properly analyse multi-core processors for use in avionics:

- Results focused on cache coherence only. For accurate measurement of the effects of interference on applications running in a multi-core, the separate analysis of sources of interference is inadequate as not only are these sources not independent, but the effects of each source

interference can, by definition, alter the behavior of the application and thus lead to the other sources of interference having a different effect on the application. Assuming a worst case for each source of interference does not ensure that the global worst case is measured (a principle known as *timing anomaly*). Thus, the contributions made in this thesis would need to be integrated into an all-encompassing framework in order to allow adequate estimation of the WCET.

4.2.2 Framework Overview

Figure 4.3 presents the general framework proposed in this thesis for the analysis of the impact of cache coherence on software running on a multi-core COTS system.

Given an architecture, the applicant performs an identification of the cache coherence mechanisms. This removes any ambiguities that may be lingering in the description of the cache coherence protocol from the architecture's documentation. This process is described in Chapter 8 and involves the use of performance monitors in order to observe the behavior of the architecture's cache coherence mechanisms and compare it with that of a hypothetical cache coherence protocol. If the two match, the applicant thus obtains an ambiguity-free cache coherence protocol description for the architecture.

Using this ambiguity-free cache coherence protocol, the temporal cost of each operation is quantified through benchmarks. Indeed, now that all relevant coherence states and behaviors are known, the applicant can be sure their measures are not missing important results. This step is not expanded upon by the thesis, as the existing literature already covers it (see Chapter 5). With these measures, a profile of the architecture's cache coherence mechanisms has been obtained.

To evaluate the impact of cache coherence on the execution of software, this thesis proposes the use of formal methods. This requires the creation of a model for the architecture (see Chapter 9), as well as the definition of the appropriate queries in order to retrieve information that can be used by the applicant through model checking (see Chapter 10).

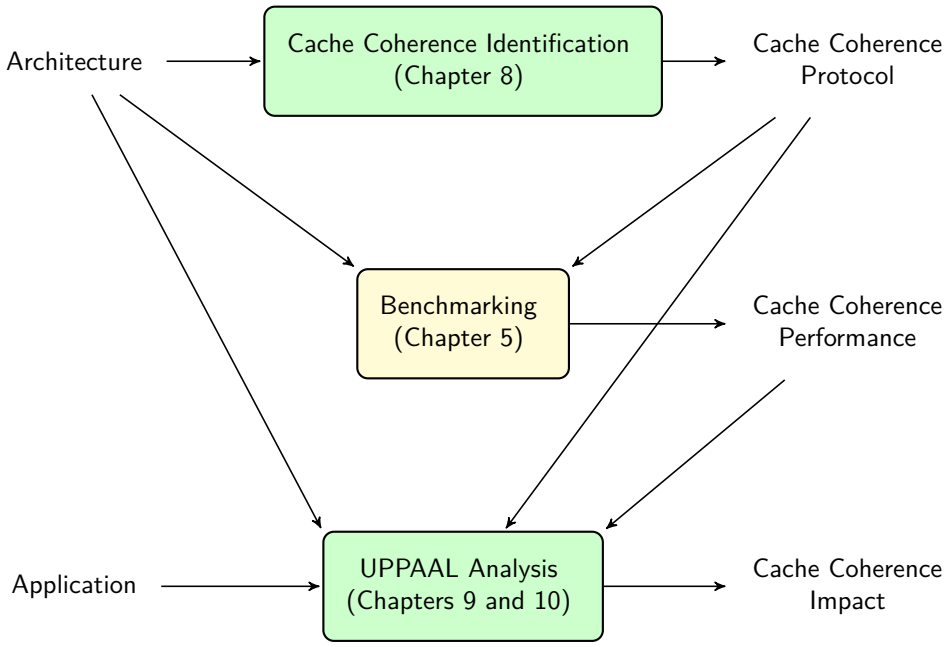


Figure 4.3 – Approach Overview

4.3 Conclusion

This chapter concludes the context part of the thesis by clearly defining the objective of this thesis and curtailing its target application.

The next part of the thesis presents a selection of the existing literature relevant to this thesis. It starts with a chapter focused on benchmarking (Chapter 5), providing a description of approaches that can be used to measure the performance of the architecture's cache coherence mechanisms once the protocol has been identified.

Literature around the broader issue of using a multi-core processor's caches in a critical real-time environment is then presented (Chapter 6). None of the existing solutions allow the use of cache coherence within an aeronautical context, but there are ways to still benefit from caches by adhering to restrictions on their use.

Lastly, the related works part of this thesis features a chapter on approaches similar to that presented in this thesis for the use of formal methods to analyze an architecture (Chapter 7). It shows that the use of timed automata eases the creation of both readable and modular models.

Part II

Related Works

Chapter 5

Micro-Stressing Benchmarks

This chapter presents strategies relying on benchmarks to figure out the characteristics of an architecture, be it its speed, the capacity of its components, or other information not sufficiently detailed in the architecture's documentation. Having properly characterized the architecture is a necessary preliminary step to the analysis of interference. Indeed, to understand the effects the interference has on running software, a quantification of the architecture's components capabilities is necessary, as determining which parallel accesses would interfere with one another is otherwise impossible.

In the contributions made by this thesis, the objective of the benchmarks is not only to measure the maximum performance of the cache coherence mechanisms, but also ascertain that they are fully understood by the user. This is a vital part of the characterization process, and solutions for mechanisms much simpler than cache coherence have already been explored. For example, the first section of this chapter showcases two papers proposing solutions to detect hidden correlations between components.

Once the mechanisms have been understood, then the performance measurements can proceed, as the user is now fully aware of what is being measured. The second section of this chapter is thus about papers on the use of benchmarks for the performance analysis of cache coherence.

Lastly, this chapter presents a paper on the use of benchmarks to remedy a lack of performance monitors. Such an approach could be used to implement the solution proposed in Chapter 8 if the user cannot find the appropriate performance monitors.

Because the terms are recurrently used thorough this chapter, definitions for *execution time*, *overhead*, and *bandwidth* are provided below.

Definition 32 (Configuration) *A configuration is defined as the combination of the mapping of programs on each core, the hardware's settings, and the initial state of the architecture prior to program execution.*

Example 15 (Configuration) *Running a sequence of instructions in isolation, and running that same sequence of instructions while other programs are running on other cores correspond to two separate configurations.*

Definition 33 (Execution Time) *The execution time of a list of instructions on a certain configuration is the time between the emission of the first of the instructions and the last instruction of the list being seen as completed by the emitter.*

Example 16 (Execution Time) *In the case of two successive loads, the execution time corresponds to the time elapsed between core emitting the first load instruction to its cache, and the cache providing the data for the second load instruction.*

Definition 34 (Overhead) *Given T_A and T_B two execution times of a same list of instructions from two separate arbitrary configurations A and B, such that $T_B \geq T_A$, The overhead O of being in configuration B over configuration A is defined as:*

$$O = T_B - T_A$$

Example 17 (Overhead) *If the two configurations of Example 15 yielded an execution time of 5 CPU cycles and 13 CPU cycles respectively, the overhead incurred because of the other programs running in parallel would be $13 - 5 = 8$ CPU cycles.*

Definition 35 (Bandwidth) *Bandwidth is the amount of data (e.g. bits, bytes, words) that can be transferred from one component to another within a given time frame.*

Example 18 (Bandwidth) *A CPU attempting to write a sequence of data with a size of 512MB in the memory would have to pass the information through all the mechanisms between itself and the memory. The amount of data that went through all the mechanisms within either a CPU cycle or a microsecond is considered to be the bandwidth.*

5.1 Detecting Component Correlation

5.1.1 Evaluating Interference Through Resource-Stressing

[44] presents a strategy to characterize the sensibility of a shared resource to temporal interference. The general idea behind the approach is to perform a stressing benchmark on an architecture's resource with only that single thread running (i.e. running in isolation), and compare the result with the same test in which multiple threads of the stressing benchmark run in parallel, in a metric called *slowdown factor* (see Definition 36).

Definition 36 (Slowdown Factor) *Given a benchmark targeting a specific component, its application using a single thread in isolation resulting in a execution time n, and its application while other threads are stressing components (potentially the same one) yielding a execution time of m, the slowdown factor f is defined as:*

$$f = \frac{n}{m}$$

This slowdown factor indicates how sensible the component targeted by the measured benchmark is to interference, and thus how large the execution time variations caused by parallel access to that component will be. By having the other threads target different components, correlation between them can be exposed: if the slowdown factor is higher than 1.0, some interactions occur between these components, and the other components are able to generate interference on any application using the component targeted by the measured thread.

The benchmarks of [44] are implemented following the pattern shown in Figure 5.1: a simple loop applying the appropriate instruction numerous times. For memory access benchmarking, a slightly different strategy is used (see Figure 5.2): each loaded memory element contains the address of the next memory element to load, a principle known as pointer chasing.

Line	Source code	Explanation
001	movl %1, %ecx	initialize loop counter *ecx* (%1 is an input parameter)
002	label_intAdd:	beginning of the loop
003	add %eax, %ebx	target instruction
004	add %ebx, %eax	target instruction
...	...	...
...	...	...
252	add %ebx, %eax	target instruction
253	decl %ecx	decrement loop counter
254	cmp %ecx, $0	compare loop counter with 0
255	jne label_intAdd	if (counter != 0) jump to the beginning of the loop

Figure 5.1 – *intAdd* Benchmarking Code (taken from [44])

```
1  for(cnt=0; cnt < array_size; cnt+=stride) {
2     if(cnt<array_size-stride) {
3        // Each array element contains the address of ...
4        // ... the following array element we want to access.
5        array[cnt] = (int)&array[cnt+stride];
6     }
7     else {
8        // The last accessed element in the array points ...
9        // ... to the first element of the array that we access.
10       array[cnt] = (int)array;
11    }
12 }
```

Figure 5.2 – Memory Benchmark Initialization Code (taken from [44])

		Pipeline front end	Pipeline back end (Execution units)						Cache memory			Off-chip resources
		nop	intAdd	intMul	intDiv	fpAdd	fpMul	fpDiv	L1 data	L1 icache	L2	mem_bw
Pipeline front end	nop	2.0	2.0	2.0	1.8	1.6	1.7	1.6	1.7	1.8	1.7	1.6
Pipeline back end (Execution units)	intAdd	2.0	2.0	1.2	1.2	1.2	1.0	1.0	1.0	1.1	1.0	1.0
	intMul	1.3	1.3	1.3	1.1	1.1	1.1	1.0	1.2	1.2	1.1	1.0
	intDiv	1.2	1.2	1.2	1.2	1.1	1.1	1.8	1.2	1.1	1.2	1.1
	fpAdd	1.1	1.2	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
	fpMul	1.0	1.0	1.0	1.1	1.0	1.0	1.0	1.1	1.0	1.1	1.0
	fpDiv	1.0	1.0	1.0	1.5	1.0	1.0	2.0	1.0	1.0	1.0	1.0
Cache memory	L1 dcache	1.0	1.0	1.2	1.0	1.0	1.1	1.0	6.2	1.0	3.6	3.6
	L1 icache	1.1	1.1	1.0	1.0	1.0	1.0	1.0	1.0	2.7	1.0	1.0
	L2	1.0	1.0	1.0	1.0	1.0	1.0	1.0	2.3	1.0	14.1	15.3
Off-chip resources	mem_bw	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	2.5	2.5

Figure 5.3 – Slowdown factor on the Intel Atom Z530 (taken from [44])

Figure 5.3 shows the approach of [44] working at its best. This is the result on an Intel Atom Z530, which features a single core capable of running two parallel threads. The lines correspond to the benchmark being measured, the columns correspond to the target of the benchmark being used as a source of interference.

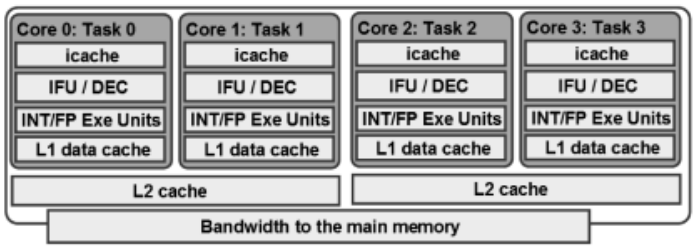


Figure 5.4 – The Intel Core2Quad architecture (taken from [44])

		Pipeline front end	Pipeline back end (Execution units)						Cache memory			Off-chip resources
		nop	intAdd	intMul	intDiv	fpAdd	fpMul	fpDiv	L1 data	L1 icache	L2	mem_bw
Pipeline front end	nop	1.0	1.0						1.0			1.0
Pipeline back end (Execution units)												
Cache memory	L1 dcache											
	L1 icache											
	L2										14.4	14.4
Off-chip resources	mem_bw										1.1	1.1

Figure 5.5 – Slowdown factor on the Intel Core2Quad within the same cluster (taken from [44])

The application on a multi-core processor, however, is not as informative: Figure 5.5 shows the same experiment performed within a cluster of an Intel Core2Quad (two separate cores that share the same L2 cache, see Figure 5.4). The results indicate no unexpected slowdowns from the simultaneous use of components. Information on slowdown due to simultaneous use of caches points is still relevant and points to running two parallel cache intensive programs being slower than one after the other.

	Homogeneous stressing workload			Mixed stressing workload			
C1 Target	L2	mem_bw		L2	L2	mem_bw	mem_bw
C2 Target	L2	mem_bw	Max	L2	mem_bw	L2	L2
C3 Target	L2	mem_bw		mem_bw	mem_bw	L2	mem_bw
STAP Radar	1.05	1.06	1.06	1.05	1.05	1.05	1.05
H264 encode	1.07	1.07	1.07	1.08	1.07	1.06	1.07
CoreMark	1.02	1.03	1.03	1.04	1.04	1.02	1.04

Figure 5.6 – Standard benchmark interference on the Intel Core2Quad (taken from [44])

[44] does not provide the analysis of potential interference channels with more than two components in use simultaneously, but it does analyze the slowdown factor suffered by some standard benchmark applications running on one core when other components are being stressed by all the other core. The results are shown in Figure 5.6. The observed slowdown factors are very small, regardless of the components being stressed by the other cores. This points to the use of standard benchmarks being poor indicators of the worst slowdown that can be suffered because of interference. Indeed, none of these results reflect the high slowdown factor that was obtained when even just two cores stressed the same L2 cache. Thus, the effects of interference on an application that uses the L2 cache more than those standard benchmarks is likely to be much higher than what the results from Figure 5.6 would lead to believe.

To summarize, the strategy employed here forms the basis of covert interference channel identification, as it exposes potentially unknown links between components, but it also provides some quantification of the architecture's capabilities through the slowdown factor, and argues for the use of micro-stressing benchmarks over that of standard ones for a true observation of the maximum impact of interference, including for interference related to the use of caches.

The next paper expands on this approach, by using performance monitors to measure more than cycle counts, and thus learn more about how components interact.

5.1.2 Architecture and Application Characterization

The approach presented in [11] can be seen as a follow up to the one of [44]: the correlations between components that were exposed are used to tailor the benchmarks performed on the application. The process is summarized in Figure 5.7.

When profiling the architecture, the objective is to characterize the hardware itself, in order to identify shared hardware resources (including some that may be missing from the architecture's documentation), determine their behavior when in contention, potentially uncover unsuspected interactions between hardware components, measure pertinent execution times and maximum bandwidths. To do so, the approach starts with a comparison of each benchmark running in isolation and running with other copies of itself running in parallel, as in the approach from Section 5.1.1 (the top-left diagonal in Figure 5.3). In a second step, benchmarks targeting different features are launched together. This provides information on which parallel resource accesses affect one another. Here also, the approach is similar to the one in Section 5.1.1 (all cases other than the top-left diagonal in Figure 5.3). Note that this similarity with the previous section is to be expected, as both approaches make similar assumptions on the availability of standard monitoring resources. The main difference being the use of hardware monitors other than the cycle counter in [11], where performance counters such as "number of L1 hits" are also used in order to obtain a clearer understanding of the architecture, and thus of the causes of the interference instead of just observing its effects. This can be used to uncover relations between components despite not having measured a change of slowdown factor upon their concurrent benchmarking.

Hardware monitors are counters that can be set to increase on the occurrence of a given event, such as an access to the interconnect, a cache miss, a cache eviction, and many other. The architecture's documentation generally provides a list of events that can be monitored. These monitors can be reset, temporarily frozen, or set to track another type of event during the execution of programs, making them a very useful debugging and analysis tool.

[11] proposes using this information to limit the analysis made on programs to what is truly necessary. Indeed, the architecture characterization phase having already determined which combinations of benchmarks are redundant for the analysis of interference between tasks, only a subset needs to be employed to fully understand the impact of resource sharing on the timing of the applications.

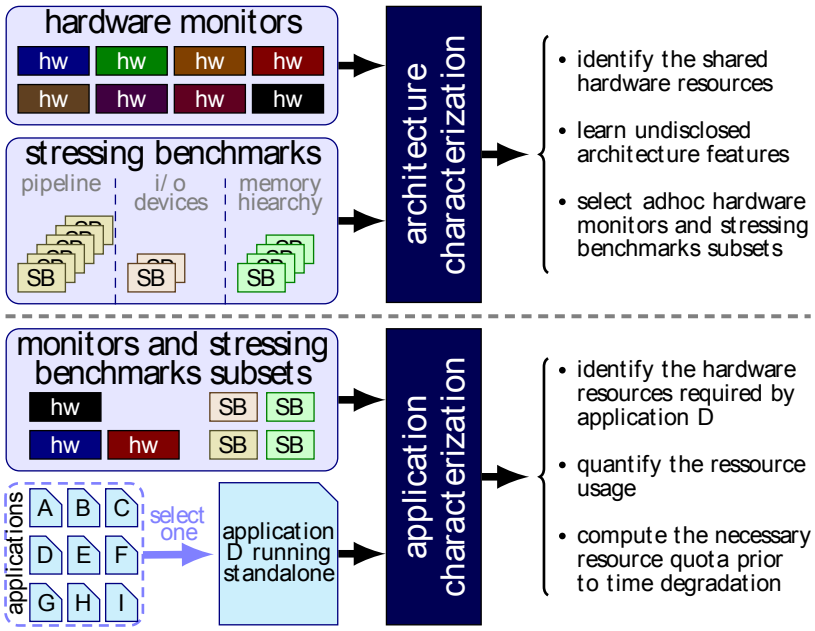


Figure 5.7 – Strategy overview for [11] (Figure taken from the paper)

This subset of benchmark combinations is then used to analyze how much usage of a particular resource is needed before an application slows down. In effect, this indicates the minimal share of each resource needed for an application to not suffer drastic worst case execution time slowdowns, and thus helps effective scheduling of applications on a multi-core processor.

```
1  LOOP (i=0;i<N;i++)
2  {
3      FREEZE_Counter();
4      INITIALIZE_Counter();
5      CONFIGURE_Counter();
6      UNFREEZE_Counter();
7      {Benchmark execution...}
8      FREEZE_Counter();
9      COLLECT_Counter();
10 }
```

(a) Performance Counter Management

```
1  LOOP (i=0;i<TABLESIZE-1;i+=STRIDE*UNROLLED)
2  {
3      ACCESS_TABLE(i,OPERATION)
4      NOP
5      ACCESS_TABLE(i+STRIDE,OPERATION)
6      NOP
7      ...
8      ACCESS_TABLE(i+(STRIDE*(UNROLLED-1)),OPERATION)
9      NOP
10 }
```

(b) Benchmark execution

Figure 5.8 – Benchmark algorithm used in [11] (extracted from [10])

While [11] does not provide any code extracts, the algorithm can be found in the associated thesis ([10]). Figure 5.8 shows an overview of algorithm used by [11]. Figure 5.8a indicates how the performance monitors are handled. The use of a loop to perform multiple measures of the same experiment allow capture of the result's variability. The benchmarked operations take the form shown in Figure 5.8b: it features an unrolled loop (with UNROLLED corresponding to the size of that inner loop) within another loop, all performing the same operation (writing or reading). The

only reason for there to be an unrolled loop within the primary loop is to reduce the amount of computations made during the iteration: the combination of both loops simply makes the operations be performed across the whole TABLESIZE addresses. The STRIDE parameter corresponds to the gap between accesses, which is there to control the number of times the same cache line is accessed. [10] indicates that the use of the NOP parameter is there to have idle time between accesses: it corresponds to a varying number of nop operations, and can thus be used to see the effect of temporally spacing out the accesses.

The cache coherence identification part of this thesis (Chapter 8) uses an algorithm similar to the one shown in Figure 5.8a to observe the behavior of caches. In effect, Chapter 8 is a specialized part of the *learn undisclosed architecture features* section of the *architecture characterization* process shown in Figure 5.7. Instead of simply detecting interaction It goes beyond the analysis of simple interactions to really understand

5.2 Analyzing Cache Performance

The previous section showed approaches to the detection of potential interference channels that could be applied to any component, but whose genericity ran the risk of failing to detect complicated relations between components. This section focuses on approaches that use benchmarks to characterize cache coherence performance and/or interference channels.

5.2.1 Cost of Cache Coherence

[39] places itself in a context similar to this thesis: the issues of interference in multi-core architecture preventing their use in avionics. More precisely, the paper proposes an analysis of an architecture through benchmarks to ensure that tasks running in parallel can correctly be time partitioned. The tasks in question involve mixed-criticality, meaning that some tasks are more important than others, and thus lower importance tasks must not impede higher importance ones.

The studied architecture is a NXP QorIQ P4080, featuring eight single-thread cores with their own L1 and L2 caches, and interconnected through a CoreNet Fabric implementing the MESI cache coherence protocol. There are also two L3 caches accessible through the interconnect.

The main sources of interference studied in [39] are simultaneous accesses to the interconnect and the main memory, as well as the latencies induced by cache coherence. The idea is to have one core act as the active observer, meaning that it is where time is measured, but it also performs some operation. The other cores only act as a source of interference, and multiple benchmarks are performed to see how increasing the number of secondary cores influences the time measured by the observer core. This is similar to the approaches presented in the previous section, with the difference being that instead of targetting a specific component, the benchmarks are solely focused on memory access, and the comparison is made on cores performing either *read* or *write* on memory elements.

For the evaluation of the latencies induced by cache coherence, three categories of cache coherence benchmarks are considered:

- **disabled:** No cache coherence enabled (i.e. the baseline), where its mechanisms are disabled and the cores do not target any of the same memory elements.
- **static:** Cache coherence mechanisms are enabled, but the cores still do not target any of the same memory elements.
- **dynamic:** Cache coherence enabled, and cores only target shared memory elements.

This is made possible by the fact that the architecture supports disabling cache coherence all together. For all three categories, sub-benchmarks are performed, in which the observer core either reads or writes memory elements while the secondary core also perform either reading or writing (thus resulting in four combinations for each category of cache coherence: read read - *rr*, read - write, *rw*, *wr*, and *ww*).

```
1  for(i=0; i<NMEAS; i++){
2      flush_caches();
3      sync();
4      time();
5      meas_loop(OPERATION, NBYTES, GAP);
6      time();
7  }
8
9  #define __asm_write_loop_gap64 \
10         __asm_meas_init \
11         \
12         "mr             __addr, %0; \
13         \
14         1: \
15         stb             __wr_val, 0x000(__addr); \
16         stb             __wr_val, 0x040(__addr); \
17         stb             __wr_val, 0x080(__addr); \
18         stb             __wr_val, 0x0c0(__addr); \
19         stb             __wr_val, 0x100(__addr); \
20         stb             __wr_val, 0x140(__addr); \
21         stb             __wr_val, 0x180(__addr); \
22         stb             __wr_val, 0x1c0(__addr); \
23         add.            __addr, __addr, __gap; \
24         cmpw            7, __nbytes, __addr; \
25         bgt             7, 1b;" \
26         \
27         __asm_meas_clean
```

Figure 5.9 – Algorithm overview for [39] (taken from the paper)

Figure 5.9 shows the algorithm used by [38]. Prior to each measurement, the caches are flushed, then the cores synchronize with each other. Time is recorder using utilities from the cores, which avoids making time() be a source of interference. The measured operation takes three parameters: OPERATION corresponds to either reading or writing, NBYTES is the number of bytes accessed in a measurement, GAP is the distance between the accesses made in order to target separate cache lines and thus ensure that all accesses are cache misses. The number of measures is controlled by NMEAS. The __asm_write_loop_gap64 corresponds to a meas_loop for a writing operation with a gap of 64 bytes.

The results of these benchmarks are shown in Figure 5.10.

Comparing *Disabled* and *Static* Cache Coherence: *Static* cache coherence (mechanisms enabled, but each core access different memory elements) does have an overhead compared to no cache coherence at all when performing read operations, regardless of the operation being performed by the other cores. For writing, this overhead is much lower. This is interesting, as the *static* benchmarks are the same experiment as the *disabled* cache coherence, meaning that effectively measures the minimal overhead induced by having cache coherence mechanisms active, regardless of how their are used. Thus, even when not having any use for it, keeping cache coherence mechanisms enabled

coherency	operation	execution time depending on number of active cores [us]							
		0	1	2	3	4	5	6	7
disabled	rr	69	70	69	70	70	71	71	72
	rw	69	69	71	89	86	83	83	84
	wr	60	62	62	62	63	63	64	64
	ww	60	60	75	91	92	93	94	97
static	rr	82	83	84	86	87	90	94	101
	rw	81	82	85	99	95	97	101	105
	wr	60	62	64	66	68	71	75	81
	ww	60	62	77	94	97	101	105	111
dynamic	rr	82	83	86	88	89	93	95	98
	rw	82	82	84	84	86	87	89	90
	wr	158	146	130	114	111	107	103	99
	ww	158	158	157	156	156	155	155	155

Figure 5.10 – Observed execution times depending on cache coherence and number of active cores (adapted from [39])

does lead to a increase in execution times. This makes an argument for taking extra steps and disabling cache coherence for memory elements that are not shared.

Comparing *Static* and *Dynamic* Cache Coherence: *Dynamic* cache coherence (cache coherence enabled, and accesses made to only shared memory elements) leads to the reverse: read operations yield better execution time compared to writes. The best execution times are obtained by reading while the other cores write, and the worst by writing while other cores are writing as well. This is a surprising result (*wr* and *ww* being slower than *rr* and *rw*), but these benchmarks do not take coherence state of memory elements into account. The order in which cores get to perform their operation changes that coherence state, and thus it changes the cache's response to other cores' operations, which in turn changes the execution time results. Because of this, the results for *dynamic* cache coherence analysis cannot be exploited for a precise measurement of the effects of cache coherence on running software.

To summarize, [39] does provide an interesting metric, which is found by comparing the execution time between the *disabled* and *static* benchmarks. The overhead caused by unrequited cache coherence queries are considered to be a form of interference. Indeed, this corresponds to the *minor* interference of Chapter 10 (see Definition 60).

The lack of information on coherence states prevent the *dynamic* benchmarks from being reused in the context of this thesis. However, a more appropriate form of benchmarking for cache coherence mechanisms with shared memory elements is explored in the next section.

5.2.2 Cache Performance Analysis

[38] is a paper on benchmarks performed on the Intel Nehalem architecture (see Figure 5.11), which uses the MESIF cache coherence protocol and has an inclusive cache hierarchy. For clarification's sake, the cache coherence protocol is active below the L3 cache, meaning that this dual processor architecture effectively results in only two caches being made coherent through MESIF.

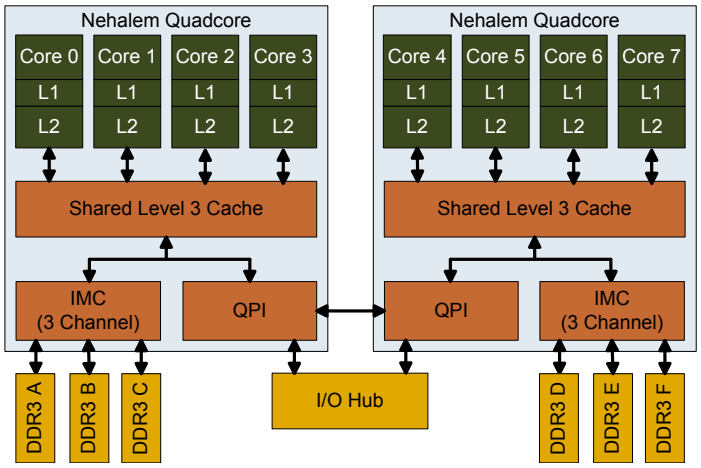


Figure 5.11 – Interconnected Intel Nehalem processors (extracted from [38]).

To perform the benchmarks comparing the effect of coherence states of memory element on writing and reading performance, their benchmark library makes it so they are able to put data in the desired stable coherence state in the cache they want, potentially using the other processor to attain the desired state. In effect, this allows control of the coherence states of the system prior to the start of the benchmark. To reach the desired state in the caches of core N, the following strategy is employed:

Modified Core N writes the data.

Exclusive Core N writes the data (thus invalidating the other cores' data), flushes, then reads the data.

Shared Core N reaches the *Exclusive* state, then another core reads the same data.

There is an interesting omission: the *Forward* state is not considered. The authors indicate expecting the *Forward* state to only become an improvement for systems in which there are more than two processors and is thus assumed not to have any effect on the benchmarks of their dual processor architecture. This is incorrect, but, as explained later, the effects of the *Forward* are not seen in their results, since the benchmarks which would have been affected were also omitted.

On the other hand, the strategy to reach a selected state for core N described above is still valid, even when ignoring the *Forward*. Indeed, with this process, the *Forward* state only appears when reaching the *Shared* state, but it is reached by the core that "reads the same data", not core N. In reality, having a *Forward* state in an architecture with only two caches does actually have some benefits: if the two caches have read certain data, then one wants to write to it. Without *Forward* state, the cache wanting to write will have to fetch data in RAM, whereas with the *Forward* state available, it might be provided by the other cache or simply ot have to fetch it in RAM (depending on which cache performed a write first, and choices of implementation). Similarly, it has an impact if a cache line is read by both caches, but the cache not in the *Forward* cache evicts it at some point, them re-acquires it. This would have had an observable result when measuring the writing execution time. Indeed, writing when core N is in the *Shared* state and another core is in the *Forward* state should result in faster execution time than when the other core is in the *Shared* state. Since The paper does not feature a table proving a list of execution time when writing, but only for

benchmarks that reading, this difference cannot be seen and one might erroneously assume that the *Forward* state does indeed have no impact on systems with coherence maintained between only two caches.

```
1  thread 0: warm-up TLB
2  if (N>0): sync of thread 0 and N
3  thread N: access data (-> E/M/S)
4  if (N>0): sync of thread 0 and N
5  all threads: flush caches (optional)
6  thread 0: measure execution time
```

(a) Latency benchmarks

```
1  all threads: access data (-> E/M)
2  all threads: flush caches (optional)
3  all threads: barrier synchronization
4  thread 0: define start_time in future
5  all threads: wait for start_time
6  all threads: measure t_begin
7  all threads: access data (read/write)
8  all threads: measure t_end
9  duration = max(t_end) - min(t_begin)
```

(b) Bandwidth benchmarks

Figure 5.12 – Algorithm overview for [38] (taken from the paper)

Figure 5.12a shows an overview of the algorithm used by [38] to measure execution times. The steps described are more about what is done prior to the measurements themselves. Warming up the transaction look-aside buffer means pre-loading all entries in order to avoid this loading being taken into account in the resulting execution time. The accesses made for the measured part of the benchmark correspond to a pointer chasing algorithm, just like in [44] (see Figure 5.8b).

The bandwidth benchmarking algorithm is shown in Figure 5.12b. Synchronization between the threads is more thoroughly controlled in this one: by memorizing the exact window upon which each thread made its accesses, the window corresponding to the period in which all threads were performing accesses can be obtained. The bandwidth can then be obtained from the number of accesses successfully performed within this window.

	Exclusive cache lines			Modif ed cache lines			Shared cache lines			
Source	L1	L2	L3	L1	L2	L3	L1	L2	L3	RAM
Local	1.3 (4)	3.4 (10)	13.0 (38)	1.3 (4)	3.4 (10)	13.0 (38)	1.3 (4)	3.4 (10)	13.0 (38)	65.1
Core1 (on die)	22.2 (65)			28.3 (83)	25.5 (75)		13.0 (38)			
Core4 (QPI)	63.4 (186)			102 - 109			58.0 (170)			106.0

Figure 5.13 – Latencies of reading from Core 0, results in "nanoseconds (cycles)" (Figure taken from [38])

Figure 5.13 shows the results of benchmarks measuring the time required for the core 0 to read data held in various locations, according to the state of the data in the remote location. Unsurprisingly, accesses made to the local L1 and L2 caches was not impacted by the state of the data in the L3 cache. It appears this also holds true for the local L3 cache itself. Access to data held in the caches of another core on the same processor does vary depending on the coherence state. According to [38], this is explained by the L3 cache having to check on the L2 and L1 caches of that other core, as, unless the state is *Shared*, the L1 and L2 cores may hold a more up-to-date value. For data held in the other processor, the results are as expected, with the cost of traversing the bridge between both processors being added when accessing memory elements in either the *Exclusive* or *Shared* state, but also having a higher cost when accessing *Modified* memory elements: as explained in [38] and Chapter 3, *Modified* memory elements are wrote back to the memory prior to being sent as a reaction to a *GetS* query.

	Reading							Writing						
	Exclusive			Modif ed			RAM	Exclusive			Modif ed			RAM
	L1	L2	L3	L1	L2	L3		L1	L2	L3	L1	L2	L3	
Local	45.6	31.1	26.2	45.6	31.1	26.2	10.1	45.6	28.8	19.9	45.6	28.8	19.9	8.4
Core1	19.3	19.7		9.4	13.2			23.4	22.2	17.6	9.4	13.0		
Core4	9.0	9.2		5.6			6.3	9.0			8.3		9.6	5.5

Figure 5.14 – Bandwidth for access from Core 0, results in GB/s (Figure taken from [38])

[38] does however perform more analyses on the performance of cache accesses. The second half of the paper is dedicated to bandwidth analysis, of which the results are shown in Figure 5.14. These results are coherent with their equivalent in the execution time benchmarks. They do provide more information than what was available in the documentation however, such as actual maximum bandwidth instead the theoretical maximal one.

The approach presented in [38] is a good solution for the *benchmark* part of the framework described in Section 4.2.2. Indeed, by taking into account the coherence state of the targeted memory elements, [38] ensures the system state that led to the recording of the execution time is understood and thus, that the correct execution time will be expected when attempting to predict the architecture's behavior.

5.3 Finding Elusive Hardware Monitors

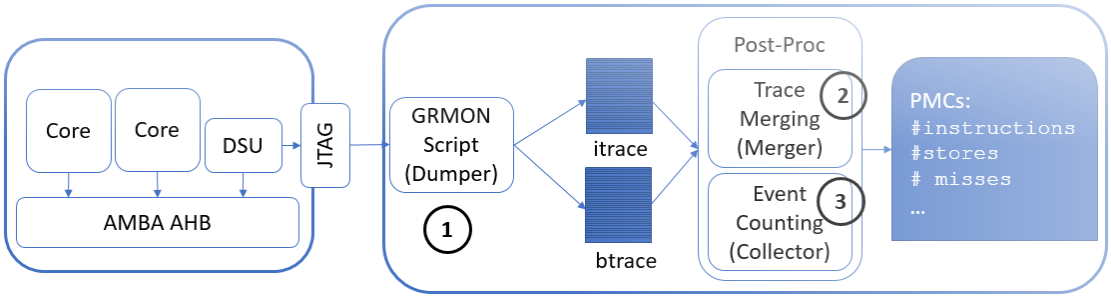


Figure 5.15 – Trace collection process (extracted from [40])

[40] presents a case study for the profiling of architectures where performance monitors are not available. This is interesting, because architecture profiling usually assumes that they are available. Having a way to profile architectures without would extend the range of architectures that can be considered for critical real-time contexts.

The architecture being studied in [40] is system-on-chip with a dual-core processor. While it does not have performance monitors, it does have some hardware debugging features, which can be accessed using a *JTAG* port.

These hardware debugging features can snoop the messages passing through the interconnect, as well as the instructions executed by each core (and their program counter). In the approach proposed by [40], the platform is configured so that these two sources of information are stored into circular buffers (within the chip). The *JTAG* connection is used to continuously copy these buffers

into an external analysis platform (see Figure 5.15). The speed at which these buffers are filled is higher than that of the *JTAG* connection, meaning that information would be lost if the programs were running normally. Thus, the *GRMON* script takes control of the execution once the section of the programs under analysis is reached, and proceeds to running the applications in a *step-by-step* manner, which ensures no information is lost.

Once this event capture is completed, the post-processing steps begin, starting with a clean-up phase (*Trace Merging*), which clears out redundant data. Then comes the *Event Counting* phase, which matches instructions to bus events in order to recognize patterns that fit known events. For example, data cache load misses are identified by seeing a load instruction be performed by a core one cycle *after* seeing the matching query go through the interconnect.

Thus, instead of relying on performance counters, it is possible to obtain an accurate description of the behavior of the platform through capture and analysis of execution traces. Such strategies extend the range of platforms upon which profiling is possible, including for the purpose of identifying cache coherence using the process described in Chapter 8.

5.4 Conclusion

The approach shown in Section 5.2.2 is adequate for the second step of the framework presented in this thesis (see Section 4.2.2), in which the performance of the architecture's cache coherence mechanisms are measured in order to feed them to a model. Indeed, it does provide interesting information about single instruction execution time according to the type of instruction being performed (`load` or `store`) and the cache coherence state. It does not, however, attempt to analyze the internals of the cache coherence mechanisms used by the architecture. In fact, some of them are explicitly ignored (*Forward* state). Thus, while being an important step, it remains insufficient to provide the user with enough information to understand what interference could be generated by cache coherence mechanisms, unless the identification step proposed in this thesis is applied.

Furthermore, approaches such as the one described in Section 5.3 can be used to apply the framework presented in this thesis to architectures with more restricted monitoring facilities.

Chapter 6

Handling Cache Interference in Safety-Critical Systems

This chapter explores existing solutions to the issue of the interference generated by cache coherence in safety-critical systems. Three approaches are considered: Limiting either the capabilities of either the architecture or the software running on it in order to limit the generation of these interference (Section 6.1); Modifying the architecture's hardware or adding new hardware components in order to lessen the unpredictability (Section 6.2); and lastly, solutions that involve running software in safety-critical systems by showing that the effects of the interference remain acceptable.

6.1 Through Restrictions

The crudest approach to the handling of interference generated by cache coherence is to not have any because all caches are disabled. While this solution tremendously improves the predictability of the system, it also tremendously decreases its execution speed, to the point where it may be preferable to use a single-core architecture with caches instead.

One step above is a solution in which the caches are enabled, but their content is locked, making their usefulness severely limited but without compromising the system's predictability.

In this section are presented strategies that allow the use of caches in a limited manner in order to achieve reasonable execution speed while keeping the system as predictable as possible.

6.1.1 Shared Cache Partitioning

[29] presents two algorithms for a condition sufficient to prove schedulability, it is meant for tasks with time and cache space constraints in the context of a shared L2 cache in which space is partitioned in order to avoid contentions. Thus, this work is more about proving that the measures taken in order to address interference are sufficient rather than a strategy to control the interference itself. For these schedulability tests, the tasks are assumed to be non-preemptive and to have a fixed priority.

The paper's first algorithm is based on constraint programming. It basically considers that there is a task missing its deadline, which implies that either all cores are being used, or that too little cache space is available. Intervals at which either of these conditions is true are searched for in order to find their maximal impact of the hypothetical task that missed its deadline. If this maximum impact is lower than the slack (longest affordable delay) of this hypothetical task, then the tasks are

schedulable. The second algorithm is very similar, but simplifies the constraints by simply assuming the worst possible interference from the other tasks in all criteria with much less regard for whether the configuration leading to this interference is actually possible. This effectively creates a more scalable schedulability test, at the cost of being much more pessimistic.

6.1.2 Cache Coloring to Curtain Interference

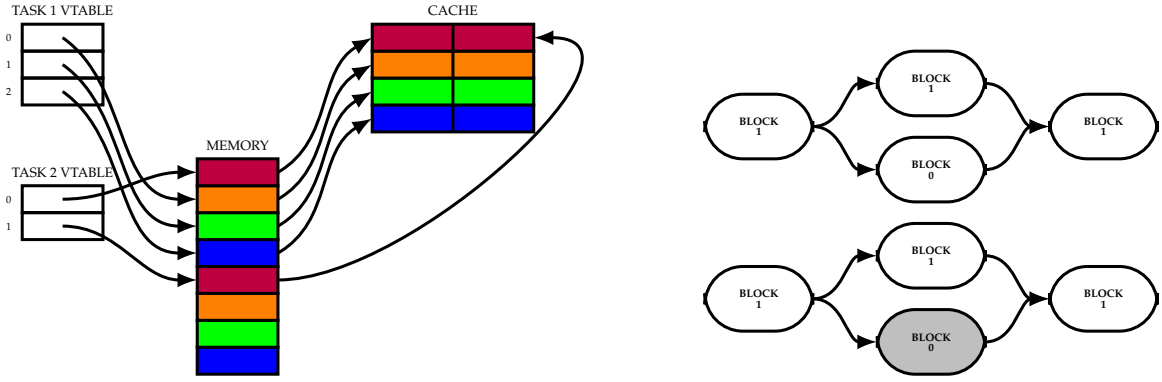


Figure 6.1 – Cache coloring aware scheduling, from [13]

When using a set-associative cache, the cache is divided in equal sets of cache lines. The memory element's physical address determines which of these sets of cache lines it will go to. The cache eviction policy then applies solely inside that set. The idea behind cache coloring is to exploit this, by determining which memory element will find itself in which set of lines, and assigning virtual addresses to physical addresses in a manner that will ensure a maximum of virtual addresses will find themselves in the cache.

[13] uses cache coloring in the opposite manner. It instead assigns a single color (thus, set of cache lines) per program. While this is not an efficient use of the cache for any one program, it means that each program is effectively curtained into the set of caches of its color. Using careful scheduling, the number of programs running in parallel that share the same colors can be reduced (see Figure 6.1), thus removing the interference they can inflict on one another.

6.1.3 Limited Shared Resources

One approach to dealing with cache coherence in multi-core processors for environments requiring certification is to simply use the caches in a manner that cancels the need for such a feature. Indeed, since the applications for single core are already available, it stands to reason plenty of them could run on multi-core processors without the need for shared memory elements. For example, [34] presents *Marthy* (see Figure 6.2), a hypervisor aiming to enforce robust partitioning between applications running on a COTS multi-core processor. This hypervisor permanently lives in the cache of each core, using cache locking mechanisms to avoid being inadvertently removed, and makes use of the MMU to take over whenever an instruction triggers a cache miss, stalling that instruction until access to the system's shared resources (i.e. any shared component, not just memory) is exclusively granted to the core by a TDMA. All shared memory elements must be read-only, which does indeed remove the need for cache coherence mechanisms.

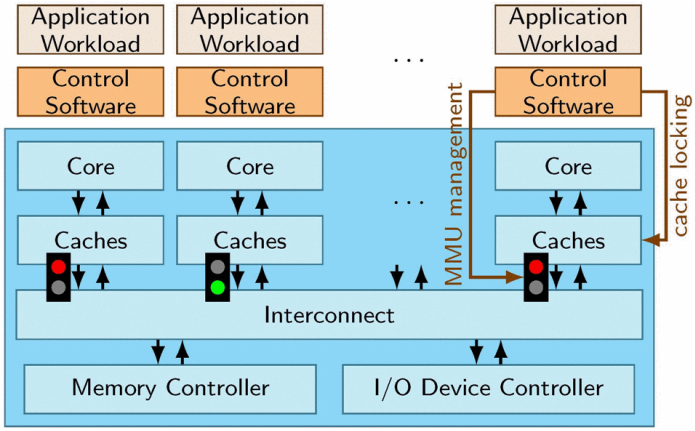


Figure 6.2 – System running Marthy (Control Software in the figure), a figure extracted from [27].

6.1.4 Isolated Communications Through Scheduling

In [12], careful scheduling is used to make calculating the worst-case execution time easier. This scheduling requires programs running on each cores to have been sliced into computation blocks and communication ones. Computation blocks are built so that while a core is in one, the other cores cannot access the same data (or only as a read-only copy if all tasks are only reading this data). Communications nodes indicate fetching and writing periods (no computation), which are similarly limited by any task currently performing a computation block. The paper provides a strategy to automatically slice the programs intended for a core into a scheduling of these types of blocks. Figure 6.3 shows an example of two cores having their tasks scheduled in this manner. White blocks indicate time reserved for computation; grey blocks are for flushes; and black blocks are for fetches.

Because they do not access any shared resource, calculating the worst-case execution time of the computation blocks is equivalent to doing so on a single core processor. [12] proposes a solution for the calculation of the worst-case time of communication blocks, including the possibility for them to occur in parallel with other communication blocks from other cores. It relies on the creation of a UPPAAL model to account for all possible interleavings of any communications that may occur during that particular communication block. The aforementioned scheduling ensures that all communications that can happen at that time are known.

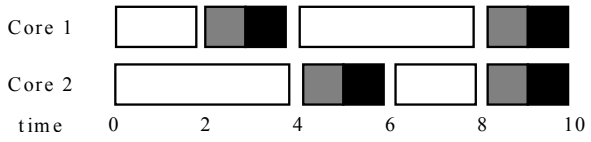


Figure 6.3 – Overview of the strategy presented in [12] (taken from the paper)

6.2 Through Hardware Modifications

6.2.1 Predictable MSI

While not exactly a hardware modification per se, the use of a coherence protocol designed for its predictability is not likely to be available on COTS hardware, which is why the work presented in [32] is included in this section.

[32] exposes the sources of unpredictability in the MSI protocol, and proposes a solution for each one of them. These solutions require some parts of the hardware connected to cache coherence mechanisms to be predictable. Combined with, PMSI (Predictable MSI), the coherence protocol they introduce, these solutions are indicated as sufficient to allow calculation of the worst case delays that may be incurred by each instruction.

Translated to the formalism from Chapter 3, the restrictions imposed on the hardware are as follow:

1. Access to the interconnect is regulated by time slots. Caches may only send queries to the interconnect during their assigned time slot.
2. For a same memory element, the coherence manager sends replies to queries in the order of their arrival.
3. Caches reply to queries in the order of their arrival. This is equivalent to imposing $\mathtt{D}_{\mathtt{out}}$ to be a FIFO queue.
4. A write request to a cache for a memory element that it does not currently hold with read-and-write permissions can only occur during that cache's time slot.
5. Writing to a memory element not currently held with read-and-write permissions requires all other caches' pending queries for that memory element to be processed first. Basically, the write is stalled if there are any queries in $\mathtt{Q}_{\mathtt{in}}$ for that memory element.
6. Caches are predictible in the ordering of their processing of instructions and external queries.

Adapting `PMSI` to fit Chapter 3 is made easy by the fact that they also based their description on the notations introduced in [49]. Figure 6.4 shows how the cache controller behaves for each memory element. This figure is almost identical to the one found in the paper presenting `PMSI`, with the following modifications:

- A single *Own Query* event is used instead of having one per type of query. This makes the table more compact, as there is never any state in which multiple categories of query have a different behavior when it comes to the handling observing their own queries on the interconnect.
- The `Upg` query type, used to move from `S` to `M`, has been merged into `GetM`, as the differentiation did not appear to have any relevance to the protocol's behavior.
- Emission of a `PutM` has been added as an action when receiving data in either $\mathtt{IM}^{\mathtt{D}}\mathtt{I}$ and $\mathtt{IM}^{\mathtt{D}}\mathtt{S}$, as it would otherwise be impossible to exit the state they lead to. The need for this has been confirmed during exchanges with the authors.

[32] does not include a table for the coherence manager, which is instead simply described as servicing the oldest pending query for a memory element every time it receives data. Figure 6.5 shows a coherence manager that would implement this behavior.

By comparison to the `MSI` protocol described in Section 3.3, `PMSI` features much fewer transient states. This is because the restrictions imposed on the system have removed:

State	Core Request			Own Query	Data Reply	Received Queries		
	load	store	evict			GetS	GetM	PutM
I	GetS? IS^D	GetM? IM^D				-	-	-
S	hit	GetM? SM^W	I			-	I	
M	hit	hit	PutM? MI^{WB}			PutM? MS^{WB}	PutM? MI^{WB}	
IS^D	stall	stall	stall		S	-	IS^DI	
IM^D	stall	stall	stall		M	-	IM^DI	
SM^W	stall	stall	stall	m!data I		IM^DS	IM^DI	
MI^{WB}	hit	hit	stall	m!data I		-	-	
MS^{WB}	hit	hit	MI^{WB}	m!data S		-	MI^{WB}	
IM^DI	stall	stall	stall		PutM? store hit MI^{WB}	-	-	
IM^DS	stall	stall	stall		PutM? store hit MS^{WB}	-	IM^DI	
IS^DI	stall	stall	stall		load hit I	-	-	

Figure 6.4 – Split-Transaction PMSI Automaton for Cache Controllers

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

Figure 6.5 – Possible Split-Transaction PMSI Automaton for the Coherence Manager

- The possibility of receiving a reply to a query not yet handled: $\texttt{IS}^{\texttt{BD}}$, $\texttt{IS}^{\texttt{B}}$, $\texttt{IM}^{\texttt{BD}}$, $\texttt{IM}^{\texttt{B}}$, $\texttt{SM}^{\texttt{BD}}$, and $\texttt{SM}^{\texttt{B}}$.
- Cache to cache data messages, which merge cases that were separated because of whether data was sent only to another cache or also to the main memory: $\texttt{IM}^{\texttt{D}}\texttt{SI}$, $\texttt{SM}^{\texttt{D}}\texttt{SI}$.
- Sending data as a reaction to seeing another cache's query: $\texttt{II}^{\texttt{B}}$ (now undistinct from $\texttt{MI}^{\texttt{B}}$, which is now $\texttt{MI}^{\texttt{WB}}$).

In PMSI, receiving data for a memory element currently held in a cache that may have modified it (e.g. M state) always requires waiting for that cache's time slot in the TDMA, as it will not perform a write back without first broadcasting its own query indicating that it is about to do so. The cache then sends a data message to the system's main memory, which only then is able to reply to the original demand. While this is clearly penalizing in terms of performance, it does lessen the variability in time required for acquisition of data.

In conclusion, [32] addresses the issue of the predictability of memory access latency by replacing the cache coherence protocol and performing some other hardware modifications in order to have a generally slower but more easily computed and less fluctuating memory access time. No modification of the software is required.

6.2.2 Limited Cacheability

One approach to the control of interference generated by cache coherence is to limit which memory elements are affected by it. [5] argues for letting developers indicate, upon memory allocation, whether to allow the new memory elements to be either cacheable as usual, not cacheable at all, or only cacheable in caches shared by all cores (refered to as INC-OC). Figure 6.6 summarizes where memory elements can be stored depending on the attribute they have been given. INC-OC, which is the main contribution of the paper, is intended to remove these memory elements from the cache coherence while not having to suffer the full cost of a cacheless system upon their access. This does indeed result in more easily predicted memory accesses for these particular memory elements, as they now behave as they would in a single core system running concurrent programs: another core/program may still evict them (either directly, or by allocating more memory and triggering the automated eviction policy), but the possibles system-wide states of those restricted memory elements (and thus possible access latencies) are much fewer that they would otherwise be. This can thus allow approaches for the analysis of memory latencies in single-core system to be applied to these memory elements, a problem for which the available literature is more prominent than for multi-core systems, with the added issue of bandwidth sharing between the cores for access to that last-level cache.

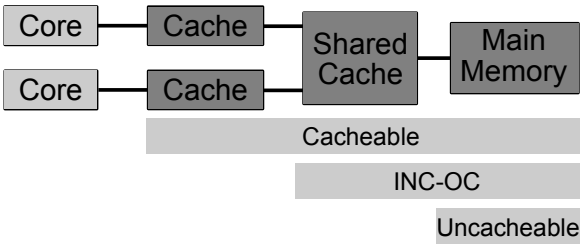


Figure 6.6 – Cacheability Levels

As for limitations, the most oblivious one is that this is only applicable to systems which do indeed implement a last-level cache being accessed by every core. Furthermore, the addition of a new type of memory leads to hardware modifications: translation look-aside buffers need to take into consideration a new attribute, and so do all sent memory access queries. The authors argue that some of these hardware modifications can sometimes be minimized through the use of the architecture's instructions, and that this approach has the advantage of being entirely orthogonal to the cache coherence mechanisms, thus not requiring any modification of the admittedly complex coherence controllers.

In effect, the approach presented in [5] requires some hardware, as well minor operating system and hypervisor, modifications, in order to let designers simplify the analysis, and lower the variation, of the access time for any memory elements they choose. This does come at the cost of the speed at which these memory elements are accessed, and it does require the designer make a decision as to which memory elements should be handled this way.

6.2.3 On-Demand Cache Coherence

[42] presents ODC^2 (On-Demand Coherent Cache), a strategy to limit the interference of cache coherence on the execution of real-time software (see Figure 6.7). The general idea is to have software delimit sections during which they access shared data (*Shared Mode*), and to have that shared data be evicted from the cache as soon as the software leaves the section (*Restore Procedure*). Any new data loaded during Shared Mode is marked as being *shared* in the cache line, making their Shared data cannot be accessed outside of these sections, which makes the code outside these coherence enabled sections (called *Private Mode*), much simpler to analyze.

A follow-up paper, [43], performs the WCET analysis of some well established algorithms (Dijkstra algorithm, Fast Fourier Transform, Matrix Multiplication) by modifying an existing WCET computation framework (OTAWA) to add support to ODC^2. The resulting WCET is compared with the ones obtained when using no caches, when using *Magic* (cache coherence without any cost, which represents the best theoretical performance), and an approach that simply invalidates the full cache upon entering any synchronization point. The results show, somewhat unsurprisingly, ODC^2 obtains a lower computed WCET than the other approaches, *Magic* excepted.

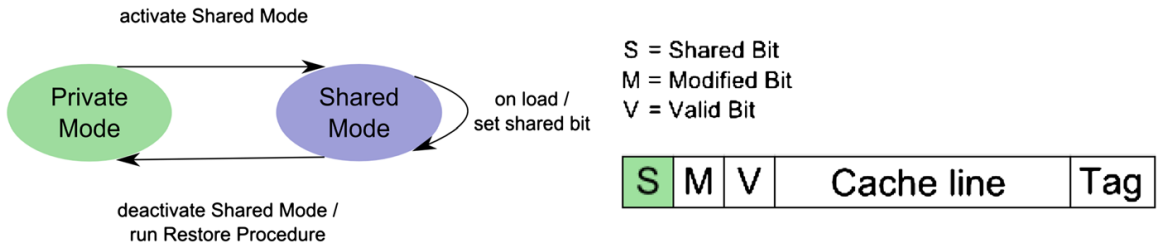


Figure 6.7 – Overview of ODC^2, as seen in [42].

6.2.4 Dynamic Verification of Cache Coherence

[15] proposes a strategy to implement fault detection in cache coherence systems, addressing both permanent and temporary issues by adding hardware (See Figure 6.8) that replicates a simplified version of the cache coherence logic and tests it for consistency with the real one.

The approach proposed by [15] recommends the addition of a new interconnect dedicated to the validation of the cache coherence protocol, to avoid disrupting the existing one. This new interconnect is solely used by caches to broadcast that they entered a stable state for a given memory element so as to allow the other caches to check that their local state for that memory element is not in conflict with the broadcasted one. For example, if a cache broadcasts that it has reached the M state for a memory element, then all other caches seeing that broadcast would test that they do not have that memory element in any state other than I.

This new verification performed on each cache partaking in the coherence protocol requires the addition of some hardware to each cache. In addition to checking for consistency with the other caches through broadcasts, this new hardware maintains a simplified version of the coherence state of each memory element. It does not incorporates transient states and, since it only considers state changes and whether they allow a given instruction to be performed, the resulting coherence protocol is that of one operating on an atomic interconnect with atomic operations (such as the one in Section 3.2.1). In effect, the simplified protocol only sees events once the transaction they were part of has been completed. At that point, it receives address, event, initial and final stable state and compares these with its own record. To detect timeout issues, a watchdog is also added to the caches.

Thus, while it requires rather complex hardware modifications, [15] does provide a way to detect the occurrence of errors in the cache coherence protocol's behavior.

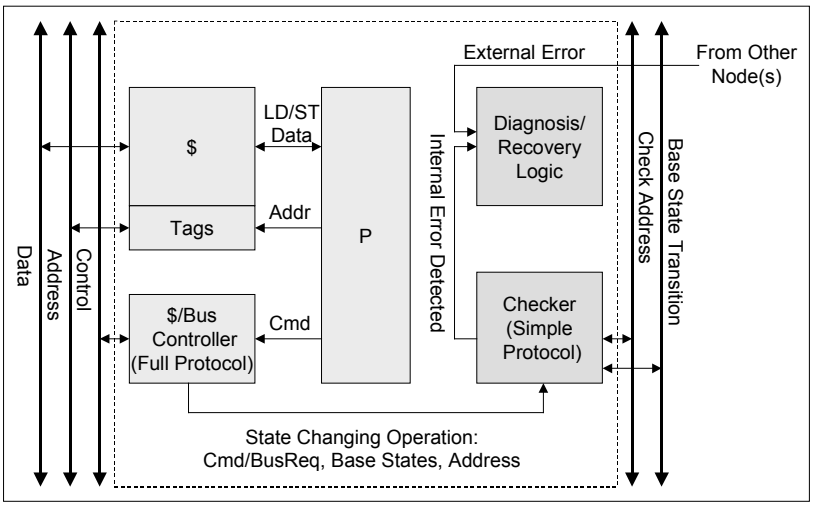


Figure 6.8 – Overview of the altered cache, as seen in [15].

6.3 By Accepting It

The most common approach to the analysis of interference in caches relies on the static analysis and abstract interpretation approach originating from [24], which categorized accesses to caches depending on whether they always found the value, never did, or if that could not be determined. This was then taken into account account for the computation of the WCET. The publications that followed usually propose the addition of a new categorization, or remark on the incorrectness or incompleteness of the way these categorizations evolve during the analysis of the program's CFG, and propose corrections for mistakes made on previous attempts at doing so (up to, and including, what has been presented in [24]). While otherwise prevalent, this particular approach is not the one focused on in this thesis, and thus only very few of these papers have been included in this section. Readers interested in learning more about this subject are encouraged to read [50], and especially its well presented related works section.

6.3.1 Instruction Cache Analysis

Most papers handling WCET analysis on multi-cores only consider instruction caches, bypassing the need to address cache coherence by indicating that instructions cannot be modified. While this makes them slightly outside the scope of this thesis, the sheer prevalence of this restriction in WCET papers would make their complete omission feel amiss.

WCET analysis

[31] proposes a way to compute WCET for multi-core architectures, with considerations for shared instruction caches with multiple levels of hierarchy. It also tries out a *bypassing scheme*, which relies on the identification of single-use program blocks within shared instruction caches to compile programs in a way that reduces interference between tasks.

The proposed WCET computation strategy is based on existing approaches for multi-levels instruction caches in single core processors. It uses static analysis to categorizes all accesses as either:

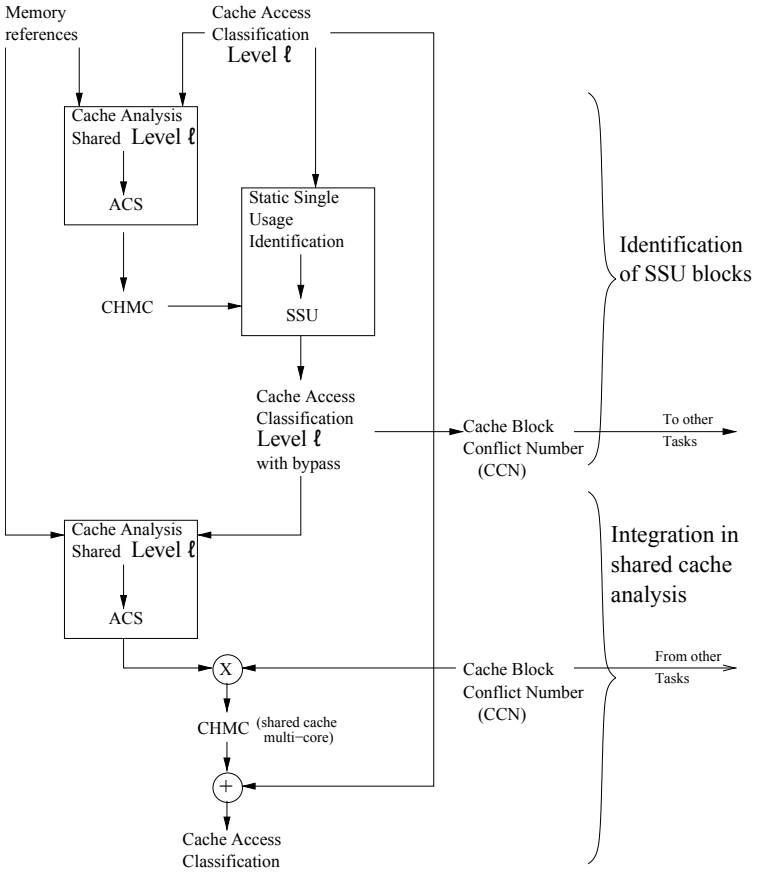


Figure 6.9 – Overview of the strategy presented in [31] (taken from the paper)

always-miss, always-hit, first-miss (first access is unknown, but all following accesses will hit), and not classified (for accesses that fail to be categorized in the other patterns). This categorization (referred to as *CHMC* in the paper) is done for every cache level. Furthermore, the likeliness of an access actually reaching a level is also evaluated and categorized with similar labels (always acceded, never, always after the first access, unknown). Moving this strategy to multi-core processors is indicated as having the potential of changing some always-hit and first-miss into first-miss and *not classified*. Accesses made by different tasks on the same level where the categorization indicates that the access is not *never* done are considered as interfering with one another (regardless of when the accesses are made). This is then used to re-evaluate the hit/miss categorization so that it has the interference taken into account. The result of the interference identification analysis is referred to as *C*ache block *C*onflict *N*umber. To improve the precision of the result, the possibility of having code shared among tasks (such as libraries) is taken into account. An overview of the whole process for a given cache level ℓ is shown in Figure 6.9.

The decisions made in what to cache generally follow the locality of reference principle, where the proximity of elements in memory is assumed to translate to a proximity is usage times, and that processors receptively access the same elements within short time periods. In [31], static analysis is used to find elements which are only used once (called *S*tatic *S*ingle *U*sage), in order to avoid having

them pollute caches and risk being considered as a source of interference needlessly. This relies on a bypass mechanism, that allows fetching data without altering the caches: if it is not found within any cache, the value is retrieved but not added to any cache; if it is found within a cache, the caches in-between do not get any copy, and the copy that was used is not updated to indicate a recent access.

Unified WCET Analysis Framework

[18] presents a framework for WCET analysis on multi-core platforms, which distinguishes itself from other solutions by the number of features it takes into consideration. Indeed, the framework accounts for shared caches, pipelines, and branch prediction. It also does not require the assumption of a timing-anomaly-free architecture. An architecture free of timing anomalies are architectures for which performing an operation faster on a core cannot result in a longer execution time for the program compared to if that operation took longer. The oppose would, for example, if the instruction activates cache coherence mechanisms which would not have been necessary if the in faster execution of that instruction. Readers interested in more possible causes for timing anomalies are encouraged to read [46]). Figure 6.10 provides an overview of the approach. The assumptions that it does are as follow: use of a TDMA-based round-robin arbitration policy, cores each have a private L1 cache. Data and instructions are assumed to be fully separated (separate caches, separate buses). Only instruction caches are taken into account (no cache coherence, no possibility of modifications). Caches are assumed to be non-inclusive and to use the LRU replacement policy.

The approach analyzes the WCET for each core separately. Using abstract interpretation, memory accesses are categorized according to whether they always hit, always miss, or are marked as being unclassified. This is done for both the L1 and L2 caches. The L2 cache can be a shared one, in which case accesses susceptible to inter-core interference may move from always hits to unclassified. This inter-core interference appears to be the only considered direct interference from other cores, due to the assumptions made on the architecture. Speculative execution, the result of branch prediction, may also have effects on the contents of the L1 and L2 caches, as well as the pipeline's behavior. A model of the shared bus's TDMA is taken into account when creating the pipeline's model, adding a latency to certain stages. This pipeline model creation process appears to be fairly complex, proceeding in an iterative manner to compute the WCET of each basic computation block. Having the WCET of each basic block, a model of the branch prediction mechanics and that of the TDMA are used in the definition an linear optimization objective that will return the WCET of the whole program.

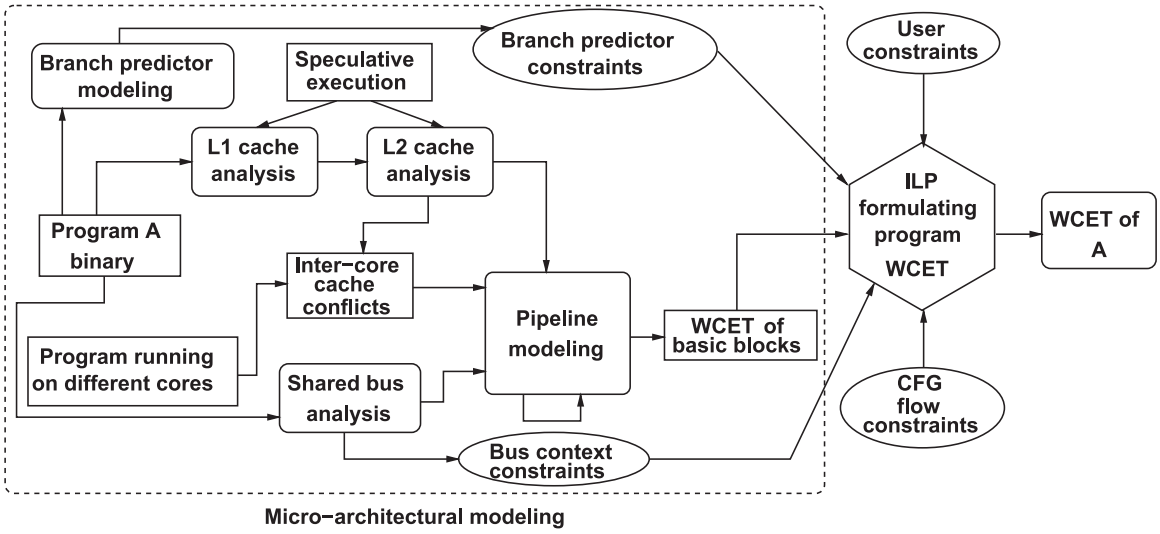


Figure 6.10 – Overview of the strategy presented in [18] (taken from the paper)

6.3.2 Data Cache Analysis

Bypass Heuristics

[35] uses a similar approach to [31], this time focusing on data caches instead. Cache coherence protocols are not considered. Indeed, all accesses to shared data is assumed to bypass private caches (something that could be implemented by [5], for example) and all shared-caches are assumed to be write-through. Data and instructions are assumed to not interfere with each other. In effect, the main change from what is presented in [31] is how accesses to a memory bock are determined: in the case of an instruction cache, a single instruction always points to the same memory blocks; whereas in data caches, the relation between instruction and memory block is murkier, as addresses may be aliased. This makes the detection of Static Single Usage cache blocks, which were the target of bypass mechanisms in [31], much more complex. Multiple heuristics on which elements to bypass at shared cache level ℓ are provided: any instruction for which none of the targeted memory references have statically been detected to be leading to a sure hit in ℓ in the future; any instruction for which the target cannot be statically computed, in order to increase determinism; and one that bypasses any access of a task until it only occupies a given number of cache ways, which allows for conflict reduction through control of the maximum occupied space by each task.

Write-Back Data Caches

[51] tackles the issue of WCET computation in multi-core processors that use data (or unified) shared caches with write-back policies. Although not explicitly stated in the paper, this approach does not consider separate caches of the same hierarchy: all caches are shared by all cores that could access the data they contain. Ergo, no cache coherence, but a cascade of non-inclusive shared caches.

In addition to the standard categorization of a cache line in a particular cache level (always hits, never hits, and so on...), the *dirtiness* is also modeled. The possible values are: dirty (i.e. has been written to, but changes were not yet propagated), clean, or possibly dirty. The main challenge is then to estimate when the write back will occur. [51] indicates how to take this new attribute into

account when performing the usual cache static analysis, and the constraints it adds to the linear optimization problem representation of the WCET computation.

6.4 Conclusion

The approaches presented in Section 6.1 are improvements on the predictability of the system, but at the cost of its performance. Those from Section 6.2 do not generally influence the system's performance in order to make it more predictable, but, by their very nature, are not compatible with the need to use existing, commercial off-the-shelf architectures. Lastly, Section 6.3 presented existing strategies to determine when the interference inherent to the use of a multi-core processor does not lead to unacceptable execution times. This last section is what this thesis contributes to. Indeed, none of the existing approaches tackle cache coherence, and are instead focused the sharing of caches. The strategy chosen in this thesis for determining the impact of interference caused by cache coherence on the system's performance is the use of formal methods.

Chapter 7

Analyzing Performance Through Formal Methods

The most common use of formal methods in relation to cache coherence is to validate protocol correctness (i.e. that a protocol verifies the properties set in Section 3.2.2). Indeed, the use of model checking for that purpose is the source of many publications. For example, [21] describes a parameterized model checker by focusing on the verification of a cache coherence protocol, by taking its description from yet another paper using model checking to verify it ([6]) and thus allowing a comparison between the approaches.

However, proving the correctness of a cache coherence protocol is not the subject of this thesis. We assume the protocols to be correct, and are instead interested by the impact the cache coherence has on the real-time properties of the system. Thus, in chapter, we look at real-time systems using formal methods to analyze the real-time properties of an architecture. As this is a fairly restrictive criterion, the results include approaches meant for single-core processors in addition to those for the more on-topic multi-cores.

7.1 Single-Core Processors

7.1.1 METAMOC

The first tool that made use of UPPAAL for the computation of WCET by modeling hardware was Modular Execution Time Analysis using Model Checking (METAMOC), introduced in [22]. The general idea behind the approach is shown in Figure 7.1. The modularity can clearly be seen in how the models for the cpu pipeline, main memory, and cache specifications are kept separate in order to facilitate their replacement when analyzing for a different processor. The other input is that of the analyzed executable, directly in its binary form, albeit with some annotations regarding loop bounds.

Given these inputs, METAMOC generates a control flow graph for the program, which is in fact yet another UPPAAL model to combine to the ones representing the architecture. The value analysis statically determines the address of memory elements, and is disabled if the modeled architecture does not feature any data cache.

The pipeline model corresponds to a collection of automata, one for each stage: *fetch*, *decode*, *execute*, *memory*, and *writeback*. The parallel nature of pipelines translates fairly well into a network

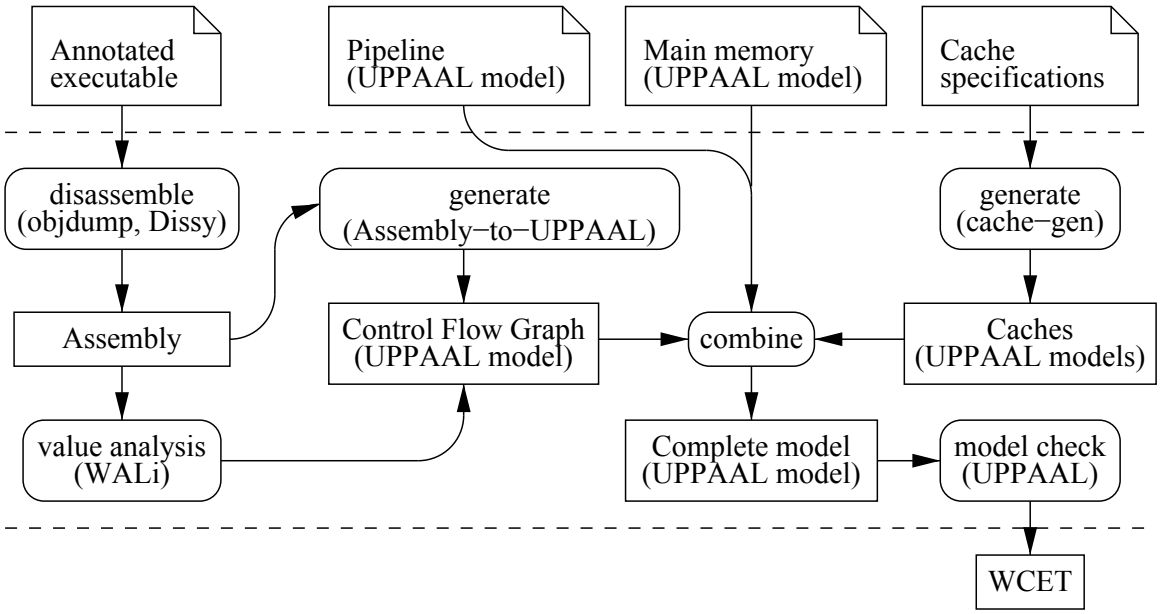


Figure 7.1 – Using METAMOC to compute WCET (from [22])

of automata communicating through channels, making the writing of such automata accessible for the user wanting to add their own in hopes of modeling another architecture. The main challenges then come in determining what can stall the pipeline on the real architecture and how long each stage should take. Indeed, the authors of [22] indicate that the documentation for the processor they modeled explicitly states that it does not contain an exhaustive list of all possible stalls.

The default model for caches considers separate instructions and data caches, matching the architecture they made their approach around. Caches are set-associative, and implement an LRU eviction policy.

Figure 7.2 shows one half of the automaton modeling a cache in METAMOC. The complete automaton has a second half mirroring this one, with *read* instead of write. Upon receiving a request for a cache write, the automaton determines from its internal state whether the request is a cache hit or if the instruction must be added to the cache. This corresponds to cache_contents(instrArd) being equal to -1 (cache miss) or not (cache hit). In the case of a cache hit, a delay corresponding to the time spent writing in the cache is introduced (controlled by the CACHEFETCH clock). write_hit_wait corresponds to the number of main memory accesses to be performed. Such accesses can still occur even after a cache hit, if a write-through policy is in place. In the case of a cache miss, a number of accesses to the main memory are performed. This number can be higher than one if a cache line had to be evicted and the cache follows a write-back policy. Once all accesses are performed, the cache synchronizes on **instructionCacheWrite** to indicate that the request was completed.

Figure 7.3 shows an example fragment of an automaton corresponding to a program after processing by METAMOC. The fragment in question corresponds to a loop, and shows how METAMOC supports branching and iterating. The *loop_counter_1* and *loop_bound_1* variables ensures that its execution terminates. The bounds of such loops have to be annotated in the program's source code.

Their solution to limit search-space explosion is to put a caveat stating that the architectures are assumed to be time-anomaly free, which lets them consider only the locally worst time for some

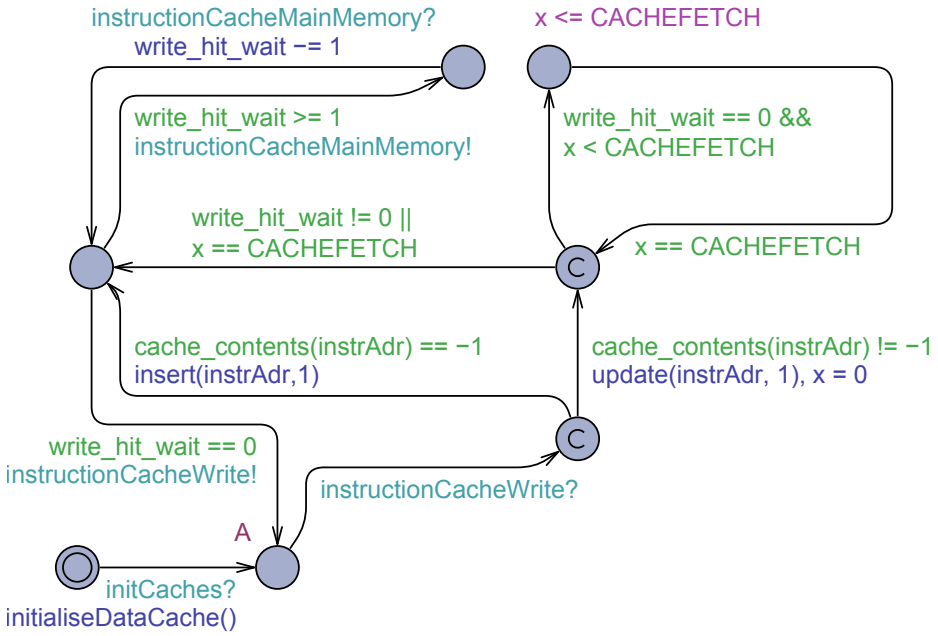


Figure 7.2 – Half of Cache in METAMOC (adapted from [23])

operations, instead of having to explore all possible timings in case a better local time leads to a globally worse one.

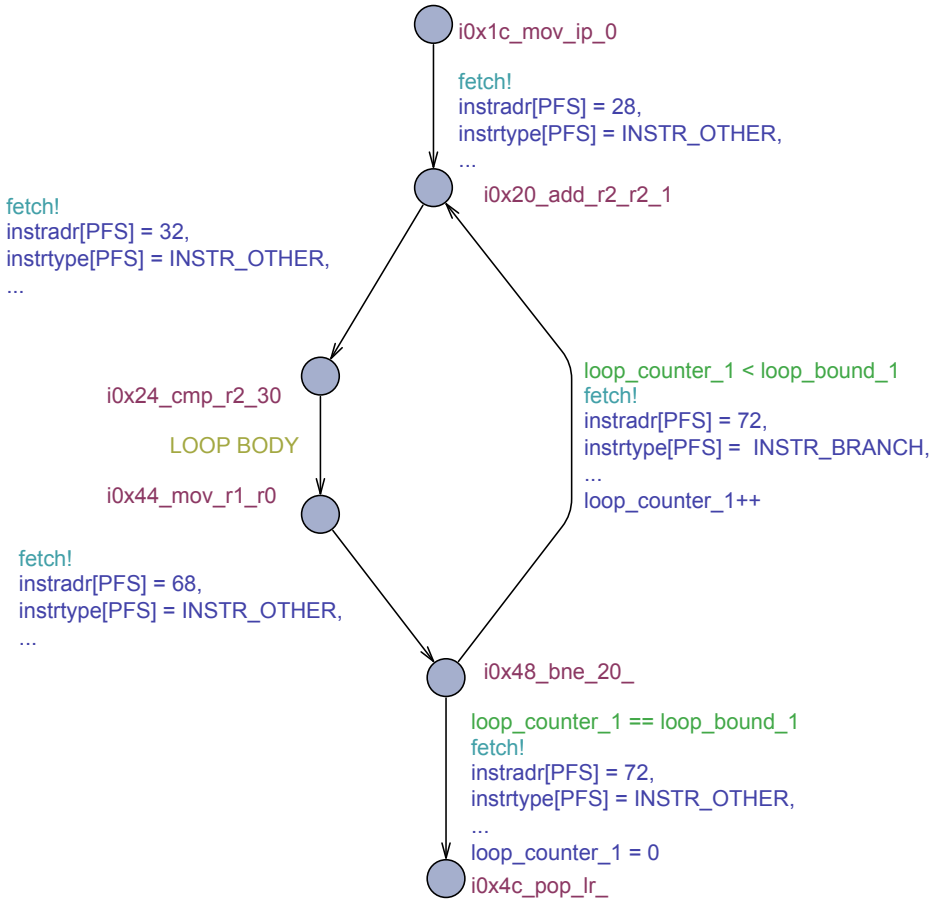


Figure 7.3 – Fragment of Program Automaton in METAMOC (from [23])

7.1.2 WUPPAAL

[16] describes WUPPAAL, another approach to using UPPAAL in order to compute the WCET of a program running on a single-core processor. The main novelty of [16] is that it combines simulated execution with the model checking. Indeed, as can be seen in Figure 7.4, it allows UPPAAL to interact with *qemu* (an architecture simulation tool) through *gdb* (a debugging tool) and *libgdbuppaal* (a library of their own making). The *pre-analysis* step shown in the figure corresponds to the annotation of a binary program with information in order to help the extraction its *Control Flow Graph*. This approach aims at improving the memory usage of model checking, as well as making the approach fit other architectures very easily (by just changing qemu parameters).

These annotations allow the generation of an over-approximation of all valid program runs from the CFG. This has some surprising results, such as deterministic programs having multiple separate runs because the over-approximation does not consider actual values for any computation involving input parameters. Instead, all outcomes are considered, even if some sequences of outcomes cannot follow one another (e.g. the exact same test failing once, then succeeding). For such runs to be finite, loops cannot be controlled by input parameters (i.e. all loops have a known amount of iterations).

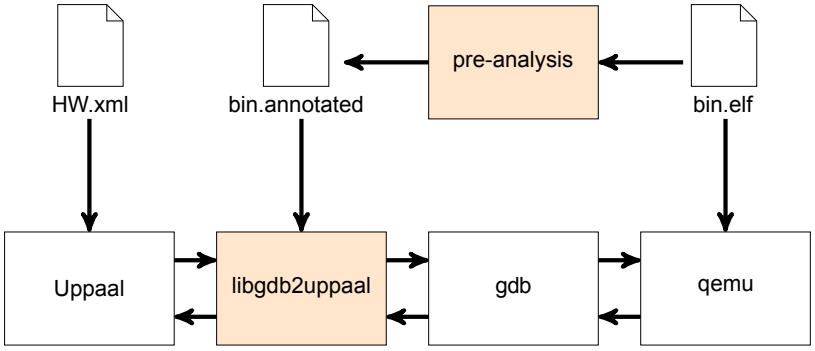


Figure 7.4 – Overview of WUPPAAL's components (taken from [16])

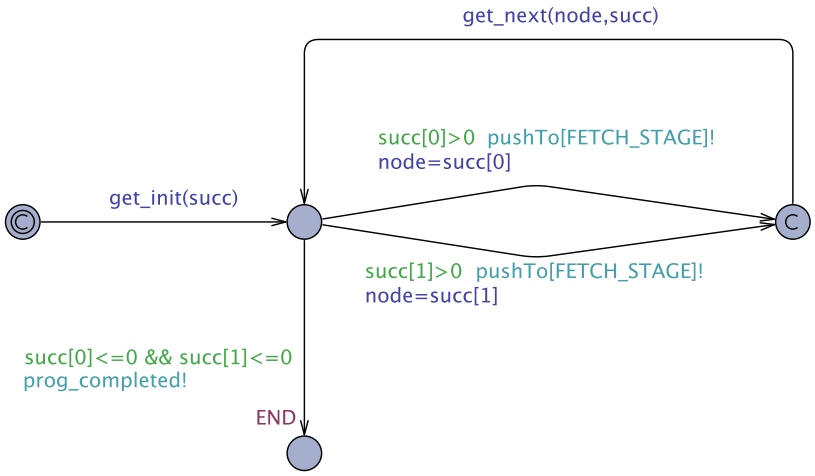


Figure 7.5 – Automaton interacting with *libgdb2uppaal* (taken from [16])

UPPAAL is used to perform queries among all possible execution paths. However, in WUPPAAL, the automaton corresponding to the program is not a purely UPPAAL one. Figure 7.5 shows the automaton in question. It does not store full program states, and instead only keeps an identifier and the annotations for current instruction. The function calls seen on the automaton actually trigger another component from Figure 7.4: *libgdbuppaal*. This component is where program states are stored. *libgdbuppaal* uses *qemu* to compute the program state resulting from the application of the instruction, as well as all possible next execution nodes. The program state inserted back in UPPAAL indicates whether the instruction was found in the cache and how long its execution stage lasts.

UPPAAL is then involved in the computation of the WCET for these annotated traces. Indeed, it models the pipelines, with one automaton per stage, as well as a main memory automaton and an instruction cache one (see Figure 7.6), to model their access times. Since the information about whether the instruction cache contains the instruction or not is already in the annotated execution trace, these last two automata are kept very simple.

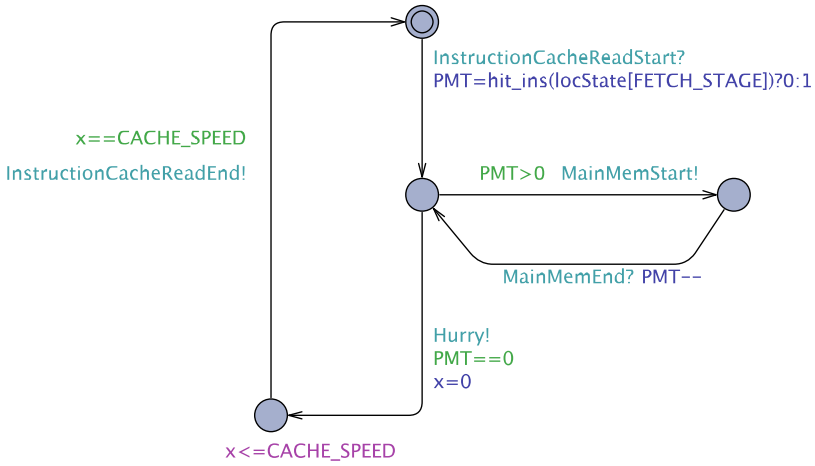


Figure 7.6 – WUPPAAL Instruction Cache Automaton (taken from [16])

Figure 7.6 shows the automaton corresponding to an instruction cache in WUPPAAL. The initial state is the top one. Upon receiving a request for reading on **InstructionCacheReadStart** (considering the rest of the variables, writing likely uses the same channel), the automaton checks if the information is already in the cache. If not it performs as many synchronizations with the main memory automaton as is needed (indicated by *PMT*), then waits a time corresponding to that of a cache access before using **InstructionCacheReadEnd** to signal the completion of the request.

7.2 Multi-Core Processors

7.2.1 Modeling Shared Buses

The approach proposed by [36] combines abstract interpretation and model checking to compute WCET in multicore, with a special focus on the interconnect. The abstract interpretation is found in the analysis of the caches, which is done in a fashion similar to those presented in Section 6.3, including in its remarks of a previous approach being unsafe and in its omission of data caches (and thus, of cache coherence). The overall strategy for this approach is shown in Figure 7.7.

The resulting model considers program instructions only as their categorization (*Always Hits*, *Always Misses*, *First time Misses*, *Not Categorized*). The paper proposes an automaton for each category (see Figure 7.8), indicating how an instruction of this category will use the interconnect. Models for program are thus constructed by replacing each instruction in the control flow graph by the automaton corresponding to its synchronization with the interconnect (*TA Construction* in Figure 7.7). An example of resulting automaton can be seen in Figure 7.9.

To showcase how modular the approach is in its modeling of the interconnect, [36] provides both a model for a TDMA bus and for a FCFS (*First Come, First Served*, which means requests are completed in the order of their arrival) bus.

Figure 7.7 mentions an ILP model being computed as well, however, the paper does not make any mention of it.

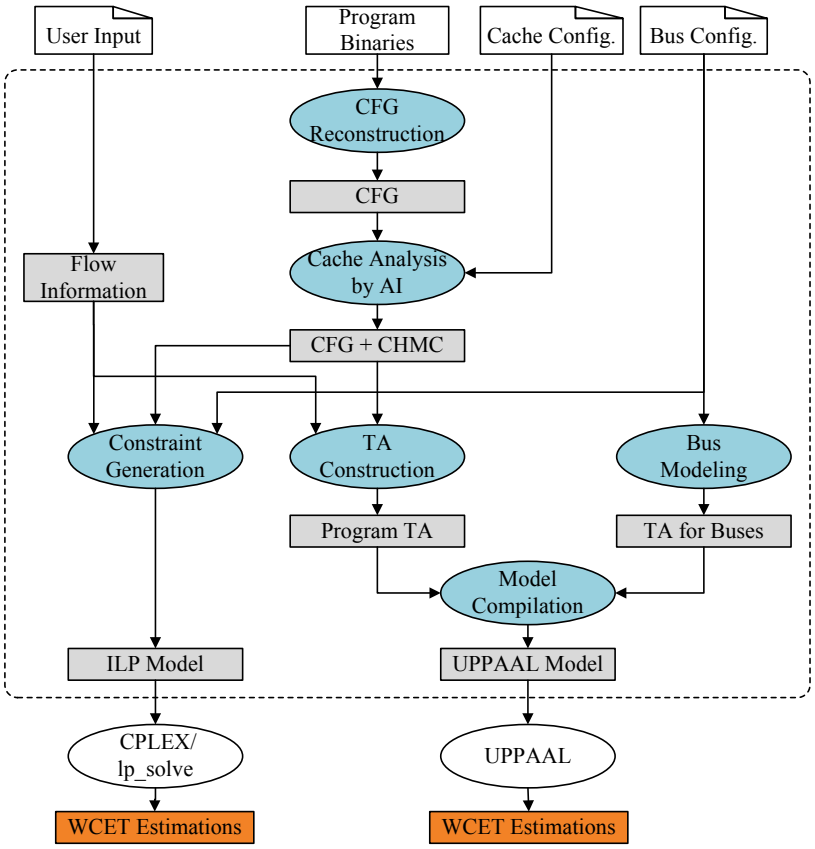
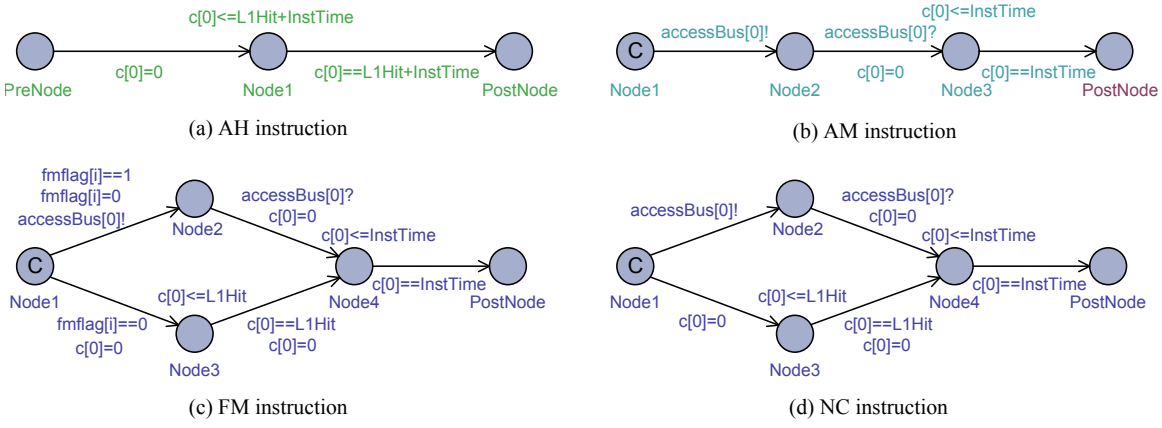


Figure 7.7 – Overall strategy for [36] (taken from the paper)



(a) AH instruction

(b) AM instruction

(c) FM instruction

(d) NC instruction

Figure 7.8 – Program Model Building Blocks (taken from [36])

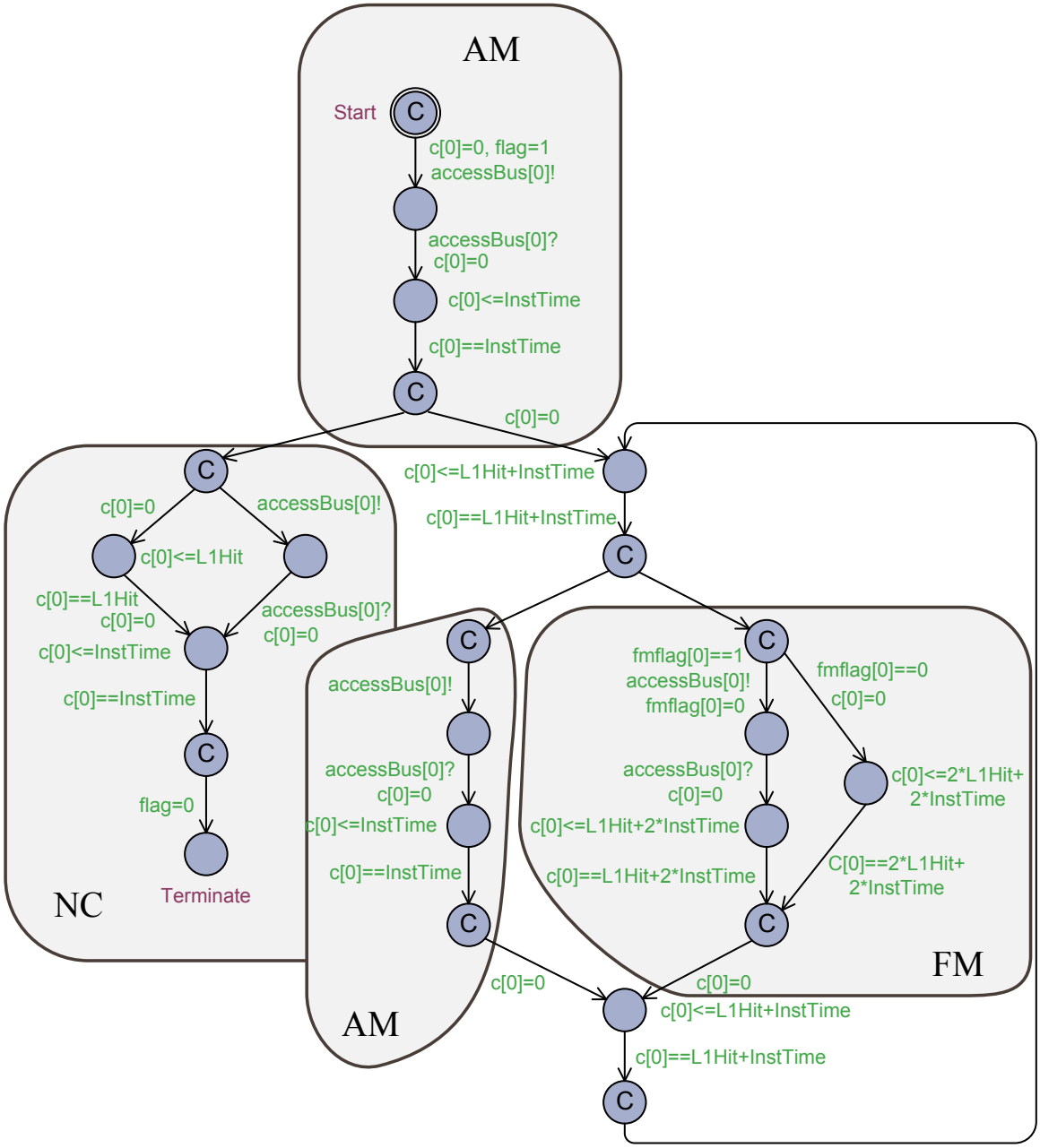


Figure 7.9 – Program Example (adapted from [36])

7.2.2 Multi-Core Analysis using only UPPAAL

The authors [30] present an approach using UPPAAL to perform computation of the worst-case execution time for software running on multi-core processors. While it does not feature cache

coherence, it does support hierarchical (and shared) caches, as well as some sort of instruction pipelining.

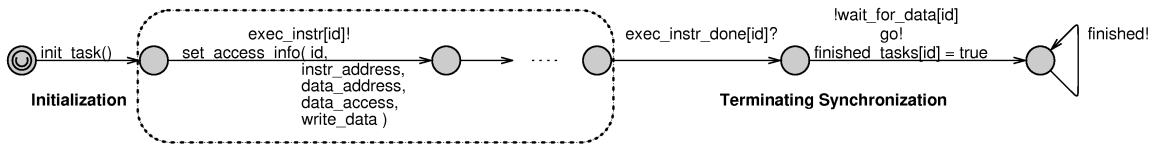


Figure 7.10 – Program Automaton (taken from [30])

In [30], programs are represented by their own automata (See Figure 7.10), sending instructions through shared variables by synchronizing with a core on a dedicated channel. This approach allows programs to feature branching and non-memory-related instructions, by simply adding numerical variables and making the automaton more than a simple sequence of states. In Figure 7.10, the framed part corresponds to where the program's instruction graph should be. The *id* variable targets a particular core on which to execute the instruction, and *set_access_info* populates the global variables characterizing the instruction: *instr_address* for the address of the instruction in the modeled memory, *data_address* for the address of any data being accessed, *data_access* to indicate if this instruction accesses data, and *write_data* to differentiate between a read and a write access. The *Terminating Synchronization* part is here to ensure the task is not considered completed until it has indeed completed all accesses for its final instruction (and not just sent the instruction).

Cores are modeled as automata that go through the pipeline required for an instruction to be completed, which includes accessing the instruction cache as well as, potentially, the data cache. Interestingly enough, this model does not use one automaton per stage of the pipeline. Instead the whole core logic is modeled in a single automaton.

Each L1 instruction cache has its own automaton, managing the possibility for instructions to have already been cached or sending a request for that instruction to the L2 cache. The L1 data caches are similar (see Figure 7.11), with the added possibility of writing data to a memory element, which invalidates it from every other L1 data cache.

The automaton for the L2 cache is very straightforward, a simple loop going over each request, determining whether the data is supposed to be in the L2 cache or not, and delaying the reply accordingly.

Unsurprisingly, the hampering factor is scalability when the number of modeled cores is increased. There appear to have very little in the way of synchronicity between the cores (such as using a round-robin for L2 cache accesses), which is likely to be aggravating the issue.

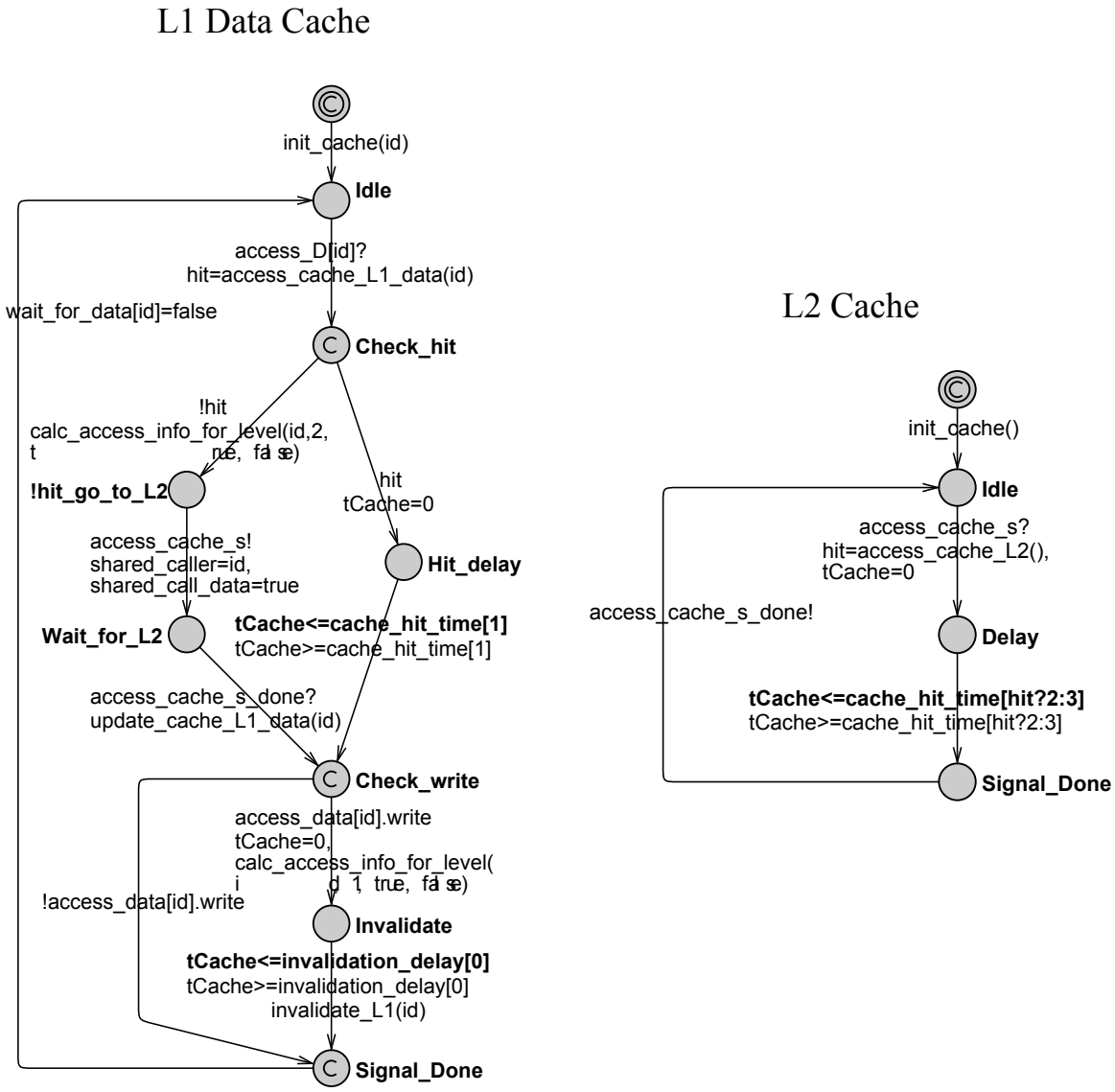


Figure 7.11 – Cache Automata (taken from [30])

7.3 Conclusion

The use of timed automata to analyze the real-time properties of an architecture has already been done in a few publications, both for single-core and for multi-core processors. The two main advantages of these approaches is the certainty that all scenarios are taken into account, and the modularity of the models, which facilitates their adaptation to most architectures.

However, these approaches have a tendency to suffer from combinatorial explosion, which limits the size of the models they can be used on. In addition, none of the existing literature exploits timed

automata to model the effects of cache coherence on the system. Instead, as in the analyses of the previous chapter, cache coherence is assumed to be disabled.

This last point is remediated in this thesis, by the proposition of a multi-core architecture model with a focus on cache coherence. For the model to be accurate however, the details of what is modeled need to be known. This information is obtained through benchmarks (Chapter 5) and a proper identification of the cache coherence protocol, which is the subject of the next chapter and first contribution of this thesis.

Part III

Contributions

Chapter 8

Identifying Cache Coherence

The first step toward the analysis and the control of interference caused by the use of cache coherence is to ensure that the mechanisms of cache coherence used on the targeted architecture are well understood.

Chapter 5 presented papers on benchmark-based strategies to expand the information available on the architecture beyond what the documentation provides. That chapter was limited to performance (namely, execution time and bandwidth) analysis. In this chapter, benchmarks are used to clarify and expand the information available to the user about the architecture' cache coherence protocol.

This chapter includes the results of applying this strategy to the NXP QorIQ T4240 architecture. This architecture is documented as implementing a MESI cache coherence protocol, and as able to perform *cache intervention*. The application of the cache coherence protocol identification strategy on this architecture reveals a MESIF protocol. The work presented in this chapter has been published in [48].

The identification strategy uses benchmarks to identify a cache coherence protocol by comparing them with the protocol the user believes it to be. In order to so, it is important to distinguish the following three notions:

Definition 37 (The Architecture's Coherence Protocol) *The actual coherence protocol implemented by the architecture, which is the one the applicant needs to identify. It may not be directly observable.*

Definition 38 (The Observed Coherence Protocol) *The* ***observed*** *cache coherence protocol is the partial view of the architecture's coherence protocol that is observed by performing a given set of benchmarks. As these benchmarks cannot be exhaustive, the observed coherence protocol is potentially incomplete.*

Definition 39 (The Hypothetical Coherence Protocol) *The user believes the architecture to be implementing a certain protocol. This corresponds to the* ***hypothetical*** *cache coherence protocol. It originally comes from the user's understanding of the architecture's documentation.*

The purpose of the identification strategy is to show that the hypothetical protocol is indeed the architecture's cache coherence protocol. This verification is done by resolving the observed cache coherence protocol and analyzing it through comparison with the hypothetical protocol.

The chapter starts by a presentation of the strategy itself, in an architecture-agnostic manner (Section 8.1), then follows up with its application to the NXP QorIQ T4240 architecture: Section 8.2

presents how the benchmarks were implemented, Section 8.3 presents a hypothetical MESI protocol implementation, Section 8.3.1 shows the result of applying the strategy until the point where the protocol is found to be a mismatch, Section 8.4 presents a hypothetical MESIF protocol implementation, and Section 8.4.1 shows the results pointing out the slight discrepancies in implementation choices.

8.1 Identification Strategy

This section presents the strategy used to identify the cache coherence protocol of an architecture. As a reminder from Chapter 3, cache coherence protocols are defined around a single memory element. Thus, the identification strategy only considers a single memory element in its process. Moreover, the following hypotheses are made:

- The architecture encodes stable cache states using binary flags attached to each cache line (see Definition 46).
- The user is able to observe cache line attributes (at least the aforementioned binary flags and the associated memory element address).

In effect, this identification strategy tests whether, as far as the user's observation capabilities allow, the architecture's cache coherence protocol implements all (and only) the behaviors of the hypothetical cache coherence protocol. The identification strategy is split in several steps, described hereafter.

8.1.1 Defining the Hypothetical Cache Coherence Protocol

The first step is to define the hypothetical cache coherence protocol using the notation presented in Chapter 3. A good starting point is to use the information available in the architecture's documentation in order to know which cache coherence protocol to define.

Taking for example the NXP QorIQ T4240 architecture, the user would consult the architecture's documentation [26]. This document does not indicate the cache coherence protocol in use on the architecture. However, it does mention *cache intervention*: queries which generate a cache hit on another cache can be made to provide the data reply. Reading on the core's documentation [25], the L2 cache coherency model is indicated to be the MESI protocol. As a result, in this case, the hypothetical cache coherence protocol would be a MESI protocol.

8.1.2 Defining the Observable Cache Coherence Protocol

The observable cache coherence protocol is the amalgam of the results from each performed benchmark. This subsection defines each relation involved in the observable protocol, and points out their equivalent from Chapter 3, when applicable. Indeed, the definitions from Chapter 3 cannot be re-used as-is for the observed cache coherence protocol, because the observations are done on binary flags and performance monitors, whereas the notations in Chapter 3 are more abstract.

Definition 40 (Observable Coherence State) *On a real architecture, each line of a cache or coherence manager has k binary flags providing information on its state. We can then define $\mathtt{C}_{\mathbb{B}} : \mathtt{Ccs} \to \mathtt{Addr} \to \mathbb{B}^k$, which indicates the valuation of these binary flags for a given memory element (see Definition 17) in a given cache (see Definition 20), and $\mathtt{M}_{\mathbb{B}} : \mathtt{Addr} \to \mathbb{B}^k$ the coherence manager equivalent.*

Example 19 (Observable Coherence State) *In an architecture with two caches (CC_1 and CC_2), an observable coherence manager, $k = 3$, and 42 being the address of a memory element, observations may reveal $\mathtt{C}_{\mathbb{B}}(CC_1, 42) = \langle \mathtt{true}, \mathtt{true}, \mathtt{false} \rangle$ at some point.*

If the caches and the coherence manager do not use the same number of binary flags to encode states, k is considered to be the maximum of the two, with the extra flags being set to `false`.

Definition 41 (Observable System State) *An analogue to $\mathtt{System}$ (see Definition 29) can be made for the observed cache coherence. Given an arbitrary memory element E and $\mathtt{cc}$ being the number of caches in the system, $\mathtt{System}_b$ is the set of all $\langle CC_1, \ldots, CC_{\mathtt{cc}}, CM \rangle$ such that: $\mathtt{M}_{\mathbb{B}}(E) = CM \wedge \forall c \in 1..\mathtt{cc},\ \mathtt{C}_{\mathbb{B}}(c, E) = CC_c$.*

Example 20 (Observable System State) *In the system of Example 19, an example of plausible observable system state would be:*
$\langle \langle \mathtt{true}, \mathtt{true}, \mathtt{false} \rangle, \langle \mathtt{false}, \mathtt{false}, \mathtt{false} \rangle, \langle \mathtt{true}, \mathtt{false}, \mathtt{false} \rangle \rangle$

Definition 42 (Observable System Transitions) *$\mathtt{reach}_b$ is the analog to $\mathtt{reach}$ (see Definition 30) and is declared as $\mathtt{reach}_b : \mathtt{System}_b \to \mathtt{Instr}^{\mathtt{cc}} \to set(\mathtt{System}_b)$.*

Example 21 (Observable System Transitions) *In the system of Example 19, an example of plausible observation transition would be:*
$\mathtt{reach}_b(\langle \langle \mathtt{false}, \mathtt{false}, \mathtt{false} \rangle, \langle \mathtt{true}, \mathtt{false}, \mathtt{false} \rangle, \langle \mathtt{false}, \mathtt{false}, \mathtt{false} \rangle \rangle, \langle \mathtt{load}, \mathtt{evict} \rangle) =$
$\{ \langle \langle \mathtt{true}, \mathtt{false}, \mathtt{false} \rangle, \langle \mathtt{false}, \mathtt{false}, \mathtt{false} \rangle, \langle \mathtt{false}, \mathtt{false}, \mathtt{false} \rangle \rangle \}$
In this case, all benchmarks performing a $\mathtt{load}$ on the first cache and an $\mathtt{evict}$ on the second, when the system was in the given state, only yielded a single resulting system state.

Definition 43 (Monitorable Activity) *The architecture's documentation lists activities that can be monitored through performance monitors. The activities that can be monitored solely depend on the architecture, with some architectures not capable of monitoring any activities (see Section 5.3). The meaning behind each monitored activity is assumed to be understood by the user. The activity observed on an architecture does not strictly correspond to what is defined as either an event (see Definition 27) or an action (see Definition 28) in Chapter 3, it might include elements from both, but is usually something that refers to the outcome of actions.*

Example 22 (Monitorable Activity) *"L1 Cache Miss" and "External Query" are two examples of activities that may be monitorable on an architecture.*

Definition 44 (Performance Monitors) *The observation of the architecture's activity is done through* performance monitors*. A performance monitor holds a number that counts the occurrences of a certain monitorable activity. Each core is assumed to have its own monitors. With $\mathtt{Perf}^{\mathtt{Mon}}$ the set of monitors available on every core, $\mathtt{Act}^{\mathtt{Mon}} : \mathtt{System}_b \to \mathtt{Instr}^{\mathtt{cc}} \to \mathtt{Perf}^{\mathtt{Mon}} \to \mathbb{N}^{\mathtt{cc}}$ indicates, for each core, the number of occurrences of each monitorable activity when performing the given instructions from a given system state.*

Example 23 (Performance Monitor Event) *Still considering the system from Example 19, the following is a plausible example of performance monitors valuation:*
$\mathtt{Act}^{\mathtt{Mon}}(\langle \langle \mathtt{true}, \mathtt{true}, \mathtt{false} \rangle, \langle \mathtt{false}, \mathtt{false}, \mathtt{false} \rangle, \langle \mathtt{false}, \mathtt{true}, \mathtt{false} \rangle \rangle, \langle \mathtt{store}, \mathtt{nop} \rangle,$
L2 Cache Hits$) = \langle 1, 0 \rangle$
In this case, the benchmark indicates that having the first core perform a $\mathtt{store}$ leads to a L2 Cache Hit in its cache (confirming that the value is there). The other core, which performed nothing, observes no L2 Cache Hits activity.

8.1.3 Naive Exploration of the Observable Protocol

We have defined how the observable protocol will be described, we now need to construct the partial view of the architecture protocol. To do so, the first benchmark steps perform a state exploration on the architecture by executing a single instruction at a time.

The general algorithm for these steps can be seen in Figure 8.1. Starting from the initial situation where all cache controllers consider the memory element to be invalid (*init*), it explores observable system states (see Definition 41) by applying a single instruction on one of the cache and recording both the resulting observable system states and a count of the monitorable activities on each cache (see Definition 44).

The steps being performed at each iteration of this naive exploration correspond to the functions state_search (Step 1), decode (Step 2), and monitors (Step 3). In order to facilitate readability, these functions are defined in their respective sub-section following this one.

```
init_state_search()
init_decode()

DstStates ← {init}
WaitList ← {init}
while (WaitList ≠ ∅):
    SrcState ∈WaitList;
    WaitList ←WaitList \ SrcState;
    foreach k ∈ 1..cc
        foreach instr ∈ {load, store, evict}
            SysInstruction ← single_instruction_on(instr, k)
            ⟨DstState, PerformanceCounters⟩ ← benchmark(SrcState, SysInstruction)

            handle_state_search(SrcState, SysInstruction, DstState) // Step 1
            handle_decode(SrcState, SysInstruction, DstState) // Step 2
            handle_monitors(SrcState, SysInstruction, PerformanceCounters) // Step 3

            if DstState ∉ DstStates
                DstStates ← DstStates ∪ {DstState}
                WaitList ← WaitList ∪ {DstState}
```

Figure 8.1 – General State Exploration Algorithm

Definition 45 (The Benchmark Function) *The benchmark function,* $\texttt{benchmark} : \texttt{System}_b \rightarrow \texttt{Instr}^{cc} \rightarrow (\texttt{System}_b \times (\texttt{Perf}^{Mon} \rightarrow \mathbb{N}^{cc}))$*, corresponds to a benchmark being performed on the architecture and returns a pair containing the resulting observable stable system state, as well as a valuation for each of the performance monitors.*

Example 24 (The Benchmark Function) *An example of result for the* `benchmark` *function could be:*
$\texttt{benchmark}(\langle\langle \texttt{true}, \texttt{true}, \texttt{false}\rangle, \langle \texttt{false}, \texttt{false}, \texttt{false}\rangle, \langle \texttt{false}, \texttt{true}, \texttt{false}\rangle\rangle, \langle \texttt{store}, \texttt{store}\rangle) =$
$\langle\langle\langle \texttt{false}, \texttt{false}, \texttt{false}\rangle, \langle \texttt{true}, \texttt{true}, \texttt{false}\rangle, \langle \texttt{false}, \texttt{true}, \texttt{false}\rangle\rangle,$
$\langle\langle \textit{L2 Cache Hits}, \langle 1, 0\rangle\rangle, \langle \textit{L2 Pushes}, \langle 1, 0\rangle\rangle, \langle \textit{L2 Reloads}, \langle 0, 1\rangle\rangle\rangle$ *This would indicate that performing a* `store` *instruction on both cores when the system is in the*
$\langle\langle \texttt{true}, \texttt{true}, \texttt{false}\rangle, \langle \texttt{false}, \texttt{false}, \texttt{false}\rangle, \langle \texttt{false}, \texttt{true}, \texttt{false}\rangle\rangle$ *state results in the system ending up in the*
$\langle\langle \texttt{false}, \texttt{false}, \texttt{false}\rangle, \langle \texttt{true}, \texttt{true}, \texttt{false}\rangle, \langle \texttt{false}, \texttt{true}, \texttt{false}\rangle\rangle$ *state, that this also results in the*

first core to observe one L2 cache hit and one L2 Push, whereas the other core observes one L2 reload instead.

While the hypotheses made ensure that cache lines are observable, the coherence manager might not be. In such cases, all valid coherence manager states must be considered, which can result in multiple reachable system states. When dealing with this special case, the sequence of instructions that led to reaching SrcState is used to infer the expected state of the coherence manager when calling `benchmark`. This resolves the issue, as it again ensures a single possible reached system state.

8.1.4 State Exploration & Reachability

Step 1 catalogs the observable coherence states (see Definition 46), as well as the valid system coherence states (see Definition 41).

Definition 46 (Valid Observable Coherence State) *It is likely not all combinations of valuations for $\mathbb{B}^k$ are valid states (i.e. some combinations may not correspond to any state and are thus never used, making them invalid). $V_s \subseteq \mathbb{B}^k$ denotes the set of all valid states. This makes V_s the codomain of both $C_{\mathbb{B}}$ and $M_{\mathbb{B}}$.*

Step 1 (Reachability) *To compute V_s, the algorithm shown in Figure 8.1 records all observed system and coherence states. $reach_b$ is built according to the transitions observed during the state exploration. This is handled by the init_state_search and handle_state_search procedures, defined as follows:*

```
def init_state_search ()
    V_s ← tuple_to_set(init)
    System_b ← {init}

def handle_state_search (SrcState, SysInstruction, DstState)
    reach_b(SrcState, SysInstruction) ← {DstState}
    if DstState ∉ System_b
        V_s ← V_s ∪ {tuple_to_set(DstState)}
        System_b ← System_b ∪ {DstState}
```

8.1.5 Matching Observed States with Hypothetical States

The states and transitions observed in Step 1 are then compared with those expected to be observed on an architecture implementing the hypothetical protocol. Step 2 binds the observed coherence states with the stable states from the hypothetical protocol, thus creating `decode` (see Definition 47).

Definition 47 (Decode Relation) *With $S_c{}^s \cup S_m$ being the set of all hypothetical stable coherence states (see Definitions 21 and 26 from Chapter 3), a relation to link V_s and hypothetical stable coherence states can be defined as: $decode \subseteq V_s \times (S_c{}^s \cup S_m)$. This relation matches elements of V_s to their corresponding element in $S_c{}^s \cup S_m$. This can, in turn, be used to link C_s (see Definition 24) and M_s (see Definition 26) to $C_{\mathbb{B}}$ and $M_{\mathbb{B}}$, respectively.*

Example 25 (Decode Relation) *In a system tested for an MSI protocol, with $k = 3$ the following is a possible decode relation:*
$decode = \{\langle\langle false, false, false\rangle, I\rangle, \langle\langle true, false, false\rangle, S\rangle, \langle\langle true, true, false\rangle, S\rangle,$
$\langle\langle true, true, true\rangle, M\rangle, \langle\langle true, false, true\rangle, M\rangle\}$

Definition 48 (Injective Relation) *A relation $R \subseteq A \times B$ is said to be injective iff:*
$\forall x \in A,\ \forall y \in B,\ \forall z \in A,\ (\langle x, y\rangle \in R \wedge \langle z, y\rangle \in R) \implies (x = z)$

Part of the flags in $\mathtt{V_s}$ may be unrelated to the coherency state of the memory element. As a result, multiple elements of $\mathtt{V_s}$ distinguished only by these flags unrelated to cache coherence can be associated with the same hypothetical coherence state. Thus, `decode` is not necessarily an injective relation (see Definition 48), and whether it is nor not does not invalidate the hypothetical protocol.

Step 2 (Observed Coherence State Decoding) *The matching between observed and hypothetical states is done during the state exploration algorithm described in Figure 8.1, by constructing* `decode` *according to what the hypothetical protocol indicates should be the system state upon application of the given instruction on the already decoded initial system state. The system starts in a state where no cache holds the memory element, nor does the coherence manager. Thus, the init state can already be decoded. This step is performed by the* init_decode *and* handle_decode *procedures, defined below:*

```
def init_decode ()
    decode ← ⟨init, < I, ..., I >⟩

def handle_decode (SrcState, SysInstruction, DstState)
    ⟨SrcState, DecodedSrcState⟩ ∈ decode
    {DecodedDstState} ← reach(DecodedSrcState, SysInstruction)
    decode ← decode ∪ {⟨DstState, DecodedDstState⟩}
```

Definition 49 (Surjective Relation) *A relation $R \subseteq A \times B$ is said to be surjective iff:*
$\forall b \in B,\ \exists a \in A \text{ s.t. } (\langle a, b\rangle \in R)$

Property 4 (Decode must be surjective) *In any successful match,* `decode` *must be surjective (see Definition 49).*

In a successful match, the hypothetical protocol cannot feature stable coherence states that are not found on the architecture. In other words, `decode` has to be surjective (Property 4).

Definition 50 (Functional relation) *A relation $R \subseteq A \times B$ is said to be functional iff:*
$\forall x \in B,\ \forall y \in A,\ \forall z \in B,\ (\langle x, y\rangle \in R \wedge \langle x, z\rangle \in R) \implies (y = z)$

Property 5 (Decode is functional) *In any successful match, the* `decode` *relation also needs to be functional (see Definition 50).*

Furthermore, for the identification to be successful, the `decode` relation also needs to be functional (Property 5). Indeed, a same element of $\mathtt{V_s}$ pointing to more than one element of $\mathtt{S_m} \cup \mathtt{S_c}$ is indicative of a transition not leading to the same destination stable coherence state in the observed protocol compared to the hypothetical protocol. In other words, it would prove there is a mismatch.

Property 6 (Reachability simulation) *In any successful match,* `reach` *must simulate* $\mathtt{reach}_b$ *through* `decode`*.*
$\forall i \in \mathtt{Instr}^{cc},\ \forall s \in \mathtt{System}_b,\ \forall d \in \mathtt{System}_b,\ \forall s' \in \mathtt{System},\ (d \in \mathtt{reach}_b(s, i)) \wedge (\langle s, s'\rangle \in \mathtt{decode}) \implies \exists d' \in \mathtt{System} \text{ s.t. } ((\langle d, d'\rangle \in \mathtt{decode}) \wedge (d' \in \mathtt{reach}(s', i)))$

At this point, the identification strategy may be able to detect a mismatch between the two protocols:

- If `decode` does not bind any observed stable cache state to one of the hypothetical stable states, despite the hypothetical protocol allowing this state to have been reached by the algorithm of Figure 8.1 given the used means of observation. This corresponds to a violation of either Property 4 or Property 6.
- If `decode` binds the same observed stable cache state to multiple hypothetical stable states, a violation of Property 5. This is because of the hypotheses of cache states being fully encoded through observable binary flags, and of not having redundant hypothetical stable states. Indeed, this observed stable cache state would need to behave as all the hypothetical states it matches, despite them being assured to each behave differently in some way.

If Properties 4, 5, and 6 are verified, no mismatch has been detected so far. However, bi-simulation between `reach` and $\texttt{reach}_{\text{b}}$ through `decode` cannot be ensured at this point, as many of the transitions of `reach` have not been explored due to the restriction to a single instruction per benchmark.

8.1.6 Activity Comparison

To confirm the matching between observed and hypothetical coherence states, the activities detected for each of the observed stable states have to be compared to the ones expected from the hypothetical cache coherence protocol, according to the available performance monitors. Once the user has determined the expected results (see Definition 51), Step 3 compares them to those observed on the architecture.

Definition 51 (Performance Monitor Oracles) $\mathit{Act_{Hyp}}^{Mon} : \mathit{System} \rightarrow \mathit{Instr}^{cc} \rightarrow \mathit{Perf}^{Mon} \rightarrow \mathbb{N}^{cc}$ *is the analogue of* Act^{Mon} *applied to the hypothetical cache coherence protocol.* $\mathit{Act_{Hyp}}^{Mon}$ *thus serves as an oracle, indicating what* Act^{Mon} *is expected to yield for the cache coherence protocols to match.*

Example 26 (Performance Monitor Oracles) *Testing for the MSI protocol defined in Chapter 3 on a system with two caches and a coherence manager, an example of performance monitor oracle would be:*
$\mathit{Act_{Hyp}}^{Mon}(\langle \mathit{I}, \mathit{I}, \mathit{I} \rangle, \langle \mathtt{load}, \mathtt{nop} \rangle, \textit{External Queries}) = \langle 0, 1 \rangle$ *This would correspond to the second core observing one external query (and the first core observing none) when performing a* `load` *on the first cache when the system is in the* $\langle \mathit{I}, \mathit{I}, \mathit{I} \rangle$ *state.*

Step 3 (Activity Matching) *To compute* Act^{Mon}*, the exploration algorithm queries the performance monitors after each transition, as they hold the number of occurrences of each monitored activity. This step is performed by the* handle_monitors *procedure, defined below:*

```
def handle_monitors(SrcState, SysInstruction, PerformanceCounters)
    Act^Mon(SrcState, SysInstruction) ← PerformanceCounters
```

Property 7 (Activity Simulation) *In any successful match, the observed performance monitor values correspond to what the hypothetical cache coherence protocol would generate:*
$\forall o \in \mathit{System}_b,\ \forall act \in \mathit{Instr}^{cc},\ \forall mon \in \mathit{Perf}^{Mon},\ \forall o' \in \mathit{System},\ \langle o, o' \rangle \in \mathtt{decode} \implies$
$\mathit{Act}^{Mon}(o, act, mon) = \mathit{Act_{Hyp}}^{Mon}(o', act, mon).$

The results of Step 3 should verify Property 7. For any monitor contradicting this property, the user must either find the reason behind the mismatch, or consider the hypothetical cache coherence protocol as disproved.

As `decode` is not required to be an injective relation, it is possible for Property 7 to be verified by some observed stable states, despite other observed stable states bound to the same hypothetical state state violating the property. This is still sufficient to disprove the hypothetical protocol. Indeed, in such cases, if the monitored activity is relevant to cache coherence, the violating observed stable states actually correspond to a different stable state than the one they are bound to. This other stable state might even not be any of the one currently found in the hypothetical protocol.

By the end of Step 3, if the hypothetical protocol has not been disproved, the hypothetical protocol replicates all observed coherence behaviors. This completes the first naive exploration of the architecture's cache coherence mechanisms. To confirm that the architecture implements the hypothetical protocol's behaviors, the next step will perform a follow-up exploration, this time guided by the hypothetical coherence protocol.

8.1.7 Exploration Guided by Hypothetical Protocol

Definition 52 (Stable State Change Path) *A stable state change path is a path between two stable states of the cache controller protocol definition table. It is formed of a stable state followed by a cycle-free sequence of transient states terminated by a stable state, with an event (see Definition 27) separating every state. The two stables states in a path may in fact be the same one. Transitions without any action (cells noted – in the tables) are not allowed within a path.*

Example 27 (Stable State Change Path) *An example of stable state change path for the MSI protocol defined in Chapter 3 is:* $M \xrightarrow{evict} MI^B \xrightarrow{GetM} II^B \xrightarrow{Qry_{own}} I$

This next step of the identification process verifies whether the architecture's cache coherence protocol replicates all of the hypothetical protocol's behaviors. It relies on an exhaustive list of hypothetical stable state change paths (see Definition 52), which can be generated by the tool described in Section 9.9. The general idea is simple: reproduce each of these paths on the architecture and compare the observations with what the hypothetical protocol indicated should have happened. On the other hand, implementation of this step is more difficult: the user has to perform benchmarks that will reproduce a sequence of events in the right order. Unlike in the naive exploration:

- Multiple instructions may be used simultaneously in order to generate the desired events. Furthermore, each benchmark may involve sequences of instructions, instead of just applying a single instruction and observing the results.
- The analysis is focused on a single cache, not the whole system. Instead, the other caches are used to generate the appropriate events.
- The exploration is not blind: the set of benchmarks to perform comes from the hypothetical protocol. The main difficulty lies in obtaining the correct sequence of events when implementing the benchmark.

Step 4 (Complex Behaviors Validation) *For each stable state change path, implement and perform a benchmark. Record the resulting system state, and performance monitors. Compare the results with what the hypothetical protocol indicates should be found.*

Once this exploration is completed, if all the hypothetical behaviors have successfully been replicated on the architecture, then the architecture's cache coherence protocol is guaranteed to implement all of the hypothetical cache coherence mechanisms. Combined with the results from the naive

exploration showing that all the observed behaviors are implemented by the hypothetical coherence protocol, this strategy ensures the user has a good understanding of the coherence protocol implemented by the architecture.

One possibility is for the benchmarks performed in this guided exploration to have revealed new observable states. If such is the case, the fact that these states were not found previously in no way prove that they will invalidate the hypothetical coherence protocol. Neither can they simply be assumed to be exactly what the hypothetical protocol expects them to be. Indeed, the behavior of such new observed states must still be compared to what the hypothetical protocol expect. In effect, the steps of the naive exploration have to be applied to these newly discovered observable states in order to validate that they match the hypothetical states the guided exploration would bind them to. If no mismatch occurs at that point, the identification process is completed.

The next section applies this strategy to the NXP QorIQ T4240 architecture, where performing the naive exploration reveals a mismatch between hypothetical and observed protocol.

8.2 Benchmark Implementation

This section explains how the strategy presented in Section 8.1 has been implemented on the NXP QorIQ T4240 architecture.

8.2.1 The NXP QorIQ T4240

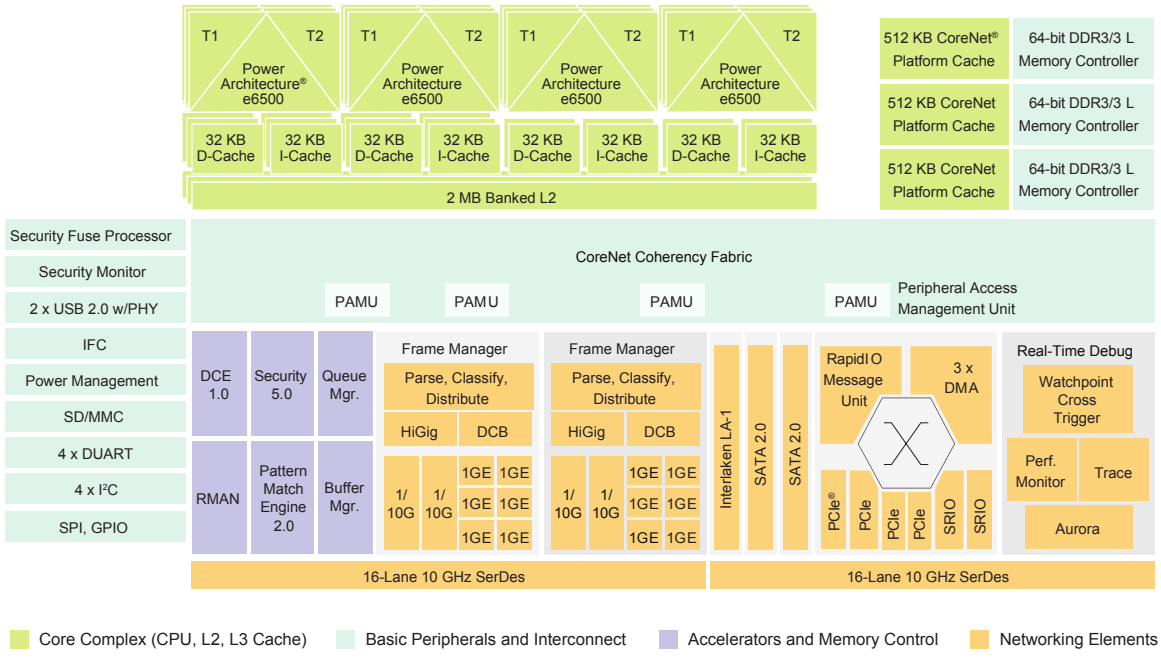


Figure 8.2 – Architecture Block Diagram (Figure taken from [26])

Figure 8.2 gives an overview of the NXP QorIQ T4240 architecture, a PowerPC featuring twelve e6500 cores, each of which is capable of running two simultaneous threads. The cores are equally

distributed among three clusters, with one shared 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, [25], these clusters implement the MESI cache coherence protocol. The architecture's documentation, [26] indicates that the caches are able to perform *cache intervention* (caches may provide a data reply if they hold the relevant memory element).

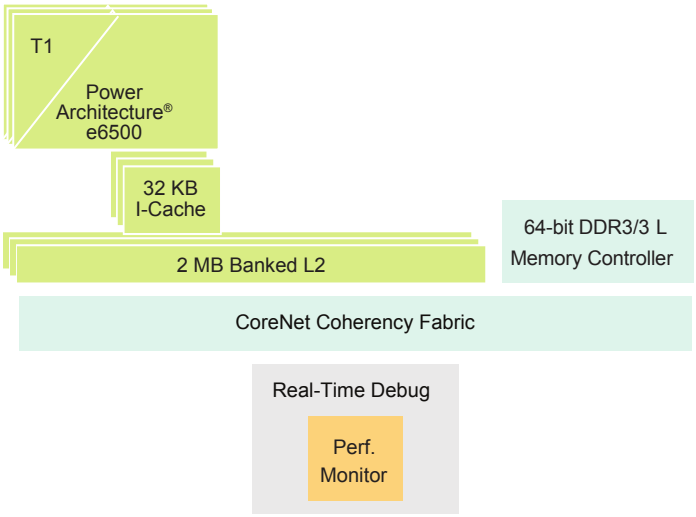


Figure 8.3 – Components used in the Identification Process

In order to limit the mechanisms observed to the L2 cache coherence, the architecture's L1 Data caches were deactivated during this process. Furthermore, only a single core (and execution thread) per cluster (and thus, per L2 cache) was considered. In an attempt at reducing the impact of instruction fetching, the L1 Instruction caches stayed enabled. Lastly, the system only used a single memory controller. Thus, the resulting configuration resembled the one shown in Figure 8.3, the remaining hardware configuration having been left to what it is by default.

Limitation 1 (No Observation Available from the Coherence Manager) *The coherence manager, if one is present, cannot be observed directly through the available means.*

8.2.2 Naught

To perform the benchmarks, a small bare-metal library was created: *Naught*[1]. It simplifies access to the architecture's performance monitors by providing C functions equivalent to those used in the algorithm of Figure 5.8a. Naught offers a very crude implementation of barriers, making it possible to synchronize the code running on each core. It also has a similarly crude implementation of semaphores, ensuring that any output made by the software does not end up garbled.

After some testing, the architecture's monitors were found to be slightly imprecise: even with all monitors of a core being set to track the same activities, they would yield slightly different results (staying generally within a difference of 10 recorded occurrences). In order to address this issue, and

[1] `https://github.com/nsensfel/naught`

to reduce the impact of the extra activities performed by Naught's code logic (loop iterator increase, for example), the benchmarks are applied over a set of 8000 memory elements.

All cores access these same 8000 memory elements. Each memory element has a size corresponding to that of a cache line, and is properly aligned so as to be fully stored within one (i.e. no false sharing is occurring, see Appendix A).

```
1  SrcStates = [STATE_A, STATE_B, STATE_C];
2  reach_states(core_id, SrcStates);
3  start_monitors();
4  SysInstruction = [INSTR_A, INSTR_B, INSTR_C];
5  perform(instructions[core_id]);
6  join(barrier);
7  store_monitor_results();
```

Figure 8.4 – Implementation of the `benchmark` function

The benchmark being run on the architecture corresponds to an implementation of the `benchmark` function (see Definition 45), and is executed by all cores of the architecture in parallel. In effect, this corresponds to a single iteration of the inner loop from Figure 8.1. Figure 8.4 provides an overview of the `benchmark` implementation.

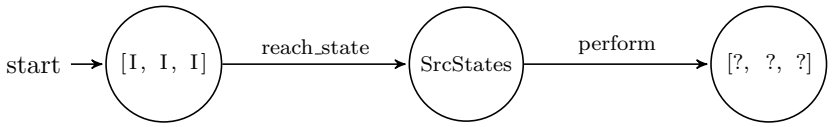


Figure 8.5 – Evolution of caches coherence states

The architecture starts in its initial state (*init*), and must thus reach the SrcStates prior to the instructions being executed (see Figure 8.5). This is done by reach_states and explained in Section 8.2.3. Once the caches have reached the relevant states, the start_monitors function enables the performance monitors, as detailed in Section 8.2.4. The relevant instructions (SysInstruction) can then be executed. The chosen assembly instructions are explained in Section 8.2.5. A barrier is used to ensure all cores have completed their instructions before results are reported. store_monitor_results is explained in Section 8.2.6, and is how the performance monitors and resulting system state are retrieved by the user.

8.2.3 Initializing the Caches (Lines 1 & 2 of Figure 8.4)

SrcStates indicates, for each of the three caches, the stable coherence state that the memory element must be in before the measures start. There are two possible cases:

- The SrcStates is [INVALID, INVALID, INVALID], and nothing needs to be done.
- SrcStates was previously reached somehow, and a sequence of instructions for each core leading to it is thus known and was added to the reach_states function.

In effect, the reach_states function follows similar idea to the one from Section 5.2.2: the different cores cannot simply reach their target state by themselves, and must thus be coordinated.

The reach_states initially does nothing, and instructions are added as new system states are reached so that they may be reached again from the *init* ([INVALID, INVALID, INVALID]) state. Let us now consider the reach_states function as it is defined after the application of the identification process on the T4240 has completed.

```
1  operation_a();
2  join(barrier_a);
3  operation_b();
4  join(barrier_b);
5  operation_c();
6  join(barrier_c);
```

Figure 8.6 – Overview of the reach_states function

Figure 8.6 provides an overview of the reach_states function. This function is performed by each core, but the operation_a(), operation_b(), operation_c() are defined depending on both the executing core's target SrcStates and its left neighbor's.

At the end of the identification process, the reach_states function was defined with to the following rules. If the core has to:

- **Reach Modified:** operation_a() writes the memory elements, the other operations do nothing.
- **Reach Exclusive:** operation_a() reads the memory elements, the other operations do nothing.
- **Reach Shared:** operation_a() reads the memory elements, the other operations do nothing.
- **Reach Forward:** operation_a() and the others do nothing, but operation_b() reads the memory elements.
- **Reach Invalid:** Given the state of the cache on the left (computed using modulo(core_id − 1, 3)):
  - **Modified:** nothing is done.
  - **Exclusive:** nothing is done.
  - **Shared:** operation_b() reads the memory elements and operation_c() empties the cache.
  - **Forward:** operation_a() reads the memory elements and operation_c() empties the cache.
  - **Invalid:** nothing is done.

8.2.4 Enabling the Performance Monitors (Line 3 of Figure 8.4)

Property 8 (T4240 Performance Counters) *Below is a list of the monitorable activities of interest, as well as their meaning based on what I understand them to be.*

- ***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***

The start_monitors() operation configures each of the available performance counters of the core to track the occurrence of different monitorable activities (see Property 8), then resets their value and activates them. Unlike with the papers in Chapter 5, all cores record the activities, not just a single one. Since there are 8 different activities to monitor and each core has four performance monitors, two runs of each benchmark is necessary in order to capture all the relevant information.

8.2.5 Performing Instructions (Lines 4 & 5 of Figure 8.4)

The perform(instructions[core_id]) call will perform either `load`, `store`, or `evict` on each of the 8000 memory elements, depending on which instruction was set for core_id in the SysInstruction array.

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 the purposes of cache coherence identification). Since the benchmarks employed here are very small programs dealing almost exclusively with the set of experimental memory elements, the application of an `evict` on all of the memory elements was replaced by a simple invalidation of the whole local cache, which still does involve cache coherence.

The `store` is implemented using *stw* (*Store Word*), which writes a zero to the memory element.

The `load` is implemented using *lwz* (*Load Word and Zero*), which writes a reads the memory element's value and stores it into an otherwise unused register.

8.2.6 Data Recording (Lines 6 & 7 of Figure 8.4)

Once every core has completed their operations and join(barrier), the store_monitor_results() prints the number of recorded values for each type of monitorable activity on each core. This data is retrieved through a serial connection.

Property 9 (T4240 Observable Flags) *Cache flags can be observed using CodeWarrior, the official debugging suite for this architecture. The observed cache line Boolean flags have the following names: Dirty, Valid, Share, Exclusive, and LastReader.*

Using CodeWarrior, the state of the caches is also recorded and stored in files, so that the flags can be analyzed afterwards. Because the cache replacement policy comes into play, the content of each cache is not simply that of the 8000 memory elements. Furthermore, the caches can seemingly contain multiple lines for the same memory element, as long as at most one indicates a state other than INVALID. The flags (see Property 9) are easily extracted using a simple Python script, as CodeWarrior allows exporting the content of all caches to plain-text CSV files.

This observable information is exactly what is needed for the new entries of both $\texttt{System}_{\text{b}}$ and $\texttt{Act}^{\texttt{Mon}}$.

The next section starts a description of the application of the identification strategy on the NXP QorIQ T4240 using this benchmark implementation.

8.3 Hypothetical Split-Transaction 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 ← nc, S	r!data, m!no-data, r ← nc, 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 ← nc, I	load hit, r ← nc, 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 ← nc, I		-	-	
$IM^{D}S$	stall	stall	stall		store hit, r!data, m!data, r ← nc, S		-	$IM^{D}SI$	
$IM^{D}SI$	stall	stall	stall		store hit, r!data, m!data, r ← nc, 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 ← nc, I		-	-	
$SM^{D}S$	hit	stall	stall		store hit, r!data, m!data, r ← nc, S		-	$SM^{D}SI$	
$SM^{D}SI$	hit	stall	stall		store hit, r!data, m!data, r ← nc, 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	hit, MI^{B}	stall	m!no-data, I			m!no-data, s!data, II^{B}	s!data, II^{B}	

Figure 8.7 – Description of the cache controller for the MESI protocol

Following the information available in documentation of the architecture ([26] and [25]), the

Coherence Manager						
State	**Received Queries**				**Data Reply**	
	GetS	GetM	PutM (Owner)	PutM (Other)	data	no-data
I	read, s!data-e, o ← s, M	s!data, o ← s, M		-		
M	o ← nc, S^D	o ← s	o ← nc, 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	o ← nc, S	o ← s	o ← nc, I	-		
S	read, s!data	s!data, o ← s, M		-		

Figure 8.8 – Description of the coherence manager for the MESI protocol

first step is to define an ambiguity-free description of the MESI protocol (Figures 8.7 and 8.8). Introduced in [41], 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.

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

Example 28 (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 not to 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, receiving the `data` message leads it to the `E` state instead of the `S` state.*

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 Figure 8.7, 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.

Example 29 (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.*

8.3.1 Strategy Application for a MESI Protocol

This section presents the application of the naive exploration of the strategy being applied to the NXP QorIQ T4240 architecture. The steps presented here have a slight deviation from those indicated in the strategy description. Indeed, the strategy saw an improvement between its application as described here and its addition to the thesis: at the time, the absence of observable coherence manager led to its state being ignored rather than it being assumed to match the hypothetical protocol. The updated strategy detects mismatches faster. In this application of the strategy, a mismatch in coherence manager behavior is likely to only be detected during the guided exploration.

8.3.2 Coherence State Matching

	⟨**Load**,-,-⟩		⟨**Store**,-,-⟩		⟨**Evict**,-,-⟩	
Origin	**Destination**		**Destination**		**Destination**	
	Observ	**Match**	**Observ**	**Match**	**Observ**	**Match**
$\langle I_b, I_b, I_b \rangle$	$\langle E_b, I_b, I_b \rangle$	$\langle E,I,I \rangle$				
$\langle E_b, I_b, I_b \rangle$	$\langle E_b, I_b, I_b \rangle$	$\langle E,I,I \rangle$			$\langle I_b, I_b, I_b \rangle$	$\langle I,I,I \rangle$
$\langle M_b, I_b, I_b \rangle$	$\langle M_b, I_b, I_b \rangle$	$\langle M,I,I \rangle$				
$\langle I_b, I_b, M_b \rangle$	$\langle \varphi_b, I_b, \chi_b \rangle$	$\langle S,I,S \rangle$			$\langle I_b, I_b, M_b \rangle$	$\langle I,I,M \rangle$
$\langle I_b, I_b, E_b \rangle$	$\langle \varphi_b, I_b, \chi_b \rangle$	$\langle S,I,S \rangle$			$\langle I_b, I_b, E_b \rangle$	$\langle I,I,E \rangle$
$\langle \varphi_b, I_b, I_b \rangle$	$\langle \varphi_b, I_b, I_b \rangle$	$\langle S,I,I \rangle$			$\langle I_b, I_b, I_b \rangle$	$\langle I,I,I \rangle$
$\langle \chi_b, \varphi_b, I_b \rangle$	$\langle \chi_b, \varphi_b, I_b \rangle$	$\langle S,S,I \rangle$			$\langle I_b, \varphi_b, I_b \rangle$	$\langle I,S,I \rangle$
$\langle \chi_b, \chi_b, \varphi_b \rangle$	$\langle \chi_b, \chi_b, \varphi_b \rangle$	$\langle S,S,S \rangle$			$\langle I_b, \chi_b, \varphi_b \rangle$	$\langle I,S,S \rangle$
$\langle \varphi_b, \chi_b, \chi_b \rangle$	$\langle \varphi_b, \chi_b, \chi_b \rangle$	$\langle S,S,S \rangle$	$\langle M_b, I_b, I_b \rangle$	$\langle M,I,I \rangle$	$\langle I_b, \chi_b, \chi_b \rangle$	$\langle I,S,S \rangle$
$\langle \varphi_b, \chi_b, I_b \rangle$	$\langle \varphi_b, \chi_b, I_b \rangle$	$\langle S,S,I \rangle$			$\langle I_b, \chi_b, I_b \rangle$	$\langle I,S,I \rangle$
$\langle I_b, I_b, \varphi_b \rangle$	$\langle \varphi_b, I_b, \chi_b \rangle$	$\langle S,I,S \rangle$			$\langle I_b, I_b, \varphi_b \rangle$	$\langle I,I,S \rangle$
$\langle \chi_b, I_b, I_b \rangle$	$\langle \chi_b, I_b, I_b \rangle$	$\langle S,I,I \rangle$			$\langle I_b, I_b, I_b \rangle$	$\langle I,I,I \rangle$
$\langle I_b, I_b, \chi_b \rangle$	$\langle \varphi_b, I_b, \chi_b \rangle$	$\langle S,I,S \rangle$			$\langle I_b, I_b, \chi_b \rangle$	$\langle I,I,S \rangle$
$\langle I_b, \varphi_b, \chi_b \rangle$	$\langle \varphi_b, \chi_b, \chi_b \rangle$	$\langle S,S,S \rangle$			$\langle I_b, \varphi_b, \chi_b \rangle$	$\langle I,S,S \rangle$
$\langle I_b, \chi_b, \chi_b \rangle$	$\langle \varphi_b, \chi_b, \chi_b \rangle$	$\langle S,S,S \rangle$			$\langle I_b, \chi_b, \chi_b \rangle$	$\langle I,S,S \rangle$
$\langle \chi_b, \chi_b, I_b \rangle$	$\langle \chi_b, \chi_b, I_b \rangle$	$\langle S,S,I \rangle$			$\langle I_b, \chi_b, I_b \rangle$	$\langle I,S,I \rangle$

Figure 8.9 – T4240 L2 Cache state transitions

.

Applying Step 1 of Figure 8.1 on the T4240 yields the stable state transitions summarized in the *Origin* and *Observ* columns of Figure 8.9. The *Origin* column corresponds to the observed system state in $\texttt{System}_b$ prior to the operation being performed, and each *Observ* corresponds to the element of $\texttt{System}_b$ observed upon application of the given instruction (see top row) on the first cache. The 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, as these cannot be observed, as explained in Limitation 1.

Once these observations are completed, Step 2 can start: the observed states can be matched with the hypothetical ones: the observed $\mathtt{M_b}$, $\mathtt{E_b}$, and $\mathtt{I_b}$ states 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, starting from $\langle \mathtt{I_b,I_b,M_b} \rangle$ and performing a `load` operation on the first cluster, leads to two different states, $\varphi_\mathtt{b}$ and $\chi_\mathtt{b}$, where the `S` state equivalent would have been expected. The same occurs when starting from $\langle \mathtt{I_b,I_b,E_b} \rangle$. By itself, this observation is not sufficient to conclude that there is a discrepancy between hypothetical protocol and the observed one. Indeed, the two observed states are marked as separate, their difference may very well not be related to cache coherence (e.g. hint for cache eviction) and the two states may actually simply correspond to our `S` state. It is only by comparing how those now exposed two states differ, if at all, in their behavior that we may conclude whether they represent an inconsistency.

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 $\varphi_\mathtt{b}$ state: the first cluster to reach the $\varphi_\mathtt{b}$ moves to the $\chi_\mathtt{b}$ state upon seeing the other's query. Neither is it possible to have all three clusters in the $\chi_\mathtt{b}$ state: the last cluster to load from $\mathtt{I_b}$ always enters $\varphi_\mathtt{b}$, and there is no way to reach $\varphi_\mathtt{b}$ other than doing exactly that.

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

Figure 8.10 – Observable Coherency Cache States of the T4240 L2 Caches Protocol

A benefit of using CodeWarrior is that the fields defining the state of a cache line are labelled. Figure 8.10 indicates the fields corresponding to each observed cache state.

8.3.3 Coherence Activity Matching

While having both $\chi_\mathtt{b}$ and $\varphi_\mathtt{b}$ mapped to the `S` state does not contradict the hypothetical protocol, it does raise suspicions of a protocol mismatch. This suspicion is confirmed by the activity analysis (Step 3 of Figure 8.1).

Figure 8.11 shows the non-null values returned by the performance monitors when loading a dataset of 8000 unique memory elements from the $\mathtt{I_b}$ state on a cluster, depending on the coherence state for these memory elements on another cluster. 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 between the two tables of the *Origin* state column and that of operation performed. The activity analysis results for $\langle \mathtt{I_b,I_b,I_b} \rangle$ are given as a reference point. Indeed, as it was so far assumed that $\chi_\mathtt{b}$ and $\varphi_\mathtt{b}$ are equivalent to an `S` state, the results ought to have been the same in all the lines of this first table.

The first surprising observation is that the amount of L2D accesses is consistently twice the expected number. While it is odd and I have not found the reason behind this discrepancy, I chose

	⟨load, -, -⟩	
Origin	**Behavior**	
	Expected	**Observed**
⟨I_b,I_b,I_b⟩	8000 L2D Accesses, 8000 Reloads From CoreNet	16000 L2D Accesses, 8000 Reloads From CoreNet, 1166700 CPU Cycles
⟨I_b,I_b,φ_b⟩	8000 L2D Accesses, 8000 Reloads From CoreNet	16000 L2D Accesses, 8000 Reloads From CoreNet, 850600 CPU Cycles
⟨I_b,I_b,χ_b⟩	8000 L2D Accesses, 8000 Reloads From CoreNet	16000 L2D Accesses, 8000 Reloads From CoreNet, 1172600 CPU Cycles

	⟨-, -, load⟩	
Origin	**Behavior**	
	Expected	**Observed**
⟨I_b,I_b,I_b⟩	8000 External Snoop Requests	8000 External Snoop Requests
⟨φ_b,I_b,I_b⟩	8000 L2 Snoop Hits, 8000 External Snoop Requests	8000 L2 Snoop Hits, **8000 L2 Snoop Pushes**, 8000 External Snoop Requests, **8000 SINTs**
⟨χ_b,I_b,I_b⟩	8000 L2 Snoop Hits, 8000 External Snoop Requests	8000 L2 Snoop Hits, 8000 External Snoop Requests

Figure 8.11 – Focusing on the Activities of φ_b and χ_b

to not consider it to be a sufficient contradiction of hypothetical protocol, as this factor holds true for every single one of the benchmarks.

Much more interesting is the hint of truly unexpected activity found in the upper table, where the ⟨I_b,I_b,φ_b⟩ benchmark is performed using less CPU cycles than the others. Looking at what happens on the bottom table for the symmetrical line, it can be seen that the cache holding the memory elements in the φ_b is actually providing them to the demanding cluster. This is in clear contradiction with the hypothetical protocol. Furthermore, this is not simply a case of having different activity for what should be the S state: the ⟨χ_b,I_b,I_b⟩ line of the bottom table indicates that no such thing is happening for memory elements in the χ_b state. This proves that φ_b and χ_b are, in fact, two completely separate stable states. In other words, the NXP QorIQ T4240 architecture does not actually use MESI as its coherence protocol.

8.4 Hypothetical Split-Transaction MESIF Protocol

As the hypothetical protocol has been invalidated, a new one matching the observations must be defined. The difference of behavior between φ_b and χ_b points to a MESIF protocol.

Figures 8.12 and 8.13 show a formal definition of the MESIF protocol. Introduced in [28], this protocols adds a *Forward* stable state, which is equivalent to a *Shared* state with the added constraint of being responsible for the propagation of the memory element's current value. This

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 ← nc, S	r?data, m?no-data, r ← nc, S	-	$IS^{D}I$	
$IS^{D}I$	stall	stall	stall		load hit, r?data, r ← nc, I	load hit, r?data, r ← nc, 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 ← nc, I		-	-	
$IM^{D}S$	stall	stall	stall		store hit, r?data, m?data, r ← nc, S		-	$IM^{D}SI$	
$IM^{D}SI$	stall	stall	stall		store hit, r?data, m?data, r ← nc, 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 ← nc, I		-	-	
$SM^{D}S$	hit	stall	stall		store hit, r?data, m?data, r ← nc, S		-	$SM^{D}SI$	
$SM^{D}SI$	hit	stall	stall		store hit, r?data, m?data, r ← nc, 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	hit, MI^{B}	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}	

Figure 8.12 – Description of the cache controller for the MESIF protocol

Coherence Manager						
State	**Received Queries**				**Data Reply**	
	GetS	GetM	PutM (Owner)	PutM (Other)	data	no-data
I	read, s?data-e, o ← s, M	s?data, o ← s, M		-		
M	o ← s, F^D	o ← s	o ← nc, 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	o ← s, F	o ← s	o ← nc, I	-	write	-
S	read, s?data, F	s?data, o ← s, M		-		
F	o ← s	o ← s, M	o ← nc, S	-	write, IoF^B	IoF^B

Figure 8.13 – Description of the coherence manager for the MESIF protocol

makes it possible to avoid reading from the system's main memory even when multiple caches hold the same memory element. However, unlike the *Exclusive* state, it does not allow the cache to upgrade to a *Modified* state by itself, since the other caches still do 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. This is unclear in the definitions of the protocol I have seen so far, but this version of the protocol assumes 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 FM^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.

8.4.1 Strategy Application for a MESIF Protocol

Using the new hypothetical protocol, the identification process can continue. φ_b is now identified as matching the `F` state, and it thus renamed F_b. Likewise, the χ_b state is now named S_b, as it does appear to correspond to the `S` state.

Overall, the benchmark results confirm a MESIF protocol, albeit differing in some of the implementation choices, as explained below.

8.4.2 No `store` Optimization on `F`

The hypothetical MESIF protocol 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 are actually received again (CoreNet Reloads) and that the cache holding the memory elements in the F_b state is not sending them to itself (which would lead to Snoop Pushes activities

being observed). As indicated in Section 8.4, whether this optimization is part of the protocol or not is unclear. This is exactly the kind of difference between implemented and hypothetical protocol that needs to be known by the architecture's user when looking for interference causes.

	⟨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

8.4.3 Odd Results with `evict` on `M`

Eviction from M_b yields 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, the `evict` instruction is not implemented as a single access but rather as a general eviction of all lines in 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

8.4.4 Better Coherence Manager

Because the state of the coherence manager was ignored during the naive exploration in this application of the strategy, the behaviors that cannot be reach through naive exploration when observable system states are considered to be only defined by their caches have yet to be analyzed.

The guided exploration still ensures these behaviors are indeed analyzed, however. Indeed, this corresponds to both of the following stable state paths:

- $\mathtt{I} \xrightarrow{\mathtt{load}} \mathtt{IS}^{\mathtt{BD}} \xrightarrow{\mathtt{Qry_{own}}} \mathtt{IEoS}^{\mathtt{D}} \xrightarrow{\mathtt{data\text{-}e}} \mathtt{E}$
- $\mathtt{I} \xrightarrow{\mathtt{load}} \mathtt{IS}^{\mathtt{BD}} \xrightarrow{\mathtt{data\text{-}e}} \mathtt{IE}^{\mathtt{B}} \xrightarrow{\mathtt{Qry_{own}}} \mathtt{E}$

While the coherence manager cannot be directly observed, attempting to expose the issue mentioned in Example 28 would reveal if it shares the same limitations as the one from the hypothetical

protocol. As it happens, the benchmarks show that the $\texttt{E}_\texttt{b}$ state is reached after a `load`, even if all caches just performed an eviction of the memory element from either a $\texttt{S}_\texttt{b}$ or $\texttt{F}_\texttt{b}$ state. Thus, the architecture either features a better coherence manager than the one described in the hypothetical protocol, or some other co-ordination strategy is used to detect the possibility of reaching the $\texttt{E}_\texttt{b}$ state.

8.5 Conclusion

This chapter presented an approach to the identification of the cache coherence mechanisms implemented by an architecture. Such a step is necessary to expose and resolve any ambiguities the user may have on those mechanisms. It does not perform an exhaustive profiling of speed and bandwidth offered by the cache coherence, but instead ensures that the user has a complete list of the mechanisms for which to perform this profiling.

The application of this approach on the NXP QorIQ T4240 fully vindicated the need for this additional step in the profiling of an architecture. Indeed, the results showed that the difference between what I believed the architecture to implement and what it actually does are far beyond the level of negligible detail: an additional coherence state was found. Without realizing that this state exists, the speed and bandwidth analysis may have obtained false results by attributing performance measurements to the wrong state and thus incorrectly profiling the architecture without realizing it.

There were in fact multiple contributions in this chapter: the approach itself, which is the main contribution; the NXP QorIQ T4240 example, which serves as a warning not to overlook the necessity of such a step; and the formalization of two protocols, which are minor contributions in themselves.

Having reached the end of this chapter, the issue of detecting sources of interference in the cache coherence mechanisms is resolved. Being able to determine the effects of this interference on a running program will require tools. The approach chosen in this thesis is to rely on models. The next chapter thus introduces how a model of the system is created.

Chapter 9

Modeling Cache Coherence

This chapter presents a UPPAAL model[1] for the analysis of the effects of cache coherence in multi-core processors. The goal is to create a formal model in order to perform automatic analyses (which are described in Chapter 10), while ensuring that:

- The model is as generic as possible in how it models the coherence protocol, making it easy to switch protocol.
- The protocols are modeled in detail, taking into account all transient states and being defined for split-transaction buses.

The approach chosen is similar to the papers presented in Chapter 7: use a network of small timed automaton, each representing a component, so that the model of the system ends up easily readable, modular, and re-usable.

The chapter starts by an overview of the modeling strategy (Section 9.1). The model is then seen through its communication channels in Section 9.2, which provides an understanding of how the components interact. The sections that follow each provide a precise description of each component's automaton. Once all components have been covered, Section 9.9 presents a tool to allow the coherence protocol being used by the model to be automatically changed to another.

9.1 Modeling Strategy

To create a model for the architecture, the chosen strategy was to base it on the principles explained in Chapter 3. In effect, the architecture is reduced to elements directly relevant to cache coherence. The assumption is that any user requiring more detail on (or the addition of) a specific component's model can either adapt it from another UPPAAL representation of architecture (such as the ones described in Chapter 7), or port the cache coherence mechanisms modeled here to that other model. The coherence protocol to be modeled is chosen outside of the UPPAAL model, using CoProSwi, a Java program I made in order to make protocol switching trivial (see Figure 9.1). This tool is explained in more details in Section 9.9.

Each component has its own automaton. These automata are designed around the synchronizations they perform with other automata. As a result, automaton transitions are mainly present for the purposes of synchronizing, with coherence behaviors being modeled solely using the C-like

[1] Available at `https://github.com/nsensfel/phylog-cache-coherence`

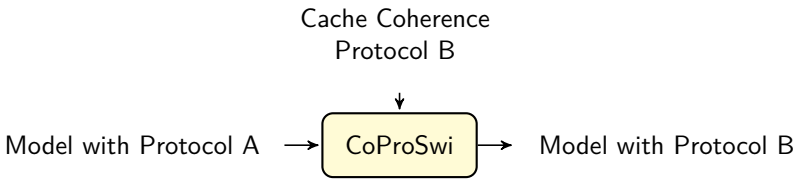


Figure 9.1 – Co(herence) Pro(tocol) Swi(tcher) utility

UPPAAL language (and thus not directly visible on the automata). This results in smaller, more readable automata.

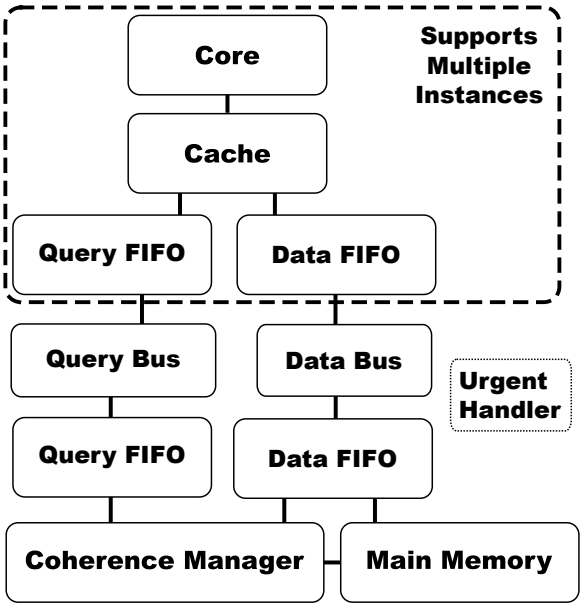


Figure 9.2 – Overview of the Model's Automata

Figure 9.2 shows all the automata defined in the model. Compared to the archetype target architecture presented in Chapter 4, the differences are minor. Indeed, the *split-transaction interconnect* has been split into its two composing parts: a *query bus* and a *data bus*. Additionally, a *data FIFO* automaton has been added, which is shared by the coherence manager and the main memory. Lastly, an *Urgent Handler* automaton is present. It does not correspond to any component, but is simply here as an utility for `urgent` synchronization.

To facilitate communications, a unique identifier is assigned to some components (namelly caches, cores, the coherence manager, the main memory, and the query bus). This identifier is used to select the correct sub-channels in some synchronizations, as well as identifying the sender and recipient of a message. Furthermore, in the model's system declaration, automata can be linked by being given as parameters the identifier of the automata they should contact (e.g. a query FIFO and its cache).

Data transfers between automata is done by synchronizing on any channel, the sender sets a dedicated shared variable with the content of the message, including the sender field. The other

fields correspond to recipient id, type of transfer (equivalent to type of event, using the definition from Chapter 3), and the address of the relevant memory element (if applicable). During the synchronization, the recipients copy the shared variable into their local variables, ensuring that the message is not overridden by a future transition from another automaton.

The model is designed following the assumptions made in Section 4.2.1, and can be further tailored to fit the relevant architecture by modifying the model parameters described in Appendix B.

9.2 Synchronization Channels

Figure 9.3 shows how synchronizations between the automata are organized. Some synchronizations, only intended to ensure the proper function of the model, are omitted from the figure, namely **SYS_INIT**, which initializes each automaton, **FORCE_URGENT** and **FORCE_EXTRA_URGENT**, which are used to make a transition *urgent* and increase its *priority*. These last two synchronizations are performed using the automaton shown in Figure 9.4.

The synchronization channels shown in the figure can be categorized as follows.

Query Transfer:

- **QUERY_BROADCAST:** `urgent`, `broadcast` channel used to synchronize with automata waiting for an incoming query.
- **QUERY_TO_BUS:** `urgent` channel used to send a query to the query bus. One sub-channel per component ID. The ID corresponds to the query's emitter, and is used to ensure that the bus only accepts the synchronization if it comes from the current bus master.
- **QUERY_IN:** `urgent` channel used by query FIFOs to send a query to their associated cache. One sub-channel per component ID. The ID corresponds to the receiving component.
- **QUERY_OUT:** `urgent` channel used by caches to send a query to their associated query FIFO. One sub-channel per component ID. The ID corresponds to the component emitting the query.

Data Transfer:

- **DATA_IN:** `urgent` channel used by data FIFOs to send a data message to their associated cache. One sub-channel per component ID. The ID corresponds to the receiving component.
- **DATA_TRANS:** `urgent` channel used by the data bus to a data message to a data FIFO. One sub-channel per component ID. The ID corresponds to the receiving component.
- **DATA_OUT:** `urgent` channel used by caches to send a data message to their associated data FIFO. One sub-channel per component ID. The ID corresponds to the data emitter.
- **DATA_TO_BUS:** `urgent` channel used by data FIFOs to send a data message to the data bus.

Instruction Transfer:

- **CPU_REQ:** `urgent` channel used by cores to send a request to a given cache. One sub-channel per component ID. The ID corresponds to the target cache.
- **CPU_ACK:** `urgent` channel used by caches to confirm completion of a request to a given core. One sub-channel per component ID. The ID corresponds to the target core.

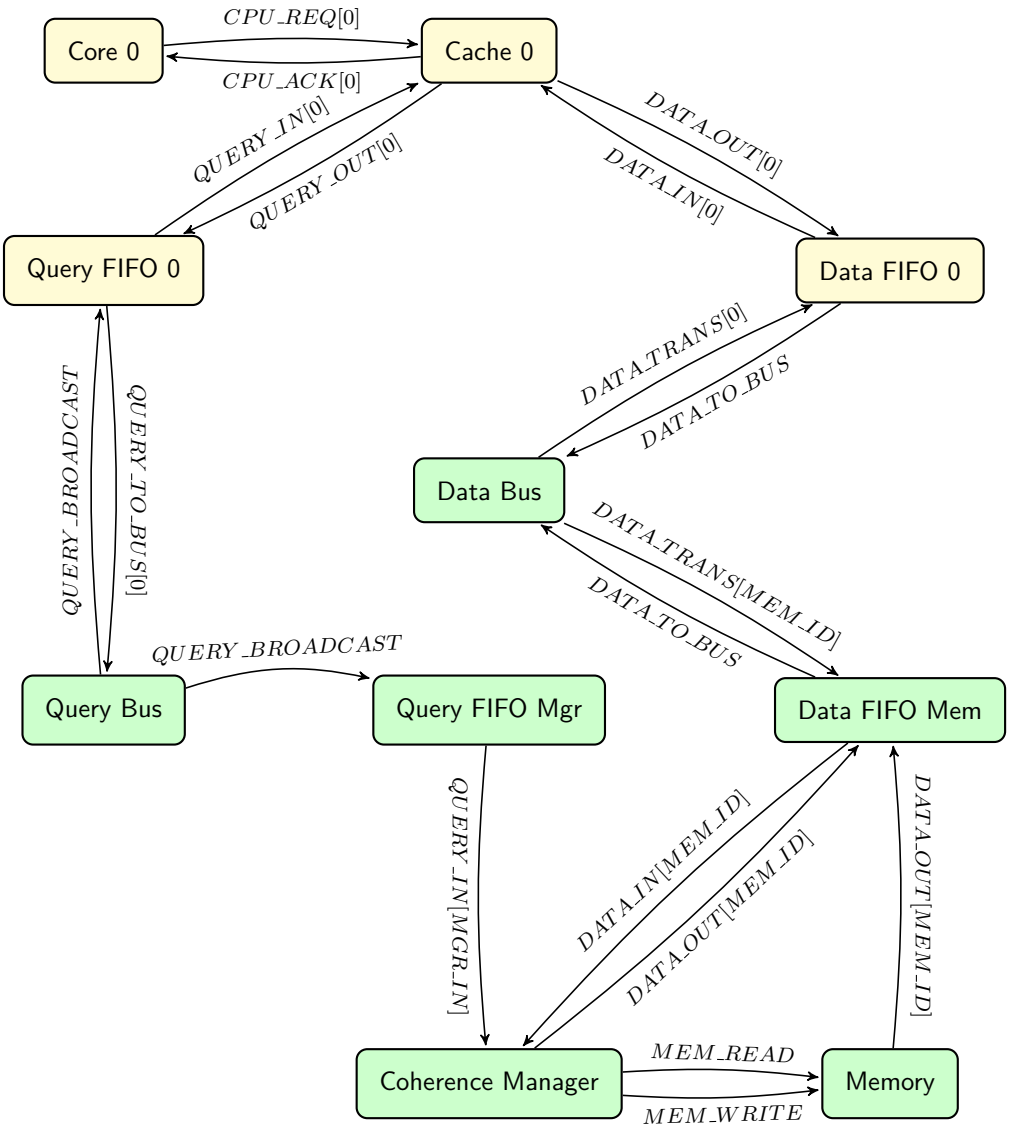


Figure 9.3 – Recurring Synchronizations in the Model

Figure 9.4 – Purely Utilitarian Automaton

Main Memory Access:

- **MEM_READ** `urgent` channel used by the coherence manager to communicate the need to read to the memory.
- **MEM_WRITE** `urgent` channel used by the coherence manager to communicate the need to write to the memory.

Two additional categories of channels are not shown in the figure, as they are either only used for initialization, or only there to manage clocks and transition priorities:

Initialization:

- **ADD_BUS_MASTER:** channel used by query FIFOs to register on the query bus as users.
- **SYS_INIT:** `urgent`, `broadcast` channel used to signal the last step of the initialization process.

Model Utility:

- **FORCE_URGENT:** `urgent` channel used to make a transition `urgent`.
- **FORCE_EXTRA_URGENT:** `urgent` channel used to make a transition `urgent`, but also increase its priority.
- **MAX_PRIORITY** `broadcast` channel used to give a transition the maximum priority.

```
1 default priority <
2 QUERY_IN, DATA_IN, MEM_WRITE, MEM_READ, QUERY_TO_BUS, QUERY_BROADCAST <
3 FORCE_URGENT <
4 QUERY_OUT, CPU_REQ <
5 CPU_ACK <
6 FORCE_EXTRA_URGENT <
7 MAX_PRIORITY;
```

Figure 9.5 – Channel Priorities

To reduce irrelevant interleaving and prevent problematic executions, the channel priorities listed in Figure 9.5 are set. The removal of these interleaving also reduces time and memory consumption when performing model checking. Not all interleaving can be avoided, as adding more priorities may lead to valid executions being cut off. Thus, the chosen priorities were set so that, as far as I know, no valid execution was removed. The reasoning being each priority is as follows:

- The MAX_PRIORITY priority is used to bypass an UPPAAL limitation. Indeed, locations with an invariant cannot have exiting transitions that synchronize on an `urgent` channel. Considering nearly all communication channels are `urgent`, this means that synchronizing from a location is very restricted. This limitation is mitigated by ensuring that a non-`urgent` transition leads from the location that has an invariant to one that does not, and that this transition has the highest priority.

- The FORCE_EXTRA_URGENT priority is used to perform transitions that should be done as soon as possible, but do not allow clocks to progress. This is used to prioritize transitions which do not involve synchronization with other automata (other than the *Priority Handler* automaton).

- CPU_REQ < CPU_ACK is here to ensure the CPU acknowledges completed requests before sending new ones.

- FORCE_URGENT < QUERY_OUT ensures that bus masters that need to use the bus indicate that need before the bus checks for it.

- default priority < QUERY_IN, DATA_IN, MEM_WRITE, MEM_READ, QUERY_TO_BUS forces automata to enter waiting locations as soon as possible.

Workarounds for UPPAAL synchronization limitations:

- Synchronizations on broadcast channels do not require receivers to be able to synchronize. As a result, making a broadcast **QUERY_BROADCAST** without ensuring that all automata that are meant to receive a query are ready to do so would risk some of the automata not getting all queries. Indeed, one possiblity is for an incoming query queue to be full. To address this issue, a Boolean table, is_ready_for_bus is shared by all automata. It has as many slots as there are component IDs, and all slots start with true as value. Whenever a component which is supposed to receive queries is unable to do so, it sets its dedicated is_ready_for_bus slot to false, thus informing all other automata that synchronization on **QUERY_BROADCAST** should not occur.

- It is not possible to have a transition that depends on another automata not being able to synchronize. This is an issue when attempting to synchronize with a specific automaton unless that automaton is unable to do so, as is the case with the query bus when it attempts to receive a query from the current bus owner. To revolve this, a has_need_for_bus array is used, with one slot per component ID. Those slots start with the value false, and the automata use their dedicated slot to inform the query bus that they have at least one query to send. The query bus is thus able to know if the current bus master needs the bus or if the next bus master should be served instead.

9.3 Models of Core and Programs

This section presents the automaton used to model a core, and the manner in which programs are modeled.

Programs are sequences of instructions, without any jumps or branchings. The available operators are INSTR_LOAD (`load`), INSTR_STORE (`store`), INSTR_EVICT (`evict`), and INSTR_END (signaling the end of the program). Each operator targets a memory element, specified by its address.

Definition 53 (Instruction Model) *Instructions are modeled by* $\langle instr, addr, min\_calc\_time, max\_calc\_time \rangle$ *tuples, with* $instr$ *corresponding to the operator,* $addr$ *the targeted memory element, and* min_calc_time *and* max_calc_time *define the interval during which the core is busy following the instruction, with 0 indicating that the default CPU cycle time should be used (a value set in the model parameters indicated in Appendix B).*

```
1  const program_line_t program_101 [7] =
2      {
3          {INSTR_LOAD,      4, 0, 0},
4          {INSTR_LOAD,      5, 0, 0},
5          {INSTR_STORE,     6, 0, 0},
6          {INSTR_LOAD,      6, 0, 0},
7          {INSTR_STORE,     4, 0, 0},
8          {INSTR_EVICT,     4, 0, 0},
9          {INSTR_END,       0, 0, 0}
10     };
```

Figure 9.6 – Example of Program

Example 30 (Instruction Model) *{INSTR_STORE, 42, 3, 9} would indicate a* `store` *applied to the memory element 42, and that after emitting this request, the core would become unavailable for between 3 and 9 time units.*

Figure 9.6 shows an example of program model.

All cores share the same automaton template, model's system description initializes each core by passing as a parameter an integer corresponding to the program they are to run. The program number 0 is reserved, and indicates a program made of random instructions.

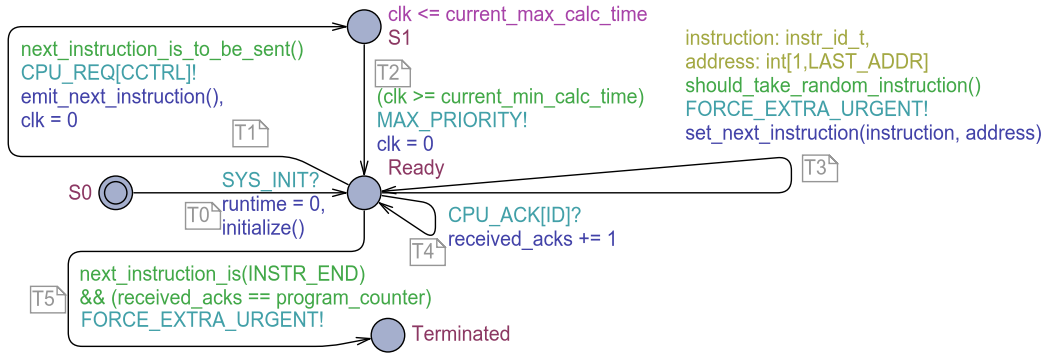


Figure 9.7 – Automaton for a Core

Figure 9.7 shows the automaton representing a core.

A core automaton has the following internal variables and clocks:

Clocks & Variables for Core

- clk is a clock used to control the time spent being busy once an instruction has been sent.
- runtime is a clock measuring time since the moment the core performed its initialization.
- program_counter is an integer corresponding to the index of the next instruction in the program's array. It is updated by emit_next_instruction().
- current_program_line contains a copy of the current instruction line. Indeed, each program is stored in its own array, and since pointers are not available in UPPAAL, finding the current

instruction line requires a chain of if/else tests so that the right array is selected before the program_counter can be used as index. Thus, to optimize, the instruction line is search for once, then copied to current_program_line. This variable is updated and used by all functions of this automaton.

- current_max_calc_time and current_min_calc_time correspond to the maximal and minimal time the core remains inactive after sending an instruction.
- received_acks is a count of the number of the acknowledgments received from caches, which indicate that an instruction sent has been completed.
- has_taken_random_instruction indicates whether the current instruction line was randomly generated. This is used by both all functions in the T_3 transition.

The automaton in Figure 9.7 starts in the S_0 location, wherein it awaits the **SYS_INIT** broadcast.

Transitions for Core

$S_0 \xrightarrow{T_0}$ `Ready` Upon synchronization on the **SYS_INIT** broadcast channel, the core resets the runtime clock and sets the current_program_line variable to the value of the first instruction of the program and also updates the current_max_calc_time and current_min_calc_time variables accordingly.

`Ready` $\xrightarrow{T_1} S_1$ For the T_1 transition to be available, the next instruction has to correspond to a memory access (i.e. `load`, `store`, `evict`). Furthermore, if the model requires all previous requests of a core to be completed prior to sending new ones, received_acks must be equal to program_counter. The T_1 transition sets the information passing shared variable to match the current instruction, then synchronizes the core's target cache **CPU_REQ** sub-channel, thus transmitting the request to the cache. The core automaton also increment the program counter and updates its local variables with the data from the next instruction line.

$S_1 \xrightarrow{T_2}$ `Ready` After having sent an instruction, the core automaton stays waiting in the S_1 location for a time in the interval defined by current_max_calc_time and current_min_calc_time.

`Ready` $\xrightarrow{T_4}$ `Ready` The core's cache will synchronize on the core's **CPU_ACK** sub-channel to signal completion of a request. This simply increments the received_acks counter.

`Ready` $\xrightarrow{T_3}$ `Ready` A special case is when the program is random. The next instruction is randomly chosen by the T_3 transition. The current_max_calc_time and current_min_calc_time are kept to their default value. Furthermore, the has_taken_random_instruction ensures that once a random instruction has been chosen, it is acted upon before a new one is chosen.

`Ready` $\xrightarrow{T_5}$ `Terminated` If the next instruction indicates the end of the program (INSTR_END), and all sent requests have been acknowledged as completed. The core automaton uses the T_5 to enter the `Terminated` location, which completes its execution.

9.4 Model of the Caches

This section presents the automaton that corresponds to a cache.

This is where the table defining the cache's behavior in the cache coherence protocol description is implemented. However, this table corresponds to the cache's behavior for a single memory element, whereas the cache has to handle a great number of them. As a result, the automaton presented in this section does not resemble the one described in a cache coherence protocol table. In effect, the transitions of this automaton are focused on what corresponds to events in the table, and actions that lead to such events being emitted.

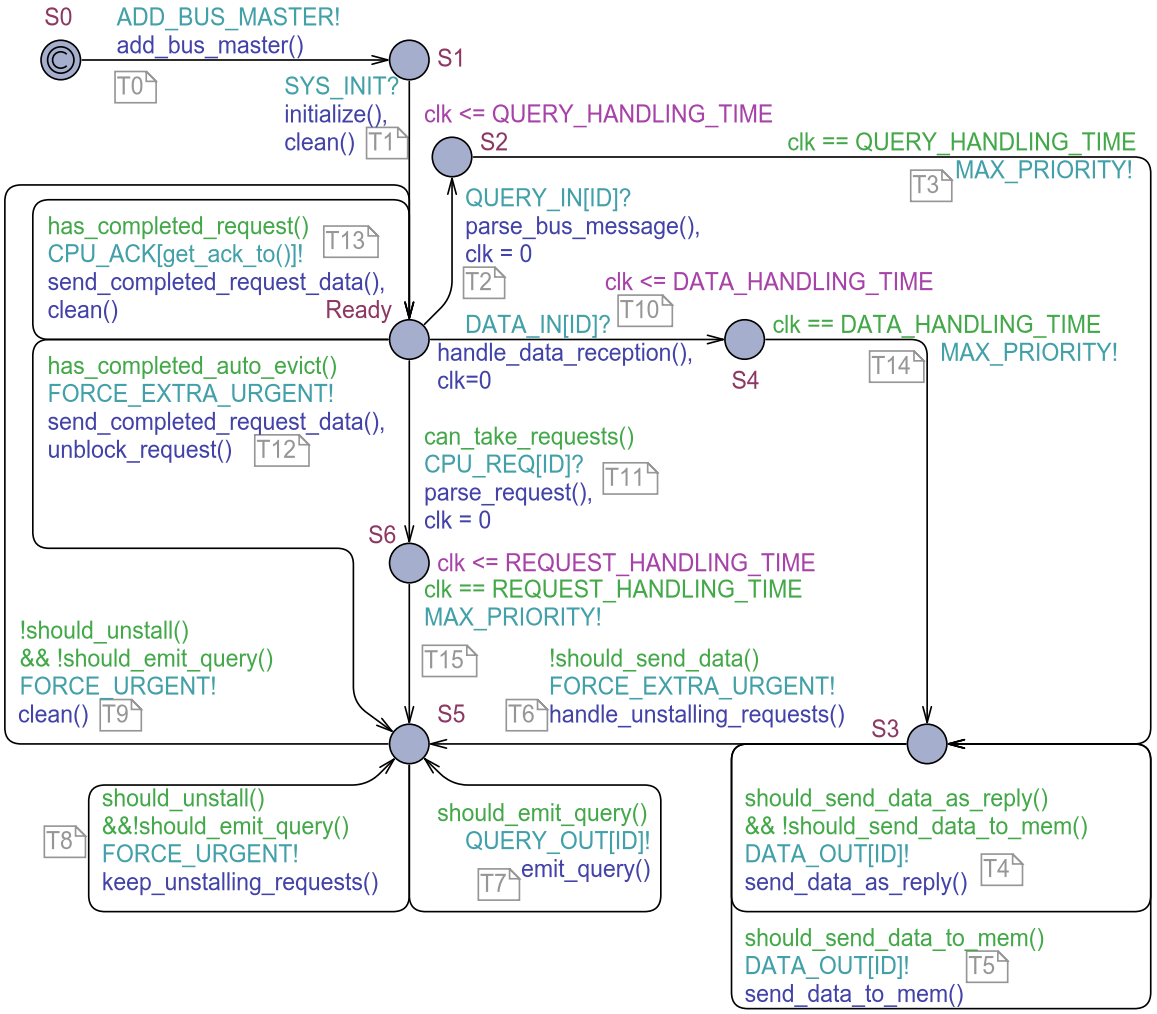


Figure 9.8 – Automaton for a Cache

Figure 9.8 shows the automaton corresponding to a cache. This is by far the most complex automaton in the model.

9.4.1 Initialization

Cache automata feature the following clock:

Clocks & Variables for Cache Initialization

- clk is a clock used to manage times during which the cache automaton is to be considered inactive.

The automaton starts in the S_0 location.

Transitions for Cache Initialization

$S_0 \xrightarrow{T_0} S_1$ The automaton starts by synchronizing with the query bus on the **ADD_BUS_MASTER** in order to be added to the list of components that can emit queries on that bus. This uses the information transmission shared variable in order to communicate to the other automaton the component ID of this cache. After this, the automaton enters the S_1 location in order to wait the **SYS_INIT** broadcast.

$S_1 \xrightarrow{T_1}$ `Ready` Upon synchronization on the **SYS_INIT** channel, the automaton sets all its internal variables to sane default values. For example, all cache lines are set to the *Invalid* state. Each cache line has its last_use value set to its index in the cache lines array.

9.4.2 Cache Lines

In cache automata, cache lines are modeled using the following variables:

Clocks & Variables for Cache Lines

- cache_lines is an array of LINES_PER_CACHE elements containing the information pertinent to every cache line (see Definition 54).
- current_line is the index of the cache line relevant to whatever operation is in progress. Thus, the line only needs to be found once, and its index can be used across multiple transitions of the automaton.

These two variables are used in just about every function of the automaton, as they are, in effect, modeling the cache's current state.

Definition 54 (Cache Line Model) *Caches lines are defined as* $\langle$`addr`, `c_state`, `last_use`, `reply_to`$\rangle$ *tuples, with* `addr` *corresponding to the address of the memory element being held,* `c_state` *its coherence state (including transient states),* `last_use` *indicating how many lines were accessed more recently than this one, and* `reply_to` *being the identifier of a cache (*`r` *from Definition 24, Chapter 3).*

Example 31 (Cache Line Model) *A cache line with a value of* $\langle 91, \mathit{MODIFIED}, 31, 3\rangle$ *corresponds to a copy of the memory element* 91*, in the Modified state. 31 cache lines have been accessed since this one was last accessed. The cache with component ID 3 has been associated with this memory element.*

Caches frequently have to find the cache line corresponding to a particular memory element. This is done either to consult the current state of the memory element in this state, or to change it to a new one. The cache line index returned when trying to find a memory element corresponds to either:

- The cache line currently holding the targeted memory element.

- The first cache line for which the coherence state is INVALID, if the targeted memory element is not held by the cache.
- A special value (-1) if the targeted memory element is not held by the cache, and there is no cache line in the INVALID state.

When attempting to consult the current state of the targeted memory element, that last case will be considered to yield INVALID, since the memory element is not in the cache. When attempting to change the state associated with the targeted memory element however, having neither the memory element nor any cache line in the INVALID state means that a cache line has to be evicted. This thus triggers the LRU eviction policy.

9.4.3 Modeling the LRU policy

The LRU policy uses this local variable from the cache automata:

Clocks & Variables for Cache Replacement Policy

- least_recently_used_line keeps track of the least recently used cache line's index. This variable is used whenever an automated eviction occurs (and those are triggered by parse_request()), and updated whenever an access is made to a cache line (which is the case in parse_request() and the unstalling functions).

Definition 55 (Cache Line Access) *A cache line is considered to be accessed when either a* `store` *or a* `load` *request is applied to it. Thus,* `evict` *is not counted as an access.*

As indicated in Definition 54, each cache line has an integer corresponding to the number of cache lines that were accessed since it itself was last accessed. To maintain this, any access to a cache line leads to the algorithm shown in Figure 9.9 being executed.

```
threshold ← cache_lines[line_being_used].last_use
i ← 0
while (i < LINES_PER_CACHE)
   if (cache_lines[i].last_use < threshold)
      cache_lines[i].last_use ← cache_lines[i].last_use + 1
   if ((cache_lines[i].last_use == (LINES_PER_CACHE - 1)) && (i != line_being_used))
      least_recently_used_line ← i
   cache_lines[line_being_used].last_use ← 0
```

Figure 9.9 – LRU Algorithm

In effect, cache lines that were already less recently used than the line currently being accessed are untouched. The cache lines that were more recently used have their last_use value incremented by one, while the one being accessed has it set to zero. This does indeed ensure that last_use still indicates how many cache lines were accessed since the one at line_being_used was accessed.

9.4.4 Handling Requests

The handling of requests is managed using the following local variables:

Clocks & Variables for Request Handling

- pending_requests is an array of REQ_BUFFER_SIZE pending requests (see Definition 56). It corresponds to requests from cores that have not been fully completed yet. It may also hold an automatically generated `evict` request.
- completed_requests is another array of REQ_BUFFER_SIZE pending requests. This time, it corresponds to requests that are already completed, but for which acknowledgment has yet to be given.
- blocked_request is a pending request. This variable is used whenever an incoming request requires the replacement policy to evict a cache line. In such cases, blocked_request is set to value of the new request, and an `evict` for the least_recently_used_line is handled instead, the blocked_request being considered only after that `evict` completes.
- current_request keeps track of the index of the element from pending_requests being handled.
- current_query is a variable containing the data for a query to send out.

Definition 56 (Pending Request Model) *Pending requests are defined as* $\langle$`requester`, `addr`, `cmd`$\rangle$ *tuples, with* `requester` *being the identifier of the component that made the request,* `addr` *being the memory element targeted, and* `cmd` *being the instruction to apply.*

Example 32 (Pending Request Model) *A pending request with a value of* $\langle 3, 91,$ `load`$\rangle$ *corresponds to a request from the component of ID* 3 *making a* `load` *on the memory element* 91*.*

Transitions for Request Handling

`Ready` $\xrightarrow{T_{11}} S_6$ The T_{11} transition corresponds to the reception of a request from a core. The request is obtained by synchronizing on the **CPU_REQ** sub-channel corresponding to the cache's identifier. For this transition to be allowed, the pending_requests must have at least two free slots. The extra slot is required in case an automatic eviction must be added to the array. Reception of a request makes the cache unavailable for a small amount of time, hence clk being set to zero.

Upon reception of the request, the cache line corresponding to the relevant memory element is located. If no cache line is available, the request is put into blocked_request and the execution continues as if the new request had been an `evict` emitted by this very cache for the memory element held in the least_recently_used_line cache line.

The addr of the cache line that was chosen is set to the address targeted by the request. This ensure that if it was an unused cache line, the stored address is the right one.

The actions then performed as those indicated by the cache protocol for the reception of a request of that type, on a memory element with the coherence state currently held in the identified cache line (see Section 9.4.6 for more details). If the actions do not include a `hit`, the request is then added to the pending_requests array as if in a queue. Otherwise, the `hit` will have added it to the completed_requests array.

$S_6 \xrightarrow{T_{15}} S_5$ The cache becomes active after having waited the REQUEST_HANDLING_TIME delay caused by the reception of a demand from the core, and is now able to complete the handling of the new request.

$S_5 \xrightarrow{T_7} S_5$ If the request being handled had an action that required it to send a query (current_query targets a memory element other than NULL), the sending is done by this transition. It synchronizes on the **QUERY_OUT** sub-channel corresponding to this cache's identifier in order to communicate the query in current_query to the cache's query FIFO automaton.

$S_5 \xrightarrow{T_8} S_5$ If no query have to be sent because of the handled request, all actions for this request that needed to be done at this point have been done. However, if the state of the targeted memory element has changed, there may be some older pending request for that memory element that could be un-stalled. If this older request has actions other than `stall` when the memory element is in its current coherency state, then that older request becomes the current request and those actions are applied (here also, the actions are described in Section 9.4.6). If this is not the case, the current request is unchanged, and the T_9 transition is ready to be taken.

$S_5 \xrightarrow{T_9}$ `Ready` If neither the T_7 transition nor the T_8 one can be taken, the T_9 transition simply resets all the internal variables that were used to quickly access whatever array line was relevant (e.g. current_line). The cache automaton is then able to return to its `Ready` state.

`Ready` $\xrightarrow{T_{13}}$ `Ready` If there is an element in the completed_requests array, whose `requestor` ID is not that of this cache, a synchronization on the **CPU_ACK** sub-channel of the `requestor` is used to inform of the completion of a request. The information transmission shared variable is set to contain the request, although this information is not actually used by the core automaton in this model. As it has been communicated, the request is removed from the completed_requests array

`Ready` $\xrightarrow{T_{12}} S_5$ If the top element in the completed_requests array has a `requestor` ID corresponding to that of this cache, it means it was for an automatic eviction, which is now completed, and that a request is waiting in blocked_request. This transition removes the top element of completed_requests, then loads the internal variables as if the request from blocked_request just arrive. In effect, it acts as T_{11}, but obtaining the request from blocked_request instead of the information sharing global variable.

9.4.5 Handling Messages

Messages are handled using the following local variables:

Clocks & Variables for Data & Query Handling

- should_send_data_to_mem_flag is a Boolean indicating if a data message should be sent to the coherence manager.
- data_to_mem contains the type of data to send to the coherence manager.
- data_to_reply contains the type of data to send to a cache.
- should_send_data_as_reply_flag is a Boolean indicating if a data message should be sent as a reply to the cache that sent the message currently being handled.

Transitions for Data & Query Handling

`Ready` $\xrightarrow{T_2} S_2$ Upon synchronization with the **QUERY_IN** sub-channel corresponding to this cache's identifier, an incoming query for the this cache's query FIFO is retrieved. The corresponding cache line is located. If there is no such line, the coherence state for this line is considered to be *Invalid*. No line is allocated for the memory element, and so there cannot be any automated cache eviction occurring. The actions prescribed by the cache coherence protocol upon reception of the communicated type of query when in this memory element's coherence state are applied. Refer to Section 9.4.6 for details on how each action is modeled.

$S_2 \xrightarrow{T_3} S_3$ Parsing an incoming query makes the cache inactive for QUERY_HANDLING_TIME. Once this period is elapsed, the actions performed in reaction to the query that interact with other components (namely sending data messages) can be performed.

`Ready` $\xrightarrow{T_{10}} S_4$ T_{10} is the analogue of T_2, but for data messages instead of queries.

$S_4 \xrightarrow{T_{14}} S_3$ T_{14} is the analogue of T_3, but for data messages instead of queries, so the waiting period is DATA_HANDLING_TIME instead.

$S_3 \xrightarrow{T_4} S_3$ If should_send_data_as_reply_flag is set to `true`, but should_send_data_to_mem_flag is set to `false`, a data message of the type indicated in data_to_reply for the relevant memory element is generated, targeting the ID of the sender of whatever message led to this location being reached. The data message is transferred to this cache's data FIFO outgoing queue through a synchronization on the **DATA_OUT** sub-channel corresponding to the ID of this cache. should_send_data_as_reply_flag is then set to `false`. Sending the reply *after* any data message meant for the memory is a modeling choice in order to reduce possible executions. Unfortunately, this has an impact on timing. Prioritizing one or the other should be done by the user so as to avoid having the model checking explore executions where the order changes.

$S_3 \xrightarrow{T_5} S_3$ If should_send_data_to_mem is set to `true`, a data message of the type indicated in data_to_mem for the relevant memory element is generated, targeting the coherence manager. It is added to this cache's outgoing data FIFO queue through synchronization on the **DATA_OUT** sub-channel corresponding to the ID of this cache. should_send_data_to_mem is then set to `false`.

$S_3 \xrightarrow{T_6} S_5$ If there is no (longer any) data to send, the incoming message is considered to have been handled. Since the actions performed may have change the state of the memory element, an un-stalling process of pending requests similar to the one from transition T_8 takes place.

9.4.6 Modeling Actions

This subsection describes how each action defined in Section 3.1.3 is performed by this automaton.

- **Stalling:** In the context of the un-stalling process, this has no other effect than preventing the un-stalling process from continuing. Otherwise, the request is simply put in the pending request queue (pending_requests), same as any request that is not the target of a `hit` action.
- **Completing a core's request:** The oldest request for this memory element that matches the given type of instruction is pulled out of the pending request queue (pending_requests), and added at the end of the completed request queue (completed_requests).

- **Preparing a query:** This can only be performed as a reaction to a request, and only a single query can be sent per request. This limitation is explained in Section 11.2.2. By setting the current_query targeted memory element to anything but the NULL address, the should_emit_query() from T_7 becomes true. The type of the query to send is also stored in current_query, and since its targeted memory element is set to the one from current_line, it will not be NULL.
- **Changing state:** The cache line at current_line has its c_state field set to the new value.
- **Preparing a data reply:** This can only be performed as a reaction to either an incoming query or data message, and only a single data reply can be sent per message. This limitation is explained in Section 11.2.2. The category of data reply is stored in data_to_reply, and should_send_data_as_reply_flag is set to `true`.
- **Preparing a data message to the coherence manager:** This can only be performed as a reaction to either an incoming query or data message, and only a single such message can be sent per incoming message. This limitation is explained in Section 11.2.2. The category of data message is stored in data_to_mem, and should_send_data_to_mem is set to `true`.
- **Memorizing a cache:** The value of the reply_to field of the cache line at current_line is set to the last received message's emitter ID. If this was a reset, the ID is set to a special component ID corresponding to NULL.

9.5 Models of FIFOs

This section presents the automata corresponding to message FIFOs. Having the FIFOs kept separate from the components greatly simplifies the components' automata. Indeed, it would otherwise be needed for every state of the components' automata to be ready for a synchronization related to message exchanges. Instead, two very similar automata are used, both having an incoming and outgoing message queue. The automata are given the component ID of the component they act for, instead of a dedicated one.

9.5.1 Query FIFO

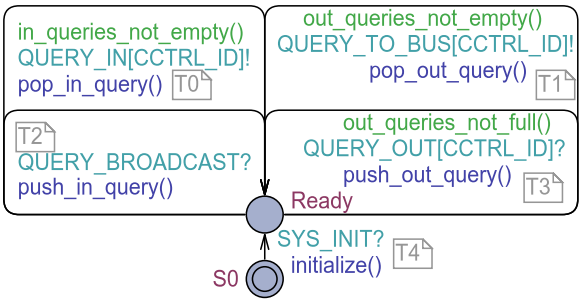


Figure 9.10 – Automaton for a Query FIFO

Figure 9.10 shows the automaton for transfer of queries. It simply maintains two arrays of queries with first-in-first-out access policies. These two arrays are the variables defined as follows:

Clocks & Variables for Query FIFO

- in_queries is an array of IN_QUERY_BUFFER_SIZE useable slots, which contains incoming query messages, and is maintained in a FIFO order.
- out_queries is an array of OUT_QUERY_BUFFER_SIZE useable slots. It stores outgoing query messages in a FIFO order.

The Query FIFO automaton starts in the S_0 location, wherein it awaits a broadcast on the **SYS_INIT**.

Transitions for Query FIFO

$S_0 \xrightarrow{T_4}$ `Ready` Upon synchronization on the **SYS_INIT**, both in_queries and out_queries have their slots set to a default value which indicates that the slot is available.

`Ready` $\xrightarrow{T_3}$ `Ready` If out_queries has available spots, the cache associated with this query FIFO can synchronize on the **QUERY_OUT** sub-channel corresponding to the cache's component ID. This transition will retrieve the query that the cache put in the information sharing global variable and insert it at the back of the out_queries queue. In addition, the slot of the has_use_for_bus global variable corresponding to the cache's component ID is set to true, allowing the query bus to know that this FIFO has content to send.

`Ready` $\xrightarrow{T_1}$ `Ready` If out_queries is not empty, the automaton synchronizes with the query bus using the **QUERY_TO_BUS** sub-channel corresponding to the cache's component ID. The query bus will only allow the synchronization if this FIFO is the current bus master. This transition removes the oldest element of out_queries and puts it in the information sharing global variable to communicate it to the bus.

`Ready` $\xrightarrow{T_2}$ `Ready` One would expect to see a in_queries_not_full () guard on this transition, however, the **QUERY_BROADCAST** channel is set to broadcast, meaning that preventing this transition from firing does not prevent the broadcast from being made. Thus, it would lead to the query never being received by this FIFO. Instead, the is_ready_for_bus global variable is used to ensure that all components that need to see queries are ready. Thus, if the transition occurs, the in_queries queue is sure not to be full.

Upon synchronization on the **QUERY_BROADCAST** channel, the query held in the information sharing global variable is copied to the back of the in_queries queue. In addition, the slot of the is_ready_for_bus global variable allocated to this FIFO's component's ID is set to true if in_queries is not full, and to false otherwise.

`Ready` $\xrightarrow{T_0}$ `Ready` If the in_queries queue is not empty, synchronization on the **QUERY_IN** sub-channel corresponding to this FIFO's component's ID leads to the oldest element of in_queries being removed and placed in the information sharing global variable. Furthermore, the is_ready_for_bus slot relevant for this FIFO is set to true, as a free spot has just been created.

9.5.2 Data FIFO

Figure 9.11 shows the automaton corresponding to a Data FIFO. It works like the one for the queries, but is simpler, as it does not have to handle any broadcast synchronizations. Thus, there is no need to handle is_ready_for_bus updates. These two arrays variables are thus also present, but defined as follows:

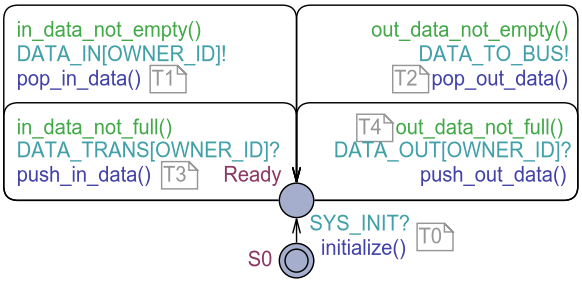


Figure 9.11 – Automaton for a Data FIFO

Clocks & Variables for Data FIFO

- in_data is an array of IN_DATA_BUFFER_SIZE useable slots, which contains incoming data messages, and is maintained in a FIFO order.
- out_data is an array of OUT_DATA_BUFFER_SIZE useable slots. It stores outgoing data messages in a FIFO order.

Transitions for Data FIFO

$S_0 \xrightarrow{T_0}$ **Ready** Upon synchronization on the **SYS_INIT** broadcast channel, both in_data and out_data have their slots set to a default value which indicates that the slot is available.

Ready $\xrightarrow{T_4}$ **Ready** If out_data has available spots, the component associated with this data FIFO can synchronize on the **DATA_OUT** sub-channel corresponding to the component's ID. This transition will retrieve the data message that the component put in the information sharing global variable and insert it at the back of the out_data queue.

Ready $\xrightarrow{T_2}$ **Ready** If out_data is not empty, the automaton synchronizes with the data bus using the **DATA_TO_BUS** sub-channel corresponding to this FIFO's component's ID. This transition removes the oldest element of out_data and puts it in the information sharing global variable to communicate it to the bus.

Ready $\xrightarrow{T_3}$ **Ready** Since the synchronization on **DATA_TRANS** is not a broadcast, there is no need for a global variable to control whether this transition can be fired. Instead, the guard allows the transition whenever the in_data queue is not full. This transition takes the data message held in the information sharing global variable and puts it at the back of the in_data queue.

Ready $\xrightarrow{T_1}$ **Ready** If the in_data queue is not empty, synchronization on the **DATA_IN** sub-channel corresponding to this FIFO's component's ID leads to the oldest element of in_data being removed and placed in the information sharing global variable.

9.6 Model of the Interconnect

This section presents the automata corresponding to the interconnect. As the split-transaction bus architectures are the target, it is unsurprising that the model of the interconnect finds itself also split into two parts: a query bus, and a data bus.

9.6.1 Data Bus

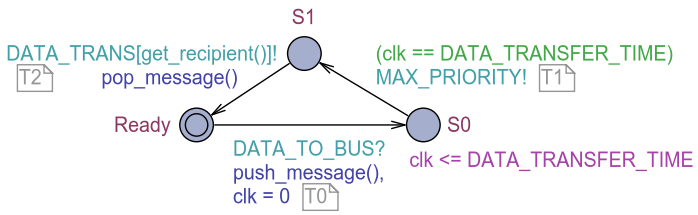


Figure 9.12 – Automaton for the Data Bus

Figure 9.12 shows the automaton used for the data bus. It uses a single variable and a single clock:

Clocks & Variables for Data Bus

- clk is a clock controlling the time the data bus stays inactive before transmitting a data message to its target.
- msg is a message to communicate.

As it does not have need for an initialization transition, the `Ready` location is the initial one.

Transitions for Data Bus

`Ready` $\xrightarrow{T_0}$ S_0 Upon any automaton synchronizing on the **DATA_TO_BUS** channel, the message in the information sharing global variable is copied to msg and clk is set to zero, as the data transfer waiting period starts.

S_0 $\xrightarrow{T_1}$ S_1 As soon as the DATA_TRANSFER_TIME period is elapsed, the automaton moves to the S_1 in order to synchronize with the message's target automaton.

S_1 $\xrightarrow{T_2}$ `Ready` By synchronizing on the **DATA_TRANS** sub-channel that corresponds to the ID of the recipient of msg, the data bus automaton stores msg in the information sharing global variable so that the other automaton is able to receive it. Once this is done, the data bus automaton becomes once again available for a new data message transfer.

9.6.2 Query Bus

Figure 9.13 shows the automaton used for the query bus. It is considerably more complex than the data one, as it features an access policy. The access policy it models is a simple round-robin one, with the order being defined once prior the **SYS_INIT** signal being emitted, then followed during the whole execution. The round-robin policy is not strict: if the next bus user in line does not need to use the bus, its turn is skipped.

The query bus automaton has the following clock and variables:

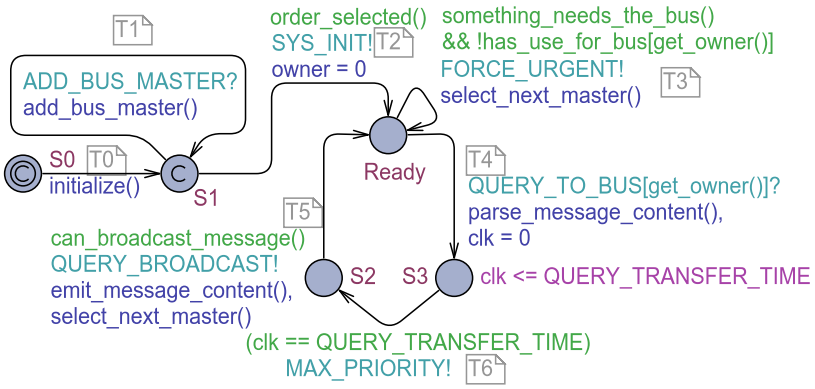


Figure 9.13 – Automaton for the Query Bus

Clocks & Variables for Query Bus

- clk is a clock controlling the time the query bus stays inactive before broadcasting a new query.
- msg is a copy of the message to broadcast.
- bus_master_order is an array indicating the order in which components can use the bus. It stores component IDs.
- owner is the index for bus_master_order corresponding to the current bus owner.

The query bus has a fairly complex initialization procedure, as it needs to define its access policy before the model is fully initialized. It starts in the S_0 location.

Transitions for Query Bus

$S_0 \xrightarrow{T_0} S_1$ The query bus automaton initializes many global variables in this transition: the default value of the information passing global variable is set; the has_use_for_bus array is initialized with false in every slot; and the is_ready_for_bus array is initialized with true in every slot.

The local variable owner is set to zero. The bus_master_order array is filled with the component ID of the query bus, a sane value to use as default.

Once all these variables have been set to sane defaults, the query bus is ready to register bus masters.

$S_1 \xrightarrow{T_1} S_1$ By synchronizing on the **ADD_BUS_MASTER** channel, other automata (namely cache automata), can register as bus master. There is no order defined for these synchronizations to take place: automata seeking to be added as bus master are all attempting to do so simultaneously. Since this is not a broadcast channel, a non-deterministic order is chosen and will be followed for the rest of the automata's execution.

This transition copies the component ID of the emitter of the message in the information transmission global variable, and stores it in the next available bus_master_order slot. The owner variable is used to keep track of how many slots have been used so far.

$S_1 \xrightarrow{T_2}$ **Ready** Completion of the bus master selection is detected by seeing that the number of slots allocated in bus_master_order, as indicated by owner, has reached CORE_COUNT.

Once this happens, owner is set back to zero and the **SYS_INIT** broadcast occurs, leading all other automata to perform their final initialization step. Likewise, the query bus automaton becomes ready to accept incoming queries.

Ready $\xrightarrow{T_4} S_3$ The query bus awaits an incoming query from its current bus owner (bus_master_order[owner]). Thus, that component's automaton has to synchronize on its dedicated **QUERY_TO_BUS** sub-channel. This synchronization leads the query bus to copy the content of the information sharing global variable to msg, set clk to zero and start waiting for the query transfer time.

$S_3 \xrightarrow{T_6} S_2$ Once the query bus has stayed inactive in S_3 for a duration of QUERY_TRANSFER_TIME, it immediately moves to the S_2 location in order to broadcast the query held in msg.

$S_2 \xrightarrow{T_5}$ **Ready** To be able to perform this transition without risking the query not reaching one of the automata that needs to receive it, the guard ensures that all slots of is_ready_for_bus hold true.

As soon as this is the case, the content of the query in msg is put in the information sharing global variable and broadcasts on the **QUERY_BROADCAST** channel.

During this transition, the owner variable is incremented by one, or set back to zero if it had reached the end of the bus_master_order array. The query bus is then once again ready to accept a query, this time from the new bus master.

Ready $\xrightarrow{T_5}$ **Ready** But if the current bus master has not set true in its dedicated slot in has_use_for_bus, yet some other component has set true in their dedicated slot, the query bus uses this transition to increment the owner variable so that it points to the next bus owner. Since the destination location is unchanged, this process is repeated until a component with a need for the bus is chosen as bus master.

Note that ensuring that at least one component needs the query bus is crucial, as the transition would otherwise be activated ad infinitum once all programs have completed.

9.7 Model of the Coherence Manager

This section presents the model for the coherence manager. Since, like the cache, it implements the behavior as defined by the coherence protocol across all memory elements, it is a rather complex automaton.

Figure 9.14 shows the automaton corresponding to the coherence manager.

Definition 57 (Coherence Manager Line Model) *A coherence manager line is defined as* $\langle$`addr`, `m_state`, `owner`$\rangle$ *tuple, with* `addr` *being the address of the memory element,* `m_state` *its state in the coherence manager (including transient states), and* `owner` *being the component ID attributed to it (see* `o` *in Definition 26 from Chapter 3).*

Example 33 (Coherence Manager Line Model) *The entry* $\langle 93, \mathit{Shared}, 8\rangle$ *would indicate that the memory element* 93 *is considered as Shared by the coherence manager, and that its owner is the component for which the ID is* 8.

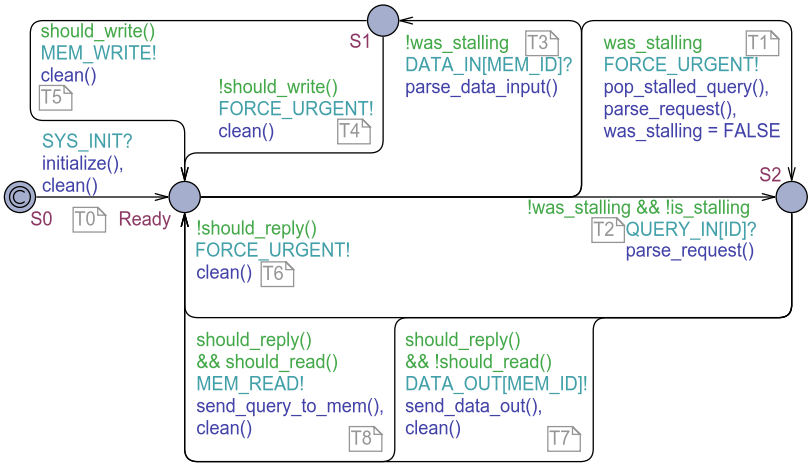


Figure 9.14 – Automaton for the Coherence Manager

As with the caches, the automaton does not directly correspond to the one used to define the behavior of the coherence manager in the protocol definition, it is instead reproducing the behavior of that automaton for each memory element.

The coherence manager automaton has the following variables:

Clocks & Variables for Coherence Manager

- cache_line_status is an array of coherence manager lines of a size equal to the sum of the size of all caches' cache_lines arrays.
- current_line is the index of the coherence manager line being currently in use.
- stalled_query is the variable in which a stalled query can be put. As indicated in Chapter 3, the coherence manager can `stall` queries, but since it is an action taken only after the query has been parsed and thus has left its FIFO, it needs a place to remain until queries are once again allowed.
- is_stalling indicates that incoming queries should not be handled for now.
- was_stalling indicates whether the coherence manager just stopped stalling and should thus act upon the query stored in stalled_query.
- reply_data_type indicates the type of data reply that should be emitted next.
- should_act_flag indicates if a data reply should be sent.
- should_mem_flag indicates if the coherence manager should require the memory controller performs an action (either reading or writing).

The coherence manager's automaton starts in the S_0 location.

Transitions for Coherence Manager

$S_0 \xrightarrow{T_0}$ `Ready` Upon synchronization on the **SYS_INIT** channel, the coherence manager sets its internal variable to sane defaults: all cache_line_status indicate the *Invalid* state, there is no stalled_query, was_stalling and is_stalling are both false, reply_data_type is an arbitrary default value, should_act_flag and should_mem_flag are both set to false.

The coherence manager is then ready to handle incoming messages.

`Ready` $\xrightarrow{T_2} S_2$ If the coherence manager is not stalling (is_stalling set to false) and is not supposed to handle what is stored in stalled_query (was_stalling set to false), then it may handle incoming queries.

To retrieve a pending incoming query, the coherence manager synchronizes with its query FIFO's dedicated **QUERY_IN** sub-channel. This transition retrieves the incoming query from the information sharing global variable and applies actions according to what is prescribed by the coherence protocol. Section 9.7.1 describes the effect of each actions in details.

Three outcomes are possible: no output is required, a data reply should be sent by the coherence manager, or a data reply should be sent by the memory controller after reading.

$S_2 \xrightarrow{T_6}$ `Ready` If no output is required (should_act_flag is set to false), the coherence manager simply returns to the `Ready` location. It is possible for this transition to have been selected because the query lead to a `stall`. In such cases, query handling has been disabled until the next `resume` action.

$S_2 \xrightarrow{T_7}$ `Ready` If a data message must be sent, but no `read` action was handled, then the coherence manager can send the data reply without consulting the memory. This corresponds to should_act_flag set to true and should_mem_flag set to false.

To send a message, the coherence manager sets the information sharing global variable so that it contains a data message of the type indicated in reply_data_type for the memory element that is relevant to the handled query. Synchronization on the **DATA_OUT** sub-channel corresponding to the memory controller's ID allows the correct data FIFO to handle the outgoing data message.

$S_2 \xrightarrow{T_8}$ `Ready` If a data message must be sent and a `read` action was handled, then the coherence manager has to rely on the memory controller to send the data message. This corresponds to both should_act_flag and should_mem_flag set to false set to true.

In this case also, the coherence manager sets the information sharing global variable so that it contains a data message of the type indicated in reply_data_type for the memory element that is relevant to the handled query. However, the synchronization on the **MEM_READ** channel, making it one that is also performed by the memory controller's automaton and not by a data FIFO. The memory controller will retrieve the data message and send it to the data FIFO once the appropriate waiting period has elapsed.

`Ready` $\xrightarrow{T_3} S_1$ Unlike queries, data messages cannot be stalled by the coherence manager. Thus, the only reason for the transition that pulls from the incoming data messages to not be allowed to fire is if was_stalling has been set to true. Indeed, if a stalled query has just been un-stalled, it is handled before anything else.

By synchronizing on the **DATA_IN** sub-channel corresponding to the memory controller's component ID, the oldest pending data message can be retrieved from the information sharing

global variable. This leads to the actions prescribed by the coherence protocol being performed. As with query handling, these actions are described in Section 9.7.1.

There are two possible outcomes, depending on whether data should be written in the main memory or not.

$S_1 \xrightarrow{T_4}$ `Ready` If there is no need for data to be written (should_mem_flag set to false), all actions related to the data message have been completed, and so the coherence manager simply returns to being able to handle the next message.

$S_1 \xrightarrow{T_5}$ `Ready` If there is a need for data to be written (should_mem_flag set to true), a synchronization on the **MEM_WRITE** channel ensures that the memory controller stays busy for the approriate amount of time. The coherence manager immediately becomes available for the next message however.

`Ready` $\xrightarrow{T_1} S_1$ A special case of "next message" is when the handling of a data message led to a `resume` action. Indeed, this indicates that, if there is a query held in stalled_query, the coherence manager is now able to act on it. The was_stalling variable being set to true indicates that the `resume` action has just occured.

Handling the stalled query is done in the same way as that of a query being pulled from the query FIFO: the actions are performed according to what is prescribed by the coherence protocol (see Section 9.7.1). The difference being that, once the actions have been performed, stalled_query is set back to its default value, and was_stalling is set to false.

9.7.1 Modeling Actions

- **Stalling:** The query is stored in stalled_query and the is_stalling variable is set to true. The query will only be processed upon a `resume` action for the relevant memory element. No other query will be considered in the meantime.
- **Changing state:** The m_state field of cache_line_status [current_line] is set to the new value.
- **Preparing a data reply:** The value of reply_data_type is set to the requested data type, and should_act_flag is set to true in order to indicate that a data reply ought to be sent.
- **Memorizing the current owner:** The owner field of cache_line_status [current_line] is set to the component ID of the sender of the message that led to this action being taken. The special component ID NULL is used if this field is to be reset.
- **Reading and Writing:** It is assumed that reading and writing never both occur as the result of the same message: queries can lead to a `read`, and data messages can lead to a `write`. Thus, setting should_mem_flag to true indicates that a synchronization with the memory controller should be performed as soon as possible. The type of synchronization performed depends on whether this was a query or a data message.

9.8 Model of the Memory

This section presents the model for the system memory. Its purpose is to model reading and writing times.

Figure 9.15 shows the automaton corresponding to the memory. It uses the following variables:

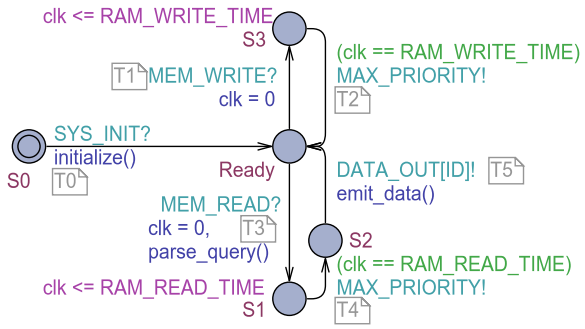


Figure 9.15 – Automaton of the Memory Controller

Clocks & Variables for System Memory

- clk is the clock controlling periods of inactivity that emulate either reading or writing times.
- msg is a copy of the message that must be sent once the waiting period is over.

The memory controller automaton starts in the S_0 location.

Transitions for System Memory

$S_0 \xrightarrow{T_0}$ **`Ready`** Upon synchronization on the **SYS_INIT** broadcast channel, msg takes an arbitrary default value. The memory controller is then ready to synchronize with the coherence manager.

`Ready` $\xrightarrow{T_1} S_3$ Upon synchronization on the **MEM_WRITE** channel, clk is set to zero in order to prepare an inactivity period corresponding to a `write`.

$S_3 \xrightarrow{T_2}$ **`Ready`** As soon as the RAM_WRITE_TIME period has elapsed, the memory controller becomes available to the coherence manager again.

`Ready` $\xrightarrow{T_3} S_1$ Upon synchronization on the **MEM_READ** channel, clk is set to zero in order to prepare an inactivity period corresponding to a `read`. Additionally, the message held in the information sharing global variable is stored in msg so that it can be sent once the inactivity period is over.

$S_1 \xrightarrow{T_4} S_2$ As soon as the RAM_READ_TIME period has elapsed, the memory controller enters a location from which it will attempt to send the message held in msg.

$S_2 \xrightarrow{T_5}$ **`Ready`** Upon synchronization on the **DATA_OUT** sub-channel corresponding to the memory controller's component ID, the value of msg is put in the information sharing global variable in order to have it be received by the data FIFO automaton, which will add it to its outgoing data message queue.

Once this is done, the memory controller is once again available for the coherence manager to synchronize with.

9.9 Switching Coherence Protocol

To make the model more easily adapted to a different architecture, it is accompanied by a tool called CoProSwi, that makes it possible to switch to a different cache coherence protocol automatically.

This tool takes as input the model and a description of the target cache coherence protocol under the form of a text file describing the tables that define the behavior of caches and of the coherence manager (as in Figures 3.3 and 3.4).

Protocols described for CoProSwi indicate:

- A list of data message types. For example, (add_data_type DATA_MSG).
- A list of query message types. For example, (add_data_type GET_SHARED).
- A definition of the cache controller's behavior.
- A definition of the coherence manager's behavior.

A notable omission are the instructions: CoProSwi assumes all protocols are defined around the `load`, `store`, and `evict` instructions. The $\texttt{Qry}_{\texttt{own}}$ event is also implicitely declared.

When defining either the behavior of either the cache controller or the coherence manager:

- Each coherence state is declared. These declarations indicate whether the state is transient or stable. For example, (add_state stable MSI_INVALID) declares the *Invalid* stable state.
- The default coherence state for memory elements is also indicated. It is considered to be the *Invalid* state's representation. For example, (set_default_state MSI_INVALID) sets MSI_INVALID as the *Invalid* state. The state must have been declared beforehand.
- For each state, the list of actions to performed upon observation of an event is then defined. In the case of the coherence's manager definition, a if_is_owner operator is added, allowing the definition of two different action lists according to whether the message received came from the component currently set as the memory element's owner or not.
- If no actions are to be taken, the (none) action must be indicated. Indeed, the tool assumes that if a case is missing, it is because of an user error, and not because no actions are to be taken.
- The coherence manager only handles query and data message events, not instructions nor reception of its own query (since it assumed to be unable to send any).

CoProSwi takes care of all the otherwise tedious aspects of describing the protocol directly in UPPAAL, such as generating separate actions to be performed during the un-stalling process in caches (in order to correctly manage request queues). In effect, no knowledge of the model's inner workings is needed to describe the protocol or to use CoProSwi in general, since the protocol definition only uses notions from Chapter 3.

The modification of the model file is done through simple search and replace: only the automata's actions and variables needs to be modified. Locations and transitions are neither added nor removed. Thus, the model's file contains tags that points to the code that needs modification by the tool upon switching of protocol.

Figure 9.16 provides an example of tagged code section. This particular snippet corresponds to the coherence manager's states declaration. When changing the coherence protocol, CoProSwi replaces the content found between the CMGR_STATES_DECLARATION tags so that it matches the

```
1  /*[CMGR_STATES_DECLARATION]*/
2  const mem_state_t MEM_I = 0;
3  const mem_state_t MEM_I_D = 1;
4  const mem_state_t MEM_S = 2;
5  const mem_state_t MEM_I_O_S_B = 3;
6  const mem_state_t MEM_S_D = 4;
7  const mem_state_t MEM_M = 5;
8  /*[/CMGR_STATES_DECLARATION]*/
```

Figure 9.16 – Example of Tagged UPPAAL Code

new protocol. As a result, the UPPAAL file can be easily modified by the user without having to wonder what will break compatibility with CoProSwi. Furthermore, this means there is no master template: the output of CoProSwi can be re-used as-is if the protocol needs to be changed again.

In addition to being able to modify the model, CoProSwi is also able to list all stable state change paths. This is useful when applying the coherence protocol identification process described in Chapter 8.

9.10 Step-by-Step Simulation

UPPAAL provides a model simulations tool, allowing the user to generate model executions by selecting each transition to take. This can be used to assert that a particular execution is indeed a valid trace of the model. It also helps to understand and debug the model. Furthermore, when using UPPAAL's model checker, the generated counter-examples can be explored in the step-by-step simulation tool. This helps understanding exactly what allowed the counter-example to be reached.

It should be noted that using the extremum operators (`sup` or `inf`, described in Section 2.2.2) in a model checking query never generates a counter-example: the property is always considered as having been verified. The resulting value is simply given to the user in the form of a pop-up dialog. To obtain a trace in the step-by-step simulator that would yield this extremum, the user has to perform a new verification. Indeed, a solution is then to ask UPPAAL to verify that the system never reaches a state in which the target clock or variable has the extremum value. This will generate a counter example, and thus one trace for which the model reaches this value.

Figure 9.17 shows UPPAAL's step-by-step simulation interface. Using this, the user is able to simulate an execution of the model for which they are able to decide the order of any transition: the available transitions in the current state are displayed in (A); those that were previously taken are shown in (B) and the system can be reverted to any of them with a simple click; the valuation of each variable and the current clock constraints are displayed in (C); the current location of each automaton is displayed in (D); and the synchronizations that have occurred so far are displayed in (E).

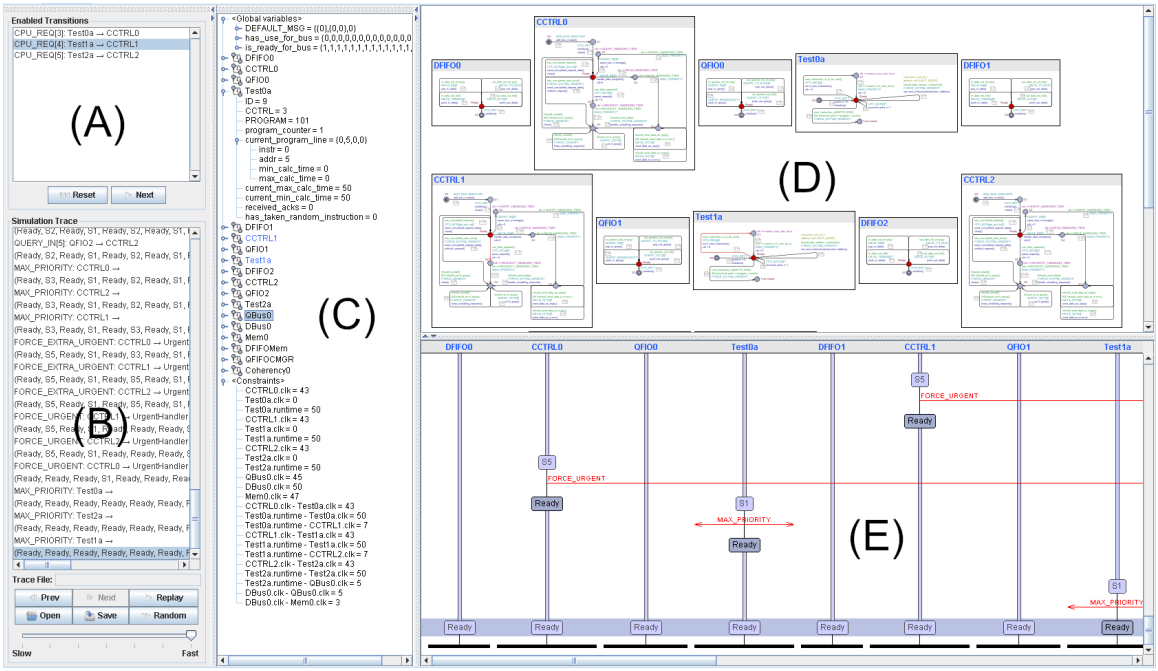


Figure 9.17 – UPPAAL Step-by-Step Simulation Interface

9.11 Conclusion

This chapter presented a way to model a multi-core architecture with a focus on cache coherence mechanisms.

The use of a network of timed automata proved to be effective in allowing a readable and modular description of the system. Indeed, each component can be modeled independently, with only synchronizations with other components needing to be accounted for.

The model presented in this chapter can be configured to match the user's architecture through easily configurable parameters. The addition of an arbitrary number of cores with their caches and FIFOs is also supported and is made easy by UPPAAL's automaton template features.

The coherence protocol is defined outside of the model, solely using the notions from Chapter 3. This makes it possible for the user to change the resulting model's cache coherence protocol without having to understand the way it is modeled. Furthermore, switching from one protocol to another can be done automatically using the provided tool, making the whole process very approachable.

Once the platform has been modeled, formal analyses can be performed. The next chapter not only explores the results that can be obtained with the model as described here, but also introduces definitions for the interference generated by cache coherence and explains how the model can be used to expose them.

Chapter 10

Exposing Interference

The previous chapter described a model of multi-core architecture with a focus on cache coherence. This chapter explores the ways this model can be analyzed through formal methods in order to expose and quantify the impact of cache coherence interference. Part of the work presented in this chapter was published in [47].

10.1 Overview of the Analyses

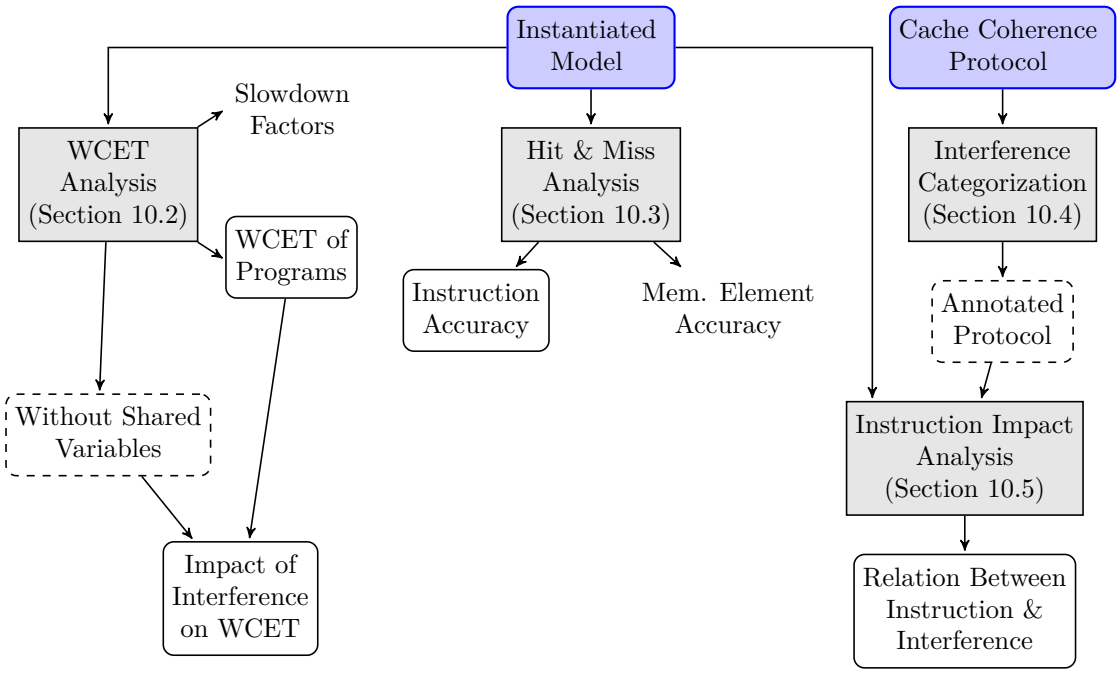


Figure 10.1 – Overview of Analyses in Chapter 10

Figure 10.1 provides an overview of the analyses performed in this chapter. Rectangles with

a gray background correspond to analyses and those without background are main results. Nodes without borders are auxiliary results, meant to provide extra information. Rectangles with dashed borders are intermediary results, which are not meant to be used on their own.

For the analyses presented in this chapter to provide relevant information to the user, the model from Chapter 9 has to be instantiated to match the user's chosen architecture (*Instantiated Model* in the figure). This instantiation corresponds to setting the model parameters listed in Appendix B according to the results of architecture profiling benchmarks. Examples of such benchmarks have been listed in Chapter 5.

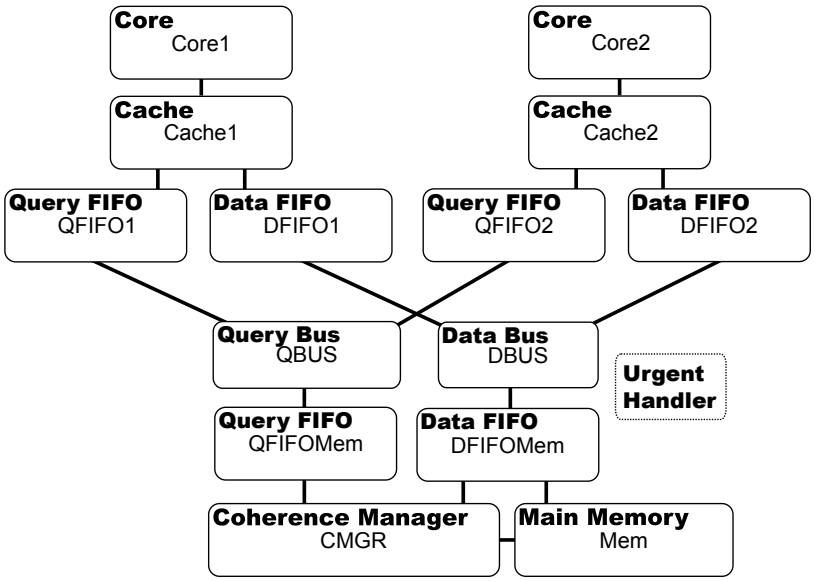


Figure 10.2 – Overview of the Instantiated Model

```
1  {INSTR_STORE,  1, 0, 0},
2  {INSTR_STORE,  2, 0, 0},
3  {INSTR_LOAD,   1, 0, 0},
4  {INSTR_STORE,  1, 0, 0},
5  {INSTR_LOAD,   3, 0, 0},
6  {INSTR_STORE,  2, 0, 0},
7  {INSTR_LOAD,   1, 0, 0},
8  {INSTR_STORE,  1, 0, 0},
9  {INSTR_LOAD,   2, 0, 0},
10 {INSTR_STORE,  2, 0, 0},
11 {INSTR_END,    0, 0, 0}
```

(a) Program Model for Core 1

```
1  {INSTR_STORE,  1, 0, 0},
2  {INSTR_STORE,  3, 0, 0},
3  {INSTR_LOAD,   3, 0, 0},
4  {INSTR_STORE,  2, 0, 0},
5  {INSTR_LOAD,   1, 0, 0},
6  {INSTR_STORE,  2, 0, 0},
7  {INSTR_LOAD,   3, 0, 0},
8  {INSTR_STORE,  1, 0, 0},
9  {INSTR_LOAD,   2, 0, 0},
10 {INSTR_STORE,  3, 0, 0},
11 {INSTR_END,    0, 0, 0}
```

(b) Program Model for Core 2

Figure 10.3 – Program Models

The instantiated model chosen to illustrate the analyses of this chapter is shown Figure 10.2. The model of each program can be seen on Figure 10.3, with Core1 running the program from Figure 10.3a

and Core2 the one from Figure 10.3b. The model uses its default parameters, as listed in Figure B.1. It uses the MESI protocol from Section 8.3.

Instantiated models still allow many different executions. For example, the order in which caches access the query bus remain undecided, leaving all possible combinations as a source of divergence between valid executions. These valid executions are analyzed using of model checking, providing information on the modeled system. In effect, each analysis is made of a number of formulas written using the syntax detailed in Section 2.2.2. The result of each analysis is an interpretation of the results of these model checking queries.

WCET Analysis

In Section 10.2, analyses are made on the instantiated model to seek worst-case program execution times. The first result taken from these analyses is the execution time of each program with cache coherence taken into account. While deviating from interference analysis, this also makes it possible to compare architecture configurations (see Definition 32) and their respective slowdown factors (see Definition 36). There is a particular configuration which should be studied regardless of its validity on the real system: running the programs without any shared variables. While this result is likely to be of use on its own, comparing it with the results from the analysis that has shared variables quantifies the impact of interference on the execution time of the programs.

With knowledge only about program execution times, the applicant would not be sufficiently informed to address the issue of cache coherence interference. As a result, the other analyses focus on program instructions. Namely, how interference affects them, and how they generate interference.

Hit & Miss Analysis

Section 10.3 uses the instantiated model to categorize each of the programs' instruction according to whether it finds the memory element in the cache or not (cache hit or cache miss, referred to as *accuracy* thorough this chapter). This provides the user with an understanding of which instructions are likely to have been impacted by cache coherence and points out which ones have execution times that may vary because of it. An auxiliary analysis looks at the accuracy of memory elements, which might point out particularly troublesome memory elements.

Interference Categorization

These accuracy analyses do not properly expose cache coherence interference. Indeed, while the effects of the interference are found within the categorization of instruction accuracy, the causes of the interference are not. Section 10.4 proposes a categorization of the effects of cache coherence interference. This makes it possible to annotate the cache coherence protocol with the transitions that can cause interference and the type of interference they generate.

Instruction Impact Analysis

Using this annotated cache coherence protocol and the instantiated model, Section 10.5 describes analyses that point out to the applicant exactly which instructions can generate interference, what category of interference they generate, which instructions can be affected, and whether this occurs on all possible executions or only some of them. This provides the information on instructions that was missing from the analyses of Section 10.3 and thus expose all cache coherence interference to the applicant with sufficient details to be the basis on which mitigation can be planned.

10.2 Analyzing Impact on Program Execution Time

This first analysis looks at the effects of interference on the programs themselves by computing their worst-case execution time. Note that when using the limited representation of the programs and of the architecture in the model proposed in Chapter 9, the execution times resulting from the analyses performed in this chapter are unlikely to be those of the real applications and should therefore only be used in comparisons with other similarly obtained execution times. Indeed, such results can be used to compare the execution time of programs in different configurations, such as for computation of slowdown factors (see Definition 36). Additionally, it is possible to obtain an estimate of the execution time that is due to cache coherence interference by comparing the execution time of each program with that of a configuration in which there are no shared variables. This configuration might not be viable in the real system, but its analysis provides valuable information as a reference point, since it corresponds to the programs neither benefiting nor suffering from cache coherence while still having to account for concurrent accesses to the system's main memory.

When looking at the execution time of programs without shared variables, the results do not include much of the interference from cache coherence, but still has the cost of concurrent memory accesses. This particular cost may actually be lowered by cache coherence. Indeed, having programs running in parallel and sharing data using cache coherence can lead to run-times increasing because cache permissions are lost, but it can also lead to run-times decreasing because more values are available in caches and may thus be retrieved faster than when those are only in the main memory.

By removing the execution time obtained by the analysis of each programs without shared variables from their execution time with those shared variables, the time lost or gained because of cache coherence is obtained. More precisely, given W_s the execution time of a program running in parallel with other programs and having shared variable and W_p the equivalent without any shared variables, $T_{cc} = W_s - W_p$ corresponds to the execution time added by cache coherence interference.

To compute the W_s and W_p using the model described in Chapter 9, UPPAAL's model checking is made to find the maximum value of the runtime clock of a core automaton outside of the *Terminated* location (e.g. sup{not Core1.Terminated}: Core1.runtime for `Core1`).

Example 34 (Slowdown Factors on the Model from Section 10.1) *Figure 10.4 shows different worst case execution times for the model from Section 10.1. The first line corresponds to the result of sup{not Core1.Terminated}: Core1.runtime and the second is the equivalent for the other core. The Shared Variables results are obtained by simply making this query on the instantiated model from Section 10.1 as-is, with both programs running in parallel unmodified. The No Shared Variables results are obtained by shifting all addresses from the program running on Core2 in order to avoid sharing any address between the two cores. In this case, all addresses from Figure 10.3b are increased by* 3*. T_{cc} corresponds to the execution time which is due to by cache coherence interference. Lastly, an example of alternative configuration in which each program runs alone on the platform is also analyzed: the Isolation configuration is achieved by replacing the other program with one composed only of a single INSTR_END instruction.*

	Shared Variables	No Shared Variables	T_{cc}	Isolation
Core1	2652	1102	1550	702
Core2	2452	1452	1000	904

Figure 10.4 – WCET Analysis of Model from Section 10.1

The results of the analysis are as follows:

- *Core1 suffers a slowdown factor of* $2652/702 = 3.77$ *when running in parallel with Core2, compared to in isolation.*
- *Core2 suffers a slowdown factor of* $2452/904 = 2.71$ *when running in parallel with Core1, compared to in isolation.*
- *Running the two programs in isolation one after the other has a maximum execution time of* $702 + 904 = 1606$.
- *Running the two programs with their shared variables in parallel has a maximum execution time of* $\max(2652, 2452) = 2652$.
- *Running the two programs without shared variables in parallel has a maximum execution time of* $\max(1102, 1452) = 1452$.
- *Approximately* $(1550/2652) * 100 = 58.44$ *percent of Core1's worst execution time is caused by cache coherence interference.*
- *Approximately* $(1000/2452) * 100 = 40.78$ *percent of Core2's worst execution time is caused by cache coherence interference.*

The following observations can thus be made:

- *The programs in this example greatly interfere with each other.*
- *Running the two programs in isolation one after the other actually leads to a WCET lower than running both in parallel with shared variables:* $1606 < 2652$. *As a result, if the nature of the programs allow it, running the programs one after the other is preferable.*
- *Running the two programs in parallel but with no shared variables is preferable to running them in isolation one after the other (*$1452 < 1606$*). This is also a result which is only exploitable if the nature of the programs allow them to be transformed into an alternative that either uses separate copies of the shared variables or uses less shared variables altogether.*

Thus, comparing WCET in different configurations of the architecture provides the user with an understanding of how much interference is affecting the programs because of cache coherence. However, this WCET analysis shows the accumulation of all interference. In order to better control this interference, a more precise analysis of the causes of this interference has to be performed. This starts with determining which instructions are impacted by interference.

10.3 Analyzing Impact on Hit & Miss

The most impactful effect interference can have on instructions is to prevent them from being able to retrieve memory elements that would otherwise be present in the cache. Thus, one way to look at the causes behind the WCET of a program is to see, for each instruction, whether retrieving the memory element they access from the cache requires the cache to fetch it or not.

The analyses in this section provide the user with an understanding of which instructions cause caches to have to fetch memory elements (a time consumming operation), whether this fetching always has to happen, never does, or sometimes does (see Section 10.3.2). It also shows how the model can be used for a similar purpose, but focused on memory elements themselves instead of each instruction (see Section 10.3.3). In effect, these analyses point out what part of the programs should be the focus of attempts at execution time improvements.

The same idea is employed in Section 6.3, which presented papers that use abstract interpretation in a way that makes all accesses be categorized as either *always-hit*, *always-miss*, *first-miss*, or *uncategorized*. However, these papers did not account for cache coherence. Using our model, this categorization can be performed even on systems featuring cache coherence.

To avoid any ambiguity, the definition of both cache hit and cache miss used in this chapter is provided below. The reason for `stall` actions being mentioned in those definitions is that as long as an event has not led to any transition other than ones which have a `stall` action, the component is not considered to have acknowledged the event's existence.

Definition 58 (Cache Hit) *A request is considered to have resulted in a cache hit if, according to the cache coherence protocol, the first triggered transition not featuring a* `stall` *action features a* `hit` *action.*

Definition 59 (Cache Miss) *A request is considered to have resulted in a cache miss, according to the cache coherence protocol, the first triggered transition not featuring a* `stall` *action does not features a* `hit` *action.*

Example 35 (Cache Hit and Miss) *In the MESI protocol from Figure 8.7 (Section 8.3), a* `store` *instruction results in a cache hit for a memory element currently held in the* `E`*,* `M` *or even* MI^B *state. If the memory element is held in the* `S` *or the* `I` *state, the* `store` *would result in a cache miss. Lastly, if the memory element is in the* IM^D *state, the categorization of the* `store` *instruction would not be determined until some other state is reached because of the* `stall` *action.*

The model, as presented in Chapter 9, does not have any notion associated with either a cache miss or a cache hit. This makes it impossible to simply use UPPAAL's model checking in order to detect them. The model has thus been modified to allow these analyses to be performed. These additions are detailed in the next sub-section.

10.3.1 Hit and Miss in the Model

Hitherto unmentioned are some aspects of the model ensuring that each cache hit and cache miss is kept track of. Indeed, the cache automaton from Section 9.4 was made so that:

Cache Automata:

- The structure corresponding to a core request in the cache automata gets two extra fields, is_cache_hit and instruction_addr, described below.
- The is_cache_hit field defaults to true, and will, once the request is completed, indicate whether than request was a cache hit or a cache miss. Thus, a request representation can be modified to be a cache miss, but not the reverse.
- The instruction_addr field keeps track of which program line this request stems from. Indeed, requests may be completed out of order, so this index is necessary to trace the resulting categorization back to the program itself.
- There is a *cache miss* action, which marks the current request as being a cache miss.
- An array, cache_local_address_infos , is part of the local variables, and has one cell per memory element. This array keeps track of the number of both hit and miss observed by this cache for each memory element.

- When acknowledging the completion of a request, cache_local_address_infos is updated according to whether the request for that memory element was a cache hit or a cache miss.

The added *cache miss* action is not explicitly part of the cache coherence protocol. Instead, the protocol switching tool presented in Section 9.9 adds it automatically. Upon generating the actions for the cache automata, the *cache miss* action is added to any action list that contains neither a `stall` nor a `hit`.

With these variables in the model, UPPAAL's model checking can be used to analyze cache hits and cache misses.

10.3.2 Instruction Characterization

The instruction characterization analysis assigns to each instruction a category among *always-hit*, *always-miss*, and *uncategorized*. The *first-miss* category, which was present in Chapter 6.3 cannot currently be detected by the model, as programs do not feature loops, meaning that each instruction is only performed once. As a result, instructions that would otherwise be characterized as *first-miss* are indistinguishable from *always-miss* ones.

This categorization allows the user to determine which instructions are mostly unaffected by interference (all *always-hit*), which ones are strongly affected (which are among the *always-miss*), and which ones are likely to cause variation in execution time (all *uncategorized*).

This analysis requires a minimum of one model checking query per instruction, and a maximum of two per instruction (if the first one is inconclusive). The UPPAAL queries corresponding to the verification of an *always-hit* or an *always-miss* can be written for the 10th instruction of Core1 as follows:

- **Always Hits:**
  `AG`(Cache1.completed_requests[0].instruction_addr = 9 `imply` Cache1.completed_requests[0]. is_cache_hit)
- **Always Misses:**
  `AG`(Cache1.completed_requests[0].instruction_addr = 9 `imply` (`not` Cache1.completed_requests[0]. is_cache_hit))

The idea being that all requests completed by a cache are guaranteed to be in completed_requests[0] (i.e. the top of their cache's completed request queue) at one point or another, making it an ideal variable to analyze. Cache1 corresponds to the cache being analyzed, as the completed_requests variable is local to each cache automaton. 9 is the index of the 10th instruction. Thus, these queries check whether the completion of the 10th instruction in Cache1 always (or never, in the case of the *always-miss*) results in a cache hit.

This analysis does not separate the cache misses caused by cache coherence and those that are innate to the program itself (e.g. the first access to a memory element is sure to be a cache miss). However, the analysis proposed in Section 10.5 provides a way to determine what categorization is the direct result of interference.

Attempting to compare the results of this analysis with the system in different configurations (namely, in isolation compared to with shared variables) in order to figure out which cache misses are caused by cache coherence is in no way assured to work. Indeed, the interference can cause cache evictions, which in turn change the effects of the cache's replacement policy: if a cache line is freed because of interference, the cache will have one extra available slot at this point in the program compared to the run in isolation, causing a divergence that is likely to void the relation between categorization in isolation and categorization with shared variables. Indeed, this may even cause instructions to be more accurate with shared variables than in isolation if an interference ends up evicting a memory element never to be accessed later and causes a memory element that

is accessed later to not be removed by the cache replacement policy because of the freed cache line. Nevertheless, even without distinction of which cache misses are caused by interference, the categorizing the accuracy of instructions provides useful information.

Example 36 (Application to the Model of Section 10.1) *Applying the instruction characterization to the model of Section 10.1 yields the results shown in Figure 10.5. AM denotes always-miss, AH stands for always-hit, and UN is for all other uncategorized cases. In this very small example,*

```
1  {INSTR_STORE,  1, 0, 0} is AM (AM)
2  {INSTR_STORE,  2, 0, 0} is AM (AM)
3  {INSTR_LOAD,   1, 0, 0} is UN (AH)
4  {INSTR_STORE,  1, 0, 0} is UN (AH)
5  {INSTR_LOAD,   3, 0, 0} is AM (AM)
6  {INSTR_STORE,  2, 0, 0} is AM (AH)
7  {INSTR_LOAD,   1, 0, 0} is AH (AH)
8  {INSTR_STORE,  1, 0, 0} is AM (AH)
9  {INSTR_LOAD,   2, 0, 0} is UN (AH)
10 {INSTR_STORE,  2, 0, 0} is UN (AH)
11 {INSTR_END,    0, 0, 0}
```

(a) Program Characterization for Core1

```
1  {INSTR_STORE,  1, 0, 0} is AM (AM)
2  {INSTR_STORE,  3, 0, 0} is AM (AM)
3  {INSTR_LOAD,   3, 0, 0} is AH (AH)
4  {INSTR_STORE,  2, 0, 0} is AM (AM)
5  {INSTR_LOAD,   1, 0, 0} is AM (AH)
6  {INSTR_STORE,  2, 0, 0} is UN (AH)
7  {INSTR_LOAD,   3, 0, 0} is AH (AH)
8  {INSTR_STORE,  1, 0, 0} is AM (AH)
9  {INSTR_LOAD,   2, 0, 0} is UN (AH)
10 {INSTR_STORE,  3, 0, 0} is AM (AH)
11 {INSTR_END,    0, 0, 0}
```

(b) Program Characterization for Core2

Figure 10.5 – Example of Program Characterizations

the cache replacement policy does not activate, which means that the performing the analysis in isolation is able to provide a reference that will indicate which instructions are cache misses because of interference and which are not. Indeed, in the figures, the first indicated characterization on each line corresponds to that obtained in the configuration with the other program running in parallel. The second one, between parentheses, is the result of the analysis in isolation. The following results can be observed:

- *In the program on Core1, only the instruction at line 7 is sure to not be negatively affected by cache coherence (always-hit).*
- *The execution time variability of the program on Core1 is caused by the instructions at lines 3, 4, 9, and 10 (uncategorized).*
- *For the program on Core2, the instructions at lines 3 and 7 are never negatively affected by cache coherence.*
- *The execution time variability of the program on Core2 is only caused by the instructions at lines 6 and 9.*
- *The result of the analysis in isolation shows that the first access for each memory element is always a cache miss, and all subsequent accesses are always cache hits.*

The result of the analysis in isolation are unsurprising, because those first accesses obtain all the permissions for the memory elements required by the subsequent accesses. A different result would be observed if the first access what a `load` *and a subsequent* `store` *was present: both would be classified as always-miss. Note that this particular example does not feature any* `evict` *instruction nor does it trigger the caches' replacement policy. As indicated, the latter would impact the results of the analysis in isolation. Considering the amount of interference in this example, the program running on Core2 is surprisingly predictable.*

At this point, the user has a better understanding of how each instructions affect the program's execution time. Section 10.3.3 provides an auxiliary analysis, which focuses on the accuracy of memory elements instead.

10.3.3 Memory Element Accuracy Analysis

In addition of characterizing each instruction, the overall accuracy of accesses made to each memory elements can also be quantified in each cache. As with the instructions, this analysis reveals which memory elements are the cause of time variation, which are not affected by interference, and which are frequent causes of cache misses. Furthermore, some information on *uncategorized* instructions can sometimes be obtained by looking at the accuracy of memory elements.

Instead of categorizing the memory elements, this analysis looks at the minimum and maximum of cache hits and cache misses for each memory element in each cache. The UPPAAL queries to determine either the maximum or minimum of cache hits for the memory element at address 3 on Cache1 are as follow:

- **Maximum Number of Cache Hits:**
  sup: Cache1.cache_local_address_infos [3]. hit
- **Minimum Number of Cache Hits:**
  inf{Core1.Terminated and Core2.Terminated}: Cache1.cache_local_address_infos[3].hit

The Core1.Terminated and Core2.Terminated formula parameter used when calculating the minimum number of cache hits ensures that only values from after both programs have successfully completed are considered. Otherwise, the result would always be 0, as it is the initial and lowest value on all traces.

Combined with the categorization of instructions, this analysis can provide information on the *uncategorized* instructions accessing a memory element. Indeed, by removing the accesses pertaining to instructions classified as *always-hit* or *always-miss* from the results, the remaining numbers indicate the range of cache hits and cache misses distributed among all *uncategorized* instructions for this memory element.

Example 37 (Application to the Model of Section 10.1) *Using UPPAAL to query the extrema of both cache hits and cache misses on each cache for the model of Section 10.1 yields the results shown in Figures 10.6a and 10.6b. The first number in each cell corresponds to the result of the analysis for all accesses. The number between parenthesis subtracts the accesses which are not from uncategorized instructions according to Example 36.*

	Cache1		Cache2	
	Hit	Miss	Hit	Miss
Address 1	3 (2)	4 (2)	0 (0)	3 (0)
Address 2	2 (2)	4 (2)	1 (1)	2 (1)
Address 3	0 (0)	1 (0)	2 (0)	2 (0)

(a) Maximum

	Cache1		Cache2	
	Hit	Miss	Hit	Miss
Address 1	1 (0)	2 (0)	0 (0)	3 (0)
Address 2	0 (0)	2 (0)	1 (1)	2 (1)
Address 3	0 (0)	1 (0)	2 (0)	2 (0)

(b) Minimum

Figure 10.6 – Cache Hit/Miss Extrema for each Memory Element

The results of this analysis can be read as follows:

- *On Cache1, the address 1 has 1 always-hit and 2 always-miss instructions. Thus, the number corresponding to the accesses from uncategorized instructions are obtained by removing 1 from the Hit columns and 2 from the Miss columns.*
- *For address 2, the program that uses Cache1 has 2 always-miss instructions, which are thus removed from the Miss columns.*
- *Address 3 has no uncategorized accesses, so there is nothing remaining.*
- *Address 1 on Cache2 is only accessed by 3 always-miss, which are thus removed. This leaves 0 hit or miss for uncategorized accesses, as expected since there are none in Example 36.*
- *Address 2 on Cache2 only removes a single always-miss.*

From these results, the following observations can be made:

- *The memory element at address 1 is the main cause of cache misses.*
- *The memory element at address 3 is the least problematic one.*
- *The results of the analyses for Cache2 show the same values for both minimum and maximum cache hits and misses. This adds information to the categorization done in Example 36. Indeed, this means that despite being neither always-hit nor always-miss, the uncategorized instructions of Cache2 never both hit (minimum and maximum of 1 hit on any execution for uncategorized instructions) nor both miss (minimum and maximum of 1 miss on any execution for uncategorized instructions) on the same execution.*
- *On Cache1, the uncategorized accesses to address 1 can both be miss or hit on the same execution. Further analysis would be required in order to determine if every execution has them both hit or both miss.*
- *This also holds true from the uncategorized accesses to address 2 on Cache1.*

This auxiliary analysis provided some additional information on which memory elements should be the focus of the user when attempting to mitigate the interference generated by cache coherence. It also provided a way to obtain a bit more information on *uncategorized* instructions.

In order to understand what causes instructions to result in cache misses when cache coherence is active, the next section studies the external queries received by caches.

10.4 Defining Impact of External Queries

This section looks at how external queries interfere with caches and, consequently, with instructions using these caches. The results of the analyses performed in this section are not likely to be of direct interest to the user, since they have little control over queries themselves. However, this section introduces the concepts that define the interference generated by cache coherence and the analysis of queries is key to the more exploitable results of Section 10.5.

The characterization of the effects of each query on a cache provides information about what causes an instruction to result in a cache miss or to be delayed (which is not observable in the analyses of Section 10.3). In effect, two of the three proposed categories of interference specify which permissions are lost, which correlates to which instructions would miss if executed after the interference.

10.4.1 Minor Interference

No handling of an event by a cache is instantaneous: every time a cache has to process an incoming query, there is a very small amount of time during which it cannot be used by its core. As a result, all external queries lead to some kind of interference. This small delay being the least disruption that can be caused. While the effect of each delay is so small as to be considered negligible, their accumulation most definitely is not. The name *minor interference* is proposed for this unavailability period.

Definition 60 (Minor Interference) *Minor interference occurs whenever a cache becomes unavailable for core requests because it is handling an external bus query, yet the handling of that query had not effect.*

Example 38 (Minor Interference) *Figure 10.7 shows an example of minor interference. The figure indicates the current coherence state of the memory element involved for both caches, as well as their outgoing query queue. In that example, Cache2 has to process Cache1's* `GetS` *broadcast, despite that message not requiring any reply or coherence state update from Cache2.*

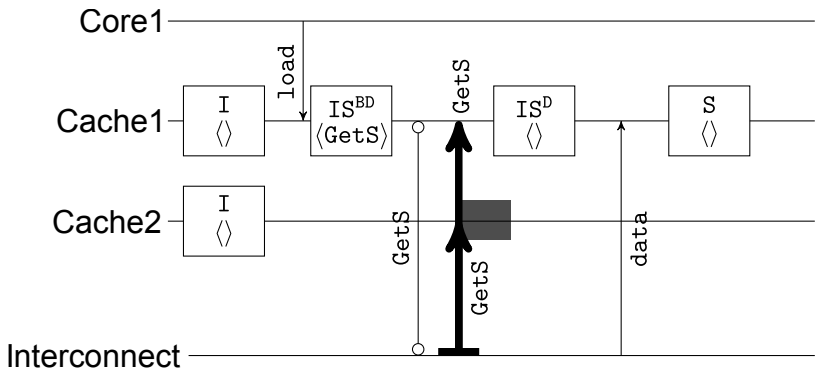


Figure 10.7 – Minor Cache Coherence Interference

Detection of this interference by the model is simply done by each cache considering any observed external query as an occurrence of minor interference. If the observed external query is later considered to be a different type of interference, it is removed from the count of minor interferences.

For a minor cache coherence interference to be considered to have had an impact, it must delay a request from a core. More specifically, the request has to become available for transmission from the core to the cache, but the cache be unavailable because it is already processing the interfering query. It is considered as having an impact solely on that delayed instruction.

While counting the number of occurrences is easily done, detecting that a minor interference had an impact on the program's execution would correspond to detecting that a synchronization for either data or request with a cache automaton (see Section 9.4) could not be performed because that automaton was in its S_2 location (waiting for the query handling time to pass). Doing so might be possible, but requires numerous changes in the model's automata, and is thus not currently supported. As a result, all occurrences of minor interferences are counted, but the model does not let the user know if they had an impact.

10.4.2 Demoting Interference

As explained in Section 3.2.2, cache coherence protocols do not allow a cache to hold a memory element with writing permissions while another cache holds that same memory element with reading permissions. Thus, acquisition of a copy the memory element by another cache leads to any cache currently holding it with writing permissions to lose them. The name *demoting interference* is proposed for this type of interference.

Definition 61 (Demoting Interference) *Demoting interference corresponds to the loss of writing permission for a memory element by a cache because of an external query, while reading permissions are kept.*

Example 39 (Demoting Interference) *Figure 10.8 shows an example of demoting interference: Cache2, receiving a demand for read access on that memory element from Cache1, has to update the value from the main memory and go from read-and-write permissions to read-only permissions.*

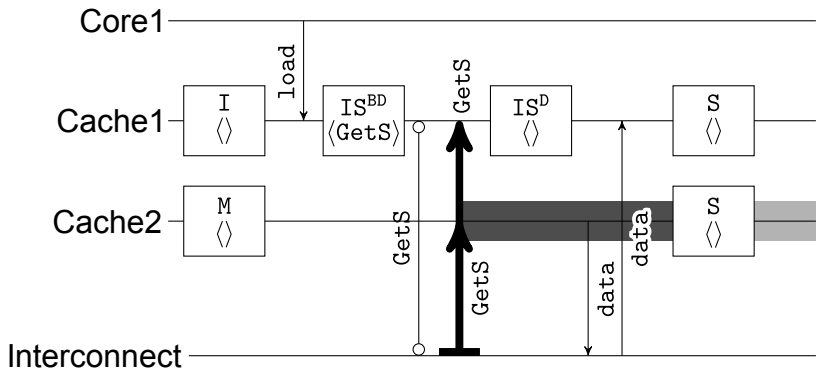


Figure 10.8 – Demoting Cache Coherence Interference

In order for a demoting interference to have had an impact, it must be followed by a `store` instruction, with any number of `load` in-between, but no `evict`. The demoting interference is only considered to have had an impact on that `store` instruction, not on the other instructions in-between nor on future instructions.

To detect demoting interference, the cache coherence protocol has to be annotated. Are annotated as demoting interference all transitions which make a memory element move from a coherence state in which `store` is a `hit` to one where it is not. To detect that a demoting interference had an impact on execution time, the model uses cache_local_address_infos to keep track of which memory elements have been affected by a demoting interference since they were last accessed for a `store`. If a `store` occurs on a memory element currently marked as such, the demoting interference is counted as having had an impact.

10.4.3 Expelling Interference

When a cache acquires writing permissions on a memory element, all other caches holding a copy of that memory element must discard it. The name *expelling interference* is proposed for this type of interference.

Definition 62 (Expelling Interference) *Expelling interference corresponds to the removal of a memory element from a cache following an external query.*

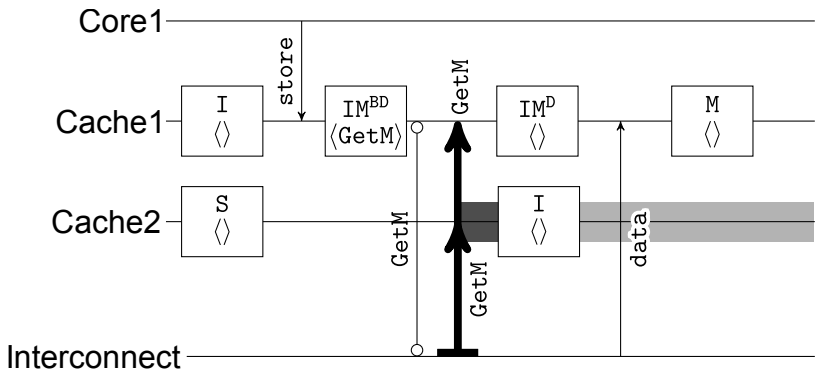


Figure 10.9 – Expelling Cache Coherence Interference

Example 40 (Expelling Interference) *Figure 10.9 shows an example of expelling interference: Cache2, receiving a demand for read-and-write access from Cache1, is forced to relinquish its read-only copy of the memory element.*

An expelling interference has an impact on the very first next instruction that is either a `store` or a `load` (not `evict`), and not any future instructions.

As with the previous type of coherence interference, detection of expelling interference is done by annotating the coherence protocol. This time, the annotation is done on transitions which make a memory element move from a coherence state in which `load` is a `hit` to one where it is not. Detection of expelling interference that affected the program's run-time is also done by using cache_local_address_infos to keep track of which memory elements have been affected by an expelling interference since they were last accessed. The interference is considered to have had an impact on execution time if either `load` or `store` occurs on a memory element currently marked as having been forcefully evicted. This may result in over-pessimism, as the `store` might not have been a `hit` regardless (e.g. expelling interference occurring in the `S` state, followed by `store`).

10.4.4 Protocol Annotations

Figure 10.10 shows the proposed interference annotation for the 3 coherence protocols presented in this thesis: MSI, MESI, and MESIF.

Explanations for the *Demoting* annotations:

- Receiving `GetS` external query for a memory element in the $\texttt{IM}^{\texttt{D}}$ state will prevent the cache from keeping the write permissions after the current `store` for this memory element completes, hence the *Demoting* annotations.
- In the `M` and `E` states, observing a `GetS` external query will result in the immediate loss of writing permissions for that memory element, which is why these transitions are marked with *Demoting* annotations.

State	Received Query		
	GetS	GetM	PutM
I	Mi.	Mi.	Mi.
IS^{BD}	Mi.	Mi.	Mi.
IS^{B}	Mi.	Mi.	
IS^{D}	Mi.	Ex.	
$IS^{D}I$	Mi.	Mi.	
IM^{BD}	Mi.	Mi.	Mi.
IM^{B}	Mi.	Mi.	Mi.
IM^{D}	De.	Ex.	
$IM^{D}I$	Mi.	Mi.	
$IM^{D}S$	Mi.	Ex.	
$IM^{D}SI$	Mi.	Mi.	
S	Mi.	Ex.	
SM^{BD}	Mi.	Ex.	
SM^{B}	Mi.	Mi.	
SM^{D}	De.	Ex.	
$SM^{D}I$	Mi.	Mi.	
$SM^{D}S$	Mi.	Ex.	
$SM^{D}SI$	Mi.	Mi.	
M	De.	Ex.	
MI^{B}	Ex.	Ex.	
II^{B}	Mi.	Mi.	Mi.

(a) MSI

State	Received Query		
	GetS	GetM	PutM
I	Mi.	Mi.	Mi.
IS^{BD}	Mi.	Mi.	Mi.
IS^{B}	Mi.	Mi.	
IS^{D}	Mi.	Ex.	
$IEoS^{D}$	Mi.	Ex.	
$IS^{D}I$	Mi.	Mi.	
IM^{BD}	Mi.	Mi.	Mi.
IM^{B}	Mi.	Mi.	Mi.
IM^{D}	De.	Ex.	
$IM^{D}I$	Mi.	Mi.	
$IM^{D}S$	Mi.	Ex.	
$IM^{D}SI$	Mi.	Mi.	
S	Mi.	Ex.	
SM^{BD}	Mi.	Ex.	
SM^{B}	Mi.	Mi.	
SM^{D}	De.	Ex.	
$SM^{D}I$	Mi.	Mi.	
$SM^{D}S$	Mi.	Ex.	
$SM^{D}SI$	Mi.	Mi.	
M	De.	Ex.	
MI^{B}	Ex.	Ex.	
II^{B}	Mi.	Mi.	Mi.
E	De.	Ex.	
IE^{B}	Mi.	Mi.	Mi.
EI^{B}	Ex.	Ex.	

(b) MESI

State	Received Query		
	GetS	GetM	PutM
I	Mi.	Mi.	Mi.
IF^{BD}	Mi.	Mi.	Mi.
IF^{B}	Mi.	Mi.	
$IEoF^{D}$	Mi.	Ex.	
IS^{D}	Mi.	Ex.	
$IS^{D}I$	Mi.	Mi.	
IM^{BD}	Mi.	Mi.	Mi.
IM^{B}	Mi.	Mi.	Mi.
IM^{D}	De.	Ex.	
$IM^{D}I$	Mi.	Mi.	
$IM^{D}S$	Mi.	Ex.	
$IM^{D}SI$	Mi.	Mi.	
S	Mi.	Ex.	
F	Mi.	Ex.	
SM^{BD}	Mi.	Ex.	
FM^{B}	Mi.	Mi.	
SM^{B}	Mi.	Mi.	
SM^{D}	De.	Ex.	
$SM^{D}I$	Mi.	Mi.	
$SM^{D}S$	Mi.	Ex.	
$SM^{D}SI$	Mi.	Mi.	
M	De.	Ex.	
MI^{B}	Ex.	Ex.	
II^{B}	Mi.	Mi.	Mi.
E	De.	Ex.	
IE^{B}	Mi.	Mi.	Mi.
EI^{B}	Ex.	Ex.	
FI^{B}	Mi.	Mi.	

(c) MESIF

Interference Type: Minor | Expelling | Demoting

Figure 10.10 – Interference Annotations

Explanations for the *Expelling* annotations:

- Observing a GetM query for a memory element in the IS^{D}, IM^{D}, SM^{D}, $IEoS^{D}$, $IEoF^{D}$, or $SM^{D}S$ state leads to states that ensure the memory element is evicted once this cache's current transaction for the memory element is completed. They are thus annotated as *Expelling* interference.
- Similarly, receiving a GetM query for a memory element in the S, M, E, or F leads to that memory being immediately evicted (and thus are also annotated as *Expelling* interference).
- The *Expelling* interference annotations on the transitions for MI^{B} and EI^{B} come from the fact that, until it observes any query for that memory element, the cache still has both read and

write permissions. These states were reached because the cache is already evicting the memory element, which makes considering its eviction an interference something that can be argued against. I chose to consider the permissions for future requests and not the result of past ones, hence the annotation.

Example 41 (Impact of External Queries on the Model from Section 10.1) *As previously stated, the direct application of the analyses proposed in this section are unlikely to produce exploitable results. Nevertheless, in order to illustrate the concepts, this subsection proposes an analysis of the queries generated by the example of Section 10.1.*

By using counters for occurrences of potential interference and those which had an impact, using model checking to obtain extrema may provide results that can be compared to those from Section 10.3. For example, the maximum number of occurrences for minor interference related to the memory element at address `target_addr` *on Cache1, the following query can be used:*

sup: Cache1. cache_local_address_infos [target_addr]. potential_interference_count [INTERFERENCE_MINOR].

	Cache1			Cache2		
	Minor	Demoting	Expelling	Minor	Demoting	Expelling
Address 1	1 (-)	1 (1)	2 (1)	1 (-)	1 (1)	2 (2)
Address 2	0 (-)	1 (1)	2 (2)	1 (-)	1 (0)	2 (1)
Address 3	1 (-)	0 (0)	1 (0)	0 (-)	1 (1)	0 (0)

(a) Maximum

	Cache1			Cache2		
	Minor	Demoting	Expelling	Minor	Demoting	Expelling
Address 1	0 (-)	1 (1)	1 (0)	0 (-)	0 (0)	2 (1)
Address 2	0 (-)	0 (0)	1 (1)	1 (-)	0 (0)	1 (1)
Address 3	1 (-)	0 (0)	1 (0)	0 (-)	1 (1)	0 (0)

(b) Minimum

Figure 10.11 – Interference occurrence count extrema for each memory element, in parenthesis is the number of occurrences that had an impact on execution times

Figure 10.11 shows the number of occurrences of each interference category, for each cache and each address. Two values are provided: the number of occurrences themselves and, in parenthesis, the number of occurrences that had an impact on execution time. As indicated in Section 10.4.1, the model is unable to determine which occurrences of minor interference had an impact on execution time, hence the lack of number for that category in the figure.

The results of Example 37 showed that the interference affected Cache2 only for the memory element at address 2, and that it led to one of two instructions being a cache hit in every execution where the other was a miss. The results from Figure 10.11 offer some further details, by showing that this is due to an expelling interference. Indeed, there is no variation between Figure 10.11a and Figure 10.11b for the number of occurrences of interference that had an impact on address 2.

Surprisingly, there is a variation for the number of expelling interference that can have an impact on address 1 in Cache2. This is not visible in the categorization from Figure 10.5b, as all accesses made to address 1 are either always-hit or always-miss. The reason for the extra interference not preventing this categorization is that, in effect, it had not impact. Indeed, performing a `store` *on a memory element in the* `Invalid` *or* `Shared` *state both result in a cache miss. However, if an*

external query evicted the memory element while it was in the `Shared` *state, this forced eviction is still considered as having had an impact on the* `store` *by the model.*

Looking at the results for Cache1 and the address 1, the difference between Figure 10.11a and Figure 10.11b only indicates a single expelling interference not always occurring. In Figure 10.5a however, the categorization for accesses to the address 1 failed on two instructions. From the fact that there is only this single expelling interference that can vary in all executions, these two instructions are either both affected or neither is. In effect, they are either both a cache hit, or both a cache miss.

The results for address 2 on the Cache1 show that there are two impactful instances of interference that may or may not happen in executions. Since the category of interference are not the same for the two instances, the results presented here are insufficient to determine a meaningful pattern. To resolve this, further model checking queries would have to be made in order to see if the two interference occurrences are somehow related.

In order to render the analysis of query interference more exploitable, the user would need to be made aware both with instruction was affected and which instruction generated the query. This is achieved in the next Section.

10.5 Analyzing Impact of Intructions on Instruction

The previous section defined the interference generated by cache coherence, and showed how the model could be used to expose it. However, knowing where and when the interference occurs isn't something that can readily be exploited.

This section proposes instead to perform analyses that will indicate for each program instruction which other instruction causes interference on it, and what kind. In effect, this corresponds to finding the sets S_A and S_E composed of $\langle I_o, E, I_t \rangle$ triplets, such that I_o corresponds to an instruction that causes an interference of type E on the instruction I_t. S_A contains all triplets for which the interference occurs in all executions, whereas S_E contains the triplets for which the interference occurs in at least one possible execution.

Combined with the results from Section 10.3.2, this provides a nearly complete understanding of the effects of cache coherence interference on instructions. The only missing information being some complex relations between occurrences of interference.

Obtaining such results using the model features indicated in Section 10.4 can be achieved by propagating an identifier for each instruction that generates a query all the way to the instruction that is affected by the interference of that query.

In effect, a triplet $\langle I_o, E, I_t \rangle$ is included in S_E if, and only if, given C_o and C_t the caches handling I_o and I_t respectively:

```
E<>(
    (C_t.completed_requests[0].instruction_addr == I_t)
    and C_t.completed_requests[0].interference_origin.author == C_o
    and C_t.completed_requests[0].interference_origin.iline == I_o
    and C_t.completed_requests[0].interference_type == E
)
```

with instruction_addr being the line of the affected instruction, interference_origin indicating an cache (author) from which the interference comes from, as well as the line (iline) of the instruction that generated it. interference_type determines if this is a *demoting* or *expelling* interference. As with the analyses of Section 10.3.2, the relevant information is stored in the representation of the request held in the cache C_t, and every request handled by a cache is assured to be found in its completed_requests[0] after it has been fully processed.

To determine if a triplet also belongs in S_A if, the following formula is used:

```
A[](
    (C_t.completed_requests[0].instruction_addr == I_t)
    imply (
        C_t.completed_requests[0].interference_origin.author == C_o
        and C_t.completed_requests[0].interference_origin.iline == I_o
        and C_t.completed_requests[0].interference_type == E
    )
)
```

Analyzing Strategy Optimization

Testing these two formulas for every possible triplets would lead to a combinatorial explosion. The following strategy is proposed to reduce the search space:

1. Instructions I_t not affected by interference can be removed from consideration. In effect, instructions I_t for which the interference origin stayed to its default nil value are removed from the search space. Testing for removal can be done using:

   ```
   A[](
       (C_t.completed_requests[0].instruction_addr == I_t) imply
       (C_t.completed_requests[0].interference_origin.author <= 0)
   )
   ```

   interference_origin .author = 0 being UPPAAL's initial value for that variable, and -1 the value used to represent nil.

2. At this point, all remaining I_t instructions are sure to be part of the results. For each of these I_t instructions, the other components of their tuple(s) have to be found. The relevant I_o instructions for each I_t instruction are those for which the associated C_o is author of the interference. Thus, for each I_t, only C_o that are author of an interference that affected I_t are kept. These C_o verify:

   ```
   E<>(C_t.completed_requests[0].interference_origin.author == C_o)
   ```

3. At this point, the relevant C_o caches for each I_t instructions are known. The search for the appropriate I_o instructions can be shortened by only considering those within a range obtained using the sup and inf operators. Indeed, the following query looks for the minimum line number for I_o instructions that generated an interference on I_t:

   ```
   inf{
       (C_t.completed_requests[0].instruction_addr == I_t)
       and (C_t.completed_requests[0].interference_origin.author == C_o)
   }: (C_t.completed_requests[0].interference_origin.iline
   ```

   By replacing inf with sup, the highest line number for an instruction I_o that generated an interference on I_t can be obtained.

4. At this point, the relevant I_t and C_o associations are known, but the I_o instructions still have a number of candidates which might not all be part of the result. Thus, each I_o candidate has to be tested individually in order to ensure it verifies:

   ```
   E<>(
       (C_t.completed_requests[0].instruction_addr == I_t)
       and C_t.completed_requests[0].interference_origin.author == C_o
       and C_t.completed_requests[0].interference_origin.iline == I_o
   )
   ```

5. At this point, the relevant I_t, C_o, and I_o are associated, and all that remains is to determine the type of interference E for each such association. As there are only two possible values for E, both should be tested in turn:

```
E<>(
    (C_t.completed_requests[0].instruction_addr == I_t)
    and C_t.completed_requests[0].interference_origin.author == C_o
    and C_t.completed_requests[0].interference_origin.iline == I_o
    and C_t.completed_requests[0].interference_type == E
)
```

6. At this point, all remaining associations form the tuples that constitute S_E. Checking if those tuples also belongs to S_A is done by testing:

```
A[](
    (C_t.completed_requests[0].instruction_addr == I_t)
    imply (
        C_t.completed_requests[0].interference_origin.author == C_o
        and C_t.completed_requests[0].interference_origin.iline == I_o
        and C_t.completed_requests[0].interference_type == E
    )
)
```

Example 42 (Instruction Interference on the Model from Section 10.1) *Figure 10.12 shows the interference between the instructions of the model from Section 10.1. The edges go from the instruction generating the interference to the one being affected.* EX *stands for Expelling interference and* DE *stands for Demoting. The dashed lines indicate interference that occurs only on some of the possible executions. The column on the left corresponds to the instructions of Core1's program. The column on the right corresponds the equivalent for Core2. The results from Example 36 have been copied to the figure in order to ease readability.*

The interference generated by cache coherence in the model from Section 10.1 can thus be understood to be affecting the instructions on Core1 as follows:

- *The first instruction on Core1 is always-miss, but not because of cache coherence. It can affect the fifth instruction on Core2 by generating an expelling interference on the memory element at address 1.*
- *The second instruction on Core1 is also always-miss, and does not generate or suffers interference other than minor one.*
- *The third instruction on Core1 is uncategorized as it can be affected by the first instruction on Core2 through an expelling interference. It is likely that this interference is linked to the one that can be generated by the first instruction of Core1, and that only one occurs in any execution.*
- *The fourth instruction on Core1 is also uncategorized, yet is not the target of any interference. This implies the accuracy of this instruction is determined by whether the third instruction is affected by interference or not. Since cache hits do not generate interference (as they do not generate queries), the potential expelling interference generated by this instruction can only occur if the third instruction was affected by interference. It is unclear whether the interference is generated whenever this fourth instruction is a miss or if it can be a miss yet not generate it.*

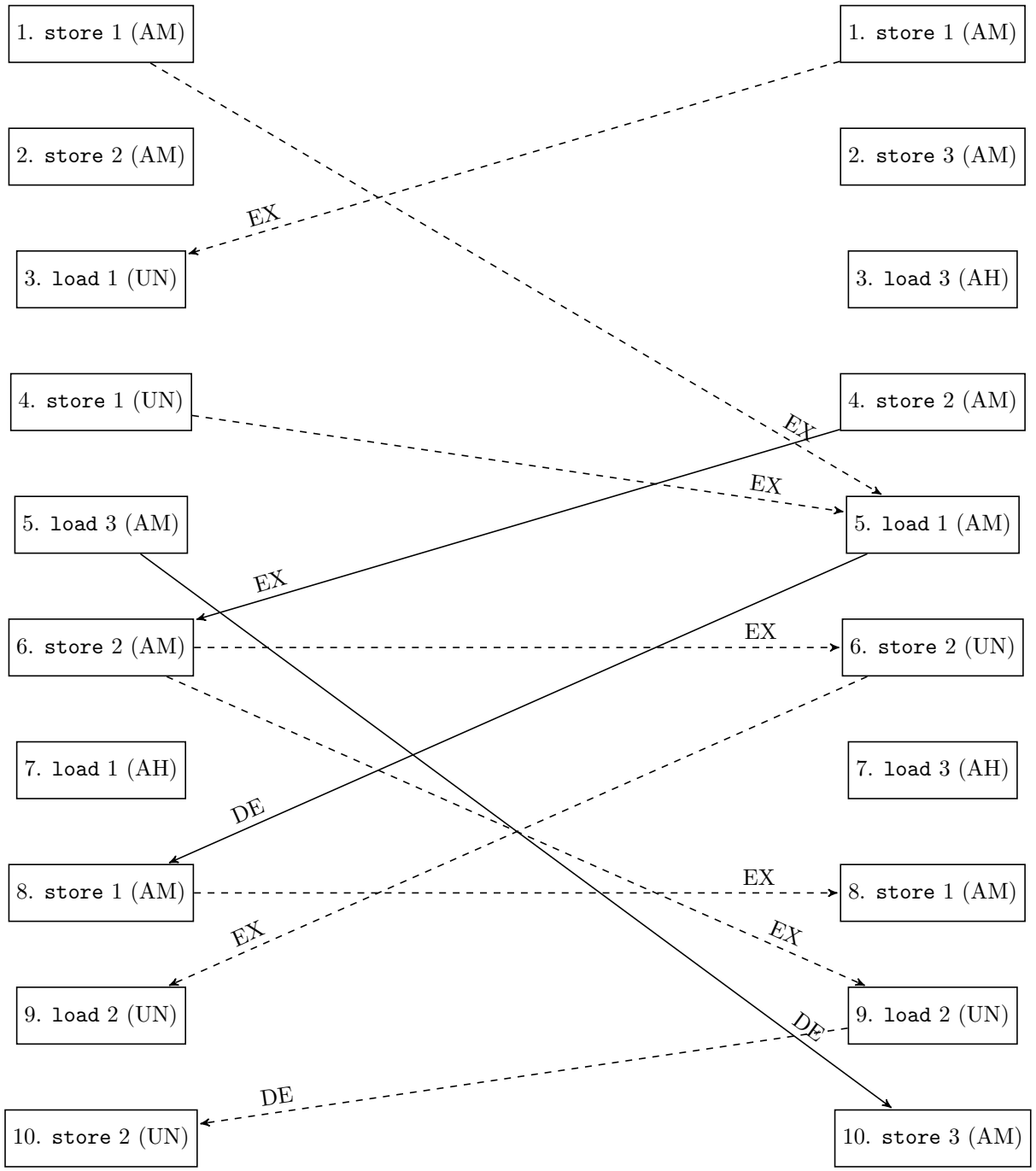


Figure 10.12 – Interference Between Instructions on Model from Section 10.1

- *The fifth instruction on Core1 is an always-miss that cannot be caused by interference. It always causes an interference for the tenth instruction on Core2, as it forces the permissions acquired by Core2's third instruction to be restricted.*
- *The sixth instruction on Core1 is an always-miss, as it always suffers from an expelling interference caused by Core2's fourth instruction. This instruction can affect Core2's ones in two places, either its sixth or its ninth. It is unclear whether it may also not generate any interference in some executions.*
- *The seventh instruction on Core1 is an always-hit, thus neither suffering from, nor generating interference.*
- *the eighth instruction on Core1 is an always-miss, as it always suffers from a demoting interference from Core2's fifth instruction. It can cause an expelling interference on the eighth instruction of Core2.*
- *The ninth instruction on Core1 is uncategorized, as it may suffer from an expelling interference from Core2's sixth instruction.*
- *The tenth and final instruction on Core1 is also uncategorized, as it can suffer from a demoting interference caused by the ninth instruction of Core2.*

Thus, the effects of interference on instructions are fully exposed. However, some uncertainties remain concerning the relation between each generated interference. For example, can the first instruction of both core generate interference in the same execution? This particular case can be easily intuited, but using model checking to ensure that any such relation is made known to the user is seemingly beyond UPPAAL's capabilities. Indeed, this would require the ability to write a formula checking that, for any given occurrences of interference, all can be found within the same execution. Since the detection of an interference requires a temporal formula, this would be a temporal formula containing a conjunction of temporal formula, which UPPAAL's query language does not support.

10.6 Model Checking Scalability Considerations

The analyses presented in this chapter rely on model checking. To validate that the model verifies a property, model checking gradually generates and explores all possible traces from the model. The time required for this exploration is the main limiting factor of the analyses of the chapter.

Indeed, checking the model described in Chapter 9 can become unreasonably time consuming. A part of the issue stems from the fact that, despite the efforts to reduce the amount of undesired non-determinism described in the previous chapter, some of it remains.

Instead of going over each property presented in this chapter, it uses the property that requires the largest exploration: In all traces, there is a state in which all cores have completed their program. This corresponds to the following formula:

```
A<>(Core1.terminated and Core2.terminated)
```

Figure 10.13 shows the execution time of the model checking process with varying number of cores in the modeled architecture. The 2 cores column corresponds to the model described in Section 10.1. The cores added for the 3 and 4 cores benchmarks are clones of the second core. The results clearly show that increasing the number of cores will quickly reach execution times beyond usable levels. This is not unexpected: even useful non-determinism such as the order of arbitration

Cores	2	3	4
Execution time (s)	0.593	12.453	425.109

Figure 10.13 – Model Checking execution time relative to number of cores

Program Size	10	20	30	40
Execution time (s)	0.593	1.297	1.906	2.266

Figure 10.14 – Model Checking execution time relative to length of programs

for the interconnect generate by itself a number of traces exponential to the number of cores (since all possible orders are explored).

Figure 10.14 shows how long model checking takes on the dual core from Section 10.1 depending on the number of instructions for the program on each core. 10 instructions corresponds to the original number of instructions. In order to add more instructions, the programs were extended by appending a copy of all their current instructions. The results show that increasing the length of programs does not lead to a rapid increase of model checking times.

10.7 Conclusion

This chapter has shown how UPPAAL's model checking capabilities can be exploited to analyze the interference caused by cache coherence on the model from Chapter 9.

This analysis starts by a computation of the WCET for each program. Useful in itself, this analysis is extended by that of the WCET for these programs with the architecture in different configurations in order to extract more information about how much of the execution time is caused by interference.

In order to more precisely understand what determine the WCET and to provide the user with information about elements of the program that can directly be manipulated, the analysis proceeds by an categorization of the accuracy of each instruction. This indicates which instructions are unaffected by the interference, which instructions are always time-consuming, and which instructions take a varying amount of time depending on the execution. By looking at the accuracy of all accesses made on each memory element, patterns for these instructions of varying execution time can sometimes be found, which results in a more predictable system.

The determining factor for the accuracy of instructions is then properly defined. This corresponds to a categorization of all external queries depending on their effects on the permissions held by a cache, and whether a loss of permission led to an instruction taking additional time. Thus, three categories of interference are defined: minor (no change of permission, but loss of time due to query processing), demoting (loss of writing permission), and expelling (loss of all permissions).

Finally, analyses are performed in order to determine how each instruction interfere with the other instructions. This results in a graph showing, for each instruction, which instruction can generate interference that will directly impact it, the category of this interference, and whether this interference occurs on all possible executions or not.

This provides the user with a clear understanding of the causes and effects of cache coherence interference on the programs' instructions, opening the way to finding means of mitigation for this interference.

Part IV

Conclusions & Perspectives

Chapter 11

Conclusion

The introduction of multi-core processors in critical avionic systems requires the ability to sufficiently predict their behavior so that certification of the system is achievable. Indeed, the parallel nature of multi-core processors leads to numerous interactions that do not strictly pertain to the objective of the applications being ran. This large amount of unprompted interactions render execution times difficult to predict and subject to large variations. Among the main causes of large time variations is cache coherence, the mechanism ensuring that data modifications performed by programs running in parallel are properly propagated. While cache coherence can be implemented with predictability in mind (see Section 6.2), this requires hardware modifications, precluding such solutions in an aeronautical context. Unfortunately, the available strategies to predict worst-case execution time of applications running on multi-core *Commercial Off-The-Shelf* processors require cache coherence to be deactivated (see Section 6.3 and Chapter 7).

This thesis proposes solutions to begin addressing the issue of cache coherence predictability in an aeronautical context. To do so, it provides the applicant with tools that can expose and explain the interference generated by cache coherence. In this chapter is provided a summary, the limitations, and suggestions of future improvements for each contribution made in this thesis.

11.1 Identifying the Protocol

The details of cache coherence mechanisms provided in an architecture's documentation do not go into sufficient details for certification purposes. The documentation can even be vague enough to cause the applicant to be misled about which protocol is implemented on the architecture. To remedy this, the first contribution made in this thesis is the cache coherence protocol identification process described in Chapter 8.

11.1.1 Summary

The proposed cache coherence protocol identification process relies on being able to observe the binary flags defining the state of cache lines, as well as having sufficient performance monitors to observe cache coherence related activity (although solutions such as the one presented in Section 5.3 may provide an alternative). The general idea being to use the documentation as a basis for the creation of a detailed hypothetical cache coherence, then validating it against observations made on the architecture by performing micro-benchmarks. These benchmarks first perform a reachability

analysis on cache states, then the hypothetical cache coherence is used to generate benchmarks corresponding to behaviors not exposed by the initial reachability analysis.

Successful application of the proposed strategy results in an ambiguity-free description of the architecture's cache coherence protocol (meaning a description following the notations presented in Chapter 3). This ambiguity-free description of the architecture's cache coherence protocol is an important information to have in order to prove that the effects of cache coherence on the system are under control. Indeed, performing benchmarks to measure access latency or bandwidth without this information is likely to result in the attribution of characteristics to the application of instructions on an mistaken system state. For example considering writing speed to be a certain value for memory elements seemingly in `Shared` in all caches of the system realizing that the protocol actually has a `Forward` state that will improve access speed.

To illustrate the need for cache coherence identification, the case of the NXP QorIQ T4240 is presented in Chapter 8. This architecture's documentation files indicate a MESI protocol (in the core documentation [25]) with *cache intervention* (in the motherboard family's documentation [26]). By applying of the cache coherence protocol identification process, the architecture is shown to implement a MESIF protocol, which is fair from being obvious from the aforementioned documentation.

11.1.2 Limitations

There are two main limitations to the cache coherence protocol identification strategy: the risk of an unreasonably large search space during the naive reachability analysis and the possibility of undetected behaviors.

The proposed naive reachability analysis performs up to $3 * \texttt{cc}$, benchmarks per observed system state (i.e. each instruction on each core, if no caches have the same state). Unfortunately, the number of observed system states can be needlessly high if the considered binary flags of cache lines contain quickly changing information not pertinent to cache coherence. For example, if flags meant to provide information to the cache replacement policy are unwittingly considered, the number of observed system states is likely to be exceedingly high.

The other limitation is that it is always possible for the architecture to have behaviors that were not observed by the benchmarks. Indeed, provided the system state is actually defined by components that cannot be observed, the naive reachability analysis might not encounter them. Likewise, nothing can be done to ensure the benchmarks guided by the hypothetical protocol would expose them either.

11.1.3 Future Works

The identification process would greatly benefit from the addition of a step in which the cache's binary flags are analyzed in order to remove what has no chance of being related to cache coherence.

The *Naught* library currently only performs a single benchmark per execution. It should be possible to improve it in order to have all benchmarks for an observable system state be performed in a single execution.

Naught could further be improved to perform the complete identification process in a single execution, if given the ability to look at the binary flags of cache lines. This would render the application of this identification process trivial and greatly improve its chances at being adopted as a common practice.

11.2 Modeling the Architecture

In order to ensure all possible cache coherence interference is taken into account, this thesis relies on formal methods to explore all possible behaviors of the architecture. This requires the creation of a model for the architecture and the programs.

11.2.1 Summary

Following the solution chosen by the papers in Chapter 7, the model is created using UPPAAL timed automata. Thus, Chapter 9 presents the network af timed automata that model an architecture implementing cache coherence.

Each of the architecture's components is represented by its own timed automaton, with synchronizations being used to interact with the other automata. This results is a model which is modular and composed of fairly small and readable automata. The timed automata formalism makes it very easy to model behaviors governed by time constraints, which is important considering that disruptions to execution time is effectively what is under study in this thesis.

The model's modularity makes it easy for the applicant to tailor it to the chosen architecture, as components can be added and removed without having to understand the model in its entirety. Furthermore, some of the model's characteristics can be made to match the architecture through a collection of parameters listed in Appendix B.

The most complex part of the model is the cache coherence protocol. However, the applicant does not have to understand how it is modeled in order to modify or replace it. Indeed, this thesis provides a tool that will transform the model to match a cache coherence protocol described in the formalism from Chapter 3.

11.2.2 Limitations

The way programs are modeled in Chapter 9 is limited to a sequence of memory accesses with a minimal and maximal delay to be respected afterwards. This corresponds to the information relevant to cache coherence, but excludes the modeling of any realistic application. Furthermore, UPPAAL's handling of priorities and `urgent` transitions may prevent the maximal delay from being considered: while being able to fire transitions with higher priority will indeed prevent lower priorities ones from firing, any `urgent` modifier from those blocked lower priorities transitions is still considered to be in effect. This will thus prevent an automaton from spending more time in a location if it can currently leave the location using a high priority transition and a lower priority transition with the `urgent` modifier is only prevented from being fireable because of priorities. This is not an issue that can be easily mitigated, as priorities are a necessary and complex aspect of the model.

Another source of limitation was from incorrect assumptions on my part of what cache coherence protocols could reasonably do. Indeed, Section 6.2.1 presents a cache coherence protocol which emits queries in reaction to incoming queries and data message. The traditional MSI protocols (MESI, MESIF, MOSI, MOESI, ...) only send emit queries because of core requests, and the model exploits this assumption to simplify the cache's automaton. As a result, the model is unable to use the protocol from Section 6.2.1 or other protocols with similarly unconventional features.

The model described in Chapter 9 only has partial support for multiple cores using the same cache. Indeed, while such configurations can easily be made because of the model's modularity, they are not considered to be within the scope defined in Chapter 4. The un-stalling procedure described in Section 9.4.4 does not account for the possibility of requests being sent by different cores.

Caches are limited to a single cache placement (fully associative) and replacement (LRU) policy. While the fully associative cache placement is not unrealistic, the LRU replacement policy is not generally used on real architectures, a pseudo-LRU policy being the most commonly used policy.

11.2.3 Future Works

It would be interesting to modify the model in order to ensure that any cache coherence protocol that can be described by the notations from Chapter 3 can also be modeled. This is not a minor modification: for example, the notations allow any number of queries to be sent during a transition in the protocol's cache automaton, but UPPAAL does not have dynamically sized lists.

The papers presented in Chapter 7 create the model for programs directly from that program's binary executable. Even with architectures using reduced instructions sets (*RISC* such as *ARM* processors), this requires a considerable amount of work, as the the behavior of each available assembly instruction must be modeled in UPPAL. It would, however, make the model able to be used on realistic programs.

The automata for caches can still be improved. For example, all commonly used cache placement and replacement policies should be added as options. Likewise, the support for multiple cores per cache would allow more architectures to be modeled.

A more difficult improvement, yet not less important, would be adding support for cache hierarchies to the model. Indeed, these are found in just about every architecture.

In its current state, the model only features components directly related to cache coherence. Components such as the pipelines described in the papers from Chapter 7 are not considered. While the programs as they are currently modeled would not be able to make use of the addition of models for the pipelines, the aforementioned improvement to an actual instruction set would benefit from the addition of pipelines. In effect, the modular nature of UPPAAL timed automata means that adapting the pipeline models from papers in Chapter 7 to the model from Chapter 9 should not present a challenge in itself.

The addition of automata for some components which are more rarely used but interact with cache coherence, such as DMAs, would also benefit the model.

Ideally, the applicant should not have to fiddle with the UPPAAL model directly. Indeed, it would be much more user-friendly to provide tools of a nature similar to the protocol switcher that would take an architecture description as input in order to generate a model that matches it.

11.3 Exposing the Interference

By making a model that matches the applicant's architecture through profiling benchmarks, model checking can be used to expose the interference generated by cache coherence.

11.3.1 Summary

Chapter 10 proposes analyses to reveal how programs are affected by cache coherence interference, and what generates it. These analyses start by a simple computation of the execution time for each program. In order to determine how much of this execution time is caused by cache coherence, an alternative of each program is created, in which no shared variable exists. This alternative does not have to correspond to anything that would realistically run on the actual architecture. Indeed, the point is to obtain the execution time of each of these alternative program and compare it to the original: the difference indicates how much of this program's execution time can be attributed to cache coherence.

The chapter proceeds by seeking information on the determining factor of execution time: the accuracy of instructions performing memory accesses. To do so, it uses model checking to categorize each instruction as being either *always-hit* (the data it uses sure to be available in the cache), *always-miss* (the data it uses sure to not be available in the cache), and *uncategorized* (there are executions in which the data is available, others where it isn't). This, in effect, points out which instructions cannot be hoped to perform faster, which should be the focus on improvement, and which cause execution time variations. An auxiliary analysis is also proposed, providing similar observations with a focus on the accuracy of memory elements instead.

To determine how cache coherence affected the accuracy of instructions, the chapter follows up by proposing categories defining the effects of external queries on the permissions held by a cache. Three categories are proposed: minor interference (no change of permissions, only incurrence of query processing times), demoting interference (loss of writing permissions, but reading permissions are kept), and expelling interference (all permissions are lost). The chapter points out where each category of interference occurs on the MSI, MESI, and MESIF protocols. It also indicates what should be considered to be an occurrence of interference that has an impact on execution time.

By tracing the source of the external query back to its originating instruction, it becomes possible to use model checking in order to determine the effects of each instruction on the other instructions. In effect, this last analyze provides the user with an understanding of, for each instruction, which instructions can cause an interference, the category (and thus, effects) of this interference, as well as if this interference occurs on every possible execution or just some of them.

11.3.2 Limitations

This thesis proposes the use of model checking to expose interference. The undeniable limitation of this approach is scalability. Indeed, the proposed analyses have only been performed on toy examples. Realistic programs are likely to be at least a hundredfold larger, and so is the number of cache lines in each cache. The number of possible executions would then result in exponentially more time and memory consumption for each analysis.

Another barrier to real-world use of the approach is the lack of automation. Indeed, these analyses require a large number of queries to be performed manually by the user. While the execution time analysis can reasonably be undertaken on large models, the categorization of each instruction takes between one and two query per instruction. Worse, obtaining the interference incurred by each instruction can take up to n^m queries, with n being the size of programs, and m the number of caches.

The available model and model checking formula do not provide the means to expose occurrences of minor interference that had an impact on execution time.

The results provide information about interference that happens on every possible execution. It also provides information on interference that happens in at least some executions. However, there is no established strategy to expose the relations between occurrences of interference that are not present on every execution. For example, the proposed formula do not provide any means to check if a set of three arbitrarily chosen occurrences of interference can all happen in the same execution.

11.3.3 Future Works

Creating tools to make all these analyses automatically should be the next step in user-friendliness. Indeed, UPPAAL can be invoked from the command line, and all the involved files are text-based, making it possible to parse and create the appropriate results and model checking queries.

Support for exposing occurrences of minor interference that have an impact on execution time might be feasible. However, considering the limited effects of minor interference on a singular instruction, this information is likely superfluous to the user.

On the other hand, being able to indicate the relations between the occurrences of interference that do not happen on all executions would provide useful information. The model is technically providing all the necessary variables to write formulas that would indicate such relations. However, these formula would require temporal operators that can be employed on temporal formula, instead of the restricted variant of computation tree logic supported by UPPAAL.

11.4 General Future Works

Overall, the framework proposed in this thesis would greatly benefit from further automation. Indeed, while the cache coherence protocol obtained through the identification process can already be automatically inserted into the model for analysis, there is still a need for numerous user actions in every contribution proposed in this thesis. Ideally, the user should only have to provide a description of the architecture through a dedicated description language and the program binaries. The framework would then generate a template for the coherence protocol identification in which only architecture specific instructions would have to be filled in by the user. The model would be automatically modified to match the architecture's description. It would also use the programs' binaries to generate corresponding automata (such as is done in the works presented in Chapter 7). The cache coherence interference analyzes would then automatically be performed, providing the user with the results.

Another point which should be the focus of future works is the range of supported architectures. This would start by taking into account more components in the model, such as the pipelines present in the approaches from Chapter 7. If programs are analyzed using their binary forms, support for different instruction sets would become a deciding factor in the usability of the framework.

Lastly, future works could involve the integration of this framework within a more general analysis of the multi-core architecture, exporting the results on cache coherence interference in a way that can be exploited by existing tools. For example, being able to interface with OTAWA [4] in order to compute more accurate WCETs.

Chapitre 12

Résumé en Français

12.1 Introduction

12.1.1 Contexte

La complexité grandissante des fonctionnalités embarquées dans les avions et l'obsolescence progressive des processeurs mono-cœur dans le commerce entrainent un besoin croissant vers l'adoption de processeurs multi-cœurs dans les systèmes aéronautiques. Or pour voler, un avion doit passer la certification, ce qui signifie qu'un avionneur postulant – que nous appellerons dans la suite un *applicant* – doit présenter un dossier argumentant que le système dans son ensemble est conforme à la règlementation en vigueur. Le cas des multi-cœurs est assez spécial et nouveau car un nouveau standard a vu le jour récemment le CAST-32A ([17]). Ce document est focalisé sur les particularités propres aux processeurs multi-cœurs. Cette thèse a été financée par le projet Phylog [9], dont l'objectif est de fournir à l'applicant une méthodologie outillée lui permettant de préparer un dossier de certification pour des systèmes s'exécutant sur des cibles multi-cœurs et donc de répondre au CAST-32A. Parmi les objectifs du CAST32-A, le *Resource Usage 3 (RU3)* nous intéresse particulièrement dans cette thèse. En effet, RU3 correspond au fait que l'applicant a identifié les canaux d'interférence qui pourraient affecter les applications hébergées sur les cœurs et qu'il a validé sa stratégie pour en atténuer les effets nocifs. Afin d'aider l'applicant à remplir cet objectif, cette thèse se concentre sur les interférences générées par *la cohérence de cache.*

Definition 63 (Interférence) *Une interférence est une modification indésirable du temps d'exécution d'une application provoquée par les actions d'une autre s'exécutant sur un cœur distant.*

Afin de répondre à *RU3*, l'applicant doit identifier les sources d'interférence et quantifier leur impact sur les applications. Cela nécessite une bonne compréhension des mécanismes présents sur l'architecture choisie. Idéalement, la solution serait de simplement consulter la documentation du processeur, où tous les mécanismes seraient décrits en détails, ainsi que tous les comportements qui induiraient de l'interférence. Cependant, la documentation des processeurs COTS n'inclut généralement pas de détails sur la cohérence de cache. C'est donc une première problématique adressée par cette thèse. Une fois cette étape d'identification menée, il faut comprendre les effets des interférences.

Definition 64 (Impact associé à une interférence) *L'impact d'une interférence est une quantification des effets temporels sur l'exécution d'une application. Plus précisément, il s'agit de la quantité de cycles processeur requis pour l'exécution d'un fragment de l'application avec des autres applications s'exécutant en parallèle comparée à son exécution en isolation.*

Prédire l'impact d'une interférence sur l'exécution des programmes n'est pas simple et il n'existe à ce jour aucune méthodologie pour répondre à cette question. La littérature propose soit des stratégies d'analyse très pessimistes ou soit des restrictions pour l'élimination des sources d'interférence pour ne pas à avoir à prendre leurs effets en compte. De plus aucune analyse ne prend en charge la cohérence de cache.

12.1.2 Contributions

Cette thèse porte sur l'identification des interférences générées par les mécanismes de cohérence de caches ainsi que sur les moyens de prédiction de leurs effets sur les applications en vue de réduire les effets négatifs temporels. Seules les architectures COTS avec cohérence de cache sont considérées. De plus, parmi les protocoles de cohérence de cache, nous nous concentrons sur ceux dits *snooping* car les plus répandus dans les architectures. D'autres restrictions ont été supposées dans le cadre de la thèse : pas de hiérarchie de cache, un cache local par cœur, politique de placement associative et politique de remplacement LRU. Un exemple d'architecture est montrée par la Figure 12.1.

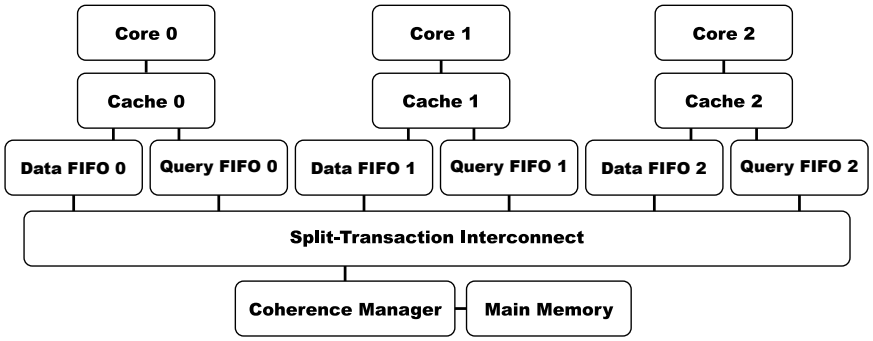


FIGURE 12.1 – Profil typique d'architecture visée

L'objectif de la thèse est de définir une méthodologie outillée de compréhension et analyse des interférences générées par la cohérence de cache sur une architecture multi-cœur COTS. La Figure 12.2 présente le framework proposé.

La première contribution adresse les ambiguïtés dans la compréhension que les applicants ont de la cohérence de cache réellement présente dans l'architecture. En effet, la documentation des architectures ne fournit généralement pas suffisamment de détails sur les protocoles. Cette thèse propose une formalisation des protocoles standards, ainsi qu'une stratégie, reposant sur les micro-benchmarks, pour clarifier les choix d'implémentation du protocole de cohérence présent sur l'architecture. Cette stratégie a notamment été appliquée sur le NXP QorIQ T4240.

Une fois le protocole correctement identifié, les techniques existantes de mesure de performance peuvent être appliquées afin d'obtenir des informations sur la performance des mécanismes de cohérence de cache identifiés. Cela ne constitue pas une contribution, d'où la coloration différente de cette étape dans la Figure 12.2.

La seconde contribution consiste à réaliser une description bas-niveau de l'architecture en utilisant des automates temporisés afin de représenter convenablement les micro-comportements et comprendre clairement comment le protocole de cohérence de cache agit. Ainsi, un framework de génération de modèles génériques a été développé, capable de supporter plusieurs protocoles de

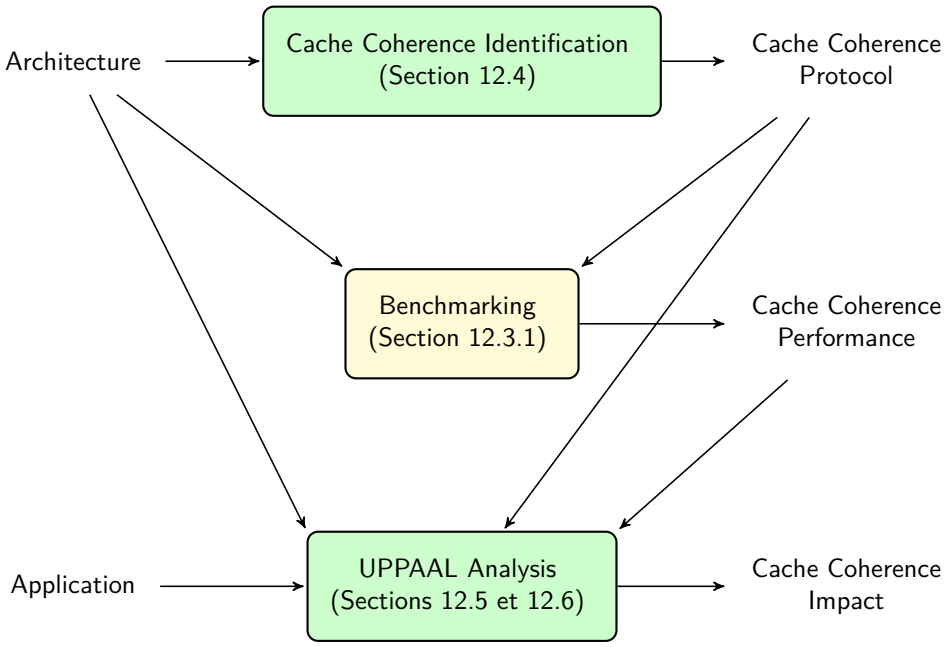


FIGURE 12.2 – Vue d'ensemble de l'approche

cohérence de cache et de représenter différents agencements d'architectures afin de mieux correspondre à l'architecture choisie par l'applicant.

La troisième contribution explique comment utiliser cette représentation de l'architecture pour exhiber les interférences. Elle propose une stratégie pour détailler les causes et effets de chaque interférence liée à la cohérence de caches sur les programmes. Commençant par une simple analyse de temps d'exécution, les résultats descendent jusqu'au niveau des instructions pour indiquer comment chaque instruction génère et souffre des interférences. L'objectif étant alors de fournir suffisamment d'information à l'applicant à la fois pour la certification, mais aussi pour définir une stratégie d'atténuation et de maîtrise des effets temporels.

12.1.3 Vue d'ensemble du résumé

Ce résumé commence par les préliminaires nécessaires à la compréhension de la problématique et des solutions proposées : les automates temporisés (Section 12.2.1) et la cohérence de cache (Section 12.2.2). Une fois ces notions présentées, le résumé présentera rapidement l'état de l'art (Section 12.3).

Les trois contributions apportées par cette thèse sont ensuite détaillées : une stratégie d'identification détaillée du protocole de cache utilisé par l'architecture (Section 12.4) ; un template de modèle d'architecture multi-cœurs supportant la cohérence de cache (Section 12.5) ; et les analyses à faire sur les modèles instanciés afin déterminer les causes et effets des interférences (Section 12.6).

12.2 Notions préliminaires

12.2.1 Automates temporisés

Les automates temporisés seront utilisés dans la suite pour réaliser un modèle formel de des architectures. Un automate temporisé est un automate étendu permettant en outre de modéliser le temps, et ce en ajoutant un nouveau type de variable appelé horloge. On rappelle qu'un automate étendu est un automate pouvant manipuler des variables entières.

Definition 65 (Horloges) *Une horloge est une variable modélisant le passage du temps. La valeur d'une horloge ne croit que dans les états car les transitions sont instantanées. La valeur d'une horloge peut être testée sur une garde d'une transition et remise à zéro après franchissement d'une transition.*

Definition 66 (Syntaxe des contraintes et des actions) *Étant donné un ensemble de variables* $\texttt{Var}$*, et un ensemble d'horloges* $\texttt{Clocks}$*, la grammaire utilisée pour l'écriture des contraintes et des actions de transitions est la suivante, avec* ***ident*** *indiquant une variable dans* $\texttt{Var}$ *et* ***clk*** *une horloge dans* $\texttt{Clocks}$ *:*
$\boldsymbol{lop} ::= \wedge \mid \vee$
$\boldsymbol{cop} ::= < \mid \leq \mid = \mid \geq \mid >$
$\boldsymbol{mop} ::= + \mid - \mid * \mid /$
$\boldsymbol{val} ::= \boldsymbol{ident} \mid \mathbb{Z}$
$\boldsymbol{mexpr} ::= \boldsymbol{mexpr\ mop\ mexpr} \mid \boldsymbol{val}$
$\boldsymbol{abexpr} ::= \boldsymbol{mexpr\ cop\ mexpr} \mid \boldsymbol{clk\ cop\ val} \mid \boldsymbol{clk - clk\ cop\ val} \mid \texttt{true} \mid \texttt{false}$
$\boldsymbol{bexpr} ::= \neg \boldsymbol{bexpr} \mid \boldsymbol{bexpr\ lop\ bexpr} \mid \boldsymbol{abexpr}$
$\boldsymbol{iexpr} ::= \boldsymbol{iexpr} \wedge \boldsymbol{iexpr} \mid \boldsymbol{clk\ cop\ val} \mid \boldsymbol{clk - clk\ cop\ val} \mid \texttt{true}$
$\boldsymbol{assign} ::= \boldsymbol{assign};\ \boldsymbol{assign} \mid \boldsymbol{ident} := \boldsymbol{mexpr} \mid if\ (\boldsymbol{bexpr})\ \{\boldsymbol{assign}\} \mid \boldsymbol{clk} := 0 \mid \boldsymbol{nop}$

Definition 67 (Automates temporisés) *Un automate temporisé* $\mathcal{A}$ *est un tuple* $\langle Q, \texttt{Inv}_Q, s_{init}, \mathcal{B}, \mathcal{E}, \texttt{Prio}_\mathcal{E}, \texttt{Var}, \texttt{Clocks}, \texttt{Act}, \rightsquigarrow \rangle$ *tel que :*

- Q *est un ensemble fini de localités.*
- $\texttt{Inv}_Q : Q \rightarrow \boldsymbol{iexpr}$ *indique l'invariant associé à chaque localité.*
- s_{init} *est la localité initiale* ($s_{init} \in Q$).
- $\texttt{Var}$ *est un ensemble fini de variables.*
- $\texttt{Clocks}$ *est un ensemble fini d'horloges.*
- $\mathcal{E} = \mathcal{E}^\alpha \cup \mathcal{E}^{sync}$ *est un ensemble fini d'étiquettes, avec* $\mathcal{E}^{sync}$ *correspondant aux étiquettes de synchronisation et* $\mathcal{E}^\alpha$ *aux autres.* $\mathcal{E}^{sync} \cap \mathcal{E}^\alpha = \emptyset$*. Les étiquettes dans* $\mathcal{E}^{sync}$ *terminent soit par par '?' qui indique la réception sur un "canal", ou par '!' qui indique l'émission.*
- $\mathcal{B} = \boldsymbol{bexpr}(\texttt{Var}, \texttt{Clocks})$ *est l'ensemble des gardes, utilisant la grammaire de Définition 66.*
- $\texttt{Act} = \boldsymbol{assign}(\texttt{Var}, \texttt{Clocks})$*, est l'ensemble des actions, utilisant la grammaire de Définition 66.*
- $\rightsquigarrow \subseteq Q \times \mathcal{B} \times \mathcal{E} \times \texttt{Act} \times Q$ *est la relation de transition.*

La sémantique de $\mathcal{A}$ *est donnée à travers ses traces d'exécution (voir Définition 71).*

Definition 68 (Valuation d'horloge) *La fonction* $h : \texttt{Clocks} \rightarrow \mathbb{R}^+$ *assigne une valuation à chaque horloge. On note* $(h + t)$ *comme abréviation pour indiquer l'incrément de la valeur de toutes les horloges dans* h *de* t *unités de temps.*

Definition 69 (Valuation des variables) *Les valuations* $v : \texttt{Var} \rightarrow \mathbb{N}$ *associent aux variables leur valeur. Étant donné une valuation* v *et une garde* $c \in \mathcal{B}$*, on note* $v \models_{PL} c$ *pour indiquer que* c

est vraie selon la valuation v. De même, étant donné $a \in \texttt{Act}$, $v[a]$ correspond à la valuation obtenue depuis v par l'application de l'action a : toutes les variables changées par a ont leur nouvelle valeur et toutes les autres gardent leur valeur précédente.

Definition 70 (Transition) *Étant donné un automate $\mathcal{A} = \langle Q, \texttt{Inv}_Q, s_{init}, \mathcal{B}, \mathcal{E}, \texttt{Prio}_\mathcal{E}, \texttt{Var}, \texttt{Clocks}, \texttt{Act}, \rightsquigarrow \rangle$, on définit $\texttt{Step}$ permettant de calculer l'ensemble des transitions valides depuis $\langle s, v, h\rangle$, avec $s \in Q$, v une valuation des variables, h une valuation des horloges et t le temps passé dans la localité courante :* $\texttt{Step}(\langle s, v, h\rangle, t) \triangleq \{\langle s', v', h'\rangle | \exists \langle s, c, l, a, s'\rangle \in \rightsquigarrow \text{ s.t. } ((\langle v, (h+t)\rangle \models_{PL} c) \wedge v' = v[a] \wedge h' = (h+t)[a] \wedge (\langle v', h'\rangle \models_{PL} \texttt{Inv}_Q(s')) \wedge \neg \exists \langle s_b, c_b, l_b, a_b, s'_b\rangle \in \rightsquigarrow \text{ s.t. } ((\langle v, (h+t)\rangle \models_{PL} c_b) \wedge v'' = v[a_b] \wedge h'' = (h+t)[a_b] \wedge (\langle v''_b, h''_b\rangle \models_{PL} \texttt{Inv}_Q(s'_b)) \wedge (l \in \texttt{Prio}_\mathcal{E}(l_b))))\}$

Definition 71 (Chemin et trace) *Un chemin est une séquence maximale de transitions $\langle s_0, v_0, h_0\rangle \rightarrow^{t_0} \langle s_1, v_1, h_1\rangle \rightarrow^{t_1} \cdots$ telle que pour chaque k, $\langle s_{k+1}, v_{k+1}, h_{k+1}\rangle \in \texttt{Step}(\langle s_k, v_k, h_k\rangle, t_k)$. La séquence est maximale dans le sens où elle est soit infinie soit de longueur N et telle que $\texttt{Step}(\langle s_N, v_N, h_N\rangle, t)$ est vide pour tout $t \in \mathbb{R}^+$. On appelle trace un chemin partant de l'état initial $\langle s_{init}, v_0, h_0\rangle$, avec v_0 la valuation initiale et h_0 la valuation telle que toutes les horloges sont à zéro.*

Definition 72 (Produit synchronisé d'automates temporisés) *Étant donné* n *automates temporisés $\mathcal{A}_i = \langle Q_i, \texttt{Inv}_{Qi}, s_{initi}, \mathcal{B}_i, \mathcal{E}_i, \texttt{Var}_i, \texttt{Clocks}, \texttt{Act}_i, \rightsquigarrow_i\rangle$ et une contrainte de synchronisation $\texttt{Sync}_{con} \subseteq (\mathcal{E}_1 \cup \{-\}) \times \cdots \times (\mathcal{E}_n \cup \{-\})$, le produit synchronisé des automates est l'automate $\mathcal{A}_s = \langle Q_s, \texttt{Inv}_{Qs}, s_{inits}, \mathcal{B}_s, \mathcal{E}_s, \texttt{Var}_s, \texttt{Clocks}_s, \texttt{Act}_s, \rightsquigarrow\rangle$ avec :*

- $Q_s = Q_1 \times \cdots \times Q_n$
- $\texttt{Inv}_{Qs}(Q_1 \times \cdots \times Q_n) = \texttt{Inv}_{Q1}(Q_1) \wedge \cdots \wedge \texttt{Inv}_{Qn}(Q_n)$
- $s_{inits} = \langle s_{init1}, \cdots, s_{initn}\rangle$
- $\mathcal{B}_s = \mathcal{B}_1 \times \cdots \times \mathcal{B}_n$
- $\mathcal{E}_s = (\mathcal{E}_1 \cup \{-\}) \times \cdots \times (\mathcal{E}_n \cup \{-\})$. *On ajoute aux étiquettes la notation $-$ pour indiquer qu'un sous-automate ne fait pas de transition,*
- $\texttt{Var}_s = \texttt{Var}_1 \cup \cdots \cup \texttt{Var}_n$, *avec* $\forall i, j \in 1 \ldots n,\ (i \neq j) \implies (\texttt{Var}_i \cap \texttt{Var}_j = \emptyset)$
- $\texttt{Clocks}_s = \texttt{Clocks}_1 \cup \cdots \cup \texttt{Clocks}_n$
- $\texttt{Act}_s = \texttt{Act}_1 \times \cdots \times \texttt{Act}_n$
- $\rightsquigarrow_s \subseteq Q_s \times \mathcal{B}_s \times \mathcal{E}_s \times \texttt{Act}_s \times Q_s$, *avec*

$$\langle\langle o_1, \cdots, o_n\rangle, \langle c_1, \cdots, c_n\rangle, \langle l_1, \cdots, l_n\rangle, \langle a_1, \cdots, a_n\rangle, \langle d_1, \cdots, d_n\rangle\rangle \in \rightsquigarrow_s$$

$$\iff \begin{cases} \langle l_1, \cdots, l_n\rangle \in \texttt{Sync}_{con} \\ \langle \forall i \in 1 \ldots n : o_i, c_i, l_i, a_i, d_i\rangle \in \rightsquigarrow_i \vee (o_i = d_i \wedge l_i = - \wedge c_i = \texttt{true} \wedge a_i = nop) \end{cases}$$

La sémantique d'un produit synchronisé est exprimée comme la sémantique d'un automate.

Example 43 (Automates temporisés) *La Figure 12.3 montre deux automates modélisant un client (sur la gauche) qui récupère des fichiers depuis un serveur (sur la droite). Dans ce scénario, le système boucle infiniment : le client initialise une demande de fichiers (**request_files**) et compte (fetched) leurs arrivées (**new_file**) jusqu'à ce que le serveur indique que tout a été transféré (**done**). A chaque requête, le serveur envoie exactement 386 fichiers (comptés par sent). Le serveur prend entre 32 et 64 unités de temps pour fournir chaque fichier, ce qui permet de modéliser les temps de transfert.*

Voici un extrait de trace valide pour ce réseau d'automates :

$$\langle\langle S_0, S_0\rangle, \{\langle first, \top\rangle, \langle fetched, 0\rangle, \langle sent, 0\rangle\}, \{\langle C_0, 0\rangle, \langle C_1, 0\rangle\}\rangle \xrightarrow{\langle \texttt{request\_file !}, \texttt{request\_file ?}\rangle}{}^{23}$$

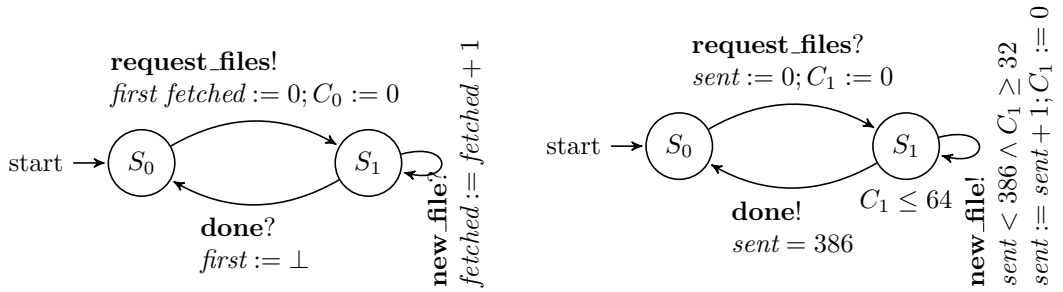


FIGURE 12.3 – Exemple de réseau d'automates temporisés

$\langle\langle S_1, S_1\rangle, \{\langle first, \top\rangle, \langle fetched, 0\rangle, \langle sent, 0\rangle\}, \{\langle C_0, 0\rangle, \langle C_1, 0\rangle\}\rangle \xrightarrow{\langle new\_file\ !, new\_file\ ?\rangle}{}^{12}$
$\langle\langle S_1, S_1\rangle, \{\langle first, \top\rangle, \langle fetched, 1\rangle, \langle sent, 1\rangle\}, \{\langle C_0, 12\rangle, \langle C_1, 0\rangle\}\rangle \xrightarrow{\langle new\_file\ !, new\_file\ ?\rangle}{}^{42}$
$\langle\langle S_1, S_1\rangle, \{\langle first, \top\rangle, \langle fetched, 2\rangle, \langle sent, 2\rangle\}, \{\langle C_0, 56\rangle, \langle C_1, 0\rangle\}\rangle \xrightarrow{\langle new\_file\ !, new\_file\ ?\rangle}{}^{32}$
...
$\langle\langle S_1, S_1\rangle, \{\langle first, \top\rangle, \langle fetched, 386\rangle, \langle sent, 386\rangle\}, \{\langle C_0, 16086\rangle, \langle C_1, 0\rangle\}\rangle \xrightarrow{\langle done\ ?, done\ !\rangle}{}^{46}$
$\langle\langle S_0, S_0\rangle, \{\langle first, \bot\rangle, \langle fetched, 386\rangle, \langle sent, 386\rangle\}, \{\langle C_0, 16132\rangle, \langle C_1, 46\rangle\}\rangle$
À la fin de la trace, $C_0 - C_1$ correspond au temps de transfert total des fichiers.

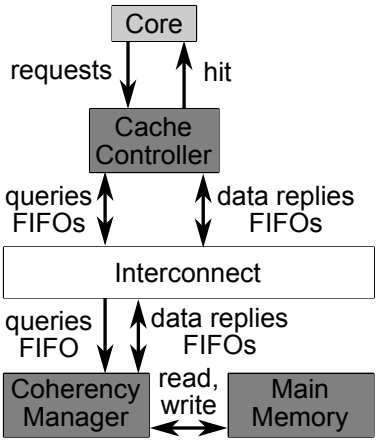
Un des intérêts d'une modélisation sous forme d'automates (temporisés ou non) est de donner à accès à des outils de vérification formelle. Ainsi, on peut définir la relation de satisfiabilité pour une propriété ϕ. On suppose que ϕ est une formule d'un sous ensemble de CTL ([19]). La satisfiabilité de $\langle s, v\rangle \models \phi$ est définie en utilisant la décomposition suivante :

$\langle s, v\rangle \models \psi \triangleq v \models_{PL} \psi$, où ψ est une expression dans **abexpr**(`Var`).
$\langle s, v\rangle \models \neg\phi \triangleq \langle s, v\rangle \not\models \phi$
$\langle s, v\rangle \models \phi \wedge \psi \triangleq (\langle s, v\rangle \models \phi)$ et $(\langle s, v\rangle \models \psi)$
$\langle s, v\rangle \models$ `AF` $\phi \triangleq$
Pour tous les chemins partant de $\langle s, v\rangle$, il y a, dans le chemin, un $\langle s', v'\rangle$, tel que $\langle s', v'\rangle \models \phi$
$\langle s, v\rangle \models$ `EF` $\phi \triangleq$
Il y a un chemin partant de $\langle s, v\rangle$ dans lequel il y a un $\langle s', v'\rangle$, tel que $\langle s', v'\rangle \models \phi$
$\langle s, v\rangle \models$ `AG` $\phi \triangleq$
Pour tous les chemins partant de $\langle s, v\rangle$, tous les $\langle s', v'\rangle$ du chemin vérifient $\langle s', v'\rangle \models \phi$
$\langle s, v\rangle \models$ `EG` $\phi \triangleq$
Il y a un chemin partant de $\langle s, v\rangle$ tel que tous les $\langle s', v'\rangle$ du chemin vérifient $\langle s', v'\rangle \models \phi$
$\langle s, v\rangle \models \phi$ `-->` $\psi \triangleq$
Pour tous les chemins partant de $\langle s, v\rangle$, tout sous-chemin partant d'un $\langle s', v'\rangle$ tel que $v' \models_{PL} \phi$ contient aussi au moins un $\langle s'', v''\rangle$ tel que $v'' \models_{PL} \psi$.

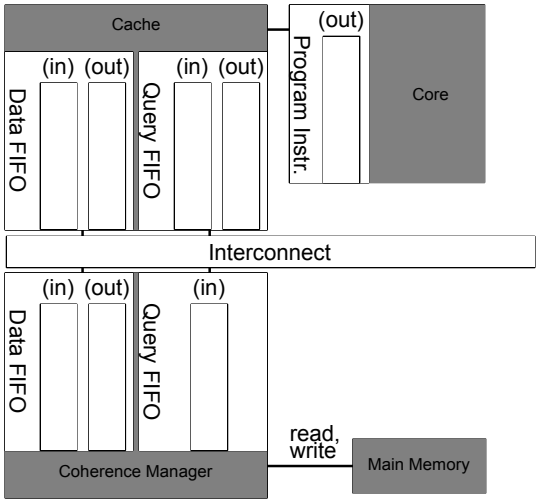
12.2.2 Fonctionnement des caches

Parmi les mécanismes complexes d'un processeur multi-coeur se trouve la cohérence de caches. Celle-ci assure que tous les cœurs lisant ou écrivant dans un même bloc mémoire ne peuvent pas aveuglement ignorer les modifications appliquées par les autres. Afin de maintenir la cohérence de

caches, le processeur suit un protocole pré-déterminé qui définit les messages à envoyer en fonction des actions d'un cœur ainsi que les actions à effectuer lors de la réception du message d'un autre cœur. Pour comprendre ce mécanisme, commençons par présenter l'architecture générale et les composants participant à la cohérence (présentés dans la figure 12.4).



(a) Vue d'ensemble



(b) Vue comprenant les FIFOs

FIGURE 12.4 – Composants ayant un rôle dans la cohérence de cache

Definition 73 (Élément mémoire) *La mémoire d'un système est découpée en blocs adressables. Dans la suite, un élément mémoire correspondra directement à un bloc mémoire. L'ensemble de tous les éléments mémoire est défini par* $\texttt{Addr} \subseteq \mathbb{N}$.

La cohérence de cache est activée lors des accès à des données partagées et donc lors des exécutions des instructions d'écriture (`store`), lecture (`load`) et éviction (`evict`).

Definition 74 (Programme s'exécutant sur un cœur) *Les opérateurs élémentaires sont* $\texttt{OPs} = \{\texttt{load}, \texttt{store}, \texttt{evict}, \texttt{nop}\}$ *et les instructions considérées sont* $\texttt{Instr} = \texttt{OPs} \times \texttt{Addr}$. *Un programme est une séquence d'instructions et donc* $\texttt{InstrQueue} = Seq(\texttt{Instr})$.

La notion de séquence sera largement réutilisée dans la suite.

Definition 75 (Séquence) *$Seq(A)$ indique une séquence finie (potentiellement vide) composée d'éléments de type A. Les séquences sont donc définies par :*

$$Seq(A) = \begin{cases} [\ ] \\ A :: Seq(A) \end{cases}$$

L'ajout d'un élément e en tête de la séquence S est noté $push(e,S)$ et correspond à $e :: S$. L'extraction de la tête de la séquence S est notée $pop(S)$, ce qui retourne $head(S)$ avant d'appliquer $S \leftarrow tail(S)$. Enfin, $isEmpty(S)$ indique si S est une séquence vide et est l'équivalent de vérifier si $S = [\ ]$.

Les caches sont chargés d'obtenir des copies des éléments mémoire afin de répondre aux requêtes d'accès mémoire de leur cœur.

Definition 76 (Ensemble des caches) *L'ensemble des caches du système est défini par* $\texttt{Ccs}$*. On note* $\texttt{Ccs}^+ = \texttt{Ccs} \cup \{\texttt{mgr}\}$ *l'ensemble des caches et le gestionnaire de cohérence (*$\texttt{mgr}$*).*

Pour gérer les requêtes du cœur, le cache doit de stocker les permissions sur chaque élément mémoire et les opérations en cours. Pour obtenir de nouvelles permissions, un cache envoie des demandes sur l'interconnect afin de se coordonner avec les autres caches et le gestionnaire de cohérence. Chaque demande ne concerne qu'un seul élément mémoire. On considèrera les demandes de copie en lecture seule de l'élément mémoire (`GetS`, faisant généralement suite à une instruction de `load`) ; de copie en écriture et lecture (`GetM`, faisant généralement suite à une instruction de `store`) ; et le signalement d'une éviction pour un élément mémoire ayant potentiellement été modifié (`PutM`, faisant généralement suite à un `evict`).

Definition 77 (Demande) *Il existe trois catégories de demande* $\texttt{Query} = \{\texttt{GetS}, \texttt{GetM}, \texttt{PutM}\}$*. Un message envoyé sur l'interconnect pour émettre une demande est défini par* $\texttt{MSG}_{\texttt{query}} : \texttt{Query} \times \texttt{Addr} \times \texttt{Ccs}$*, qui indique le type de la demande, l'élément mémoire visé et l'émetteur du message.*

Chaque demande fera l'objet d'une réponse. Une réponse peut simplement contenir une copie de l'élément mémoire (`data`) ; une réponse peut indiquant qu'aucun autre cache ne possède actuellement de copie de cet élément mémoire (`data-e`) ; et une réponse peut informer qu'aucune copie ne sera envoyée (`no-data`).

Definition 78 (Réponse) *Il existe trois catégories de réponse* $\texttt{Reply} = \{\texttt{data}, \texttt{data-e}, \texttt{no-data}\}$*. Un message envoyé sur l'interconnect pour émettre une réponse est défini par* $\texttt{MSG}_{\texttt{data}} : \texttt{Reply} \times \texttt{Addr} \times \texttt{Ccs}^+$*, qui indique la catégorie, l'élément mémoire en question, et le cache visé.*

Il est possible que des caches reçoivent des demandes auxquelles ils doivent répondre alors qu'ils n'ont pas encore reçu les informations nécessaires. Ils auront donc en charge de gérer toute cette complexité et répondre à l'ensemble des requêtes et demandes qui leur sont envoyées. Chaque cache a quatre FIFO, chacune gérant soit la réception ou l'émission des demandes et des réponses, comme le montre la Figure 12.4b.

Le gestionnaire de cohérence est un élément central, non nécessairement implanté par un composant unique dans l'architecture, permettant la coordination entre les caches. Tout comme les caches, il utilise des files FIFOs pour gérer les messages entrant et sortant. L'interconnect connecte notamment les caches et le gestionnaire de cohérence. Il broadcast les demandes des caches à tous les composants connectés, y compris le cache émetteur. Les réponses, quant à elles, ciblent un seul composant et ne sont donc reçues que par celui-ci.

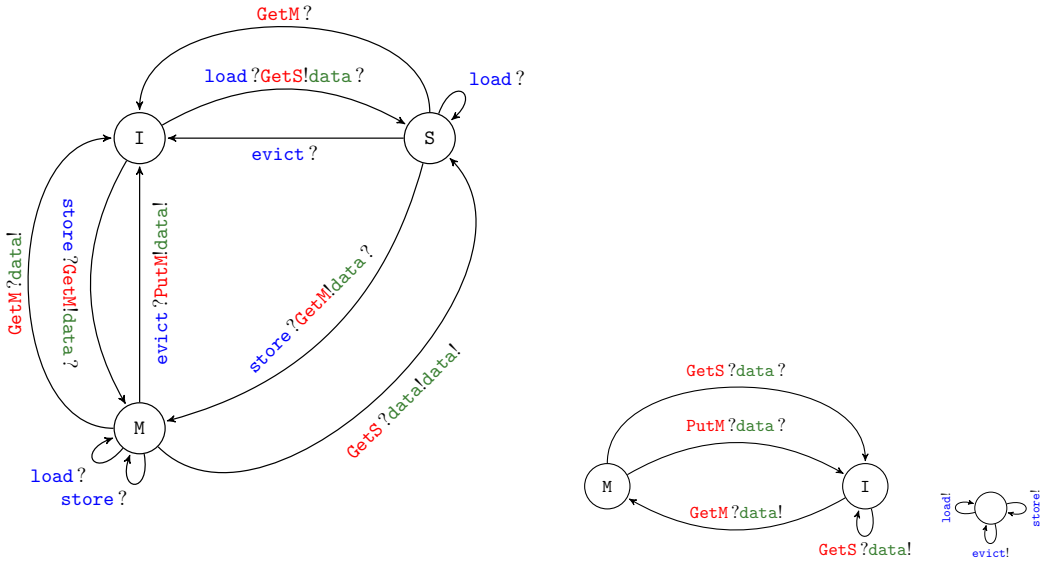
12.2.3 Cohérence de cache

Cette partie, inspirée des principes de [49], introduit la cohérence de cache. Un système avec cohérence de cache est un système dans lequel des applications utilisant des caches séparés ne perçoivent pas cette séparation lorsqu'elles lisent et écrivent sur les éléments mémoire.

Property 10 *Un protole doit vérifier les propriétés suivantes :*

1. *Les caches ont la valeur système : À tout moment, pour chaque élément mémoire, toutes les copies du même élément mémoire présentes dans un cache ont la même valeur. Cette valeur correspond à la dernière qui a été écrite pour cet élément mémoire, quel que soit le cache qui a fait l'écriture.*

2. *Un seul écrivain ou seulement des lecteurs : À tout moment, pour chaque élément mémoire, il y a soit un seul cache autorisé à écrire et il est aussi le seul à pouvoir lire, soit aucun cache autorisé à écrire et un nombre indéterminé de lecteurs.*
3. *Garder le fil : Si un élément mémoire n'a pas de copie en cache, alors la valeur qui se trouve dans la mémoire principale du système est la dernière à avoir été écrite.*



(a) Cache (b) Gestionnaire de cohérence (c) Cœur

FIGURE 12.5 – Vue d'ensemble du protocole MSI

Le protocole MSI est le protocole élémentaire de cohérence de cache et les autres protocoles connus en sont en général des extensions. Le protocole MSI contient trois états stables, qui lui donnent son nom :

- **Modified :** *Modified* indique qu'au moins une écriture a été réalisée sur l'élément mémoire. Tant qu'il possède une copie dans cet état, le cache peut librement lire et écrire. Cela correspond à être l'unique écrivain dans la Propriété 10.
- **Shared :** *Shared* correspond à un accès en lecture.
- **Invalid :** *Invalid* indique que le cache ne possède pas de copie de l'élément mémoire. Il ne peut donc ni le lire ni le modifier en écriture.

La Figure 12.5 montre les automates correspondant au cache, au gestionnaire et au cœur. Cette vision est simplifiée notamment car les FIFO ont été abstraites et les échanges sont synchrones. Dans les faits, il faut distinguer les états *stables* ceux représentés ici des états *transients*, qui représentent des étapes intermédiaires dans la résolution des demandes et des réponses.

Example 44 *Considérons un système avec deux caches, C_A and C_B, et un seul élément mémoire E tel que, initialement, C_A n'a pas de copie de E (état `I`) et C_B a une copie de E dans l'état `S`. Le gestionnaire de cohérence considère donc E dans l'état `I`. Supposons que C_A reçoive une requête de `store` de son cœur. Dans cette situation, C_A envoie une demande `GetM`, laquelle est broadcastée*

aux autres composants. Voyant le `GetM`*,* C_B *doit alors abandonner sa copie de* E *afin de maintenir la Propriété 10. De son côté, le gestionnaire de cohérence réagit au* `GetM` *en envoyant une réponse* `data` *à* C_A *et en considérant désormais* E *dans l'état* `M`*. Cette réponse* `data` *permet à* C_A *d'obtenir sa copie de* E *avec un état* `M` *et donc d'exécuter l'instruction du cœur.*

12.3 État de l'art

Pour répondre à l'objectif RU3, il faut être capable d'identifier le protocole de cache, de quantifier son impact et de maîtriser les interférences associées. L'identification et la quantification reposent souvent dans la littérature sur l'utilisation de stressing benchmarks qui seront l'objet de la première sous-partie. La deuxième rappellera brièvement les solutions actuelles pour maîtrîser les interférences et la dernière se concentrera sur l'utilisation de méthodes formelles pour l'analyse et la maîtrise des interférences.

12.3.1 Micro-stressing benchmarks

Afin d'identifier les caractéristiques d'une architecture, plusieurs travaux reposent sur l'utilisation de micro-stressing benchmarks. L'idée est de jouer sur des paramètres de configuration et mesurer avec des moniteurs de performances les effets temporels négatifs (souvent appelés surcoûts ou facteurs de ralentissement).

Definition 79 (Configuration) *Une configuration est définie comme la combinaison du placement des programmes sur chaque cœur, des valeurs de paramètres matériels, ainsi que l'état initial de la machine avant que les programmes ne s'exécutent.*

Definition 80 (Stressing Benchmark) *Un stressing benchmark est une suite d'instruction spécifiquement développée pour saturer les capacités d'un composant particulier de l'architecture afin d'en observer ses limites.*

Definition 81 (Temps d'exécution) *Le temps d'exécution d'une suite d'instructions sur une certaine configuration est le temps entre l'émission de la première instruction et la complétion (vue par l'émetteur) de la dernière instruction.*

Definition 82 (Surcoût / Facteur de ralentissement) *Étant donné* T_A *et* T_B *deux temps d'exécution de la même suite d'instruction sur deux configurations distinctes* A *et* B *tels que* $T_B \geq T_A$*.*

Le surcoût O *d'utilisation de la configuration* B *comparée à* A *est défini comme :* $O = T_B - T_A$*. Le facteur de ralentissement* f *est défini comme :* $f = \frac{T_A}{T_B}$

[44] présente une stratégie pour la caractérisation de la sensibilité aux interférences d'une ressource partagée. L'idée générale étant d'effectuer un stressing benchmark sur ladite ressource uniquement avec un programme s'exécutant en isolation, puis de comparer le temps d'exécution avec le même benchmark s'exécutant alors que d'autres programmes sont exécutés en parallèle. Cette approche forme la base de l'identification de canaux cachés d'interférence puisqu'elle révèle de potentiels liens cachés entre les composants. Le papier montre qu'il est préférable de faire des micro-stressing benchmarks (par exemple des séries de lecture ou des séries d'écriture) pour obtenir des résultats précis plutôt que des suites de benchmarks standards (par exemple MiBench avec des applications complètes), y compris pour l'interférence liée au partage de caches.

L'approche présentée dans [11] peut être vue comme la continuité de [44] : la corrélation identifiée entre composants est ensuite utilisée pour déterminer les benchmarks les plus pertinents à faire

pour déterminer les effets des interférences sur une application précise. Pour cela, au lieu de se limiter au temps d'exécution, [11] regarde également les corrélations entre composant à travers des moniteurs de performance. Les moniteurs de performance sont des compteurs qui peuvent être configurés pour compter le nombre d'occurrences d'un événement donné (par exemple, l'accès à l'interconnect). La documentation de l'architecture fournit généralement une liste d'événements pour lesquels les moniteurs de performance peuvent être configurés. Ces moniteurs peuvent être remis à zéro, temporairement gelés ou configurés pour suivre un autre type d'événement pendant l'exécution des programmes. Cela en fait un outil d'analyse très utile. [11] propose d'utiliser cette information pour limiter au strict minimum les analyses à faire sur les programmes. En effet, en observant quels événements peuvent être causés par interférence depuis un composant connexe (et surtout ceux qui ne peuvent pas l'être), des benchmarks redondants peuvent être évités.

Les moniteurs de performance ne sont pas présents sur toutes les architectures. [40] présente une étude de cas pour la caractérisation d'architectures malgré leur absence. Leur plateforme dispose de certaines fonctionnalités d'analyse qui permettent de capturer les messages passant sur le bus et les instructions exécutées sur chaque cœur. Par extraction et analyse de cette information pendant l'exécution des programmes, les auteurs de [40] parviennent à obtenir un compte des événements semblable à celui de moniteurs de performance. La stratégie d'identification du protocole de cache proposée dans cette thèse repose sur l'utilisation de moniteurs de performance.

Les travaux présentés jusque là regardent les caches d'assez loin. [39] s'intéresse également aux problèmes d'interférence dans les multi-cœurs dans l'avionique. Plus précisément, ce papier propose une analyse de l'architecture à travers des benchmarks pour s'assurer que des tâches exécutées en parallèle peuvent correctement être temporellement partitionnées. Les analyses de [39] s'intéressent aux surcoûts induits par la cohérence de cache. L'approche choisie étant de comparer le temps d'exécution d'instructions `load` et `store` selon trois configurations : sans cohérence de cache, avec cohérence de cache mais sans variables partagées, avec cohérence de cache et variables partagées. Pour chacune de ses configurations, les résultats sont ceux du temps d'exécution d'un cœur pour la lecture ou l'écriture pendant qu'un autre cœur effectue des lectures ou des écritures. Ces analyses permettent en partie de mesurer la performance de la cohérence de cache mais sans tenir compte des états de cohérence pour les éléments accédés, ce qui rend les résultats incomplets.

[38] tient compte des états de cohérence de cache et présente ainsi des travaux d'analyse de performance de la cohérence de cache qui permettraient un paramétrage précis du modèle UPPAAL. La bande passante est aussi mesurée, complétant la caractérisation de la performance de la cohérence de cache pour l'architecture étudiée. Les auteurs de ce papier affirment cependant que le protocole MESIF utilisé par leur architecture peut être considéré comme un simple protocole MESI étant donné qu'il n'agit qu'entre deux caches. Il s'avère que la différence entre les deux protocoles devrait avoir un effet, et les résultats de benchmark qui devraient le montrer (opérations d'écriture) ne sont pas présents dans le papier.

12.3.2 Gestion des interférences

Cette section présente les solutions existantes pour faire face aux interférences générées par la cohérence de cache dans les systèmes critiques. Trois types d'approches sont considérées : imposer des restrictions sur le système, modifier le matériel ou prouver que les effets de l'interférence sont acceptables.

Approche par restriction. L'approche la plus simple est de désactiver les caches entièrement. Cela rend le système beaucoup plus prévisible mais au prix d'un très fort impact négatif sur le temps d'exécution, au point de rendre parfois une exécution mono-cœur préférable. Une légère amélioration consiste à verrouiller le contenu des caches, ce qui limite fortement leur utilité mais n'entraine pas

de diminution de la prédictibilité du système. Dans [29], les lignes caches sont tout simplement partitionnés par cœur, ce qui évite théoriquement que les accès d'un cœur modifient l'espace utilisé par un autre. Le papier propose deux algorithmes pour prouver la possibilité d'ordonnancer des taches temps réel avec ces restrictions mises en place.

[13] effectue un partitionnement moins strict des lignes de cache à cache cœur. En effet, l'idée proposée dans ce papier est d'exploiter la politique de placement en s'assurant que toutes les lignes de cache utilisées par un cœur soient dans le même ensemble vis-à-vis de la politique de remplacement (principe proche du *cache coloring*). En conséquent, l'éviction automatique n'affecte pas les autres cœurs utilisant le même cache. Grâce à une stratégie d'ordonnancement minutieuse, le nombre de programmes partageant ces même lignes de caches est gardé à un minimum.

[34] propose d'utiliser les processeurs multi-cœurs pour faire tourner en parallèle des applications qui étaient jusqu'alors prévues pour des mono-cœurs. L'utilisation des caches est simplifiée, puisque les données ne sont pas partagées entre les programmes. L'hyperviseur proposé assure un partitionnement robuste entre les différentes applications et fait usage d'un TDMA pour contrôler les accès aux ressources partagées, ce qui rend le système plus facile à prédire.

Les travaux de [12] utilisent une stratégie d'ordonnancement minutieuse afin de rendre le calcul du temps d'exécution des applications sur un multi-cœur plus facile. L'approche consiste à transformer automatiquement les programmes de manière à avoir des blocs de calculs et des blocs de transferts, puis de les ordonner de façon à ce que lorsqu'un cœur est dans un bloc de calcul, les autres cœurs ne peuvent pas faire d'accès aux données qu'il utilise. Ainsi, les calculs ne peuvent pas être perturbés par les actions des autres cœurs.

Approche par modifications matérielles [32] met en avant les sources de manque de prédictibilité dans le protocole MSI et propose une solution pour chacune d'entre elles. Ces solutions demandent à ce que certains composants matériels liés aux mécanismes de cohérence de cache soient rendus particulièrement prévisibles. Un protocole de cohérence de cache, PMSI (*Predictable MSI*) est alors proposé avec des formules pour déduire les pires temps de résolution d'instruction.

[42] introduit la notion de cohérence de caches sur demande (ODC^2, *On-Demand Coherent Cache*). L'idée étant de délimiter les sections des programmes pendant lesquelles ils peuvent accéder aux données partagées. Ces données sont marquées comme partagées dans le cache, ce qui nécessite une modification matérielle. L'analyse des sections durant lesquelles aucune donnée partagée n'est accédée est donc rendue beaucoup plus simple. L'ajout de cette stratégie au framework OTAWA est présenté dans [43] et montre que le WCET obtenu avec cette approche est effectivement une amélioration comparée à une approche sans cache ou une dans laquelle tout point de synchronisation entraîne l'invalidation complète des cache.

Approche par acceptation L'approche la plus commune des caches repose sur l'analyse statique et l'interprétation abstraite en continuation des travaux de [24]. Ainsi, chaque accès mémoire est catégorisé en fonction de s'il est assuré de trouver la donnée dans le cache (hit), ne pas l'y trouver (miss), ou si l'analyse ne peut pas le déterminer (unknown). Cette méthode d'analyse n'est pas celle retenue ici pour la cohérence de caches, puisque cette simplification n'est pas compatible avec la gestion des états de cohérence.

La plupart des papiers sur le sujet s'intéressent exclusivement aux caches d'instructions, or les caches contenant des data avec de la cohérence sont ceux qui nous intéressent. A titre d'exemple sur les caches instruction, [31] propose une stratégie d'analyse WCET en catégorisant les accès à chaque niveau : est-ce qu'un certain accès atteindra ce niveau à coup sûr ? Jamais ? Pas la première fois, mais toujours après ? On n'arrive pas à déduire ? Cette catégorisation est alors utilisée pour déterminer quelles interférences potentielles sont dignes d'êtres analysées. Une approche qualifiée de *bypassing scheme* (plan de contournement) est aussi présentée, dans laquelle les données accédées une seule fois sont identifiées afin d'exploiter un mécanisme de contournement pour que cet accès

n'utilise pas inutilement de l'espace en cache.

[35] utilise une approche similaire à celle ci-dessus mais pour les caches de données. La principale différence avec [31] étant la manière de caractériser les accès : les données peuvent avoir des adresses alias qui rendent la détection des blocs à accès unique plus difficile. Cependant la cohérence de cache n'est cependant pas considérée et tous les accès aux données partagés sont supposés ne pas utiliser de caches privés (et il n'y a donc pas besoin de cohérence entre ces caches) et les caches partagés sont considérés comme *write-through* (les modifications sont répercutées sur tous les niveaux de caches jusqu'à la mémoire principale).

12.3.3 Approches formelles

L'approche choisie dans cette thèse repose sur la modélisation en automates temporisés et la vérification formelle. L'utilisation d'UPPAAL à des fins similaires a déjà fait l'objet de travaux, cependant aucun ne traite de la cohérence de cache.

Le premier outil à utiliser UPPAAL pour le calcul du WCET est METAMOC [22] (*Modular Execution Time Analysis using Model Checking*). L'approche modulaire s'applique sur du mono-cœur : le pipeline, la mémoire principale et les paramètres du cache sont isolés afin de faciliter leur remplacement en cas d'analyse d'un autre processeur. Les programmes sont analysés directement depuis leur binaire exécutable avec cependant des notations à ajouter pour la gestion des limites de boucles. [16] présente WUPPAAL, une autre approche utilisant UPPAAL pour le calcul du WCET d'un programme exécuté sur processeur mono-cœur. La principale particularité de cette solution est qu'elle combine la simulation d'exécution et la vérification de modèles. En effet, WUPPAAL fait en sorte qu'UPPAAL interagisse avec *qemu* (un outil de simulation d'architecture) à travers *gdb* (un outil de débogage) et *libgdbuppaal* (une libraire maison). L'intérêt étant de réduire la consommation mémoire de la vérification de modèles et de supporter différentes architectures (et donc ensembles d'instructions binaires) très facilement. L'approche proposée par [36] combine l'interprétation abstraite et la vérification de modèles pour le calcul de WCET sur multi-cœurs, avec une attention particulière à l'interconnect. L'interprétation abstraite est utilisée pour l'analyse des caches (en tagguant les caches hit et miss). Ici encore, la cohérence de caches n'est pas prise en compte.

Le modèle présenté dans [30] correspond le plus à ce qui est fait dans cette thèse : UPPAAL est utilisé pour le calcul de WCET sur un processeur multi-cœurs avec une représentation détaillée de chaque composant. La cohérence de caches n'est pas prise en compte. En revanche, la hiérarchie de cache et les caches partagés par plusieurs cœurs, ainsi que les pipelines, sont présents.

12.4 Identifier la cohérence de cache

Cette section présente la stratégie d'identification du protocole de cache implémenté par l'architecture choisie par l'applicant. Ces travaux ont fait l'objet d'une publication [48]. Pour éviter les confusions, trois protocoles sont définis ci-dessous :

Definition 83 (Le protocole de l'architecture) *Le protocole réellement implémenté sur l'architecture. Celui que l'applicant souhaite identifier. Il n'est probablement pas observable directement.*

Definition 84 (Le protocole observé) *Le protocole observé est la vue partielle du protocole de l'architecture obtenue à travers un ensemble de benchmarks. Puisqu'il n'est pas possible d'assurer que les benchmarks soient exhaustifs, le protocole observé est potentiellement incomplet.*

Definition 85 (Le protocole hypothétique) *Le protocole défini par l'applicant et qui correspond à ce qu'il s'attend à trouver d'après la documentation de l'architecture.*

Pour rappel, les protocoles de cache sont definis autour d'un seul élément mémoire. En conséquent, la stratégie que nous proposons ne considère qu'un seul élément mémoire. De plus, on suppose que les états stables du protocole de l'architecture sont observables.

12.4.1 Définir le protocole hypothétique

La première étape de cette stratégie d'identification est la définition du protocole hypothétique par l'applicant. Cette définition devrait utiliser les notations présentées dans la Section 12.2.2. Un bon point de départ est de se référer à la documentation de l'architecture. Par exemple, pour le NXP QorIQ T4240, la documentation indique un protocole MESI (dans [25]), avec des optimisations pour le partage de données déjà en cache (dans [26]).

12.4.2 Exploration naïve du protocole observable

Pour définir le protocole observable, on effectue une exploration des états accessibles. L'algorithme correspondant à cette exploration se trouve dans la Figure 12.6. En commençant dans un état où aucun cache ne contient la donnée (*init*), on explore les états accessibles par l'application d'une unique instruction à la fois, en notant l'état de cohérence du système une fois l'instruction complétée, ainsi que les valeurs des différents moniteurs de performance. Les étapes du protocole correspondant à state_search, decode et monitors sont détaillées par la suite.

```
init_state_search()
init_decode()

DstStates ← {init}
WaitList ← {init}
while (WaitList ≠ ∅):
   SrcState ∈WaitList;
   WaitList ←WaitList \ SrcState;
   foreach k ∈ 1..cc
         foreach instr ∈{load, store, evict}
               SysInstruction ← single_instruction_on(instr, k)
               ⟨DstState, PerformanceCounters⟩ ← benchmark(SrcState, SysInstruction)

               handle_state_search(SrcState, SysInstruction, DstState) // Step 1
               handle_decode(SrcState, SysInstruction, DstState) // Step 2
               handle_monitors(SrcState, SysInstruction, PerformanceCounters) // Step 3

               if DstState ∉ DstStates
                  DstStates ← DstStates ∪{DstState}
                  WaitList ← WaitList ∪{DstState}
```

FIGURE 12.6 – General State Exploration Algorithm

La fonction benchmark retourne l'état de cohérence des différents composants du système (caches et gestionnaire de cohérence). Dans le cas où le gestionnaire de cohérence n'est pas observable, on considère qu'il suit les règles du protocole hypothétique et on définit le protocole observé comme s'il comportait le même gestionnaire de cohérence.

12.4.3 Exploration d'état et atteignabilité

L'étape state_search catalogue les états stables de cohérence observable dans chaque composant $\texttt{V}_\text{s}$, ainsi que leurs combinaisons à l'échelle du système (c'est-à-dire l'état de cohérence du système) $\texttt{System}_\text{b}$.

```
def init_state_search ()
    Vs ← tuple_to_set(init)
    Systemb ← {init}

def handle_state_search (SrcState, SysInstruction, DstState)
    reachb(SrcState, SysInstruction) ← {DstState}
    if DstState ∉ Systemb
        Vs ← Vs ∪ {tuple_to_set(DstState)}
        Systemb ← Systemb ∪ {DstState}
```

On récupère en fait le graphe de transition des états de cohérence du système.

12.4.4 Correspondance entre état observé et hypothétique

Les états et transitions observés lors de l'étape précédente sont ensuite comparés avec ceux attendus avec le protocole hypothétique. On associe alors les états de ces deux protocoles en définissant `decode`, la relation qui permet de passer d'état observé à état hypothétique. Pour cela, on a besoin de `reach`, qui correspond au graphe de transition d'états stables pour le protocole hypothétique.

```
def init_decode ()
    decode ← ⟨init, < I, ..., I >⟩

def handle_decode (SrcState, SysInstruction, DstState)
    ⟨SrcState, DecodedSrcState⟩ ∈ decode
    {DecodedDstState} ← reach(DecodedSrcState, SysInstruction)
    decode ← decode ∪ {⟨DstState, DecodedDstState⟩}
```

Il est possible que plusieurs états observés soient associés au même état hypothétique. En effet, lorsque l'on observe l'état d'une ligne de cache, il est possible qu'une partie de l'information ne soit pas liée à la cohérence de cache. En conséquent, plusieurs états observés peuvent en fait correspondre au même état de cohérence.

Cependant, si le protocole hypothétique correspond effectivement au protocole de l'architecture, alors il ne doit pas y avoir d'état observé associé à plusieurs états hypothétiques. En effet, cela indiquerait que certains comportements du protocole hypothétique sont absents de l'architecture. Il en est de même si l'un des états hypothétiques n'est associé à aucun état observé. Si l'une de ces deux situations non souhaitées parvient, alors le protocole hypothétique est infirmé.

12.4.5 Comparaison des activités

Si le protocole hypothétique n'est pas infirmé lors de l'étape précédente, les activités observées sur l'architecture grâce aux moniteurs de performance sont comparées à celles attendues d'après le protocole hypothétique. Pour cela, l'applicant note le nombre d'événements observés lors de chaque benchmark dans $\texttt{Act}^{\texttt{Mon}}$.

```
def handle_monitors(SrcState, SysInstruction, PerformanceCounters)
    Act^Mon(SrcState, SysInstruction) ← PerformanceCounters
```

Les résultats obtenus dans $\texttt{Act}^{\texttt{Mon}}$ doivent correspondre à ce que le protocole hypothétique génère. Toute exception doit être étudiée, car elle pourrait indiquer une différence significative entre les deux protocoles. De plus, si certains états observés n'agissent pas de la même façon vis-à-vis d'événements

de cohérence alors qu'ils sont associés au même état théorique, alors les deux protocoles ne se correspondent pas. En effet, le protocole implémenté a dans ce cas plus d'états stables de cohérence que le protocole hypothétique.

Si, à la fin de cette étape, le protocole hypothétique n'a pas été contredit par les observations, alors le protocole hypothétique décrit tous les comportements du protocole observé. Pour s'assurer que le protocole de l'architecture implémente tous les comportements du protocole hypothétique, l'étape suivante fait une exploration guidée par le protocole hypothétique.

12.4.6 Exploration guidée par le protocole hypothétique

Definition 86 (Chemin de changement d'état stable) *Un chemin de changement d'état stable est un chemin entre deux états stables de la partie "contrôleur de cache" du protocole. Il est formé d'un état stable suivi d'une séquence d'états transitoires et termine par un état stable, avec une transition du protocole entre chaque état. Les deux états stables peuvent être les mêmes. Les transitions sans actions ne sont pas permises.*

Cette étape du processus d'identification s'assure que tous les comportements du protocole hypothétique sont présents sur l'architecture. Elle repose sur l'établissement de la liste exhaustive des chemins de changement d'états stables (voir Définition 86), qui peut être générée automatiquement par un outil fourni dans le cadre de cette thèse.

L'idée générale est simple : reproduire la séquence de chacun de ces chemins sur l'architecture afin de s'assurer que les activités observées sur l'architecture correspondent à celles attendues. Cependant, l'implémentation de ces benchmarks est beaucoup plus difficile que pour les étapes précédentes, puisque les transitions doivent être faites dans un ordre précis. Comparé aux étapes précédentes :

- Plusieurs instructions peuvent être appliquées simultanément (sur différents cœurs) afin de générer la séquence désirée. De plus, les benchmarks peuvent comporter des séquences d'instructions, au lieu d'une seule par cache.
- L'analyse est concentrée sur un seul cache et non l'ensemble du système. À la place, les autres caches sont utilisés pour causer les bonnes transitions.
- L'exploration n'est pas aveugle : l'espace d'états est maintenant connu. La difficulté repose sur l'obtention de la séquence souhaitée sur l'architecture.

Une fois cette exploration terminée, si tous les comportements du protocole hypothétique ont correctement été observés, alors le protocole de l'architecture est garanti de tous les implémenter. Combiné avec les résultats montrant que le protocole hypothétique décrit correctement le protocole observé, on peut conclure que cette stratégie d'identification a fourni une bonne compréhension du protocole implémenté par l'architecture à l'applicant.

12.4.7 Application au NXP QorIQ T4240

L'application de cette stratégie sur le NXP QorIQ T4240 a révélé des résultats intéressants. En effet, tenter de valider un protocole hypothétique MESI révèle un seul état de cohérence observé pour chaque état de cohérence hypothétique, sauf pour l'état hypothétique `S`, qui est associé à deux états observés ($\varphi_{\mathtt{b}}$ et $\chi_{\mathtt{b}}$).

Bien qu'avoir deux états observés pour un même état hypothétique ne soit pas suffisant pour contredire le protocole hypothétique, la suite du processus d'identification a révélé que ces deux états observés sont différents du point de vue de la cohérence de cache. En effet, comme montré dans la Figure 12.7, les états observés $\varphi_{\mathtt{b}}$ et $\chi_{\mathtt{b}}$ réagissent de manière différente à une demande provenant

	⟨-, -, load⟩	
Origine	**Comportement**	
	Attendu	**Observé**
$\langle \mathtt{I_b}, \mathtt{I_b}, \mathtt{I_b} \rangle$	8000 External Snoop Requests	8000 External Snoop Requests
$\langle \varphi_\mathtt{b}, \mathtt{I_b}, \mathtt{I_b} \rangle$	8000 L2 Snoop Hits, 8000 External Snoop Requests	8000 L2 Snoop Hits, **8000 L2 Snoop Pushes**, 8000 External Snoop Requests, **8000 SINTs**
$\langle \chi_\mathtt{b}, \mathtt{I_b}, \mathtt{I_b} \rangle$	8000 L2 Snoop Hits, 8000 External Snoop Requests	8000 L2 Snoop Hits, 8000 External Snoop Requests

FIGURE 12.7 – Anomalies pour les états $\varphi_\mathtt{b}$ et $\chi_\mathtt{b}$

d'un autre cache : l'état $\varphi_\mathtt{b}$ entraine une réponse à la demande de la part du cache, alors que l'état $\chi_\mathtt{b}$ ne répond pas. Le comportement attendu pour un état hypothétique `S` est de ne pas répondre. Ainsi, les observations montrent que l'architecture a un état stable supplémentaire et n'utilise donc pas un protocole MESI. Cette état supplémentaire semble correspondre au `F` d'un protocole MESIF et, après une seconde application de la stratégie d'identification du protocole, cela s'est confirmé.

12.5 Modéliser la cohérence de cache

Cette section offre un aperçu du modèle UPPAAL[1] pour l'analyse des effets de la cohérence de cache dans les processeurs multi-cœurs. Sont but est de créer un modèle formel afin de pouvoir faire des analyses automatiques (décrites dans la Section 12.6) tout en assurant que :

- Le modèle est aussi générique que possible dans la façon dont il modélise la cohérence de cache afin de permettre de facilement passer d'un protocole à un autre.
- Les protocoles sont modélisés en détails, prenant en compte tous les états transitoires et sont définis pour des bus *split-transaction*.

L'approche choisie est similaire à celles des papiers présentés dans la Section 12.3.3 : utiliser un réseau d'automates de tailles modérées, chacun représentant un composant, afin que le système résultant soit facile à comprendre et modulaire.

12.5.1 Stratégie de modélisation

Le modèle de l'architecture est limité aux éléments directement liés à la cohérence de cache. On suppose qu'un applicant ayant besoin de composants plus précis ou de composants additionnels peut soit les prendre depuis une autre solution, soit intégrer les composants de cohérence de cache présents ici dans cette autre solution.

Chaque composant a son propre automate. Les états et transitions de chaque automate représentent en premier lieu les communications (via des synchronisations) entre les différents composants. Le fonctionnement interne de chaque composant, comme les états de cohérence, est décrit via des variables d'états qui sont mises à jour dans des fonctions, qui utilisent une syntaxe proche du C d'UPPAAL. Cela produite des automates plus petits et plus lisibles, car les transitions sont moins nombreuses et leur actions sont faites par l'appel à des fonctions au nom significatif.

[1] Disponible sur `https://github.com/nsensfel/phylog-cache-coherence`

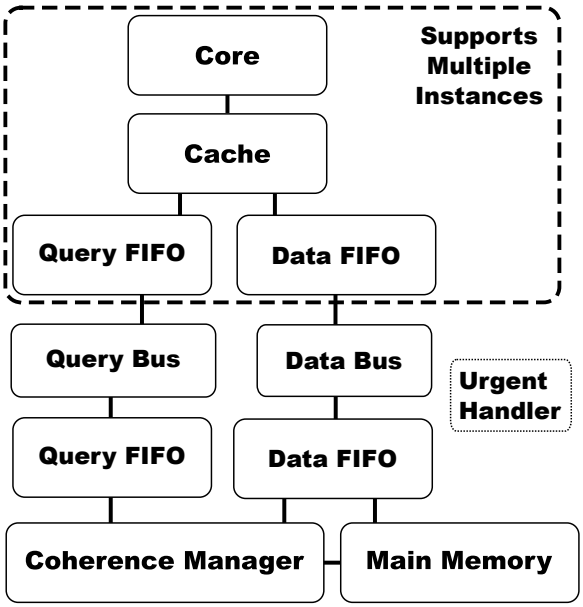


FIGURE 12.8 – Vue d'ensemble des automates du modèle

La Figure 12.8 montre tous les automates définis dans le modèle. On y retrouve les composants d'une architecture multi-coœurs, plus quelques artefacts de modélisation. En effet, l'interconnect split-transaction a été divisé en deux composantes : un bus de demande et un bus de réponse. De plus, un automate de file de messages FIFO pour les réponses a été ajouté et est partagé par le gestionnaire de cohérence et le contrôleur mémoire. Pour finir, un automate *Urgent Handler* est présent, qui ne correspond à aucun composant physique mais permet de faire des synchronisations `urgent`.

Les transferts de données entre automates sont faits par synchronisation : l'émetteur stocke la valeur dans une variable globale, qui est lue par le ou les receveurs. Cette variable globale est définie de manière à clairement identifier l'émetteur et, s'il y a lieu, le receveur. L'élément mémoire concerné par l'échange est aussi indiqué dans cette variable globale, ainsi que le type de message (par exemple `GetM`). La validité de cette variable globale n'est assurée que le temps de la transition, puisque la prochaine transition peut en changer le contenu. En conséquent, les automates la recevant vont garder une copie du message dans une variable locale.

Le modèle est fait de manière à suivre les suppositions de la Section **??** et peut être instancié en configurant les paramètres listés dans l'Annexe B.

12.5.2 Changer de protocole de cohérence

Afin de rendre le modèle plus facile à adapter à différentes architectures, celui-ci est accompagné par un outil appelé CoProSwi qui permet de faire un changement automatique du protocole de cache utilisé.

Comme indiqué dans la Figure 12.9, cet outil prend en entrée un modèle et la description du protocole de cache sous la forme d'un fichier texte. Cette description correspond aux notations de la Section 12.2.2 et donc aussi à celles du processus d'identification de la Section 12.4.

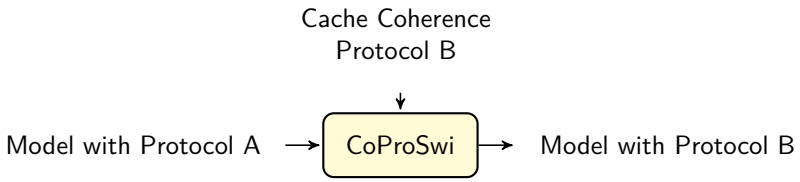


FIGURE 12.9 – L'outil *Co(herence) Pro(tocol) Swi(tcher)*

En conséquent, les protocoles décrits pour CoProSwi indiquent notamment :

- La définition du comportement du contrôleur de cache.
- La définition du comportement du gestionnaire de cohérence.

CoProSwi suppose que tous les protocoles sont définis autours des instructions `load`, `store` et `evict`.

Lors de la définition du contrôleur de cache ou du gestionnaire de cohérence :

- Chaque état de cohérence est déclaré. Ces déclarations indiquent si l'état est transitoire ou stable.
- L'état par défaut des éléments mémoire est aussi indiqué. On considère que cet état correspond à la représentation de l'état *Invalid* du protocole.
- Les actions à faire pour chaque état suite à l'observation d'un message ou d'une requête du cœur sont clairement définies.

CoProSwi prend donc en charge toutes les difficultés liées à l'adaptation du modèle à un protocole de cohérence de cache différent. Il n'est donc pas nécessaire à l'applicant de comprendre le fonctionnement interne du modèle pour en faire l'usage. CoProSwi est aussi capable de générer la liste exhaustive de chemins de changement d'états utilisée dans la Section 12.4.

12.6 Analyser la cohérence de cache

Cette section présente les analyses permettant d'utiliser le modèle de la section précédente pour mettre en évidence les interférences liées à la cohérence de cache. Une partie de ces travaux ont été publiés dans [47].

Le modèle présenté dans la section précédente comporte des paramètres (par exemple la durée d'un accès à la mémoire) qui doivent être instanciés via une campagne de benchmarks afin de mener les analyses qui sont décrites ici. Ce processus d'instanciation est en dehors du périmètre de la thèse. La Figure 12.10 fournit une vue d'ensemble des analyses proposées. Les rectangles avec fond gris correspondent aux analyses et ceux sans arrière plan sont les résultats principaux. Les éléments sans bordure sont des résultats accessoires. Les bordures en pointillés indiquent les résultats intermédiaires, qui ne sont pas censés être utiles en eux-mêmes.

Le modèle instancié utilisé pour l'illustration des analyses dans cette section est représenté dans la Figure 12.11. Celui-ci utilise un protocole MESI dont les détails sont fournis dans la version complète de la thèse (voir Figure 8.7). Le programme utilisé par chaque cœur est indiqué par les Figures 12.12 et 12.13. Ces modèles instanciés comportent toujours de l'indéterminisme, qui représente ce que l'applicant ne connaît pas ou ne peut pas contrôler sur l'architecture. Par conséquent, le modèle instancié admet toujours plusieurs traces d'exécution possibles. Les analyses étant basées sur l'algorithme de model checking d'UPPAAL, toutes ces traces sont explorées.

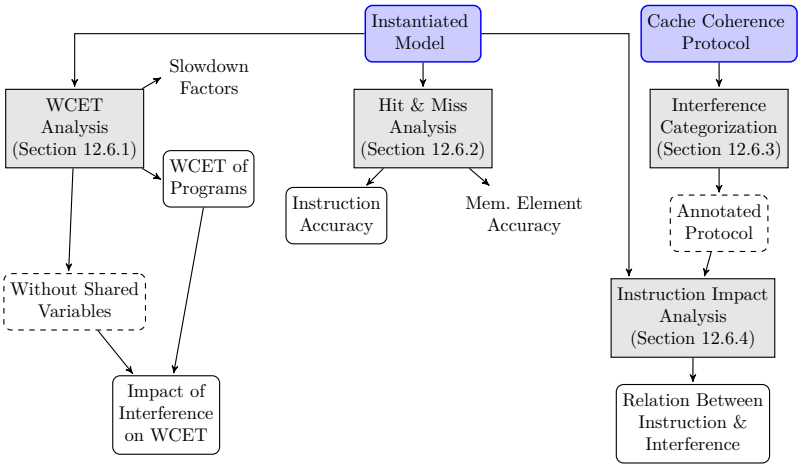


FIGURE 12.10 – Vue d'ensemble des analyses proposées

12.6.1 Analyse de l'impact sur le temps d'exécution

Cette première analyse regarde les effets de l'interférence sur le temps d'exécution total des programmes. Notons cependant qu'il s'agit du temps d'exécution calculé sur le modèle qui peut s'éloigner du temps d'exécution sur l'architecture réelle, notamment en raison des abstractions faites dans la modélisation ou d'une instanciation du modèle insuffisamment précise. Les valeurs obtenues sont tout de même intéressantes à comparer avec d'autres analyses de temps sur un modèle similaire, par exemple pour le calcul de facteurs de ralentissement (voir Définition 82). Par exemple, on peut obtenir la proportion du temps d'exécution causée par la cohérence de cache en comparant le modèle avec une version adaptée du même modèle dans laquelle aucune variable n'est partagée. Bien que cette version adaptée ne soit probablement pas réaliste, son temps d'exécution peut être utilisé comme point de référence.

Definition 87 (Impact de la cohérence de cache sur le temps d'exécution) *Soit W_s le temps d'exécution d'un programme sur le modèle instancié, et W_p son temps d'exécution sur une instance du même modèle dans laquelle toutes les variables ont été rendues privées. La part de W_s correspondant aux mécanismes de cohérence de cache peut être obtenue avec l'équation suivante : $T_{cc} = W_s - W_p$*

Pour obtenir les valeurs de W_s et W_p en utilisant UPPAAL, on emploie la vérification de modèle. En effet, ces valeurs correspondent au maximum d'une horloge mesurant le temps d'exécution du cœur étudié. Il est donc possible de la récupérer avec une formule semblable à sup{not Core1.Terminated}: Core1.runtime, qui retourne le maximum pour l'horloge Core1.runtime dans l'ensemble des états du système pour lesquels l'automate Core1 n'est pas dans la localité Terminated.

Example 45 (Exemple de mesures de temps d'exécution) *La Figure 12.14 indique la valeur maximale du temps d'exécution pour chaque cœur, avec différentes versions de l'exemple du modèle instancié. W_s correspond à celle du modèle instancié original (et donc avec les variables partagées). W_p correspond à celle d'un modèle dans lequel toutes les variables ont été rendues privées. Ici, on incrémente les adresses d'éléments mémoires du programme sur le Cœur 2 par 3 pour éviter tout partage. T_{cc} correspond à la part du temps d'exécution de W_s prise par les mécanismes de cohérence cache. Enfin, pour montrer un exemple de facteur de ralentissement, on étudie aussi le cas où chaque programme est exécuté en isolation. Les résultats de l'analyse sont donc :*

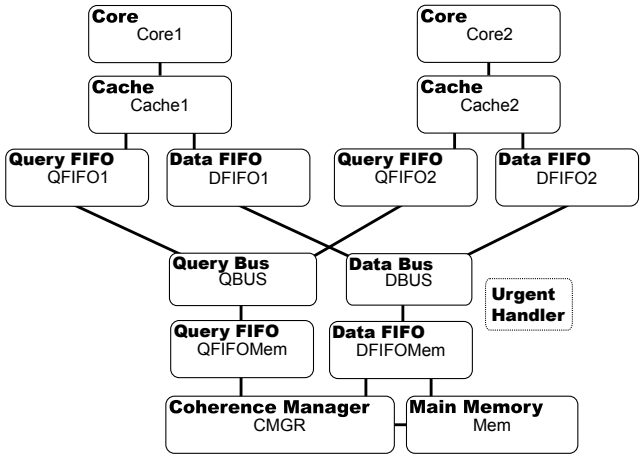


FIGURE 12.11 – Aperçu de l'exemple de modèle instancié

```
1. store 1
2. store 2
3. load 1
4. store 1
5. load 3
6. store 2
7. load 1
8. store 1
9. load 2
10. store 2
11. end
```

FIGURE 12.12 – Modèle de programme pour Cœur 1

```
1. store 1
2. store 3
3. load 3
4. store 2
5. load 1
6. store 2
7. load 3
8. store 1
9. load 2
10. store 3
11. end
```

FIGURE 12.13 – Modèle de programme pour Cœur 2

	W_s	W_p	T_{cc}	Isolation
Cœur 1	2652	1102	1550	702
Cœur 2	2452	1452	1000	904

FIGURE 12.14 – Exemple d'analyse de temps d'exécution

- *Cœur 1 souffre d'un facteur de ralentissement de* $2652/702 = 3,77$ *quand son programme tourne en même temps que celui de Cœur 2, comparé à son exécution en isolation.*
- *Cœur 2 souffre d'un facteur de ralentissement de* $2452/904 = 2,71$ *quand son programme tourne en même temps que celui de Cœur 1, comparé à son exécution en isolation.*
- *Exécuter les deux programmes en isolation l'un après l'autre aurait un temps d'exécution maximum de* $702 + 904 = 1606$ *unités de temps.*
- *Exécuter les deux programmes en parallèle a un temps d'exécution maximal de* $\max(2652, 2452) = 2652$.
- *Exécuter les deux programmes en parallèle mais sans variables partagées a un temps d'exécution maximal de* $\max(1102, 1452) = 1452$.
- *Approximativement* $(1550/2652) * 100 = 58,44\%$ *du temps d'exécution de Cœur 1 est causé par l'interférence liée à la cohérence de caches.*
- *Approximativement* $(1000/2452) * 100 = 40,78\%$ *du temps d'exécution de Cœur 2 est causé par l'interférence liée à la cohérence de caches.*

12.6.2 Catégorisation des accès au cache

Une interférence peut empêcher une instruction de récupérer une valeur dans le cache (parce que son contenu a été modifié récemment à cause d'une autre instruction). C'est ce qu'on appelle un *cache-miss*. Les analyses de cette section s'intéressent à la catégorisation des instructions en fonction des *cache-miss* observés. C'est une technique utilisée dans la littérature (voir Section 12.3.2). L'objectif de cette analyse est donc de classer chaque instruction en tant que *always-hit* (cache toujours prêt), *always-miss* (cache jamais prêt) ou *uncategorized* (le cache peut être prêt ou ne pas l'être selon l'exécution). Pour cela, on utilise l'opérateur de logique temporelle `AG` afin de s'assurer que pour toute trace d'exécution du modèle, l'instruction donnée :

- Est résolue par le cache immédiatement, dans quel cas l'instruction est classée *always-hit*.
- N'est pas résolue par le cache immédiatement et donc classée *always-miss*.
- Sinon, on la classe comme *uncategorized*.

Example 46 (Catégorisation des accès au cache)

1. `store 1` est classé `AM`.
2. `store 2` est classé `AM`.
3. `load 1` est classé `UN`.
4. `store 1` est classé `UN`.
5. `load 3` est classé `AM`.
6. `store 2` est classé `AM`.
7. `load 1` est classé `AH`.
8. `store 1` est classé `AM`.
9. `load 2` est classé `UN`.
10. `store 2` est classé `UN`.

(a) Catégorisation pour le Cœur 1

1. `store 1` est classé `AM`.
2. `store 3` est classé `AM`.
3. `load 3` est classé `AH`.
4. `store 2` est classé `AM`.
5. `load 1` est classé `AM`.
6. `store 2` est classé `UN`.
7. `load 3` est classé `AH`.
8. `store 1` est classé `AM`.
9. `load 2` est classé `UN`.
10. `store 3` est classé `AM`.

(b) Catégorisation pour le Cœur 2

FIGURE 12.15 – Exemple de catégorisation des accès cache

La Figure 12.15 montre le résultat de la catégorisation de chaque instruction de notre exemple. `AH` *correspond à always-hit,* `AM` *correspond à always-miss et* `UN` *correspond à uncategorized.*

Cette catégorisation des instructions permet de déterminer quelles instructions font varier le temps d'exécution. Cependant, certaines de ces variations ne sont pas liées à la cohérence de cache. Pour pouvoir répondre au besoin de la certification, il va donc être nécessaire d'analyser l'interférence elle-même.

12.6.3 Catégorisation de l'interférence

Pour pouvoir comprendre la cause et les effets des interférences générées par la cohérence de cache, nous proposons de les classifier en fonction de leurs effets sur le cache affecté. Cette section présente les trois catégories d'interférences proposées dans cette thèse.

Definition 88 (Interférence mineure) *On considère qu'un cache subit une interférence mineure lorsqu'il reçoit une demande en provenance d'un autre cache sans que cette demande ne nécessite d'actions de sa part. En effet, le cache a alors pris le temps de traiter une demande sans que cela n'ait eu d'utilité.*

Example 47 (Interférence mineure) *Le protocole MSI simplifié de la Section 12.2.3 présente des interférences mineures lorsqu'un cache tenant un élément mémoire dans l'état* `I` *reçoit des demandes, ou qu'il reçoit un* `GetS` *(demande d'accès en lecture) alors qu'il a l'élément en* `S`*.*

Definition 89 (Interférence de rétrogradation) *On considère qu'un cache subit une interférence de rétrogradation lorsqu'il perd les permissions d'écriture sur un élément mémoire suite à la demande d'un autre cache.*

Example 48 (Interférence de rétrogradation) *Le protocole MSI simplifié de la Section 12.2.3 présente une interférence de rétrogradation lorsqu'un cache tenant un élément mémoire dans l'état* `M` *reçoit une demande* `GetS` *de la part d'un autre cache. En effet, il passe alors à l'état* `S` *et perd ses permissions d'écriture.*

Definition 90 (Interférence d'expulsion) *On considère qu'un cache subit une interférence d'expulsion lorsqu'il perd toutes ses permissions sur un élément mémoire suite à la demande d'un autre cache.*

Example 49 (Interférence d'expulsion) *Le protocole MSI simplifié de la Section 12.2.3 présente des interférences d'expulsion losrqu"un cache tenant un élément mémoire dans l'état* `S` *ou* `M` *reçoit une demande* `GetM` *(demande d'accès en lecture et écriture) de la part d'un autre cache. En effet, il passe alors à l'état* `I` *et perd toutes ses permissions.*

12.6.4 Révéler les interférences liées à la cohérence de cache

Pour révéler toutes les interférences causées par la cohérence de cache en tenant compte des effets catégorisés, il suffit de faire en sorte que le modèle détecte les cas où l'interférence a affecté une instruction. Cela permet alors d'utiliser les outils de model checking d'UPPAAL pour déterminer quelle instruction cause une interférence sur quelle autre instruction. Ainsi, cette section identifie les interférences liées à la cohérence de cache en définissant deux ensembles finis S_A et S_E composés de triplets $\langle I_o, E, I_t \rangle$, tels que I_o correspond à l'instruction causant une interférence de type E sur l'instruction I_T. L'ensemble S_A contient les triplets pour lesquels l'interférence est certaine de se produire alors que S_E correspond à ceux pour lesquels au moins une exécution présente l'interférence en question. Combiné avec les résultats de l'analyse de la Section 12.6.2, ceci fournit à l'applicant à la fois les causes et les effets des interférences liées à la cohérence de cache dans le système modélisé.

Example 50 (Interférences dans notre exemple) *La Figure 12.16 montre les interférences entre les instructions des programmes de l'exemple. Les flèches partent de l'instruction générant l'interférence. Celles qui sont en pointillés indiquent une interférence ne se produisant pas dans certaines des traces d'exécution du modèle.* EX *correspond à une interférence d'expulsion et* DE *à une interférence de rétrogradation. La colonne de gauche correspond au programme de Cœur 1 et celle de droite à celui de Cœur 2.*

12.7 Conclusion

La documentation des architectures ne fournit pas suffisamment de détails sur la cohérence de cache pour répondre aux besoins de certification et peut même parfois induire en erreur l'applicant. Pour remédier à cela, nous avons proposé une approche d'identification du protocole de cache réellement implémenté sur l'architecture. Une fois le protocole identifié, il faut analyser le système dans son ensemble. L'approche choisie, pour répondre partiellement à cette question, a consisté à modéliser

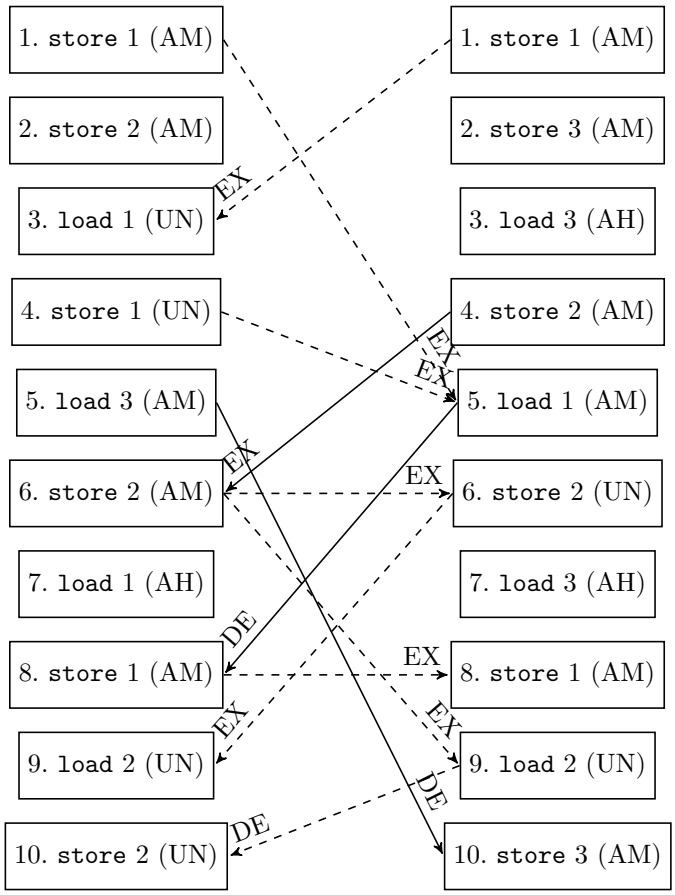


FIGURE 12.16 – Interférence dans l'exemple de modèle instancié

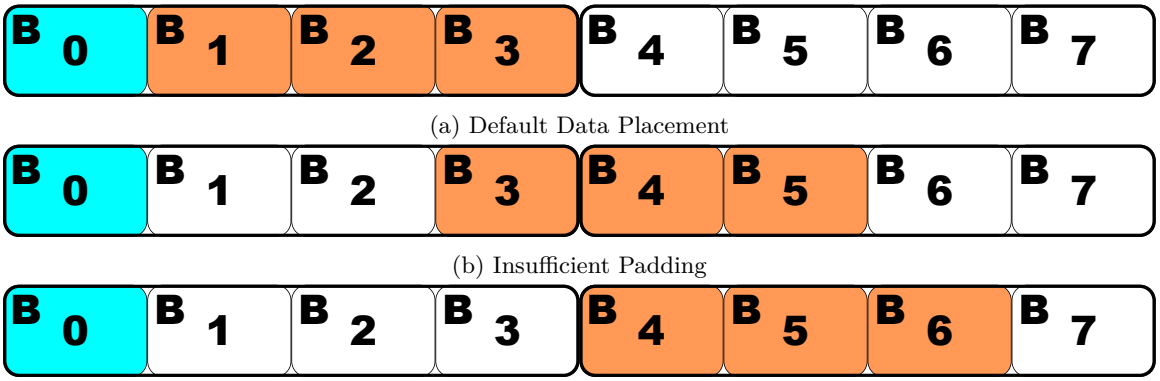
le protocole de cohérence de cache sous forme d'un réseau d'automates temporisés, ce qui permet de modéliser les composants participant à la cohérence en incluant leurs comportements bas niveau. Dans un souci de généricité, une approche modulaire a été proposée afin de facilement modifier certains composants et un outil prenant en entrée plusieurs protocoles permet d'instancier le modèle UPPAAL ad hoc. Un fois le modèle UPPAAL instancié grâce à des paramètres dont les valeurs sont à obtenir au travers de méthodes existantes, le modèle UPPAAL peut être analysé, à l'aide de techniques de vérification formelle, pour mieux cerner les interférences liées à la cohérence de cache dans le système. Il est notamment possible d'explorer toutes les traces d'exécution du modèle afin d'obtenir des informations sur les effets temporels des interférences dues à la cohérence de cache.

Plusieurs limitations ont été identifiées sur l'approche et les premiers axes d'extension de résultats seraient de réduire celles-ci. Comment prendre en compte des programmes plus réalistes et non déterministes ? comment intégrer les modèles UPPAAL uniquement focalisés sur la cohérence de cache aux frameworks plus généralistes proposant des modèles de cœurs détaillées ? Comment remonter ces informations dans le calcul de WCET et les documentations demandées par la certification ?

Appendix A

False Sharing

In the context of this thesis, memory is considered to be split into memory elements (see Definition 17), with a size corresponding to that of a cache line. This means that the minimal size that can be addressed (i.e. a byte) is that of a cache line. Realistically however, cache lines are generally able to hold 32, 64, 128 bytes. The false sharing issue stems from the fact that cache lines are the size of the blocks being loaded into caches, so loading any address within a block of that size loads data for other addresses.



(a) Default Data Placement

(b) Insufficient Padding

(c) Data Placement Without False Sharing

Figure A.1 – Examples of Data Placements

To understand what false sharing is, let us consider an architecture with two cores, $\texttt{Core}_0$ and $\texttt{Core}_1$, both having their own cache, $\texttt{Cache}_0$ and $\texttt{Cache}_1$, with cache coherence being active. Let us consider that these caches have cache lines with a size of contain 4 bytes, and that both cores are currently making numerous memory accesses to one variable each, $\texttt{Var}_0$ and $\texttt{Var}_1$. $\texttt{Var}_0$ is 1 byte long, and $\texttt{Var}_1$ is 3 bytes long. Figure A.1 shows how both variables could be placed in memory.

Caches access whole cache lines, meaning that accessing any address between 0 and 3 inclusive leads to the transfer of all bytes in this interval into the cache.

The placement shown in Figure A.1a corresponds to putting $\texttt{Var}_0$ at address 0, and $\texttt{Var}_1$ at address 1. This is the most memory efficient solution. However, this configuration, a cache accessing to either variable also performs an access to the other variable. This is not obvious from a program's

point of view, as the two variables have separate addresses. The effect is that cache coherence has to be maintained between both caches upon any operation performed on either variables, despite the fact that each core never actually addresses the other core's variable. To avoid this very costly and unrequited cache coherence, the solution is to add padding: place the variables so that a cache accessing one does not accesses the other.

Utilities to ensure a given memory block is allocated in memory with sufficient padding for a given cache line size are available in C/C++ (for dynamic memory allocation) and in popular compilers (for static memory allocations). Since this alignment is not done by default, this requires that the programmer is aware of the false sharing issue. Furthermore, it also requires the cache line be known at compilation time, which is an issue for portable programs. Figure A.1b shows a placement in which an insufficient padding has been used. By placing $\texttt{Var}_1$ at address 3, the amount of data transfers is further increased: not only is the false sharing issue still there, but now two cache lines have to be accessed when accessing $\texttt{Var}_1$.

Figure A.1c shows a padding corresponding to $\texttt{Var}_1$ being placed at address 4, which ensures that accessing either variable only transfers a single cache line and that no false sharing can occur between these two variables.

Appendix B

Model Parameters

In order to tailor the model to match the user's architecture, a number of parameters can be modified:

- LAST_ADDR, an integer, which, in effect, corresponds to the number of memory elements used by the system. The memory element 0 is reserved as a default NULL value.
- CORE_COUNT, an integer corresponding to the number of caches present in the system. The name comes from the limitation to a single core per cache.
- LINES_PER_CACHE, an integer indicating the number of memory elements that can be held in each cache.
- COMPONENT_COUNT an integer equal to the highest component ID value.
- USE_LOCK_FREE_CACHES is a Boolean controling whether cores can send new instructions before the previous ones have been resolved. If they are allowed to, the value is set to true.
- REQ_BUFFER_SIZE is an integer indicating how many pending requests a cache can handle simultaneously. Because of how automated eviction is handled, the minimal value is 2.
- IN_QUERY_BUFFER_SIZE is the number of slots available in an incoming query FIFO queue.
- OUT_QUERY_BUFFER_SIZE is the number of slots available in an outgoing query FIFO queue.
- IN_DATA_BUFFER_SIZE is the number of slots available in an incoming data FIFO queue.
- OUT_DATA_BUFFER_SIZE is the number of slots available in an outgoing data FIFO queue.

Furthermore, the time required for certain operations can be set:

- RAM_READ_TIME is the time during which the memory controller is inactive before sending the queried memory element.
- RAM_WRITE_TIME is the time during which the memory controller is inactive after having updated a memory element.
- QUERY_HANDLING_TIME is the inactivity period of a cache receiving a new query.
- DATA_HANDLING_TIME is the inactivity period of a cache receiving a new data message.

- REQUEST_HANDLING_TIME is the inactivity period of a cache receiving a new request from its core.
- DATA_TRANSFER_TIME is the delay for transfer through the bus of a data message from one component to another.
- QUERY_TRANSFER_TIME is the time needed for a query to be broadcasted by the bus.
- CLOCK_CYCLE_TIME is how long a core stays inactive after sending an instruction.

By default, the model uses the values indicated in Figure B.1, which do not correspond to any architecture in particular and were arbitrarily chosen to be small and vaguely realistic in their proportion to one another.

- LINES_PER_CACHE = 20
- USE_LOCK_FREE_CACHES = true
- RAM_READ_TIME = 200
- RAM_WRITE_TIME = 300
- QUERY_HANDLING_TIME = 4
- DATA_HANDLING_TIME = 5
- REQUEST_HANDLING_TIME = 6
- DATA_TRANSFER_TIME = 17
- QUERY_TRANSFER_TIME = 24
- CLOCK_CYCLE_TIME = 50
- REQ_BUFFER_SIZE = 3
- IN_QUERY_BUFFER_SIZE = 5
- OUT_QUERY_BUFFER_SIZE = 5
- IN_DATA_BUFFER_SIZE = 6
- OUT_DATA_BUFFER_SIZE = 6

Figure B.1 – Default Model Parameters

Bibliography

[1] R. Alur. *Techniques for automatic verification of real-time systems*. PhD thesis, Stanford University, Stanford, CA, USA, 1992.

[2] R. Alur and D.L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, April 1994.

[3] André Arnold and John Plaice. *Finite Transition Systems: Semantics of Communicating Systems*. Prentice Hall International (UK) Ltd., GBR, 1994.

[4] Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. Otawa: An open toolbox for adaptive wcet analysis. In Sang Lyul Min, Robert Pettit, Peter Puschner, and Theo Ungerer, editors, *Software Technologies for Embedded and Ubiquitous Systems*, pages 35–46, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[5] 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`.

[6] Kai Baukus, Yassine Lakhnech, and Karsten Stahl. Parameterized verification of a cache coherence protocol: Safety and liveness. pages 305–308, 04 2002. `doi:10.1007/3-540-47813-2_22`.

[7] Johan Bengtsson, Kim Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Uppaal: a tool suite for automatic verification of real-time systems. In *Proceedings of the DIMACS/SYCON Workshop on Hybrid Systems III: Verification and Control: Verification and Control*, pages 232–243, Secaucus, NJ, USA, 1996. Springer-Verlag New York, Inc. URL: `http://dl.acm.org/citation.cfm?id=239587.239611`.

[8] Béatrice Berard, Michel Bidoit, Alain Finkel, François Laroussinie, Antoine Petit, Laure Petrucci, and Philippe Schnoebelen. *Systems and Software Verification: Model-Checking Techniques and Tools*. Springer Publishing Company, Incorporated, 1st edition, 2001.

[9] Pierre Bieber, Frédéric Boniol, Youcef Bouchebaba, Julien Brunel, Claire Pagetti, Olivier Poitou, Thomas Polacsek, Luca Santinelli, and Nathanaël Sensfelder. A model-based certification approach for multi/many-core embedded systems. In *9th European Congress on Embedded Real Time Software and Systems (ERTS 2018)*, 2018.

[10] Jingyi Bin. *Controlling execution time variability using COTS for Safety-critical systems*. Theses, Université Paris Sud - Paris XI, July 2014. URL: `https://tel.archives-ouvertes.fr/tel-01061936`.

[11] 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 (ERTS'14)*, 2014.

[12] Frédéric Boniol, Hugues Cassé, Eric Noulard, and Claire Pagetti. Deterministic execution model on cots hardware. In Andreas Herkersdorf, Kay Römer, and Uwe Brinkschulte, editors, *Architecture of Computing Systems – ARCS 2012*, pages 98–110, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[13] Fabien Bouquillon, Clément Ballabriga, Giuseppe Lipari, and Smaïl Niar. A wcet-aware cache coloring technique for reducing interference in real-time systems. *CoRR*, abs/1903.09310, 2019. URL: `http://arxiv.org/abs/1903.09310`, `arXiv:1903.09310`.

[14] P. Bouyer. Lesson: An Introduction to Timed Automata, 2008-2009.

[15] Jason F. Cantin, Mikko H. Lipasti, and James E. Smith. Dynamic verification of cache coherence protocols. 2004.

[16] Franck Cassez, Pablo Aledo, and Peter Jensen. *WUPPAAL: Computation of worst-case execution-time for binary programs with UPPAAL*, pages 560–577. 07 2017. `doi:10.1007/978-3-319-63121-9_28`.

[17] CAST (Certification Authorities Software Team). Position Paper on Multi-core Processors - CAST-32, 2016. Retrieved from `https://www.faa.gov/aircraft/air_cert/design_approvals/air_software/cast/cast_papers/media/cast-32.pdf`.

[18] Sudipta Chattopadhyay, Lee Kee Chong, Abhik Roychoudhury, Timon Kelter, Peter Marwedel, and Heiko Falk. A unified wcet analysis framework for multicore platforms. *ACM Trans. Embed. Comput. Syst.*, 13(4s), April 2014. `doi:10.1145/2584654`.

[19] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite state concurrent system using temporal logic specifications: A practical approach. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '83, page 117–126, New York, NY, USA, 1983. Association for Computing Machinery. `doi:10.1145/567067.567080`.

[20] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model-checking*. MIT Press, 1995.

[21] Sylvain Conchon, Alain Mebsout, and Fatiha Zaïdi. Vérification de systèmes paramétrés avec Cubicle. In *Vingt-quatrièmes Journées Francophones des Langages Applicatifs*, Aussois, France, February 2013.

[22] Andreas E. Dalsgaard, Mads Chr. Olesen, Martin Toft, René Rydhof Hansen, and Kim Guldstrand Larsen. METAMOC: Modular Execution Time Analysis using Model Checking. In Björn Lisper, editor, *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, volume 15 of *OpenAccess Series in Informatics (OASIcs)*, pages 113–123, Dagstuhl, Germany, 2010. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. The printed version of the WCET'10 proceedings are published by OCG (www.ocg.at) - ISBN 978-3-85403-268-7. URL: `http://drops.dagstuhl.de/opus/volltexte/2010/2831`, `doi:10.4230/OASIcs.WCET.2010.113`.

[23] Andreas Engelbredt Dalsgaard, Mads Chr. Olesen, and Martin Toft. *Modular Execution Time Analysis using Model Checking: METAMOC*. PhD thesis, Aalborg Universitet, 2020.

[24] Christian Ferdinand and Reinhard Wilhelm. Efficient and precise cache behavior prediction for real-timesystems. *Real-Time Syst.*, 17(2–3):131–181, December 1999. doi:10.1023/A:1008186323068.

[25] Freescale. e6500 core reference manual, rev 0, 2014.

[26] Freescale. T4240 QorIQ: Integrated multicore communications processor family reference manual, 2014.

[27] S. Girbal, X. Jean, J. Le Rhun, D. G. Pérez, and M. Gatti. Deterministic platform software for hard real-time systems using multi-core cots. In *2015 IEEE/AIAA 34th Digital Avionics Systems Conference (DASC)*, pages 8D4–1–8D4–15, 2015.

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

[29] Nan Guan, Martin Stigge, Wang Yi, and Ge Yu. Cache-aware scheduling and analysis for multicores. In *Proceedings of the Seventh ACM International Conference on Embedded Software*, EMSOFT '09, page 245–254, New York, NY, USA, 2009. Association for Computing Machinery. doi:10.1145/1629335.1629369.

[30] 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.

[31] Damien Hardy, Thomas Piquet, and Isabelle Puaut. Using bypass to tighten wcet estimates for multi-core processors with shared instruction caches. In *Proceedings of the 2009 30th IEEE Real-Time Systems Symposium*, RTSS '09, page 68–77, USA, 2009. IEEE Computer Society. doi:10.1109/RTSS.2009.34.

[32] 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.

[33] J. E. Hopcroft and J.D. Ullman. *Introduction to automata theory, languages and computation.* Addison Wesley, 1979.

[34] Xavier Jean. *Hypervisor control of COTS multi-cores processors in order to enforce determinism for future avionics equipment.* Theses, Télécom ParisTech, June 2015. URL: https://pastel.archives-ouvertes.fr/tel-01341758.

[35] Benjamin Lesage, Damien Hardy, and Isabelle Puaut. Shared Data Caches Conflicts Reduction for WCET Computation in Multi-Core Architectures. In *18th International Conference on Real-Time and Network Systems*, page 2283, Toulouse, France, November 2010. URL: https://hal.inria.fr/inria-00531214.

[36] M. Lv, W. Yi, N. Guan, and G. Yu. Combining abstract interpretation with model checking for timing analysis of multicore software. In *2010 31st IEEE Real-Time Systems Symposium*, pages 339–349, Nov 2010. doi:10.1109/RTSS.2010.30.

[37] R. Milner. *A Calculus of Communicating Systems.* Springer-Verlag, Berlin, Heidelberg, 1982.

[38] Daniel Molka, Daniel Hackenberg, Robert Schone, and Matthias S. Muller. Memory performance and cache coherency effects on an intel nehalem multiprocessor system. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, PACT '09, page 261–270, USA, 2009. IEEE Computer Society. doi:10.1109/PACT.2009.22.

[39] Jan Nowotsch and Michael Paulitsch. Leveraging multi-core computing architectures in avionics. *2012 Ninth European Dependable Computing Conference*, pages 132–143, 2012.

[40] Xavier Palomo, Mikel Fernandez, Sylvain Girbal, Enrico Mezzetti, Jaume Abella, Francisco J. Cazorla, and Laurent Rioux. Tracing Hardware Monitors in the GR712RC Multicore Platform: Challenges and Lessons Learnt from a Space Case Study. In Marcus Völp, editor, *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*, volume 165 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 15:1–15:25, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. URL: https://drops.dagstuhl.de/opus/volltexte/2020/12378, doi:10.4230/LIPIcs.ECRTS.2020.15.

[41] 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. URL: http://doi.acm.org/10.1145/773453.808204, doi:10.1145/773453.808204.

[42] Arthur Pyka, Mathias Rohde, and Sascha Uhrig. A real-time capable coherent data cache for multicores. *Concurrency and Computation: Practice and Experience*, 26(6):1342–1354, 2014. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.3172, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.3172, doi:10.1002/cpe.3172.

[43] Arthur Pyka, Lillian Tadros, Sascha Uhrig, Hugues Cassé, Haluk Ozaktas, and Christine Rochange. WCET Analysis of Parallel Benchmarks using On-Demand Coherent Cache (regular paper). In *Workshop on High-performance and Real-time Embedded Systems (HiRES 2015), Amsterdam, 21/01/15*, page (on line), http://www.hipeac.net, janvier 2015. HiPEAC. URL: http://www.cister.isep.ipp.pt/hires2015/.

[44] Petar Radojković, Sylvain Girbal, Arnaud Grasset, Eduardo Quiñones, Sami Yehia, and Francisco J. Cazorla. On the evaluation of the impact of shared resources in multithreaded cots processors in time-critical environments. *ACM Trans. Archit. Code Optim.*, 8(4), January 2012. doi:10.1145/2086696.2086713.

[45] J.-F. Raskin. Second lecture: Basics of model-checking for finite and timed systems, Artist2 Asian Summer School - Shanghai - July 2008.

[46] Jan Reineke, Björn Wachter, Stephan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger, and Bernd Becker. A definition and classification of timing anomalies. In Frank Mueller, editor, *6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis, July 4, 2006, Dresden, Germany*, volume 4 of *OASICS*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006. URL: http://drops.dagstuhl.de/opus/volltexte/2006/671.

[47] Nathanaël Sensfelder, Julien Brunel, and Claire Pagetti. Modeling Cache Coherence to Expose Interference. In *Proceedings of the 31st Conference on Real-Time Systems (ECRTS'19)*, 2019.

[48] Nathanaël Sensfelder, Julien Brunel, and Claire Pagetti. On How to Identify Cache Coherence: Case of the NXP QorIQ T4240. In *Proceedings of the 32nd Conference on Real-Time Systems (ECRTS'20)*, 2020.

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

[50] 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.

[51] Zhenkai Zhang, Zhishan Guo, and Xenofon Koutsoukos. Handling write backs in multi-level cache analysis for wcet estimation. In *Proceedings of the 25th International Conference on Real-Time Networks and Systems*, RTNS '17, page 208–217, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3139258.3139269.