Defining and using virtual platforms traces captured for debugging MPSoCs

Marcos Cunha Pinto

Communauté
UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES

Spécialité : **Informatique**

Arrêté ministériel : 7 août 2006

ISBN : **978-2-11-129208-6**

Présentée par

Marcos Aurélio Pinto Cunha

Thèse dirigée par **Frédéric Pétrot**

préparée au sein du **Laboratoire TIMA**
dans l'École Doctorale **MTSII**

Définition et utilisation de traces issues de plateformes virtuelles pour le débogage des MPSoCs

Thèse soutenue publiquement le **29 janvier 2016**
devant le jury composé de:

M. Tanguy Risset
Professeur à Insa-Lyon, France, Président
M. Sébastien Pillement
Professeur à l'Université de Nantes, France, Rapporteur
M. Abdoulaye Gamatié
Directeur de recherche CNRS, Université de Montpellier, France, Rapporteur
M. Luis-Miguel Santana-Ormeno
Directeur "Outils de Développement Logiciel", ST Microelectronics, Examinateur
M. Frédéric Pétrot
Professeur à l'Université Grenoble Alpes, Laboratoire TIMA, France, Directeur de thèse

I dedicate this thesis to my wife, Slyrley, who has been proud and supportive to my work and has given me hope and encouragement through all challenges and uncertainties that we faced during this PhD journey. I, also, dedicate it to my children, Adelaide and Pedro Henrique, who are my motivation. They give meaning and purpose to every step I take.

Abstract

The increasing complexity of Multiprocessor System on Chip (MPSoC) makes the engineers' life harder as bugs and inefficiencies can have a very broad range of sources. Hardware/software interactions can be one of these sources, their early identification and resolution being a priority for rapid system integration. Thus, due to the huge number of possible execution interleavings, reproducing the conditions of occurrence of a given error/performance issue is very difficult. One solution to this problem consists of tracing an execution for later analysis. Obtaining the traces from real platforms goes against the recent development processes, now broadly adopted by industry and academy, which rely on simulation to anticipate hardware/software integration. Multi/many core systems on chip tend to have specific memory hierarchies, to make the hardware simpler and predictable, at the cost of having the hardware percolate towards the high levels of the software stack. Despite the developers efforts, it is hard to make sure all preventive measures are taken to ensure a given property, such as lack of race conditions or data coherency. In this context, the debugging process is particularly tedious as it involves analyzing parallel execution flows. Executing a program many times is an integral part of the process in conventional debugging, but the non-determinism due to parallel execution often leads to different execution paths and different behaviors.

This thesis details the challenges and issues behind the production and exploitation of "well formed" traces in a transaction accurate virtual prototyping environment that uses dynamic binary translation as processor simulation technology. These traces contain causality relations among events, which allow firstly to simplify the analysis, and secondly to avoid relying on timestamps. We propose a formalism to define the traces and detail an implementation to produce them in a non-intrusive manner. We use these traces to help identify and correct bugs on multi/many-core platforms. We firstly introduce a method to identify the potential cache coherence violations in non-cache-coherent platforms. Our method identifies potential violations which may occur during a given execution for write-through and write-back cache policies by analyzing the traces. We secondly focus on easing the debugging process of parallel software running on MPSoC using traces. To that aim, we propose a debugging process which replays a faulty execution using traces. We detail a strategy for providing forward and reverse execution features to avoid long simulation times during a debug session.

We conducted experiments on MPSoC using parallel applications to quantify our proposal, and overall show that complex analysis and debug strategies can be implemented over traces, leading to deterministic results in shorter time than simulation alone.

Résumé

La complexité croissante des systèmes multiprocesseurs sur puce (MPSoC) rend la vie plus difficile aux ingénieurs à cause des bugs et des inefficacités qui peuvent avoir un très large éventail de sources. L'interaction matériel / logiciel peut être l'une de ces sources, dont l'identification précoce et la résolution doivent être une priorité pour l'intégration rapide du système. Ainsi, en raison du grand nombre d'entrelacements d'exécution possibles, reproduire les conditions d'apparition d'une erreur ou d'un problème de performance est très difficile. Une approche de ce problème consiste à tracer une exécution et exploiter cette trace en faisant des analyses postérieures. L'obtention de traces à partir de vrai matériel va à l'encontre du processus de développement récent, désormais largement adoptés par l'industrie et l'académie, qui repose sur la simulation pour anticiper l'intégration matériel / logiciel. De nombreux systèmes multi-cœurs sur puce ont tendance à avoir des hiérarchies mémoire spécifiques, pour rendre le matériel plus simple et prévisible, au prix de voir percoler les contraintes matérielles vers les niveaux élevés de la pile logicielle. Malgré les efforts des ingénieurs, il est difficile d'assurer que toutes les mesures de prévention sont prises pour assurer une propriété donnée, comme l'absence de course lors de l'accès aux variables partagées ou la cohérence des données. Dans ce contexte, le processus de débogage est particulièrement pénible car il implique d'analyser des flux d'exécution parallèles. L'exécution d'un programme à plusieurs reprises est une partie intégrale du processus de débogage classique, mais le non-déterminisme du fait de l'exécution en parallèle conduit souvent à différents chemins d'exécution et donc des comportements différents.

Cette thèse détaille les défis et les enjeux derrière la production et l'exploitation des traces "bien formés" dans un environnement de prototypage virtuel qui utilise la traduction binaire dynamique comme technique de simulation des processeurs. Ces traces contiennent des relations de causalité entre les événements qui permettent, d'une part, de simplifier l'analyse et, d'autre part, d'éviter de faire confiance à des horloges globales pour synchroniser les événements. Nous proposons un formalisme de définition des traces et détaillons sa mise en œuvre qui permet de rester non-intrusif aussi bien du point de vue matériel que logiciel. Nous utilisons ces traces pour aider à identifier et corriger les bugs sur les plateformes qui ont multiple cœurs. Nous présentons tout d'abord une méthode pour identifier les violations potentielles de cohérence de cache dans des plates-formes possédant des caches mais qui n'ont pas de matériel garantissant leur cohérence. Notre méthode identifie des violations potentielles qui peuvent apparaître au cours d'une exécution donnée en analysant les traces pour les deux stratégies d'écritures de cache : "write-through" et "write-back". Finalement, Nous nous intéressons à la simplification du processus de débogage des logiciels exécutés en parallèle sur MPSoC en utilisant les traces. Dans cet objectif, nous proposons un processus de débogage qui

rejoue une exécution fautive en utilisant des traces. Nous détaillons une stratégie pour fournir des fonctionnalités d'exécution inverse pour éviter des temps de simulation élevé pendant une session de débogage.

Nous avons mené des expériences en utilisant des applications parallèles s'exécutant sur MPSoC pour quantifier notre proposition et montrer que l'ensemble des stratégies d'analyse et de débogage complexes peuvent être mis en œvre par des traces, conduisant ainsi à des résultats déterministes en moins de temps que la simulation seule.

Acknowledgments

Firstly, I would like to express my sincere gratitude to my advisor Prof. Frédéric Pétrot for the continuous support of my PhD study and related research, for his patience, motivation, and immense knowledge. His guidance helped me through all the time of research and writing of this thesis.

I wish to thank all members of the jury for their interest in evaluating this work. I wish to address special thanks to the examiners, Dr. Sebastien Pillement and Dr. Abdoulaye Gamatié, for their contribution with valuable academic feedback to polish my thesis.

I thank my fellow labmates for the stimulating discussions and for all the fun we have had in the last three years. In particular, I am grateful to Dr. Nicolas Fournel for insightful discussions during this research and to Dr. Frédéric Rousseau for his kind advice and for introduce me to SLS group.

I would like to extend great thanks to TIMA Laboratory administrative board members who offered their time, support and commitment. I thank the TIMA Laboratory and CAPES, which provided funding for this work.

I would never forget all the chats and funny moments I shared with my friends. They are fundamental in supporting me during these stressful and difficult moments.

Finally, I especially want to thank my parents, my brothers, my wife and children whose unconditional love has been the backbone for me to finish this journey of PhD study.

List of Figures

List of Tables

Contents

Chapter 1: Introduction

Contents

Walking in the footsteps of supercomputing, embedded systems for high end applications are making use of tens if not hundreds of processors today, and the growth in number of processing elements is expected to continue unaltered (see Fig. 1.1). This was made possible thanks to the amazing uninterrupted decrease in transistor size, allowing today to integrate not only a huge number of processors but also many if not all necessary non-processing components on a single chip. The aim of increasing the performance, decreasing power consumption and fitting into even the smallest form factors has been achieved with these Multi-Processor System On Chip (MPSoC). They are now mainstream solution to multimedia, mobile and automotive applications, and are even used for servers in data centers.

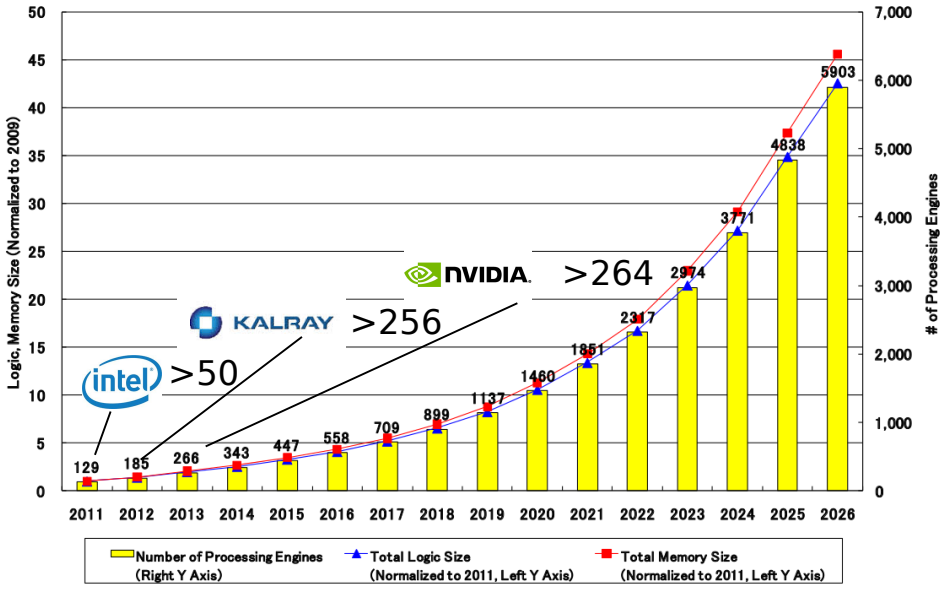


Figure 1.1: Trend in the number of processors / SoC for next years.

The increasing number of processors integrated within a single chip obliges rethinking of the complete digital design process, spanning across all hardware and software layers. Concerning hardware, a characteristic example is cache coherency, whose efficient implementation (in term of performances and resources) at a large scale is still to be demonstrated, but it is only one example among many. In making these findings, several many-core architectures were designed to avoid these hardware scalability issues, leading however to changes in programming models. At software level, programming a chip with a huge number of processors is not an easy task, and it usually leads to many, hard to find, programming mistakes. Synchronization issues, consistency issues, coherence issues, etc, are some bug sources due to a high degree of parallelism.

1.1 Development Cycle

Delivering complex architectures to an end user requires integration between the software and hardware. The classical SoC design flow in which the hardware is built and then the software is developed is not viable anymore. Indeed, time to market is such that some questions have to be answered early to start the software development process, such as "how many processors does a given application need?", "which kind of interconnection is more efficient?", "what is the most appropriate memory hierarchy?"; and so on and so forth.

Modern methodologies are indeed parallel design flows where simulation allows exploring many hardware and software combinations early. The virtual platforms play an important role to anticipate the integration phase. On the hardware side, it allows exploring different hardware configurations related to the above questions before manufacturing. On the software side, it allows integration of software in a virtual version of the final chip. It permits identifying and solving the vast majority of the potential, but very probable, problems that are identifiable only after the chip manufacturing using the sequential design flow.

1.2 Debugging Relevance

A fact is that the development of embedded software consists of 80% of MPSoC project time [1]. The debugging time can demand between 40%-70% of this time [2]. The debugging cycle therefore deserves special attention. The initial point is to understand how to well programing MPSoC and Manycore platforms and how to provide efficient debugging methods to these platforms.

Programming these architectures is related basically to two main paradigms: message passing and shared memory. Message passing is mostly applicable to distributed architectures. Using this method to share data between two nearby processors, *e.g* inside a single chip, is probably not optimal due to the quantity of messages to be exchanged. Shared memory seems to be the most adequate programming model in these architectures with high bandwidth and low latency and, indeed, has been chosen in most cases.

In shared memory systems, causes of bugs and inefficiencies in parallel computations originate mainly from synchronization and concurrent accesses to shared data. For instance, protecting data with too few locks leads to sporadic bugs whose conditions of occurrence are difficult to reproduce with conventional debugging tools. Conversely,

excessively increasing critical section lengths reduces parallelism by sequencing the execution of these sections, resulting in lower performances. A blatant example is Linux big kernel lock [3], which required 12 years of efforts to be killed. This was made possible only thanks to the development of an *ad-hoc* lock checking tool, *lockdep* [4]. Appropriate tools have shown to be useful for small scale SMP computers integrating a few tens of processors, they become mandatory for large scale many-core systems.

1.3 Objectives

This thesis aims at providing trace-based methods to efficiently debug systems. As the trace-based methods explore the information provided by traces, the actual specification of the traces is a task of great significance. First of all, we aim at capturing useful information during MPSoC simulation to produce relevant traces. In parallel executions, the order in which things happen is of utmost importance, then the traces have to be capable of providing a mean to represent parallelism. The applicability of these traces is explored by using them in two different system level debug problems. Among all system level problems that arrive during development of a new MPSoC, we choose to focus on two of them that are linked to low-level hardware/software interaction and to high-level functional software implementation.

First, the main bottleneck to scalability of MPSoCs is the cache coherence control. Hardware based solutions require a lot of space and impose the exchange of a lot of messages through the interconnection to guarantee data coherence of all caches in the system. Alternative software-based solutions have been proposed to avoid hardware cache coherence, but they rely on the programmer, and are thus susceptible to implementation bugs. We advocate that well detailed traces can aid identifying data coherence problems in complete system analysis of shared memory systems. We propose a method to identify these issues on cache non-coherent systems.

Second, debugging deterministically a parallel application is essential to correctly identify the bugs. For instance, in parallel software, the huge number of flows of execution interleaves the memory accesses in complex ways, whereas maintaining the same order of access for each execution is crucial to the debugging process. A last point is the debugging method itself. The classical debugging process usually consists of an excessive number of restarts of the application until finding the bug. Imagine the possibility of eliminating restarts, then using just one execution to represent the system behaviour and thus going forward and backward in the flows of execution. We thus propose a method capable of providing a deterministic execution of parallel programs that is also able of undoing operations, going backward in the execution. We call this method a deterministic reverse debugging.

1.4 Contributions

This thesis presents three main contributions. The first contribution consists of a formalism of traces that represents the parallel behavior of programs. These traces are captured through a virtual platform which is flexible enough to allow specifying novel relationships between events. The proposed trace formalism is based exclusively on the order of events produced in different internal components. Therefore, these traces

contain causality relation among events, which prevent relying on timestamps. The trace generation produced using a hybrid virtual platform based on DBT and TLM technologies.

The second contribution presented in this thesis is focused on detecting cache coherence violations in cache non-coherent multiprocessor platforms. Our method is based on the order of events and the causality relations detailed in our first contribution. We propose a method that identifies cache coherence issues in two different cache-write policies: write-through and write-back. A detailed set of rules using events' order to detect cache coherence violations in both policies is formalized. Each write policy is covered by a specific set of rules. The proposed rules do not take into account timestamps, since they are focused on the order of events that can be obtained on a purely functional simulation. Therefore, this analysis method relies on cache and memory accesses. When a violation is pinpointed, our method suggests corrections, according to the type of the violation, and it also proposes the right place in the source code where the correction should be inserted. Experimentations have been conducted proving the efficacy of our method. A functional prototype was implemented in order to apply the proposed rules and locate the source code correction point and the cause of the violations.

The third main contribution aims at providing a deterministic, reversible, trace-based, debugging method for parallel applications running over virtual platforms. This contribution also includes algorithm details that create and maintain an execution graph representing the behavior of the system during the parallel execution. A tool that wraps the implementation of these algorithms in *gdb*, which is an open source and broadly adopted debugger, is also proposed.

1.5 Thesis Organization

The thesis manuscript is organized as follows:

- **Chapter 2** details the context of this thesis and shows the challenges of MPSoC development;
- **Chapter 3** reviews the state of the art in this domain, focusing on how to capture traces and how to apply them to improve the debugging methods;
- **Chapter 4** presents our first contribution which addresses the definition, formalisation and capture of the traces;
- **Chapter 5** presents our second contribution which focuses on how to use the traces to identify cache coherence violation in cache non-coherent architectures;
- **Chapter 6** presents our third contribution which aims at deploying a deterministic reverse debugging to ease the debugging task of parallel applications in an MPSoC environment; and
- **Chapter 7** concludes this thesis and presents possible future directions.

Chapter 2: Problem Definition

Contents

According to the EDA vendors [5], SoC debugging phases takes the bulk of the team's efforts, up to 40% of the overall development timeline. Novel and efficient debugging tools are required to help simplify designers and engineers' day to day life.

Complete system simulation of SoCs has been and still is the subject of many research works, showing its importance in the debugging process. We advocate that simulation flexibility combined with consistent debug information help anticipate problems in subsequent development phases. The challenges this thesis proposes to focus on are discussed in this chapter.

2.1 Context

The novel SoC architectures present many architectural variations, such as number of processors, memory organization, interconnection, *etc.* These architectural differences often impact hardware and software interactions, which have to be taken into account during the debugging process. An example of novel many core architectures, Kalray's Andey manycore, is depicted Fig. 2.1. This component contains 256 cores grouped in 16 clusters of 16 cache non-coherent processor units. In this kind of platforms, complex hardware controls, such as cache coherence protocols, may not be available for cost/power reasons. Thus, software bugs of a non-usual nature happen, increasing the complexity

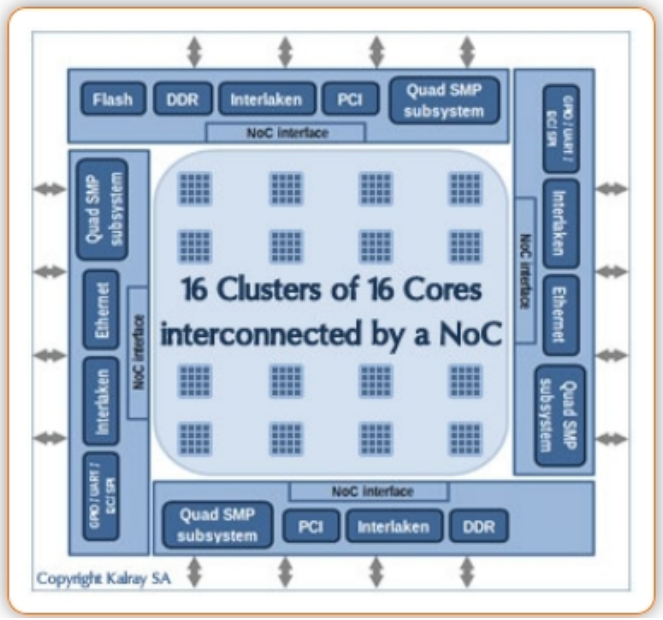


Figure 2.1: Kalray's Many Core System on Chip.

of debugging as well. This is just an example to illustrate the challenges faced by the engineers that use these new architectures.

2.1.1 Multi/Many-Core System on Chip Challenges

The increasing number of processors imposes different hardware and software challenges. In this thesis, we are interested in system level issues, mainly associated to the debugging process. We focus on three of these issues.

First, representing a trusty high parallel execution is complex, due to the massive number of components running in parallel. Capturing the behavior of these components as a set of events has been broadly used. Adding relations between events helps create a detailed vision of the system. The union of events and relations is called a *Trace*.

Usually, it is captured using a dedicated trace component in either real or virtual platforms. However, implementing a component that captures complex relationships in real platforms is merely impossible. We advocate that virtual platforms are flexible enough for capturing valuable information that could be explored to improve tools' capabilities.

Second, we observe the decreasing complexity of some hardware components allowing scalability, such as the ones present in cache control circuits. However, this complexity migrates from hardware to software. Coherent caches, despite recent technological improvements, suffer from a lack of scalability. Software managed non-coherent caches may be part of a solution. However, guaranteeing coherence in such platforms demands specific software behavior and consequently considerable verification effort. Thus, using *traces*, that represent interactions between SoC components, helps ease this process. We defend that the verification of an overall system starts with the validation of low level hardware/software interfaces, such as cache coherence control implemented by software.

Third, the functional, or high-level, software debugging process remains almost the

same tedious cyclic process. Basically, it consists of starting the application, including breakpoints/watchpoints, executing and restarting this process until finding the bug. This scenario is difficult to follow when many execution flows require the developer's attention, which can easily lead to mistakes, thus increasing even more the number of iterations. The non-determinism present in parallel software makes this scenario worst. We propose that traces can be used to improve this process, leaving the engineer focused on resolving the root of the problem, instead of caring about parallel flows of execution.

As observed, the challenges permeates low and high system levels. In this thesis, we propose debugging solutions supported by traces captured in virtual platforms to solve them.

2.1.2 Programming Models

The programming model defines the rules used to program a given platform. In novel MPSoC and many core architectures, the development of a concurrent program needs rethinking the adopted programming model. The most straightforward programming model is the shared memory model. This model is provided by Portable Operating System Interface (POSIX) Threads (Pthreads) over cache-coherent platforms. However, in cache non-coherent platforms, the most adequate programming model is still an open question.

The architecture of a multiprocessor system usually dictates the best programming model to use. There are two broadly used paradigms: *message passing* and *shared memory*, both are shown in Fig. 2.2.

In Fig. 2.2.(a), a message passing architecture is composed by many Processor-Memory pairs connected by some kind of high-speed interconnection. Each memory is local to a single processor and can be accessed only by that processor. The machines communicate by sending multi-word messages over the interconnection. As this architecture does not share any physical memory at all, the cache coherence is not an issue. However, the performance of exchanging messages between computers to manipulate shared data in these architectures depends directly on the interconnect speed. With good interconnection, a short message has a latency of $1 - 5\mu$sec [7]. Implementations of this method are standardized and available as libraries, such as the MPI (Message Passing Interface). Due

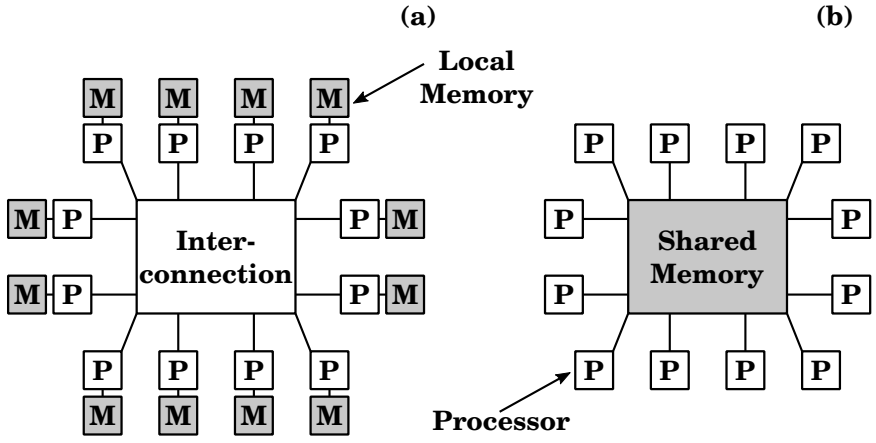


Figure 2.2: (a) Message passing and (b) Shared Memory architectures [6].

to its broadly application in multi machine clusters, some efforts are done to evaluate its performance in intra-chip cluster communication, for example using Intel's SCC platform [7].

Shared memory is present in nearly all MPSoC architectures. A simple representation of this architecture is shown in Fig. 2.2.(b). The processors communicate via shared memory. This programming model allows all processors to access the entire memory, and permits reads and writes using single LOAD and STORE instructions. The shared data access time falls to 10-50 nsec. The OpenMP and Pthread are libraries based on shared memory programming model. Programming this architecture sounds simple, actually it is far from simple when non-coherent caches are employed. From the engineer point of view, the majority of shared memory programming models relies on transparent access between cache and memory. For cache non-coherent architectures, the programming model have to be adapted or completely rethought. Programs using these new propositions have to pass through a verification phase. At the end, verification methods are important to guarantee the correct programmability of the complete system.

Hybrid approaches based on shared memory and message passing programming models are also available and they bring many improvements. This hybrid programming model consists of mixing different parallel paradigms thus maximizing the advantages. For example, considering a software stack composed of a software process and internal computations, the message passing paradigm can be applied to process level and the shared memory to internal computations. In modern cluster systems, where message passing is most applicable, memories can be treated as an additional node, which extends the concept of shared memory even in clusters. However, the mixing of programming paradigms demands expertise in applied methods.

In this thesis, we are specially interested in improving the verification process of programs developed for cache non-coherent architectures. The shared-memory systems are a sensitive case considering cache and memory interaction. We focus on this architecture due to two factors. First, the message passing approach not suffer from coherence problem./ Second, shared-memory architectures use caches as local memories which are the cause of data incoherence. We aim at providing a method to identify cache coherence violations in cache non-coherent architectures. This identification aims to be independent of the exact programming model. To do so, we rely on detailed execution traces captured in virtual platforms.

2.1.3 Virtual Platforms

Models' adoption reduce the development cost. We represent the design flow in three Design Space Exploration (DSE) levels (High, Medium and Low levels). High DSE level permits low cost exploration of many architectural configurations, thus being used at the beginning of the design flow. On the other hand, at the end of the design flow, low DSE level allows accurate verification of hardware/software interactions. Medium DSE level offers a good trade-off between accuracy and flexibility (represented by the cost of modification and configuration opportunities). The trade-offs between configuration opportunities, cost of modification and accuracy are shown in Fig. 2.3, inspired from [8].

At medium level, virtual platforms are an alternative to real prototypes, presenting several advantages. They allow embedded software development earlier than their hardware counterpart. They offer a detailed enough view of the registers and signals allowing

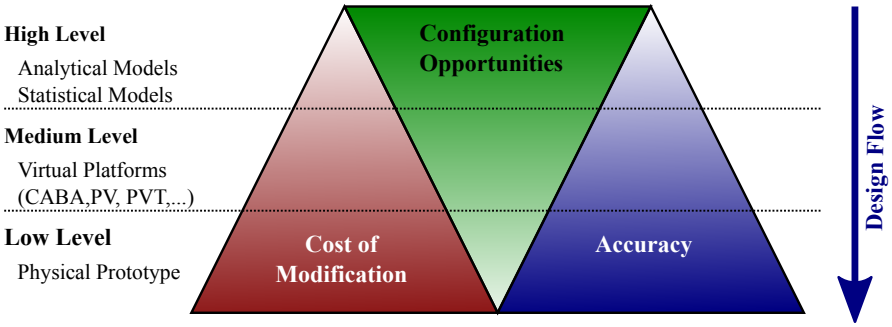


Figure 2.3: Hardware Design trade-offs at different abstract levels.

hardware accuracy. Finally, they do not suffer from physical prototype availability, which impacts testing in large teams.

Different modeling concepts are used to describe virtual platforms. Transaction Level Modeling (TLM) appears as a well accepted concept in which details of the communication between components are clearly separated from the details of the component's computation. The communication relies on the concept of channels, while computation requests take place by calling interface functions of these channel models. Implementation details of communication and computation are abstracted away in TLM and maybe added later in the development cycle [9]. Besides that, the TLM concept covers different abstraction levels: Cycle Accurate Bit Accurate (CABA), Programmer's View (PV) and PV-Timed (PVT), *etc.*

The simulation of processor requires special attention. Processors are often the most complex components within the chip, thus having high computational cost for simulation. Consequently, techniques increasing their simulation performance have been proposed. These techniques are usually different from those used to simulate other components. We present three well known techniques.

- **Instruction Accurate Interpretative Instruction Set Simulator (ISS)**: combines near to RTL accuracy, with improvement in simulation speed. It is usually one order of magnitude faster to implement and several orders of magnitude faster to simulate than RTL models. The limitation is that the simulation slows down drastically when increasing the number of processors.

- **Dynamic Binary Translation (DBT):** consists of translating the target code on host binary code, then executing it natively on the host. This translation is processed at runtime and a cache is used to store the already translated code. As the translation mechanism is fully dynamic, all computations done by the target code, even its own modification, are handled correctly.

- **Native Simulation:** is a method based on native binary code. It consists of compiling the target source code using the host toolchain thus obtaining the host binary code, instead of the target binary code, then running it on the host platform. Obviously, this method is strict and less flexible than DBT, but achieves very good simulation time.

The speed/accuracy relation is roughly depicted in Fig. 2.4. These simulation

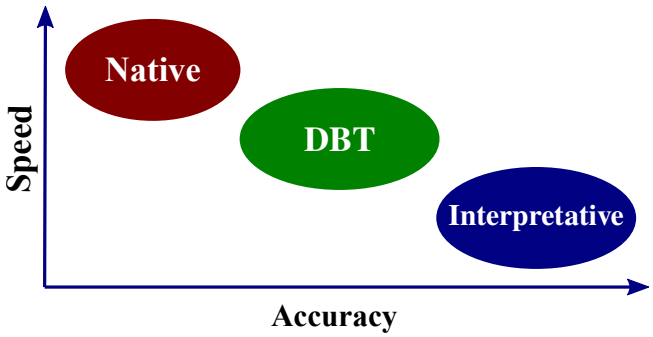


Figure 2.4: Relations between simulation speed and accuracy for processor simulations techniques.

platforms are usually available at early HW/SW integration phases, being perfectly fitted to debugging objectives.

2.1.4 Debugging and Analysis Methods

Parallel programs are a daily confrontation for engineers. Those programs consist of executing a set of logically linked instructions, also called flows of execution, in parallel. The huge number of possible flows of execution can be cumbersome for engineers, if they want to follow them in their own way. Debugging techniques can highlight errors and help find their root causes. To that aim, the broadly adopted techniques are *non-regression test*, *execution trace analyses* and *classical debugging*.

The *non-regression test* aims to verify the absence of errors in the application as compared to an older version of the application that is deemed to function properly. Its objective is to execute the application using a representative set of inputs. Then the produced new output is compared with an expected set of outputs. If a mismatch is detected, then something erroneous had happened during the system processing. This technique identifies whether an error exists or not, but it does not indicate where is the faulty point. Its main usage is for continuous integration process [10, 11].

The *execution-trace analysis* consists of capturing the execution traces, delving into them to find either abnormal behaviors or attest the good state of the system. It provides a fine grain exploration that can pinpoint the cause of the problem. It is usually addressed to investigate performance issues, system behavior adherence, *etc.* Usually, this analysis is made *post-mortem*, but it is possible to find implementations doing both operations at run-time (trace generation and analysis). The results obtained with this technique help the engineers solve problems effectively. If very low level information is present, then the size of the trace can be huge, which could be a barrier to its adoption. On the other hand, it must contain enough information for problem identification [12, 13].

The last, but not least, method is the *classical debugging*. This technique permits controlling the execution of a given application. This controllability allows the execution of an application instruction by instruction (step-by-step), addition of breakpoints and watchpoints, visualisation of the system state at any moment, and so on and so forth. The objective of breakpoints and watchpoints is, given a point inside the source code, to interrupt the execution at this desired point. This is the very classical method that is used by 100% of firmware engineers. However, the classical debug process is very slow

due to continuous human interactions [14, 15].

Those methods can be used together to leverage their strengths. At the very beginning of development, the *non-regressing testing* can be applied to verify whether a problem takes place or not. After that, if an error occurs, the *execution trace analysis* brings the token identifying which kind of error is produced and giving the approximative place inside the source code where the abnormal behavior occurs. At last, the *classical debugging* phase is necessary to identify exactly what is the cause of the problem, analysing manually the effects of previous and next instructions, the correctness of memory states, etc. In modern debuggers, the analysis is integrated inside the tool.

The analysis process is an important part of the development flow. It can be divided into two main categories: *static* and *dynamic*.

Static analysis is a process that performs the analysis without actually executing the program. The source code and object code are usually its input. Static analysis is used to formally prove the properties of a program, such as whether all variables are actually initialized before being used, or all *switch-case* structures are reached, *etc.* Furthermore, they can be used to analyse the data-flow which helps the detection of race conditions, for example. The primary advantage of static analysis is the examination of all possible execution paths and variable values, not just those invoked during execution. Thus, static analysis can reveal errors that may not manifest themselves until weeks, months or years after release. This aspect of static analysis is especially valuable in security insurance, because security attacks often exercise an application in unforeseen and untested ways [16, 17]. However, it is of a very computational complexity, and is not applicable to legacy code.

On the other hand, dynamic analysis is a process that investigates the program at runtime or using artefacts that represent its runtime behavior, such as execution traces. The primary advantage of dynamic analysis is its speed, at cost of generality, in comparison to static analysis because it covers just the runtime operations, excluding from the analysis the portions of code that are not actually performed in such execution. Dynamic analysis can play a role in security insurance, but its primary goal is finding and debugging errors [12, 18, 19].

In the context of this thesis, virtual platforms execute binary code and generate traces, as depicted in Fig. 2.5.(a). The dynamic analysis is based on these execution traces to perform the investigation. It also uses the binary code being aware of symbols, sections, and other data present in this file. Therefore, the source code can be used to feed-back information to the engineer. This scenario is presented Fig. 2.5.(b), resulting in a report containing, for example, runtime statistics and source code coverage.

2.1.5 Positioning

The context in which this thesis belongs involves multi and many processors SoC. We rely on virtual platforms to simulate the behavior of the complete system. Using TLM and DBT allow improvement in simulation speed without losing considerable accuracy. We support that such platforms can generate valuable execution traces for debugging. We believe that such traces can be used to help engineers resolve system level issues.

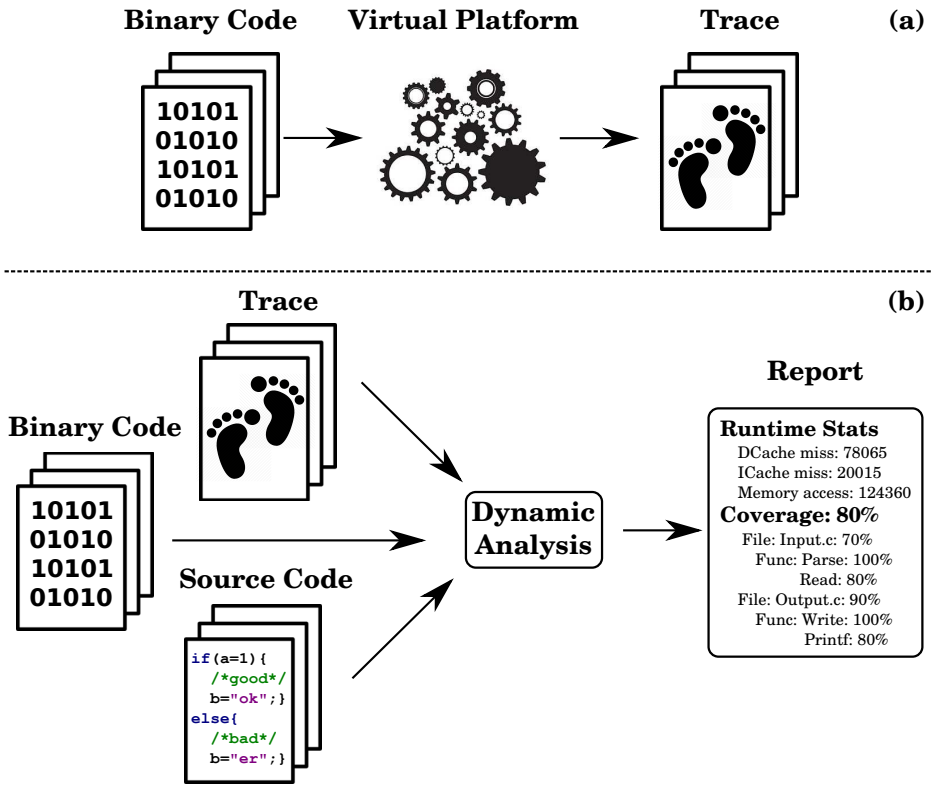


Figure 2.5: Inputs and outputs of (a) virtual platform with trace generation and (b) dynamic analysis.

2.2 Problem Definition

In this section, we describe the problems this thesis helps to resolve. We reinforce that traces improve the debugging process, however, obtaining them correctly in MPSoC is a hard task. Besides that, two other system issues drew our attention: cache coherence and classical debugging. We discuss these three topics in this section.

2.2.1 Execution Traces

Traces are a set of events and a set of relations among events that describe the behavior of a given system. The granularity of traces depends on which kind of information the engineers intend to capture. System's high-level traces are composed by the events produced at system level, which includes applications and even the operating system. In general, such a trace is used to debug the problems associated to software architecture or implementations.

A well known trace system applied to Linux is the *Linux Trace Toolkit: next generation* (LTTng), which may generate trace at kernel, library and application levels. Tools to visualise and analyse the trace are available, which can separate the events for each application, provide the callstack, memory and processor usage, for example. However, they do not capture detailed interactions between hardware components, such as those that occur between caches and memories.

Low level traces intend to catch this information giving up the processes' information

Event ID	Component ID	Type of Event	Cycle Number	Data
1	CPU_1	INSTRUCTION	1235678	PC=0x000000A0
2	CPU_2	INSTRUCTION	1235679	PC=0x000000B0
3	MEMORY_1	READ	1235680	ADDR=0xDEADBEEF
4	MEMORY_1	READ	1235781	ADDR=0xDEADBEEF
...	...	...	...	...

Table 2.1: Low level trace example.

and interactions. In this case, the events are generated for each single executed instruction, containing detailed data about the execution of the instruction, *e.g.* the modification made in memory, the new register values, time spent to execute it, so on and so forth. These traces are named *Execution Traces*. An example of low level execution trace is presented in Table 2.1.

Low level traces can be applicable to help solve hardware/software interaction problems, which are usually difficult to solve. In the MPSoC context, the increasing number of possible interactions among components make this task even more complicated.

Parallel architectures present well known complications. The concurrent accesses to memories made by two different processors at the same time cause the *congestion* problem, which can directly impact performance on massively parallel programs. The identification of such problems can be addressed using traces, when they convey the correct information. Other parallel problems, like coherence, consistency and synchronization can be also explored using traces. Effectively, for each problem the information inside the events may change. To address the congestion problem, the information present in traces given Tab. 2.1 could not be enough, since it does not indicate which component causes memory events 3 and 4.

Let's take Fig. 2.6 and Tab. 2.1 as an example. Suppose two situations where there is no explicit information about relations between events. First, Fig. 2.6 shows the bare events. Pointing out which component causes both memory events (3 and 4) is merely impossible, except when making assumptions. Fig. 2.6.(b) and Fig. 2.6.(c) represent two scenarii with different assumptions. In Fig. 2.6.(b), CPU_2 causes both memory events, thus normal memory access occurs. However, Fig. 2.6.(c) shows that both CPU_1 and CPU_2 access MEMORY_1, which delays event 4. This leads to a problem: wrong assumptions may result in wrong conclusions.

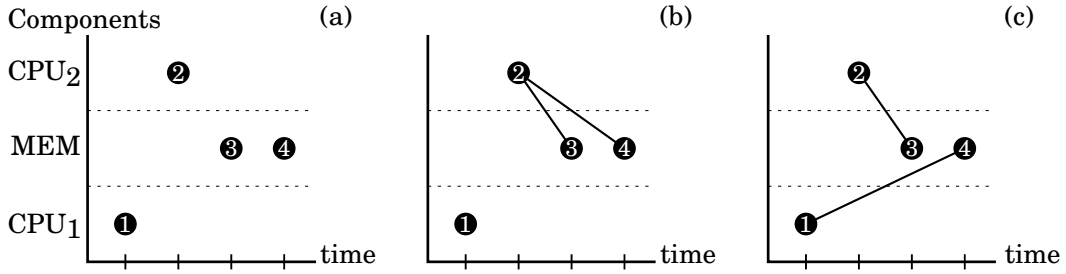


Figure 2.6: Trace examples (a) without relationships, (b) and (c) with possible relationships.

Usually, a trace is a simple dump of events which represents information similar to the one accessible to the runtime tools. These traces are either collected using code instrumentation or trace ports. In the former case, software events such as function calls or variable values are recorded, in the latter case, they are hardware events, such as instruction execution or memory accesses. The specification of the trace plays an important role to lead to conclusive analyses.

In order to reduce its complexity, the trace analysis can take advantage of additional information related to the execution: precise event descriptions, events that are otherwise difficult to gather, and relations between events. For example, concurrency of data accesses needs to know not only the initiation of the requests (execution of the load or store instructions), but also their arrival to memory, and the relation between the start and arrival of each request. Retrieving the initiation events is always possible, whereas arrival events can only be gathered at hardware level. However, the relationships are merely impossible to collect in both cases.

In real platforms, to obtain the desirable information, additional hardware is mandatory. The more detailed the required information is, the more costly it is to obtain. However, real platforms have physical limitations, such as link bandwidth to transport the traces. As expected, additional hardware results in additional power consumption. These points limit the trace capture capabilities in real platforms. On the other hand, virtual platforms already offer a great opportunity to collect detailed traces, while further model instrumentation allows maintaining causality chains between events [20]. Moreover, collecting traces through this mean is non-intrusive and does not suffer from bandwidth or memory limitations. Then, gathering all spread information present in the system in a structured trace is a role that virtual platforms can play honorably.

Another point to consider when capturing traces is intrusiveness. The intrusiveness is a characteristic that describes how "deeply" a system is modified to produce information. This modification has several meanings, it can represent either a physical modification or a source-code modification, for instance.

Approaches that have high intrusiveness are not desirable because they can alter the results. Methods employing additional hardware components can obtain different results when the trace capture is disabled.

This phenomenon is also observed at software level when source code modifications are imperative to trace generation. Consequently, the system may be exposed to bugs when the traceless version is deployed.

For parallel program debugging, an ideal trace system should be neither hardware nor software intrusive. In the context of virtual platforms, hardware intrusiveness has no impact on simulation results, whereas the increase in simulation time is observed. However, the software intrusiveness may be present because it depends on the adopted trace system.

2.2.2 Some Challenges in Debugging Multicore Systems Using Traces

In parallel programs, common problems faced by engineers are, *e.g*, race conditions, cache coherence and memory consistency. MPSoC, as parallel execution engines, are now facing these issues. All of them associated to parallel programming paradigms, which are present in MPSoC. We believe that execution traces can be used to help solve these problems due to their valuable representation of the behavior of the system.

We want to demonstrate the relevance of execution traces in MPSoC design. For that, we intend to make use of the versatility of execution traces for improving the debugging process in two ways. First, to identify hardware/software interface level problems, thus targeting low level system issues. Second, to include new functionalities to the debugging process of parallel programs, thus helping to solve high level system issues.

Cache coherence is a hardware/software level problem that arises from novel architectures. Several novel architectures do not provide hardware-based cache coherence. These platforms present good scalability for many core system on chip designs, but complex programmability. We believe that the lack of tools dedicated to help the system integration and debugging is one of the reasons that makes the adoption of such platforms difficult.

The debugging process for parallel programs is not simple as it is for sequential programs. Since execution traces allow reproducibility of a parallel program execution, we believe that using them for debugging is a part of the solution.

In the next section, we discuss about the characteristics of cache non-coherent architectures and debugging issues in parallel programs.

2.2.2.1 Cache non-coherent Architectures

In novel MPSoC and many core architectures, we observe an increasing number of features migrating from an absolute hardware control to a shared hardware/software control. Cache coherence is one of them.

The vast majority of multi/many core systems support shared memory. To access this memory usually the processor uses small intermediary memories, named caches, that reflect the memory contents. It improves the overall performance as it decreases the number of memory accesses thanks to program temporal and spatial localities. In mono-processor SoC, the most common components that write in memory are the processor and DMA, then maintaining the cache contents updated is an ease task. However, when considering MPSoCs, multiple processors may have access to multiple copies of a datum in their respective caches. When one of them proceeds a write operation, the data present in the other's cache should not be used anymore. When one processor uses it, a cache coherence violation occurs. Indeed, cache coherence ensures that a processor never catches an out-of-date value from its own cache.

The cache coherence problem has been tackled in different ways. The most broadly applied are hardware solutions. Well known protocols based on Modified-Shared-Invalid (MSI) and its variations are often found in commercial design.

However, classical hardware implementations suffer from scalability [21, 22]. Among the problems faced by hardware solutions, the most notable are high power consumption and verification complexity. First, most of the classical approaches, such as snooping protocols, incur loss of performance related to latency and power. Indeed, acknowledgment and invalidation messages may eat up more than necessary bandwidth and power. Second, the formal verification of these protocols is complex due to many transient states. The time consumed by verification discourages the optimization of well established protocols. Third, the directory protocols incur storage overhead to track sharer list.

This issue has opened a research area, which aims at looking for novel hardware protocols and software cache coherence implementations. Software solutions are viable alternatives, indeed, the idea is to remove the hardware complexity by having more

complex software, which may permeates the ecosystem, *e.g.* compilers, languages, Operating System (OS), application itself, so on and so forth. Software cache coherence has been largely studied in the past, but many-core architectures benefit from much lower latencies and higher throughput than discrete machines and many new proposals have been done recently on this topic.

Multi and Many core providers are somehow creating cache control substitutes or even removing them. Intel and Kalray provide cache non-coherent architectures. Intel has the SCC (Single-chip Cloud Computer) which is mainly available for researches and Kalray has the MPPA (Massively Parallel Processor Array) which is a commercial many core. The SCC has 48 distinct cache non-coherent physical cores, whereas the MPPA has 256 of them. Side-by-side, ARM and MIPS provide mixed architectures. In this case, their solutions are cache coherent, however, the coherence can be disabled. The cache coherence circuit area remains the same for ARM and MIPS solutions, whereas the power consumption decreases disabling the coherence control. From a software point of view, these platforms can be treated as cache non-coherent platforms

In this thesis, we also focus on identifying cache coherence issues that may occur in this kind of platforms. To identify an issue is to provide enough information to the engineer about write and read accesses that cause the problem.

Despite the participation of these important players, programming cache non-coherent architectures lacks efficient programming methods and debugging tools.

2.2.2.2 Translation Lookaside Buffers and Virtual Caches

The system components are also affected by the coherence problem. The most notable impact happens when a Memory Management Unit (MMU) is inserted in the system. In this case, the address may be physical or virtual, which forces rethinking about how to correctly handle both addresses. The vast majority of MMU implementations relies on Transaction Lookaside Buffers (TLB) to cache recent translations.

In simple architectures, *i.e* ARMv5, there is a single TLB that includes a page walk hardware. On a TLB miss, the hardware fetches the page table entry from memory to update the TLB. In ARMv7, a more complex system composed of a main and a micro TLB is implemented. The micro TLB is attached to the first level (L1) cache and provides a fully associative look-up of virtual address in a single clock cycle. The main TLB provides a centralized source of lockable translation entries, accessing it usually takes a variable number of cycles. It is implemented as a combination of a fully-associative, lockable array of four elements and a two-way associative structure. The main TLB catches the misses from micro TLBs. As observed, both architectures may suffer from coherence problems. In the first one, the valid information is available in memories and cached in each TLB. In the second one, the main TLB concentrates the information present in all micro TLBs, thus all caches must be aware of updates.

Another aspect of caches can be explored, like the adoption of Virtual Caches. Conventional and broadly used caches manage physical addresses, yet caches can also be accessed with virtual addresses. Both options are depicted in Fig. 2.7. The main advantage of using virtual caches is that the latency to recover the physical address is off the critical path. However, this technique has two main drawbacks: 1) The majority of coherence protocols operates with physical addresses and 2) the virtual caches introduce the problem of *synonymous*, *i.e.* two seemingly different virtual addresses map to the

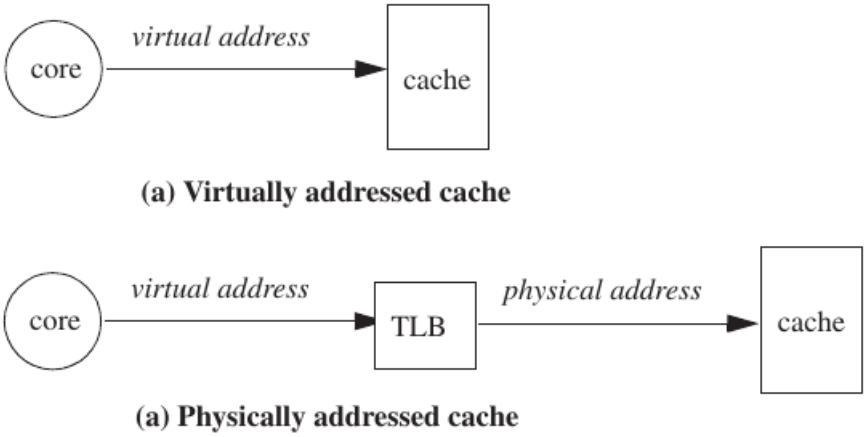


Figure 2.7: Physical and virtual addressed caches [23].

same physical address. The former case can be addressed by including a TLB in each cache, thus when a coherence request arrives a reverse translation is processed and the virtual address is recovered. The latter case is a more complex problem to deal with, since it needs a more complex mechanism to reverse the address translation, because it must know if a physical address is mapped to one or more different virtual addresses.

The TLBs and Virtual Caches represent just two factors that influence coherence, there are other, like DMAs and upper cache levels. It reinforces our argument that guaranteeing the coherence of the system is very complex. We advocate that traces help find incoherent accesses even for TLB's caches.

Assuming all hardware/software issues resolved, system bugs may delay the integration process. In the next section, we discuss the impact of system level bug and the challenges faced during the debugging process.

2.2.2.3 Deterministic Reverse Debugging

In addition to previous interface challenges, multiprocessor systems face parallel software challenges. Among those problems, the determinism of a program is one of the most difficult to deal with.

We define a deterministic program as a program in which, for each program execution, the order of operations does not change. Thus, the result produced by each operation remains the same using the same set of inputs. For example, a sequential program that multiplies two numbers always results in the same order of operations. In a sequential program, the order of operations is preserved.

However, the order of execution can change in parallel programs. The flow of execution is a sequence of logically linked instructions. Parallel programs consist of basically parallel flows of execution, which can be executed by different processors. In this case, there is no way to preserve the program's order. Thus, we classify these programs as non-deterministic programs.

We can observe two different flows of execution of a non-deterministic program Fig. 2.8. The program consists of writing the variable `X` by two different flows of execution (`FE1` and `FE2`). Let's consider a write operation as an atomic operation, *i.e.* when a flow

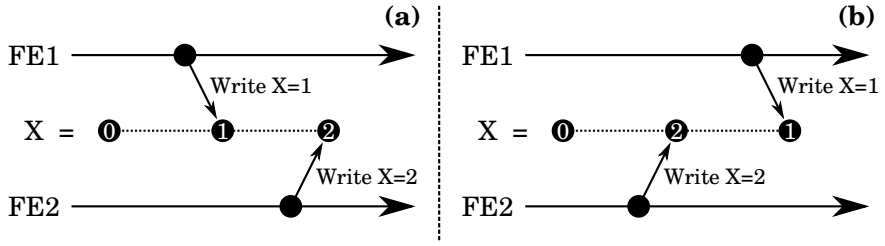


Figure 2.8: Two executions of a non-deterministic parallel program.

of execution accesses `X` another one cannot access it. Fig. 2.8.(a) represents the first execution, where `FE1` and `FE2` write `X` respectively. In the second execution, the order of execution is swapped, resulting in the second execution presented Fig. 2.8.(b). The crossing arrows represent the moment where a flow of execution writes in the variable `X`. Then, we observe different results present in `X` for each execution.

A program can be deterministic regarding the results produced instead of the /operations' order. Indeed, the results produced are more important than the order of the executed operations. Fig. 2.9 represents a variation of Fig. 2.8, however, in this case, the values obtained for variable `X` are the same for every execution. However, this definition does not fit our objective, that is to reproduce the same order of operations instead of results.

Environments of parallel program development provide support to help maintain determinism in portions of the code. This support can be provided by either hardware features, such as *test-and-set* instructions, or low-level software, such as *kernel* operations, which use hardware features. They can also be provided by kernel layers, such as function calls, that allow high-level software development. Independently of the adopted method, a parallel software is a non-deterministic program by nature. We can easily deduce that obtaining precisely the same execution helps the engineer in solving system level problems.

Parallel software usually relies on methods to guarantee parallelism. As examples we have the *scheduler* and the *deployer*. The *scheduler* decides which flow of execution has to be executed based on a given set of rules, whereas the *deployment* decides in which processor a flow of execution is executed. Debugging these portions of codes is not an easy task because they are responsible for guaranteeing somehow the determinism of an application.

Obtaining a deterministic execution from a non-deterministic software is a problem that has been addressed for long. A well adopted solution is a method called *deterministic*

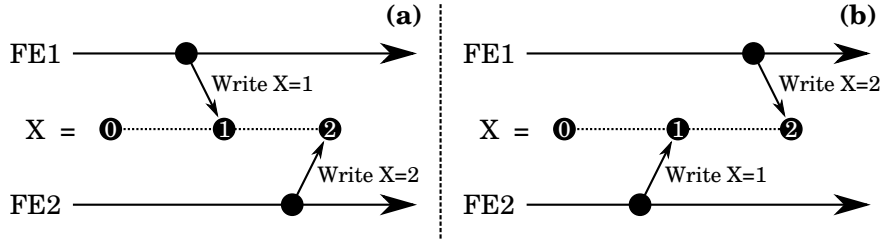


Figure 2.9: Two executions of a deterministic value program.

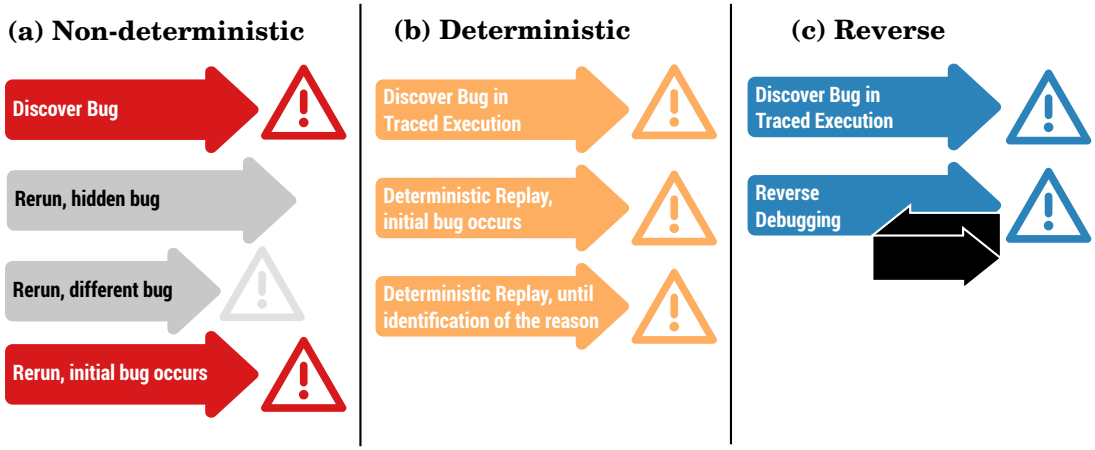


Figure 2.10: (a) Non-deterministic, (b) deterministic and (c) deterministic reverse debugging.

replay, which consists of capturing somehow the erroneous behavior, then deterministically reproduce it. After that, the engineer explores the deterministic execution to find the problem. The method to capture the behavior relies on traces. Obviously, the fidelity of the reproduction depends directly on the captured traces.

Deterministic execution improves the development of parallel software, however, the process of finding the bug remains the same conventional method. In a conventional debug approach, when the program just crashes, the developer puts a breakpoint in some place in the code before the faulty point, and re-executes the program waiting for the breakpoint to be triggered. Then, he goes forward step by step, guessing if he should or not execute a function as a whole, checking the variables' values, ..., until he reaches the problem. If the cause of the crash is found, then the source code is modified and new tests are launched. This is a perfect scenario, as the breakpoint was placed at the right place and the right modification could solve the problem in one attempt. Unfortunately, this doesn't usually work that way with large code base, many people working on the code, the maintainer and the developer not being the same person, and so on. So discovering the correct place for a breakpoint is like looking for a needle in a haystack. Consequently, often many new executions are made, modifying the breakpoint place, and redoing the step-forward process. Daily experience shows that even the most attentive engineer can make more step-forwards than necessary, leading to a complete restart. Alternatively, he can try to reverse instructions in mind, but this soon becomes obscure and unmanageable, especially when the faulty point is far from the breakpoint or in multi-threaded programs. Therefore, a lot of restarts are an intrinsic part of the debugging process, but even with many restarts, the sporadic bugs continue to be undebuggable, which is common on non-deterministic debugging as show Fig. 2.10.(a). Ensuring the determinism, at least, guarantees the same execution every time. A simplified view of deterministic debugging is given Fig. 2.10.(b).

As observed, obtaining a deterministic execution is a necessary but not sufficient action to ease the debugging process of parallel software. In this case, the developers have a process, although simplified, which is equivalent to the conventional sequential debugging method.

The bug hunting starts by pinpointing the problem, then the engineers will find and correct it. However, lack of functionalities on debug tools leaves engineers by themselves. Debugging a huge number of flows of execution without appropriate tools is one reason. Alternative functionalities must be incorporated to debuggers to help them. Deterministic replay has arisen as one solution, but it may be necessary to have many replays until spotting the root cause. Another alternative method is to integrate reverse execution in debuggers. The reverse execution consists of recovering the previous state of a given instruction, *i.e.* undo the modifications made by it.

The main advantage of reverse execution is that it allows the engineer to go forward and backward in execution without restarting the entire execution. Such restart imposes a lot of drawbacks, such as simulator reinitialization, waiting to reach breakpoints, debugger reconfiguration, *etc.* Fig. 2.10.(c) shows a simplified view of reverse debugging of a parallel software using traces.

Undoing an instruction in a multiprocessor systems is not as simple as it seems. We list three of problems researchers face for providing this functionality:

- **Past States**: intuitively all previous internal values of a processor must be restored bringing it back to its previous state, which means "un-execute" the instruction. However, almost all program instructions do not store previous states, hence, destroying them. For example, an instruction that writes a new value in a memory position overwrites the previous value. Methods to preserve or to recover the previous values are available, such as traces and checkpointing. Using traces consists of storing the state of each operation in a structured way. Checkpointing consists of recovering the state of the system until a certain saved state then execute in forward way. The travel to the past has to take into account the states of all components, such as memories, processor's register and other peripherals.

- **Non-Determinism**: the determinism is fundamental to properly debug a parallel software. The traces can be used to reconstruct deterministically the states of the system, however, they just represent system progress in forward way. Deterministic reverse representation has to be carefully studied because undoing an instruction implies recovering such information in the past. However, how far from the point the recovery process has to go through can lead to unacceptable search time. As checkpointing consists of recovering the system state based on saved state of the system, then there is usually a gap between this saved state and the breakpoint resulting in a set of instructions that have to be completely re-executed. This gap of instruction have to be also deterministically executed, then operations of non-deterministic results have to be stored, such as I/O operation results.

- **Intrusiveness**: It impacts the overall debugging process. The considerations presented in Section 2.2.1 are applied to reverse debugging. Some propositions consider code instrumentation to recover the last executed instruction, which is somehow a type of trace capture, however, they are specific to reverse debugging. We advocate that traces have to be captured in a non-intrusive way and must contain information that can be explored to eliminate system level bugs.

2.3 Conclusion

In this chapter, we present the context of this thesis. We expose the advantages of using virtual platforms with trace capturing features to help the debugging process. We also discuss the challenges engineers face during this process. Questions about how to debug novel architectures are still open. In this context, this thesis aims at answering the following questions:

- Question 1: What is the traces' definition that allows capturing the parallel behavior of applications running in simulated MPSoC platforms? Execution traces can capture the hardware/software interactions, but the way they are captured and which information must be captured in order to represent the entire execution is an important issue. Furthermore, this information has to allow the identification of problems in parallel programs efficiently.
- Question 2: How to use traces to identify cache coherence violations efficiently in cache non-coherent architectures? The software cache coherence comes back in sight in researches due to the novel architectures. Now, guaranteeing the coherence is a challenge in MPSoC systems, thus efficient ways to identify misbehavior are mandatory. It has a direct impact on improving the scalability of MPSoCs which will be achieved if and only if all subsystems scale well. In this case the coherence challenge is an issue that permeates hardware and software domains.
- Question 3: How to provide a deterministic reverse debugging method that can be applied to parallel software? Debugger tools lack bringing to engineers innovative features, like reverse debugging. Including this feature in virtual platforms avoids re-simulation of entire system while debugging. This method can improve the productivity during the debugging process of parallel programs.

Chapter 3: State of the Art

Contents

This chapter presents the state of the art of capturing traces and performing analysis and debug of MPSoC. First, we present works related to capturing relevant traces in real and virtual platforms. The information that is produced by the system, which is embedded in traces, is briefly discussed. Second, we present how a sort of analysis can be performed for MPSoC and how traces can help in this task. Third, we show how the reverse debugging is performed until now. Finally, we draw some conclusions about multiprocessor environment and how traces can be used to address effectively the problem and help ease verification and debug process.

3.1 How to capture a Trace?

Dumping hardware and software information about the execution of software is a usual practice. The purpose of these dumps is usually to provide human readable information to help have a view on system behavior. The most well-known technique used in dump generation is the use of *print* statements. As opposed to raw dumps, traces contain

structured information with clear semantics, so that they can be used by programs for profiling, debugging and so on.

Collecting traces poses several issues:

- access to the information to be traced. Which function or instruction is being executed, what is the value of a variable or memory cell?
- gathering and outputting the information. Which buffering capabilities and throughput is requited for collecting and sending data out of the system?
- relations between events. Which events occurred before a given event? Which event is the source of a given event?

As of today, these issues have been tackled by various means.

3.1.1 Code annotation

Code annotation is a process that consists of adding new code lines in source code. In the trace generation context, these new lines aim to create events that will be in the trace set. These modifications can result in system behavior modification. Usually this method implies high intrusiveness. Despite that, it is broadly used in the industry [24].

Well-known code annotation solutions available for MPSoC are LTTng and KPTrace.

LTTng is an open-source tracing framework for Linux, being able to capture events at kernel and application levels, allowing understanding of interactions between multiple software components of a given system. However, it imposes some overhead due to probing. It also presents some limitations regarding its time-stamp precision and the addition of new trace-points during the execution. This implementation is part of many Linux distribution mainstreams.

These limitations are overcome by KProbes.This tool consists of a kernel instrumentation mechanism, which provides a facility to execute a user-defined handler when an instrumentation point is hit. It can use accurate clocks based on performance counters and allows adding probes dynamically to a running kernel. However, it can only trace kernel functions. Its intrusiveness has a weak impact on execution time, however, of course, intrusiveness depends on probe density.

The events are related to functions and variables, but cannot expose instruction executions and memory accesses in detail. In general, this kind of solution relies on the relations between events which is maintained through time stamping, that requiring a global, software managed, system clock. Such software clocks are difficult to keep reliable on multiprocessor platforms. This approach is very intrusive thus modifying the software behavior.

3.1.2 Dynamic Instrumentation

An alternative to code annotation is dynamic instrumentation. It relies on including probes during the application execution to analyse the application behavior. The probes contain conditions which, when triggered, produce an event. Such conditions can be described in specific languages. Their inclusion and removal occur dynamically. It implies that when no probe is present, the intrusiveness is almost eliminated. Different from code annotation, these commands need a specific run-time. For example, in the Linux environment, *SystemTrap* provides a runtime that allows engineers to probe kernel-space

events without having to resort to instrument, recompile, install, and reboot the kernel. Basically, it consists of monitoring the interactions between the applications and the kernel [25]. These probes which are triggered when an application calls a kernel function are configured through scripts.

DTrace [26] is a dynamic tracing framework created by Sun Microsystems to help find problems in kernel and applications. DTrace provides a high level language for describing observation and event generation. Special consideration has been taken to make DTrace safe to use in a production environment. For example, there is minimal probe effect when tracing is underway, and no performance impact associated with any disabled probe; this is important since there are tens of thousands of DTrace probes that can be enabled. New probes can also be created dynamically.

This technique presents some drawbacks. The density of probes affects the performance directly.Furthermore, the hardware/software interactions cannot be explored using this technique, being mostly applied to resource monitoring, such as memory usage and processing time. Moreover, it can not extract the low level information for hardware/software analysis, such as processor and cache interactions. Thus, this technique could not be efficiently applied in memory consistency and cache coherence analyses, for instance.

Dtrace is used by [27] aiding the implementation of a proposition based on *doors* for Inter Processus Communication (IPC) in Solaris Operating System. Detailed information about this technology can be found in the book [28].

3.1.3 Hardware Trace Ports

A trace port is a hardware component that captures low level interactions between hardware components. This approach is different from the two previously presented because it relies exclusively on hardware, thus no software configuration and implementation are necessary, except for these necessary to configure the trace port itself. The trace ports are important to low level analysis because low level events can only be captured at hardware level through the use of a dedicated hardware IP for processors [29, 30], or specific hardware components[31, 32].

The trace ports can either reside completely or partially inside the chip. In the former case, an external device shares the responsibility to capture the traces. The internal and external solutions proposed by ARM are depicted in Fig. 3.1. On the left hand side, the internal Embedded Trace Buffer (ETB) increases the complexity inside the chip as it needs more internal RAM to implement a buffer, consequently increasing the silicon area. On the right hand side, the external Trace Port Analyzer (TPA) decreases the internal complexity, but obliges complex external device to capture the traces.

To cope with the hardware contrasts and implementation strategies, industrial and IEEE standardizations are proposed. A comparison around this subject can be found in [33]. A traditionally used interface for test and debug is the IEEE 1149.1 Test Access Port (TAP), which is the basis of other standards. Nexus 5001 extends the TAP by defining a message-based interface between on-silicon instrumentation and debug tools for the automotive industry. The IEEE P1149.7 interface is the successor of TAP providing retro-compatibility, while adding new functionalities. The MIPI Test and Debug working group leverages the IEEE P1149.7 interface and adds high bandwidth, unidirectional trace interfaces for smart phones, and other network devices. Other groups have adapted standards to address specific needs for each industry segment.

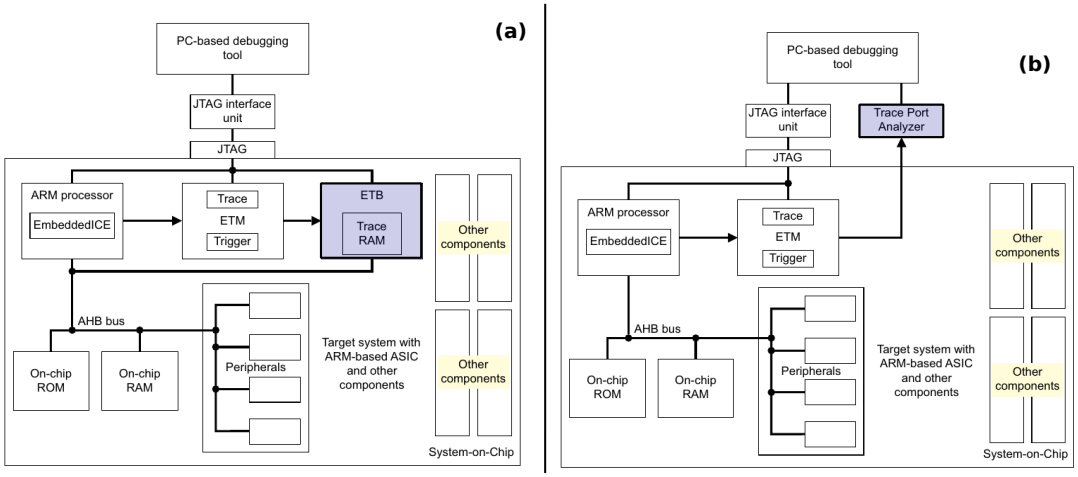


Figure 3.1: (a) Internal ETB and (b) External TPA trace capture architectures compatible with ARM processors.

In all of those cases, the internal hardware activity is captured, then an event is posted in the trace port, after that, it is exported to a host computer for analysis.

Some analyses require low level events, as it is the case for debugging concurrent programs. These IPs also export the information via physical interfaces, but in that case the amount of data is huge, as hardware resources being limited, it is necessary to select ahead of time which events to trace. Even if compression techniques can be applied [34], this amount of data is still an issue.

3.1.4 Capturing Traces on Virtual Platforms

An alternative solution is producing traces from virtual platforms, in which the simulator and the models are modified to this aim. This gives access to low level events and remains non-intrusive, thus the only effect of trace production is that it decreases simulation performance. As far as relations between events are concerned, time stamping events becomes an issue of the simulation platform. Besides, some analysis can take advantage of more advanced relationships, which cannot be recovered from timestamps, such as cause/effect relations. For example, verifying the adherence to a given consistency model by only using load/store traces is NP hard [35]. However, when considering additional information about execution, such as the order of write events, the complexity changes. For example, when the order *Read-Modify-Write* of a given address is available, the complexity of coherence identification falls from NP-Complete to $O(n)$, where n is the number of events [35]

In other cases, traces are able to link transactions, being used to identify the root causes of latency and throughput violation, as presented by [36] for TLM simulation. This method describes trace format that provides the order of events. Since it is based on TLM simulation, each event can be attached to an accurate time-stamp. Therefore, data mining techniques are applied over the trace to identify the concurrent events, thus reducing the number of events. After that, they apply classification methods to segregate the problems, which are *concurrent request, interleaving read/write and bank conflict.*

Congestion and hot-spot problems are addressed by [37] and [38] also using traces

captured in TLM platforms.

Obtaining information of that level of detail in real platforms can be extremely expensive, considering costs and software impact, and may modify the software behavior. Furthermore, using software techniques is equally problematic, as they are even more intrusive. These examples make it clear that for complex analysis the usage of real platforms and software techniques are impracticable and virtual platforms can be used as a solution that produces almost zero intrusiveness and delivers accurate traces.

Most of MPSoC processors are currently modeled at ISS/Interpretative, however, it is intrinsically limited to small systems [9]. As instruction interpretation becomes a bottleneck when the number of processors grows, and native simulation is not flexible enough, then alternative approaches have been proposed, among which DBT appears as the most promising one [39–42]. A hybrid approach, where TLM and DBT are used to speed-up the simulation, is presented in [40] In [39], the authors also propose a hybrid approach, extending functionalities, such as Dynamic Voltage/Frequency Scaling (DVFS) support. The improvement proposed by [41], in this hybrid context, provides performance estimation about the system, such as the number of executed instructions, the number of memory and I/O accesses, *etc.* A flowchart of the translation of target binary code into host code with an intermediate representation is represented in Fig. 3.2.

Binary translation is a process generating an executable host binary code based on target binary code. Therefore, before the final host code, an intermediate phase can be necessary. In this case, an intermediate code is generated. This additional phase aims at easing the addition of new targets and hosts. For example, to support a new host processor, the translation for intermediate code to host code must be done. Then, this new host supports all targets that implement translation to intermediate code.

However, to the best of our knowledge collecting system trace in a TLM simulation

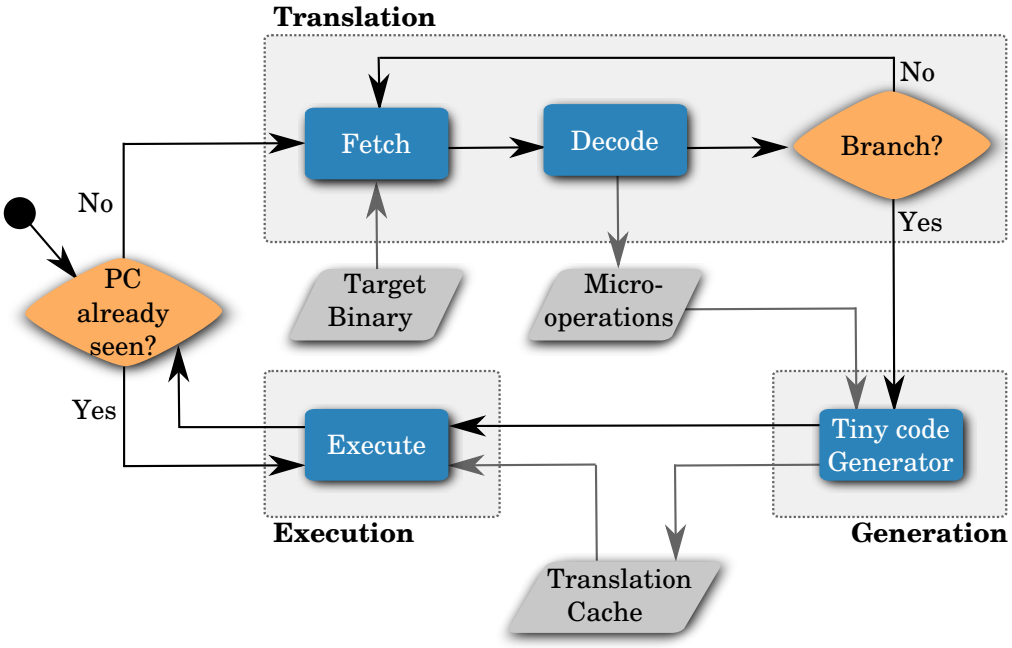


Figure 3.2: Dynamic Binary Translation using an intermediate representation simulation model [39].

framework that uses DBT for software execution has not been addressed previously.

3.2 Analysis Methods

In this section, we present methods aiming at analysing hardware/software interaction, which can be used to address the cache coherence problem. We focus on trace based models as an effective alternative to the costly exclusively formal methods. We show how they employ traces for debugging and verifying hardware/software issues.

3.2.1 Formal Analysis Methods

Formal analysis is a mathematical method that is applied for verifying and specifying systems. Two approaches are generally used for hardware/software analysis: *model checking* and *theorem proving*.

Model Checking is a formal method that permits analysing all reachable states of the system. The objective is to analyse the correctness of each state based on predefined rules. To analyse the whole system, all reachable states have to be analysed, which can demand a tremendous computational effort and can also lead to the problem of state space explosion. Despite these drawbacks, some works use this technique to address hardware/software problems, such as cache coherence verification [43–46].

Theorem proving is a formal method that determines the validity of a formula. Propositional and temporal logic are useful tools for specification and verification of such formulas, and they are especially applied in the hardware domain. Therefore, applying this technique to decompose the system in small parts, each one having its own formula, helps verify large, complex systems by reducing the verification of a problem that can be solved automatically by *model checking*. This technique uses some proving assistant tools, which are softwares that automate steps in the theorem proving process. The list of such tools is extensive, we just give a few examples of them: Coq [47], PVS [48] and LEGO [49] and Z/EVES [50].

Some tools and methods support both methods, *model checking* and *theorem proving*, like Isabelle [51, 52]. In [53] the authors propose a unified framework of these methods.

Despite the efforts to simplify these methods, they are still resource greedy, specially when considering computational cost, being complex to model hardware and software interactions. In this case, alternative methods are indeed necessary to capture this hardware/software interaction.

3.2.2 Trace-Based Analysis Methods

We can classify the trace-based methods as semi-formal methods. Semi-formal methods are a complement of formal methods that are used to decrease the cost of analysis. Generally, the semi-formal methods address initially high abstraction level, *i.e.* system level. After that, when a problem is identified in a specific block, formal or more classical debugging methods can be applied to solve it.

A comparative study of KPTrace and VisualOProfile, which is a visual profiling tool, is available in [54]. This comparative aims at exploring the visual information provided by these tools. They compare the way the statistics are shown and how they can help to solve problems in parallel applications. There are works that use these traces targeting

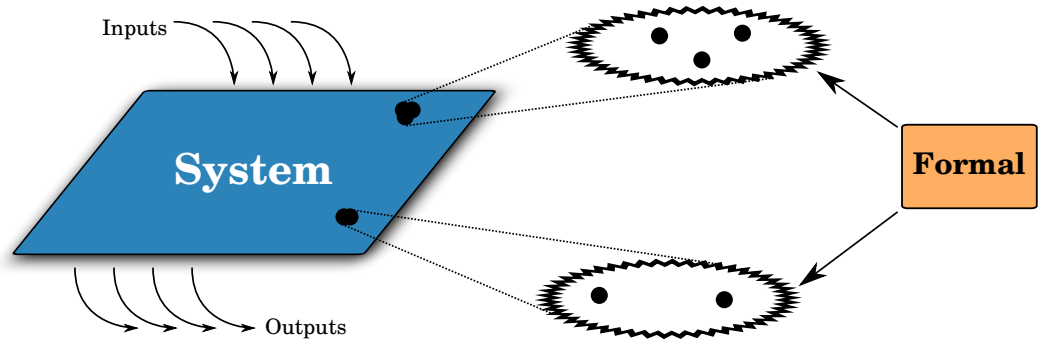


Figure 3.3: Semi formal verification methodology proposed by [56].

debugging parallel programs. One of these is [55], where they explore the load balancing in Symmetric Multiple Processor (SMP) platforms and time constraints in video decoding application.

In [56], the authors propose a semi-formal method that alleviates the verification step. This method uses formal tools just to analyse the "difficult" blocks. At system level, the simulation is still used as the primary verification engine. Their method is based on two related verification efforts: at block level and at system level. At block level, the identification of the buggy blocks is based on previous experience and feedback from the RTL engineers. Although, accurate specifications for these blocks are not available, RTL designers can identify many properties for each block to satisfy, which can be verified with formal tools. More specifically, using a model checker. At system level, the simulation aims at being checked against system specification. In this case the architecture specification is very abstract and does not specify many design details, such as the number of cycles per instruction, or the number of cycles for memory access. The overview of this method is depicted in Fig. 3.3.

The general method of semi-formal analysis can be reflected by Fig. 3.4.(a). This figure represents a closed loop approach where the illegal behavior is identified by a user or an automated method, named monitors. When a monitor is triggered and an illegal behavior is identified, then an analysis is made on formal models, whether it is a

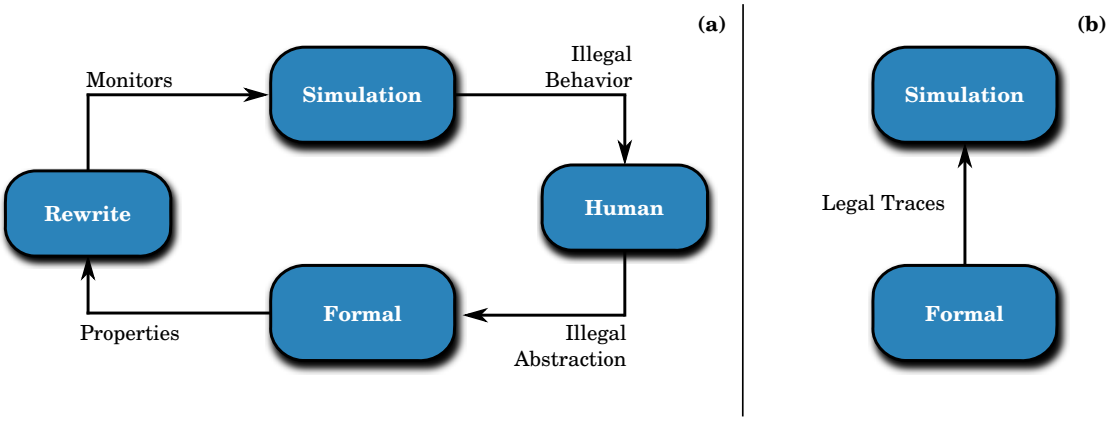


Figure 3.4: (a) Closed-loop Verification Methodology and (b) Trace for simulation engine [56].

real design error or not. If it is not a design error, then it can be introduced by illegal abstraction or wrong environment modeling. In this case, modifications on the simulator or the models are made and the process is restarted. This method uses as additional information the traces generated by the formal model to feed the simulator, which is depicted in Fig. 3.4.(b).

The efficiency of an analysis depends on the type of information provided by the traces. The order is one of these parameters. We found in order theory two candidates that are useful in traces, the *total* and the *partial* orders. Strictly speaking, total order is a binary relation on some set which is transitive, anti-symmetric and total. This relation allows comparing all events in a set. By having the total order of an execution, it is possible to analyse what happened before and after a given event. However, methods that use only the total order to represent the system generally dot no offer the interleaving of flows of execution. Partial order is a binary relation that indicates for a certain pair of events that the one precedes the other. In this relation not every pair of events needs to be related.

Tools that face state space explosion usually employ partial order reduction. Partial order reduction is applied to reduce the number of reachable system states that must be searched to verify some properties. This strategy is employed by SPIN *model checker* [57]. SPIN uses *model checking* that consists of traversing the possible states of the system, then building them, and verifying whether expressions hold or not. At the end, a reduced graph containing all possible state space is built. However, this graph is reduced but not limited to a number of states, thus even using this technique the state space explosion can occur.

Some trace-based methods do not suffer from state explosion because, intuitively, they only add edges and vertices to a given input trace, hence, there is no state space traversal. POTA (Partial Order Simulation Traces) uses traces applying formal verification based on Computation Tree Logic (CTL) expressions [19]. It applies a method called Computation Slicing, which is an abstraction technique for analyzing traces of distributed programs. Intuitively, a slice of a trace with respect to a specification p is a sub-trace that contains all the states of the trace that satisfy p. Note that the set of states that satisfy p may be large, so one could not simply enumerate all the states efficiently either in space or time. A slice contains all states that satisfy p such that it is computed efficiently (without traversing the state space) and represented concisely (without explicit representation of individual states).

The Computation Slicing consists of a dynamic graph construction which is based on a predicate definition, which are statements that describes a property. It achieves an efficient temporal predicate detection. This is obtained by checking whether the initial state of the trace is a state of the traced-slice respecting the given predicate or not, since predicate detection (similarly to *model checking*) checks whether the initial state of a system satisfies the predicate or not. Upon using the combination of slicing and predicate restriction, they alleviate an exponential problem into a polynomial one [19].

They use partial order of traces for creating slices. In Fig. 3.5.(a), a total order trace generated by a program with two processes is depicted. The events on process `P1` are generated in the order e_0, e_1, e_2, and e_3 and the events on process `P2` are generated in the order f_0 , f_1 , f_2 , and f_3. Each event is also labeled by variables `CS1` or `CS2` if the corresponding process is in the critical section; otherwise, it is not in the critical section. For example, `P2` enters the critical section at event f_1. Let f_1 be the send of a

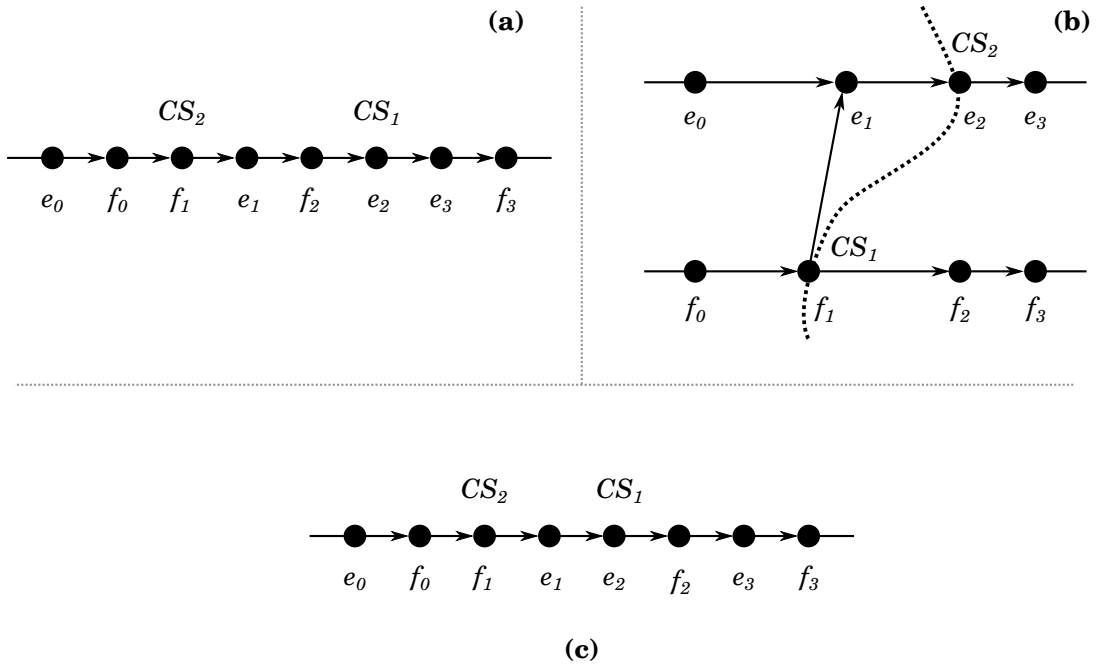


Figure 3.5: (a) A total order trace, (b) the corresponding partial order trace, and (c) a total order trace generated from the partial order trace [19].

message from P_2 and e_1 be the receive of that message by `P1`. The partial order trace corresponding to the total order trace in Fig. 3.5.(a) is depicted in Fig. 3.5.(b). Observe that the only dependencies in the partial order trace are the order of events on each process and the message send/receive dependencies. Suppose the specification is the mutual exclusion property which requires that there is no state where two processes are in the critical section at the same time. The state of a system is given by a set of events that have been executed. Observe that the specification is not violated in Fig. 3.5.(a). This is because `P1` is in its critical section at event e_2 and `P2` is in event f_2 (outside the critical section in event f_1). From the partial order trace, let's observe that events e_1 and f_2 can be executed in either order since there is no dependency between these concurrent events. If the order of concurrent events was such that e_2 had been executed before f_2, then a bug would be identified. The total order trace that represents the swap of concurrent events is shown in Fig. 3.5.(c). We observe that adding relations between processes constructing a partial order allows further means to concurrence analysis instead of using simple total order relation [19].

Other works address the usage of traces and dynamic graphs creation limiting the state space explosion and speeding up the analysis. In [20, 58], authors propose a model of low level traces that captures events generated by hardware components. This trace model allows linking events produced by different components. It aims at capturing all memory accesses and creating a causality chain using requests and acknowledgment events. Requests represent a data request usually performed by processors and caches, whereas acknowledgements represent the memory operation answering the requests. Instructions events are also captured by the trace system.

Fig. 3.6 shows some typical examples of event dependencies. We divide the figure

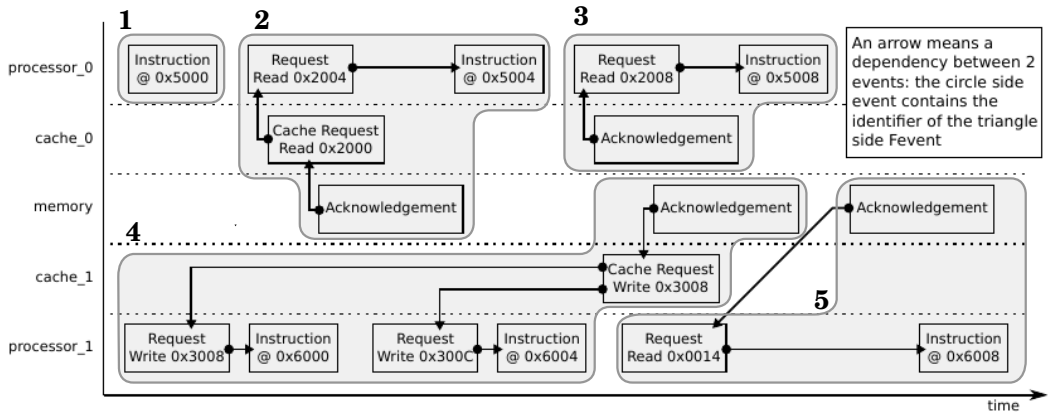


Figure 3.6: Event dependencies examples for a 2-processor platform with write-through policy caches and write buffers [20].

in five subsets of events. Subset 1 represents a simple instruction execution done by `processor_0`. Subsets 2 and 3 show memory accesses through the caches, in the former, the cache requests the data from the memory, in the latter, the access is handled by the cache. Other subsets are performed by `processor_1` instructions. In subset 4, `processor_1` first does two writes which are gathered by the cache write buffer and then, in subset 5, does an uncached read which is handled by the memory, bypassing the cache.

These traces allow capturing the relations between components in the system. The analysis, by using these relations, permits identifying interaction issues between hardware and software since it identifies, through events, the behaviour of each component in the system. In [58], the authors propose a method to verify the consistency model of different architectures. They address this using partial order and dynamic graphs creation, which consists of creating new edges and vertices when new events occur while removing useless edges and vertices. The addition of a new vertex is based on the activity done at each address. The removal of a vertex is done when it cannot, by any mean, imply the creation of a circle in the graph, which is the condition they use to identify a consistency violation. An example of dynamic graph is shown in Fig. 3.7. In this example, they represent two flows of execution, $\mathtt{SI}_1$ and $\mathtt{SI}_2$, and the accesses made at two different variables (addresses in memory). The nodes represent the events, and the edges the events order. The gray cloud bounds the edges and vertices that are used during the analysis, hence limiting the memory used during analysis.

As observed, the presented methods are supported by the idea that the analysis of all reachable states of the system can lead to state space explosion and huge analysis time, which impacts considerably multi and many processor system on chip analysis. Methods limiting memory usage have been proposed. They use traces to represent the behavior of the system, bounding the analysis. However, the interaction of components may not be completely captured by traces or exhaustively tested, which can open a space for opportunistic bugs. At the end, a performance/coverage trade-off is imposed to system verification.

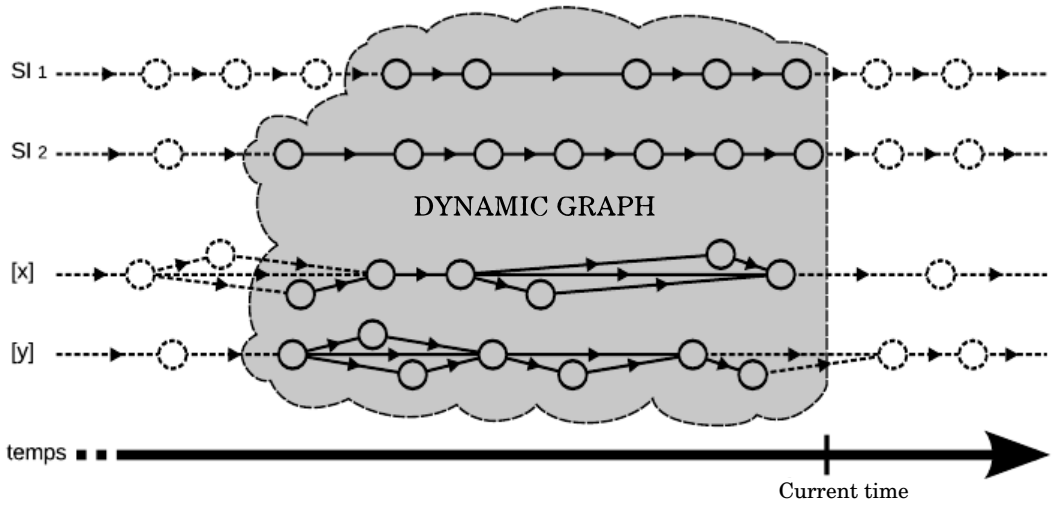


Figure 3.7: Dynamic graph of a given execution [58].

3.2.3 Summarizing

The trace based analysis methods present advantages compared to those exclusively based on formal methods, notably handling efficiently the state explosion problem. As traces can represent detailed view of the behavior of the system, such behavior can be explored to ease the verification and debugging processes. However, trace based methods can suffer from disk size scalability issues due to the huge number of events generated during a single execution. The adoption of these methods depends on the disk size available to store trace events.

3.3 Deterministic Reverse Debugging

In this section, we present techniques that provide reverse debugging capabilities. We show their features and drawbacks. There are some methods that are neither deterministic nor trace based, however, they present other relevant contributions.

The debugging process in multiprocessor systems is a hard task due to the large number of possible flows of execution. Another complicated factor is the non-determinism of such systems.

Deterministic debugging is a process that aims at providing the same order of operations of the system every time an execution takes place with the same inputs. It is a very powerful feature because it permits the reproducibility of intermittent problems, which is often a very difficult problem to solve. Deterministic execution was proved to be an efficient way to debug parallel systems by [59–61]. These works showed how to capture traces and execute them in real platforms. However, waiting for the final hardware to start the debugging process may compromise the time-to-market of certain designs.

Even supported by deterministic debugging solutions, the conventional interactive debugging process may need a lot of system restarts until correct bug identification. Then providing additional features that avoid an elevated number of system restarts, such as reverse debugging capabilities, help facilitate the debugging process. Reverse debugging

is a process that provides developers means to execute a program in a reverse way during the debugging session. Debugging reversely is not a new idea [62, 63] and comes from the 90's. Thenceforth, many strategies have been employed to provide it, such as structured backtracking, check-pointing, code instrumentation and reverse code generation. All those strategies deal with issues due to the nature of the reverse debugging, but some fit better than others the MPSoC hypothesis. In [64], a global vision about reversible methods is exposed. Next sections discuss each strategy.

3.3.1 Execution Log and Structured Backtrack

The Execution Log is a file that contains the *change-set* of the system. The *change-set* consists of all variables whose values are modified by the statements of the program. When purely sequential programs are executed, the ones without control structures, the number of elements in the *change-set* is admissible, resulting in approximately one element by line of the program. However, programs with assignments nested inside `for`s and `while`s can produce a huge number of elements because the number of times a loop iterates may depend on run-time input. Thus, the length of the execution log may not be bounded at compile time.

The structured backtracking approach is a method that incorporates more information about control structures inside execution logs. It observes the variables that could be modified inside these control structures (`for`s and `while`s). From this information, the values of these variables are stored before entering the loop, therefore, considering that the next assignment to be treated is located after a loop. Let's suppose a debugging session, where an engineer places a breakpoint before a loop, executes the program until after the end of loop, then wants to backtrack the execution until the breakpoint. In this case, the computations inside the loop are skipped and the system jumps over it, recovering the previous values inside the structured execution log. This improves the reverse execution of the code by jumping directly to the instructions before a loop, thereby avoiding its entire reverse execution. Spyder [63] uses this technique for C programs.

The execution log can be optimized in many ways, like for example, storing just relevant operations instead of very detailed instruction granularity [65].

3.3.2 Checkpoints

Check pointing techniques aim at creating periodically a file containing the system state. The checkpoints can be created either when a condition is triggered, for example, a call to a function is performed, or simply following a given time interval. Thus, when a reverse execution is started, the debugger recovers the system state from the nearest precedent state-saved point and then re-execute the code in forward way until reaching the desirable point. Effectively, the performance is impacted when the interval of checkpoints creation is too short because a lot of them will be created. On the other hand, enlarging the interval increases the number of instructions that have to be re-executed, thus it can deteriorate the performance. A good performance/periodicity trade-off is therefore necessary.

Checkpointing also permits a dead program to recover from the latest checkpoint. As the checkpoints maintain the entire snapshot of the system, this re-spawn operation is not complicated to perform. This feature is very useful when the debugging session crashes, thus a new session can be reconstructed from the last created checkpoint. The trigger

to snap a checkpoint can vary. In [66], the authors propose the usage of breakpoints and watchpoints to create restore points in addition to temporal trigger. This method is applied during simulations [66].

IGOR [62] is a tool that performs reverse debugging of C programs. Its main feature resides in recovering a program from death and still allowing reverse debugging. Modifications on some software components are necessary, like compiler, libraries and linker, and the application must run over a modified kernel. The authors advocate that these modifications do not heavily impact the system performance. The debugger for standard ML [67] uses a similar checkpoint and replays the system in a more efficient and interactive manner.

Other languages, like Python, need less effort to modify the toolchain as they are interpreted instead of being compiled. In [68], the authors propose a reverse debugging system for Python programs.

However, reverse debugging based on checkpoints needs improvements to allow deterministic execution. As checkpointing consists of recovering the system state based on the system saved state, then there is usually a gap between this saved state and the breakpoint resulting in a set of instructions that have to be completely re-executed. This gap of instruction has to be also deterministically executed, then operations with non-deterministic results have to be stored, such as I/O operation results.

To cope partially with deterministic execution, [69] proposes a tool called *Flashback*, which runs over Linux Operating System and is used to debug user level applications. This tool supports manual check-pointing control (*checkpoint*, *discard* and *replay*) instead of periodical creation. The objective is to avoid huge number of checkpoints, then when the engineer identifies a potential problem, he/she can snapshot the checkpoint (*checkpoint*). If a condition is satisfied, for example a file was opened successfully, then a restoring point is deleted (*discard*). If the faulty point is reached, such as an error opening a file, otherwise, a *replay* control takes place. In this case, the last checkpoint is recovered and a normal debugging session starts. The partial deterministic execution is guaranteed by *shadow process*, which is a snapshot of the process, in conjunction with execution logs. This *shadow process* is forked by *checkpoint* operation, suspended immediately after creation, and stored within the process structure. A *shadow process* can be removed or restored, using *discard* or *replay* respectively. The execution log stores the intercepted results of the *OS system calls* during the first execution. After that, during a replay section, each *system call* is also intercepted and the results and the side-effects are given to the current execution. Of course, it is very difficult to reproduce all side-effects and this method does not guarantee the determinism of some memory operations, and peripheral behaviors are almost impossible to reproduce deterministically.

3.3.3 Code instrumentation

Code instrumentation is another method to provide the reverse debugging feature. It consists of adding non-functional statements to functional code. In the reversible debugger case, these non-functional statements aim at producing an execution log, thus they are used to capture the modified values caused by each executed statement or instruction, *i.e.* which registers were changed or which new values were written in memory.

This technique is used in [70]. They propose a method that is free of modification at the programming language and compiler level. It is also configurable, allowing the

engineer to turn it on/off during debugging sessions. The instrumentation process is applied just in user-defined routines, which are instrumented and recompiled. Using user-defined tags permits targeting the most problematic functions, routines or methods. All routines can be also defined, allowing the reversible execution of the entire code. When the reversible operation is disabled, the intrusiveness of this method is zero. When it is enabled, in the worst case, when all routines are instrumented, the penalty during a recording session can be increased by a factor of five.

However, methods based on code instrumentation tend to be very intrusive, even adopting partial instrumentation, such as [70]. Non-deterministic execution can be produced due to the intrusiveness of the instrumentation, which can change the order of parallel operations. Another limitation is associated to availability of source-code. Only functions present in the source code can be backtracked, thus proprietary binary libraries are not supported.

This method succeeds to reproduce a fault captured in the execution log. However, the instrumentation does not exempt the method itself of inserting other faults. We do not consider the size of the execution log a problem, due to the falling cost per byte in current storage systems.

3.3.4 Reverse Code Generation

Most of the previous methods are based on execution logs, checkpoints and instrumentation. However, reverse code generation relies on a different paradigm. It is a method that consists of generating a reverse executable that undoes the operations done by a given statement. It uses the notion of *self-defined* statement. It is an expression where the operands and the final results reside in the same variable. For example, for the *self-defined* expression $x := x + 1$, the reverse operation is $x := x - 1$. This technique exploits the reverse of a statement, but generally this is a quite complex task. In addition, state-save techniques are used when reverse operation can not be performed.

In [71], the authors propose static reverse code generation. They demonstrate that this technique avoids memory explosion for reverse debugging. As a static method, it aims at exploring all reversible paths. However, it cannot cope with multithread programs, the non-determinism inherent to this paradigm imposes too many execution paths due to the interleaving of threads.

The dynamic reverse code generation consists of producing the reverse code at runtime during a debugging session. This method mixes the mathematical reverse computation techniques with state-save techniques. It allows deterministic reverse debugging of parallel programs. In [72], a dynamic approach is proposed and simple reverse operations are demonstrated. A use case demonstrating this technique is being applied to multi-thread programs in [73].

The authors advocate that these methods are less greedy in memory consumption than other methods. However, they consume more processor time than their pairs, due to the computations necessary to obtain the reverse statements. The determinism of these methods is also not clear. As code instrumentation, additional computations are executed to provide the reverse functionality, such as saving-states or calculating the reverse statements. Thus, the order of different threads can vary. Then, it is very intrusive at this point, which can impact determinism of execution. The modification in source code is zero, guarantying no intrusiveness at this point.

3.3.5 GDB Reverse

The well known open source debugger `gdb`, since version 7.0 released in September 2009, supports reverse execution commands [74]. For each platform a *target* has to be available to support reverse execution. A *target* is the execution environment occupied by a program. It can be used to implement specific functionalities, increasing the number of commands supported by a given platform, working also as a plug-in. This is the case of reverse debugging support.

On some platforms that run Linux OS, `gdb` provides a feature, named `target record`, that can record a log of a process in execution, and replay it later with both forward and reverse execution commands [75]. Furthermore, it supports native execution of `x86` architectures (`i386-linux` and `amd64-linux`).

The `target record` relies on the execution log technique. It is started, through a command, during a conventional debug session. The engineer specifies when to start recording events, and from then on can go forward and reverse up to the log starting point. While logging, `gdb` runs in single step mode, and builds up an execution log in which, for each executed instruction, all changes in memory and register state are logged. For reverse execution, it replays back the log.

There are other target implementations available for different platforms, based on `gdb target remote` protocol specification, which is mostly applied to cross-debugging. The moxie-elf simulator is a tool that simulates the Moxie general purpose bi-endian load-store processor and supports reverse execution. The SID is a simulator for Xstormy16 architectures that implements the reverse execution. Chronical-gdbserver is an implementation of `target record` protocol which talks to a database written by chronicle-recorder. What this means is that you can run the program you want to debug under chronicle, which will write a complete database of everything it does. Then you can use chronicle-gdbserver to link `gdb` with that database, and use all gdb's standard facilities to examine what the program did during its instrumented run, including `gdb` reverse commands. The UndoDB project led to creation of a company that explores commercially these technologies [76]. Furthermore, all these extensions may implement the reverse execution using different techniques, not holding on execution log, using `gdb` just as a sort of interface with engineers.

Using a well-known debugger as baseline to demonstrate the feasibility of a method help its adoption in academy and industry. Furthermore, the researches focus on the method, since the user infrastructure and communication protocols are already broadly adopted.

3.3.6 Summarizing Drawbacks

Unfortunately, almost all of these methods are applicable only to native host execution over a specific operating system, dedicated either just to x86 applications or designed for a target language [62, 63, 65, 67, 69, 72].

In [63], a granularity/speed trade-off is observed. It improves the reverse execution avoiding entire loop execution, but lacks in obtaining values inside the loop.

Oppositely, [70] shows how to reverse every single instruction, unfortunately with considerable memory allocation and execution time overhead. It imposes high overhead, as 12 instructions are added for a single instrumented instruction during forward execution. Moreover, it can change the program behavior, thereby changing the flow of execution.

Consequently, it may cause undesirable results or different order of operations from that obtained when instrumentation is disabled. Non-deterministic executions can be experienced in checkpoint techniques, such as [62, 67, 69].

In [62, 68], compiler, library and loader modifications are necessary, putting some additional information in binary code about data allocation, which can be difficult to modify or even not possible when the source code is not available.

The reverse code generation approach presented in [72] is specific to a functional language and does not support complex operations, supporting only simple arithmetic functions. Improvements in this method are presented in [73], which mixes the reverse code generation approach with execution log approach when the reversion of a statement is not successfully processed.

Furthermore, some techniques do not support threads, which is deterrent for multi/-many processor application developments [62, 63, 70].

Concerning `gdb`, `target record` is very intrusive. Thus, even better applied to sequential application debugging, it is not easily applicable for debugging full parallel software layers. Fortunately, the `gdb` architecture is extensible and new methods can use its infrastructure to deploy innovative approaches.

3.4 Conclusion

As the number of processing units increases into a single chip, methods to develop these new systems have to be rethink. Indeed, a multi domain approach is necessary to produce new adequate methods. We observe that traces play an important role in this context, as detailed intra-component relations can be captured. However, to produce them in real platforms induces huge area consumption, consequently power consumption and intrusiveness, either in hardware execution or even in software execution. On the other hand, the virtual platforms can be used to generate the traces without being intrusive. Their flexibility is important because they allow capturing the system behavior without perturbing the system. Some virtual platform technologies are very slow, such as RTL simulators, and including trace capabilities could slowdown even more their performance. Fortunately, new methods have arisen and the simulators are increasingly efficient, however, low-level traces are not present in most of these new tools.

In addition, methods to analyse systems are employed efficiently to identify system, hardware and software bugs. The use of a debugger in its essence puts a hard load on engineers because they have to manually find the potential and effective problems. The analysis tools, like those described in this chapter, aim at speeding up the development and also at providing means for engineers to correct them. We advocate that traces can bring relevant information about the system behavior. Effectively, these traces can vary in form and type, depending on the problem addressed.

The challenges faced by engineers imposed by novel multiprocessors architectures oblige rethinking about program debugging process. The forward way to debug is not enough to offer the productivity expected by system integrator and software developers. Some works have addressed this point, however, in general, they are very intrusive modifying the behavior of the system, either at software- or hardware-levels. We advocate that the detailed traces captured using virtual machines can be used to improve and increase the acceptability of new debugging methods. Efficiency methods to produce useful

information are important because the debugging process concentrates a considerable effort, thus methods that take days to provide results must be avoided.

All available methods for capturing traces and performing analysis and debugging contribute to improve the development of multiprocessor systems. However, they also present some drawbacks most of them either incur high intrusiveness or rely on real hardware. To delivery even most efficient methods, these drawbacks have to be properly treated.

Chapter 4: Trace Definition and Trace Capturing Method

Contents

This chapter details the first contribution of this thesis, presenting the trace formalism that describes important relations among traces, such as total order, partial order, and causality.

We expose the objectives of our first contribution in Section 4.1. Section 4.2 presents the formalism used to define traces and their relations. After that, Section 4.3 presents TLM and DBT, introducing the integration of both methods to provide a fast and accurate virtual platform simulator. In Section 4.4, we explain how to produce the formalised traces in this DBT-TLM simulator detailing which information is captured and how events are produced. Section 4.5 exposes the experiments we did to verify the trace system. Finally, we draw some conclusion in section 4.6.

4.1 Objectives

We work with the hypothesis that the traces have to represent the behavior of a multi-processor executions, which consist of a lot of actions happening in parallel. Thus, to

obtain a consistent trace, we divide our objectives in four categories: *abstraction level*, *parallelism*, *intrusiveness* and *speed*.

- *Abstraction level*: The first objective aims at capturing traces that allow system level representation, being applicable to the debugging process. System level models focus on functional behavior, simplifying implementation details, which help produce functional results using simulation instead of using actual hardware. At system level, the functional hardware/software interaction can be captured, for example, software instructions that results in interactions between processors and their caches, the caches and memories, and so on. These interactions can be used to identify misbehavior either in software or hardware. Another advantage is that it allows anticipating issues before the actual hardware being available, alleviating time-to-market pressure and decreasing prototyping costs.

- *Parallelism*: The second objective aims at capturing the behavior of parallel software running on MPSoCs. Misapplying parallel programming and approaches or their absence is the source of many bugs. Capturing the parallel behavior helps bug identification and consequently their elimination. The parallelism is mainly represented by multiple processors where each processor executes its own set of sequential instructions, but also accesses shared resources. The manner the parallel events happen in these systems can be represented by events and the relation among them, which is the essence of our traces. Thus, how to represent events and relations is the key objective. A total order indicating when an event occurred individually in each component is not enough to capture the parallelism because it does not take into account the events happening in other components. Thus, a method to preserve this total order and link these events to the consequences that each event causes into the system help reproduce the parallelism of events.

- *Intrusiveness*: Our third objective is to propose a non-intrusive method to capture traces. The intrusiveness can modify the order in which a program is executed. In parallel execution environments, this modification can be even worse, since it can modify the order a shared resource is accessed masking or leading to parallel problems like interconnection congestion. A low level of intrusiveness is desired which helps maintain almost the same execution order whether the capturing of traces is enabled or not. Otherwise, high intrusiveness can lead to undesirable executions, then difficult bug hunting. A non-intrusive trace system is such that there is no undesirable behavior when disabling trace generation, thus guaranteeing that the expected behavior remains even when trace generation is disabled.

- *Speed*: The fourth objective is to obtain a fast method to capture traces. The simulation of complex systems usually takes a long time to be finished due to increasing number of components, their interactions and technology limitations. If a simple simulation that produces traces takes a day long, then it is useless reaching the previous objectives to just deliver one execution trace per day. As capturing traces is just the start point to identify and correct bugs, then other phases, such as analysis and correction, have to have their slot time during a working day. Fast virtual platform simulators are thus a good choice to capture traces.

We describe how these objectives are covered by formally defining the traces and detailing how we capture them.

4.2 Trace and Event Definitions

The order of events can be expressed in many ways, such as using either time stamps, strict total order, partial order or mixing these methods. Each of these methods presents advantages and drawbacks. Time stamping produces a way to compare each event individually, generating the strict total order from the time information. However, capturing time information is a complex task in multiprocessor environments, since obtaining accurate time requires a very accurate simulator. These simulators generally do not scale-well for large systems, as discussed in Chapter 2.

Inside current MPSoCs reside a lot of components that perform operations, such as processors, interconnects, memories, interfaces, so on and so forth. A flexible trace specification helps cover a wide-spectrum of architectures. This flexibility implies hiding some architectural details, which can be achieved by rendering them implicit.

Let us take the example of Fig. 4.1. The two MPSoC architectures (Fig. 4.1.a) embed four processors, a DMA, a timer, a DRAM and interconnect. This latter element is the only one that differs between them : the top architecture uses a bus, the bottom one a Network-on-Chip. Fig. 4.1.b displays the end-to-end events on a graph whose *x-axis* represents the system order and whose *y-axis* lists the components.

In this example, two processors (CPU_1 and CPU_4) access the DRAM while the other components continue their execution. In the bus-based system, the arbiter grants the bus to CPU_1, which then performs an access that eventually reaches the memory. Then, CPU_4 is granted the bus, and in its turn, accesses the memory. In this case, the memory events order represents the fact that the bus is shared and granted following a given

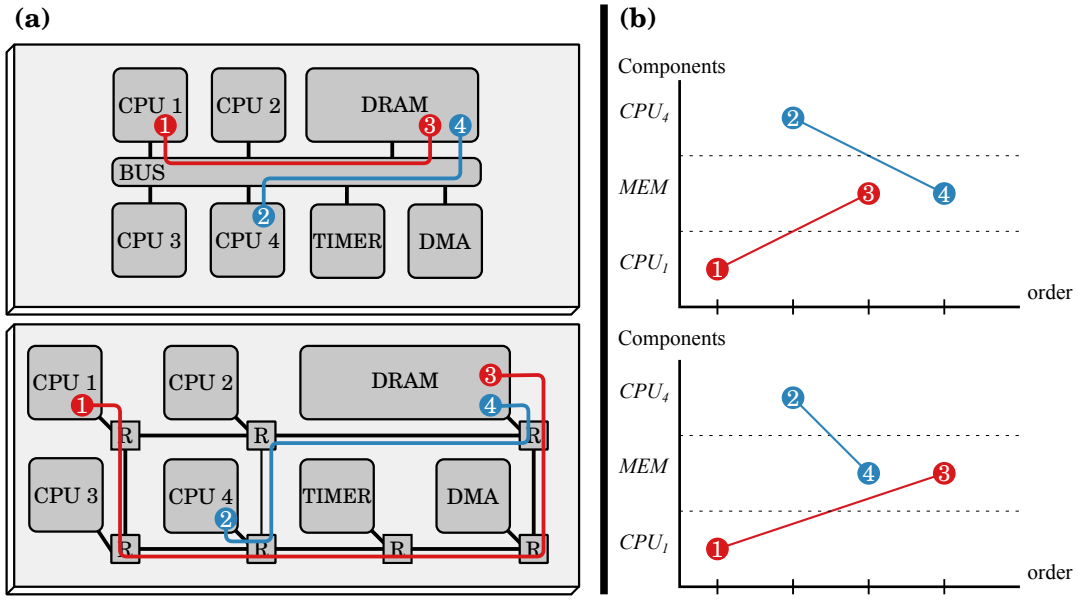


Figure 4.1: (a) MPSoC Architectures and CPU requests.
(b) Order of memory accesses.

algorithm. In the NoC-based system, the path followed by the data depends on the routing algorithm. In both cases, the interconnection-related events are useless, as the order of memory events gives the relevant information at system level while limiting the trace size. In our example, CPU_4 packet uses a shorter path than the one of CPU_1, thus reaching memory first. Hence, the order of the memory events is different in these two cases, which is observable on Fig 4.1.b.

Each system execution leads to a different behavior of the internal components. As each component produces events of its own, a special care has to be taken regarding the ordering of events in the trace. Indeed, a desirable property, especially for many on-line analysis, is that all events generated by a given hardware component appears in their order of occurrence in the trace, and that events that are the source of other events appears before the events they produce in the trace. We define a trace having this property as being "well formed". To express this property, we define it formally through two relations that are also defined after. Before that, we define formally the trace and the events.

Given C a set of hardware components, we define an event as a tuple $e = (c, t, d)$, where c is the component, with $c \in C$, t the event type, d the supplementary data when useful. Furthermore, the attribute $t \in \{I, S, M, B, L, A\}$, with I meaning a processor's instruction, S a memory or cache store, L a cache load, M a cache modify, B a cache write in the memory without cache modification (meaning a write-back) and A a memory or cache acknowledgement. The supplementary data d contains instruction-specific information. For loads and stores d contains the address of the access, for example. On the other hand, the `add` instruction have d containing register value. This attribute is defined as much general as possible because different events bring different data.

Therefore, noting E the set of events that system produces during an execution and E_c the set of events of a component c, then $E = \bigcup_{c \in C} E_c$ the disjoint union of E_c. The order in E is also the disjoint union of all original orders, which does not form a total order in E.

Interactions among components are also collected during simulation. Let's consider a simple architecture composed of a processor, L1 cache and memory. Store and load events lead to either an acknowledgement event or the creation of other store or load events, respectively. In the first case, the data is already in cache and requesting this data from memory is unnecessary, then the cache produces the acknowledgement. In the second case, the cache does not have the requested data forcing a request to memory, the request is created by the cache, and the acknowledgement is generated by the memory. Based on this knowledge, we define two relations among events.

- Component Strict Total Order ($<$): it is the order between two different events that happened in the same component. We define $<_c$, a strict total order on E_c, such that $\forall (e_i, e_j) \in E_c \times E_c | e_i <_c e_j$ when the occurrence of e_i precedes the occurrence of e_j. This order aims to represent the events that happen inside each component. In this case the strict order aims to identify precisely the occurrence of an event. This assumption does not allow the representation of simultaneous occurrence of events inside each component. The treatment of each event inside a component is considered atomic. It does not comprehend the parallel representation inside the component. However, the other relations permit capturing the parallelism of the system.

- Causality ($\leftharpoonup$): it is the relation between two events e_i and e_j, where e_j is a consequence of e_i, represented by $e_i \leftharpoonup e_j$. In the context of traces, e_i and e_j occur in different components. We note $(e_i, e_j) \in E_k \times E_l$ with $(k,l) \in C \times C$ and $k \neq l | e_i \leftharpoonup e_j$ the fact that event e_j "is due to" event e_i. The $\leftharpoonup$ relation aims at linking events that are generated in different components. Whereas the $<$ relation is restricted to events in each component, the $\leftharpoonup$ relation serves to produce the consequences of events. Both relations represent the parallelism and possible interleaving between processor operations and their actual processing at memory, for instance.

Both relations maintain the properties below:

- *Irreflexive*: $e_a \leftharpoonup e_b \implies e_a \neq e_b$ and $e_g < e_h \implies e_g \neq e_h$
- *Transitive*: $e_a \leftharpoonup e_b \wedge e_b \leftharpoonup e_c \implies e_a \leftharpoonup e_c$ and $e_g < e_h \wedge e_h < e_i \implies e_g < e_i$.
- *Antisymmetry*: $e_a \leftharpoonup e_b \wedge e_b \leftharpoonup e_a \implies e_a = e_b$ and $e_g < e_h \wedge e_h < e_g \implies e_g = e_i$.

The order of events in the system is important to highlight undesired behavior. These relations aim at rebuilding the execution to understand the causes of problems. However, the total order of the system is complex to rebuild, thus the partial order can be used to reconstruct partially the system's execution.

In fact, a single instruction can trigger several events, usually in different components. For example, a simple instruction that performs multiple store operations to memory will produce various store events. Furthermore, in MPSoCs, race conditions can influence the order of such events. If the memory receives more than one request from different processors, then the memory's events can interleave, thus the causality relation ($\leftharpoonup$) help identify the source of request. Therefore, the causality relation also defines a strict partial order in E.

Fig. 4.2.(a) shows an example of a trace obtained from a simulated platform containing two processors $CPU_{0,1}$, two data caches $DCaches_{0,1}$, and one memory MEM. CPU_0 instruction events (I) are represented by black nodes and CPU_1 instruction events by gray ones. CPU_0's instructions are a store followed by a load, and conversely for CPU_1, a load followed by a store. White nodes represent request events (S, L or B) and crossed nodes are either acknowledgement (A) or modify (M) events. The three memory accesses done by the processors through their caches result in a graph split into four parts. In each part the events are connected through the $\leftharpoonup$ relation represented by black edges.

We can observe the order relations for each component represented with gray arrows, *e.g.* $e_1 < e_7$ or $e_6 < e_3 < e_{11}$. Subgraph (1) shows a store instruction issued by processor CPU_0. This store instruction is depicted by event e_1 and then followed by a store e_2 to $DCache_0$ and to memory, hence the acknowledgment event e_3 generated by the memory. This results in the relation $e_1 \leftharpoonup e_2 \leftharpoonup e_3$. Subgraph (2) shows another processor request e_4 made by CPU_1, in this case a load. A cache miss occurs and events e_5 and e_6 are generated. Those subgraphs represent a concurrent memory access, leading to a shift in memory acknowledgement e_3. Subgraph (3) shows the second instruction generated by processor CPU_0 which results in a cache hit. In fact, CPU_0 produces a load instruction represented by event e_7 which is then pursued by an acknowledgment event e_8 produced by $DCache_0$. In Subgraph (4), there is another access to memory, which is a load

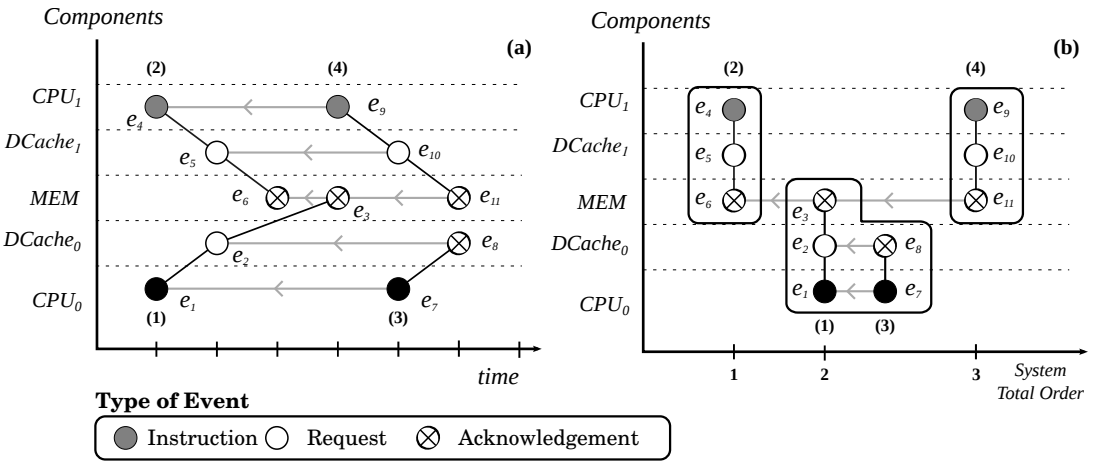


Figure 4.2: (a) Component strict total order and causality chain example. (b) System order representation.

launched by CPU_1. As can be seen, the temporal information is not used to order the events.

Two events are not comparable if they are generated in different components and do not have a causal relation. However, this comparison is crucial to cache coherence analysis, where it is important to compare events that happen in two different caches, for instance. Often events that happen in L1 caches do not trigger events in other L1 caches, except for hardware coherence control. However, in systems that do not have such hardware-supported control, such comparison is merely impossible. For example, the events e_{11} and e_8 in Fig. 4.2(a) are not comparable using causality $\hookleftarrow$ and component strict partial order $<$ relations. We use the previous definitions to introduce system total order and *shared component*.

- Shared Component: It is a component $c \in C$ that generates events $e \in E_c$ and that maintains the causality relation with at least two other different components. Noting $E_S = E_a \cup E_b \cup E_s$ the union of event sets generated by $\{a, b, s\} \in C$ such that $e_i \in E_a$, $e_j \in E_b$ and $\{e_k, e_l\} \in E_s$, then the component s is a shared component if and only if $e_i \hookleftarrow e_k$ and $e_j \hookleftarrow e_l$.

- System Total Order ($\prec$): it is a relation that allows comparing events generated in different components. It is mandatory that both components have generated events that have caused consequences in at least another common component, a *shared component*. Thus, the system order is based on the strict order of a *shared component*. We note this relation $\prec$. It has the same properties as $<$. We use the short system order term for system total order.

To illustrate this definition, Fig. 4.2(b) shows the system order of the events in Fig. 4.2(a). The events e_4, e_5 and e_6 have the same system order, which is equal to 1. The events e_1, e_2, e_3, e_7 and e_8 have a system order that is equal to 2 and for the last chain of causality, which is composed of events e_9, e_{10} and e_{11}, the value is 3. Given this relation, we can say that $e_4 \prec e_1$, even if these events were generated by two different components and do not have any causality relation.

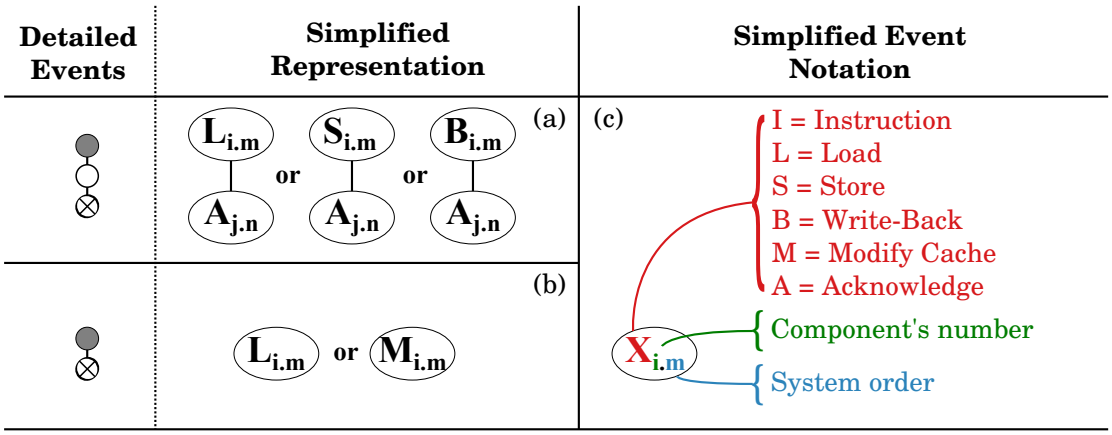


Figure 4.3: Simplified trace representation for (a) memory accesses, (b) cache accesses and (c) notation of simplified event.

In a multi-memory system it is possible that multiple system orders exist. Considering that each memory may be accessed by different components, the system order is relative to the shared component in question.

We define a *trace* as $T = \{E, <, \leftarrow, \prec\}$, with $<$ defined as $\{<_c | \forall c \in C\}$. The union of orders defined by $<$, $\leftarrow$ and $\prec$ makes a strict total order in E.

The graph presented Fig. 4.2 becomes intricate when increasing the number of components. We simplify the representation to improve the understanding of the relations between events by grouping instruction and cache events as shown Fig .4.3. They are represented with a single symbol illustrating the action, the subscript attached to the symbol designating the component which performed the action and the system order. $L_{c.o}$ is a load performed by the component c and has the system order o. If it leads to a cache miss, it links to an acknowledgement symbol $A_{d.o}$, which is an answer of component d. $S_{c.o}$ is a store.

The cache write policies (*write-through* and *write-back*) generate different trace events, thus some constraints are defined for each policy. For *write-through* caches, we assume a no-write allocate policy on miss, a store thus updates the cache if the address is present in cache, and also updates the memory, therefore a link to an acknowledgement is always

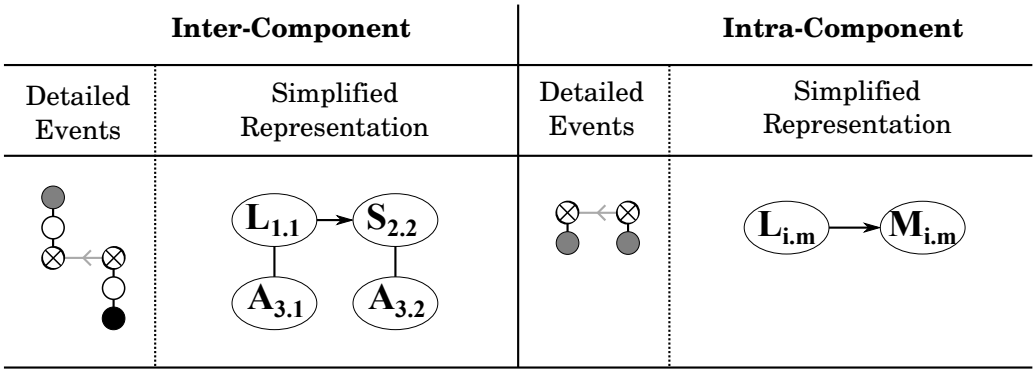


Figure 4.4: Intra and inter components event simplifications.

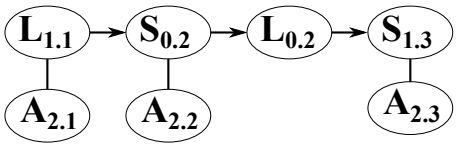


Figure 4.5: Simplified representation of Fig. 4.2.

present. For *write-back* caches, we assume a write-allocate policy, so no store (unless to uncached addresses but then cache coherency is not an issue anymore) as such occurs, and we use the symbol $M_{c.o}$ to represent a cache modification without memory modification, while $B_{c.o}$ represents the write operation to memory of a modified value. This latter action occurs only when the line containing the modification is evicted. A load leading to a cache miss, a store and a write-back are represented in Fig. 4.3(a). A cache hit and a cache modify are highlighted in Fig .4.3(b). The Fig. 4.3(c) summarizes the adopted nomenclature. Fig. 4.4 shows the representation of common inter and intra component events, which are normally differentiated by an acknowledgement.

After simplification, the graph of Fig. 4.2 becomes the one of Fig. 4.5. In this graph the identification of the different components is $Cache_0 = 0$, $Cache_1 = 1$ and $Memory = 2$. The $CPU_{0,1}$ are omitted. The causality chain (2) becomes the first in the graph due to its system order, then comes (1) and (3), due to their internal relation and their common system order. Finally, we have the causality chain(4). The simplified graph is used in Chapter 5 and the detailed representation is used in Chapter 6.

The system order of events that do not have $\leftharpoondown$ relation with an event generated by a *shared component* is determined by their type and their $<$ relation. The modify-cache event M has the same system order as the previous event that shared the access. Formally, given two components $\{a,b\} \in C$, where b is a *shared component*: the set of events generated are $E_a = \{e_1, e_2\}$ and $E_b = \{e_3\}$, where the e_2 is a cache modify. These events have the following relations: $e_1 \leftharpoondown e_3$ and $e_1 < e_2$. Thus, the system order of e_1 is the order of e_3 and we attribute to e_2 the same system order as e_1. We define this operation as *rewind* and represent it using the $\lll$ symbol. Another type of events is the cache hit event L. In this case, the system order is attributed considering the successor event that has the $\leftharpoondown$ relation with a *shared component*. For example, considering the

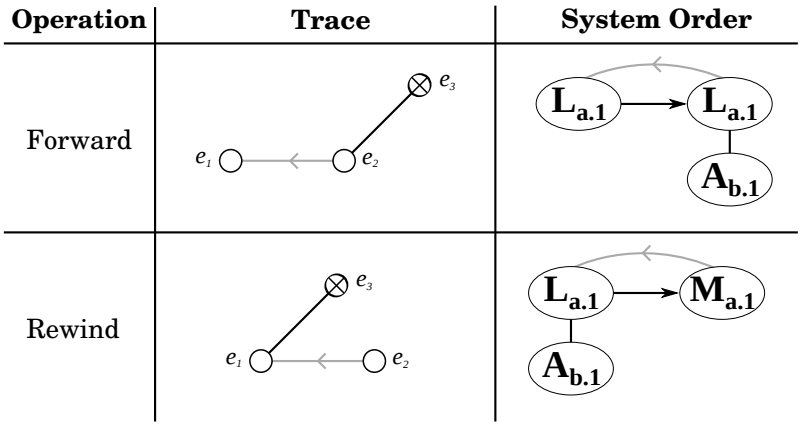


Figure 4.6: Forward and Rewind operations.

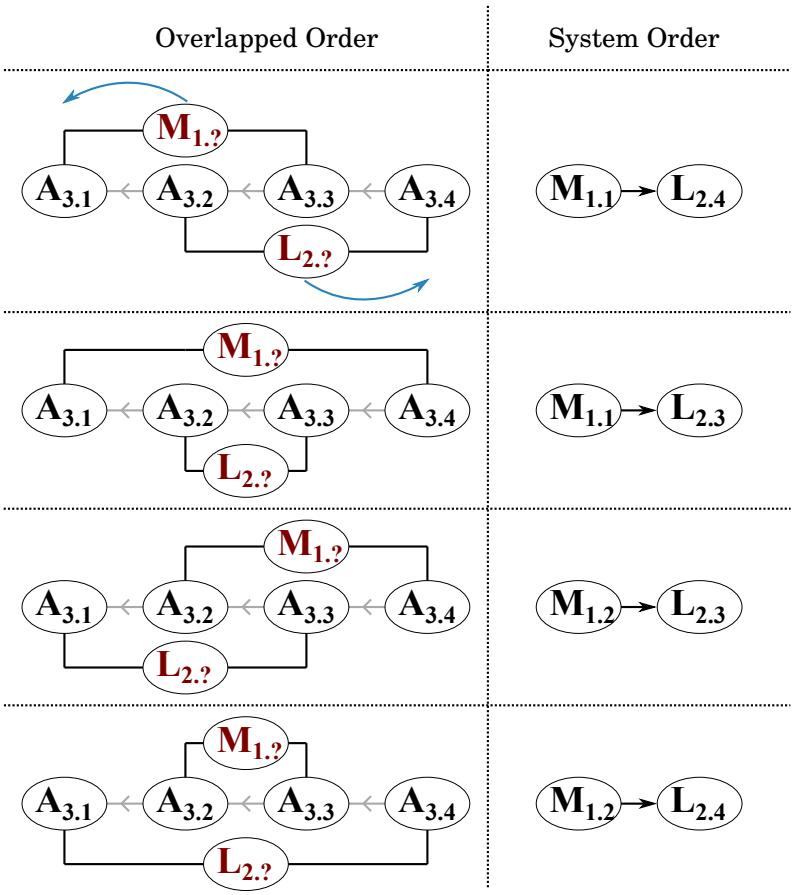


Figure 4.7: System order examples for cache events.

same components but with different sets of events and relations, $E_a = \{e_1, e_2\}$ and $E_b = \{e_3\}$, where e_1 is a cache hit. If the relations $e_1 < e_2$ and $e_2 \leftarrowtail e_3$ are constructed, then we attribute to e_1 the same system order as e_2. We represent this operation using $\ggg$ and call it *forward*. Both operations can be used on any type of events. The Fig. 4.6 shows both examples. In this example the events have the same system order which can lead to ambiguous interpretations. In this case the ambiguity is resolved through component strict order, represented by the gray arc.

The forward and rewind operations are mostly used to attribute the system order to cache events. Fig. 4.7 shows some examples of cache accesses and their ordering. The accesses that caused the acknowledgement events were omitted for better visualisation. This technique is applied to compare different cache events in order to pinpoint cache coherence violations.

4.3 DBT/TLM Virtual Platforms

This section briefly describes TLM and DBT techniques which are used to capture our traces. Then, we describe how those techniques are integrated to produce them.

4.3.1 Transaction Level Modeling

TLM is a simulation technique classified as event-driven. It models a system as a discrete sequence of events. These events are responsible for changing the state of the system. Between two consecutive events, no change of the system is assumed to occur, thus the simulation time jumps directly to the next event.

Furthermore, TLM is an approach to model digital systems where communication among the digital modules is separated from the functional implementation of the module, which permits to decouple each part and ease the specification and the verification process. Communication is modeled by channels, while transaction requests take place by calling interface functions of these channel models. Unnecessary details of communication and computation are hidden in TLM and may be added later. TLM speeds up simulation, compared to RTL simulation, and allows exploring and validating design alternatives at a higher level of abstraction [9].

Libraries are available for engineers to improve the productivity and reuse of Intellectual Property (IP). Notably, the well-known SoClib open source library is used by industrials and academics [77]. Regarding proprietary solutions, Design Ware from Synopsis provides a set of standards IP that serve as building blocks for virtual prototypes [78].

However, when applied to complex systems, such as MPSoC, the overall simulation time of pure TLM models is prohibitive, as briefly discussed Chapter 2. Because of that, hybrid approaches like the one used in this thesis, are needed. These approaches basically consist of changing the ISS to decrease the simulation time of processors.

4.3.2 Dynamic Binary Translation

DBT is a method that translates dynamically a target binary code into a host code. The name dynamic means that the target code translation and host code execution are performed concomitantly. There are other ways to perform binary translation, such as static approaches, which consist of translating the whole target code before the host code execution. However, this method does not capture runtime issues, such as self-modifying code, which is common in Java runtime and complex operating systems. DBT is used to simulate the processors in this thesis, we detail it in this section.

DBT uses the concept of basic block which is a slice of target code that will be translated. Fundamentally, the slices are basic blocks, *i.e.* a sequence of non-branch instructions terminated by a branch instruction. Each basic block contains the program counter of the first target translated instruction, which can be used as its identification number. A branch instruction and its variations, such as call and return instructions, bounds each basic block, limiting the translated block's size. There are some technicalities, such as page boundaries, instructions being the target of a goto, etc, that require changing the definition thus leading to the concept of Translation Block (TB), which represents the sequence of instructions being translated at once at runtime.

DBT has three main phases which are code translation, code generation and execution, all performed at runtime.

Before detailing all these phases, let us introduce the concept of *intermediate code*. This code is used in many DBTs. It is used to generate an intermediary representation of the target binary code, which is posteriorly translated to the host code, which is finally executed. It seems that this process adds an overhead to translation, which is obviously

true, however, it presents more advantages than drawbacks. The main advantage is the absence of direct correlation between the target and host codes, which eases the incorporation of new targets and hosts. The front-end and back-end are other terms used in DBT compilers. The front-end is a process that deals with target code while the back-end deals with host code. Therefore, using intermediate code allows sharing the front-ends and back-ends, and thus avoids having a translator for each (target, host) pair. Keeping these definitions in mind, we detail the DBT phases.

Code translation consists of fetching an instruction from the target binary image and translating it into one or several micro-operations (instructions of the intermediate representation), and doing this until a TB boundary is reached. This process represents the front-end. The subsequent phase, called *code generation*, transforms the micro-operation sequence into a sequence of host executable code. This code, the executable TB, is ended by a specific call to the DBT runtime necessary to handle the search for the next executable TB and various housekeeping activities. The executable TB is placed in a cache for future use. For example, if during the *code translation* phase an instruction has an address already present in the cache, no translation occurs (unless a write access to the TB has been performed). As all caches, the TB cache is limited in size and a replacement policy has to be defined. As of today, the most of widely used policy is to simply flush the whole cache, which is indeed quite efficient. This process represents the back-end.

The last phase is *code execution* and it consists of executing the executable TB. During this process some target processor internal structures are simulated, such as internal processor registers, TLBs for MMU and even caches. It permits the correct execution of architecture dependent instructions. For example, in the ARM architecture, there are conditional instructions that are executed under certain register conditions. Thus, when predicated instructions are executed, the DBT evaluates the internal registers to take decisions. The DBT process is depicted Fig. 3.2, Chapter 3.

4.3.3 TLM and DBT integration

In [39], the authors propose a strategy to integrate DBT and event driven simulation. They encapsulate each processor within a wrapper interacting on one side with the rest of the system at transaction level and on the other side with the DBT engine. Fig. 4.8 shows an example of such system.

To ensure proper advance in simulation time of the originally non-timed DBT ISSs, synchronization points, reached when an interaction between a processor and a TLM component is required, are employed. Synchronization is limited to external accesses, avoiding synchronization for every single instruction, including instruction fetches and loads and stores that are performed behind the hood using a so called *back door* access, *i.e.* direct access to the memory simulation process through a pointer to the appropriate instance of the memory model. However, the cost of internal computations and memory accesses must be accounted for to have correct estimates of the system simulation time.

In this case, the time between two synchronization points is estimated as cycle count, and converted to time spent in the wrapper. As we know, for some processors' instruction execution time may vary due to hardly predictable internal structures, in which case these timing are considered as approximate costs. When a synchronization occurs, the DBT simulation for the processor stops, waiting for the transaction acknowledgment.

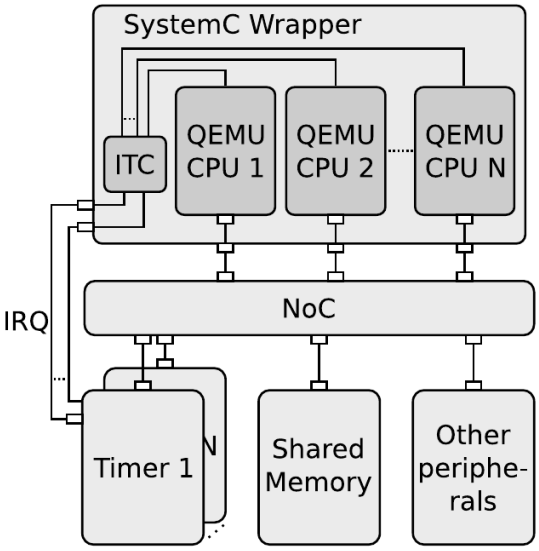


Figure 4.8: DBT/TLM integration in [39].

Meanwhile, other components can process their operations. Details about the synchronization process and how to implement it using non functional micro-operation are given in [39].

4.4 Trace Generation on DBT/TLM Virtual Prototypes

4.4.1 Issues

The generation of a "well formed" trace is challenging, because the information required is partly available within TLM simulation and partly within DBT at different points in time, and there is currently no way to share it.

The information can be either *static* or *dynamic*. *Static* information is not time related and is unmodified for every event it belongs to, such as instruction program counter, operands or instruction opcodes. This kind of data is available at compile time, thus the information can be acquired during DBT at the *code translation* phase, forwarded through *code generation* and finally used in the *code execution* phase to update the non functional data structures used to represent the event.

Some data are available only during instruction execution and are therefore *dynamic*, such as which event occurs, when it occurs, and its relations to other events. Therefore, information has to be produced during the *code execution* phase of DBT and TLM simulation.

In our event definition, the $<$, $\longleftarrow$ and $\prec$ relations are *dynamic*. The data d can also be classified as *dynamic* data. Indeed this classification of d depends of which data are necessary for a given event. Data are updated by getting the values through the response path triggered by the acknowledgement. Finally, $\longleftarrow$ is built by having each event referencing its creator (or set of creators).

4.4.2 Proposition

The production of the trace requires modifications in DBT that span *code translation*, *code generation* and *code execution* to gather and pass information to Transaction Accurate (TA) models. These TA models must be modified to incorporate the causality links creation capabilities, getting information coming from DBT level and incorporating them into the new events they create, then maintaining the relation $\hookleftarrow$ among events. This doesn't interfere with the functional behavior, and is thus non intrusive both in function and timing. This process is illustrated in Fig. 4.9 and described hereafter.

As DBT relies on an intermediate micro-operation, the idea is to add two new, non functional, instructions *event_insn* and *event_commit*. An *event_insn* allocates the event and assigns the tuple elements known at the execution time of the instruction. An *event_commit* indicates that all the elements of the event are up to date, and outputs the event.

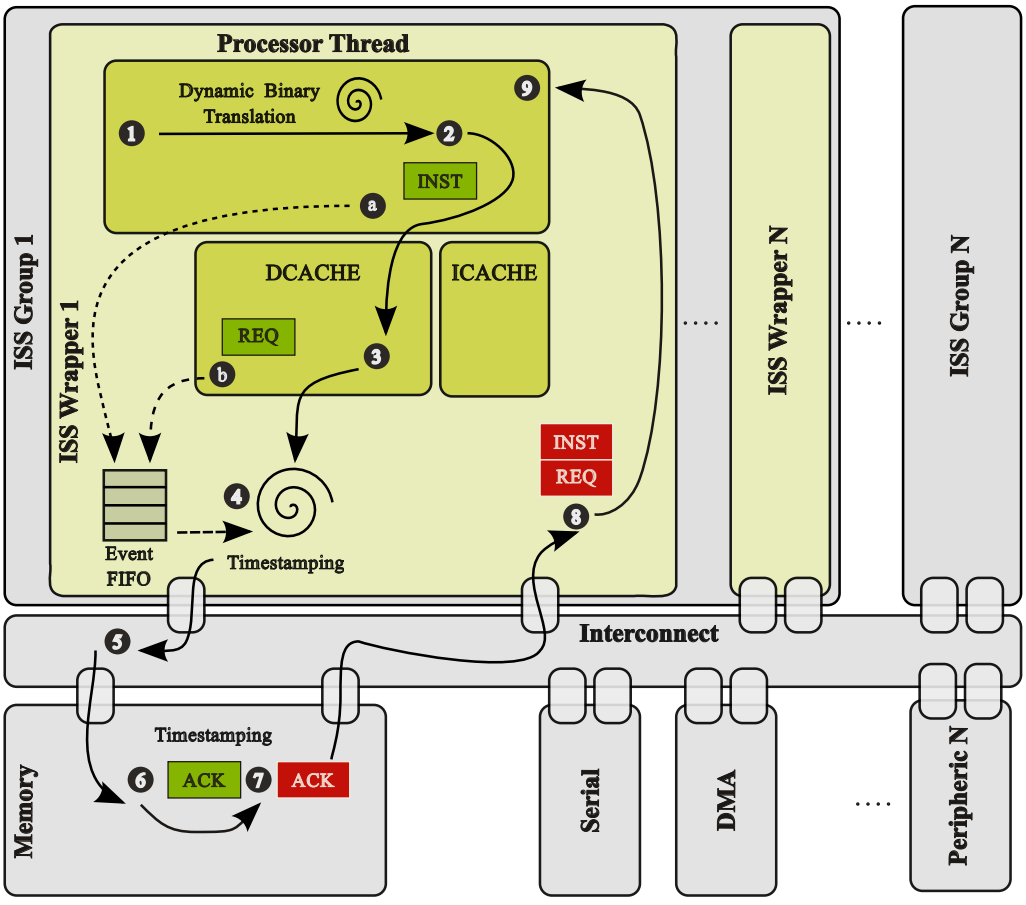
During *code translation* (1), for each target instruction, an *event_insn* is added as the first micro-operation instruction of the intermediate representation. This is valid for every instruction, even if the instruction is not actually executed. The Program Counter (PC) and the opcode are *static* data passed as parameters of *event_insn* instructions. Following the same principle, at the end of every translated instruction, a micro-operation instruction *event_commit* is inserted.

During *code execution* (2), *event_insn* creates an instruction event. As the execution produces an instruction fetch and possibly a data-cache access, cache requests are produced (3). The summary of properties filled in events is presented below:

1. **Conditional Execution:** predicated instructions are analyzed during execution and they are executed only if the conditions are met,
2. **Aligned Access:** unaligned memory accesses trigger more accesses to memory and caches than normal accesses,
3. **Multiple Access:** Some processor instructions can trigger several memory accesses,
4. **Number of Instruction Cycles**: effective number of cycles used to process the target instruction, or a *nop* if the predicate is false,
5. **Exclusive Load/Store Access:** information necessary for some later analysis,
6. **Data Cache, Instruction Cache, or Uncached Accesses:** classifies the access performed, if any.

As there are events in the TLM part that are causally dependent on the instruction event, a circular buffer is used to share them between both parts. At the end of an *event_insn* execution, the event (or its reference) is inserted in this buffer (a). There is one such buffer per processor simulation model, which is also used to store the processor data cache and instruction cache events (b). The order in which the events appear in the buffer reflects their order of occurrence. For example, when a processor accesses an address and a cache hit occurs, two events are placed in the buffer, the first one e_i for the processor instruction, and the second one e_c for the cache access, also constituting the $e_i \hookleftarrow e_c$ relation.

When a processor synchronization occurs, DBT passes control to TLM simulation (4), in which the rest of the system progresses and from which the processor's event buffers can be accessed. The events are extracted from the shared buffers and the causality chains are built. Then, the number of cycles associated to the instruction event is converted



!h

Figure 4.9: Implementation details to collect traces in virtual DBT/TLM environment.

to time units, and added to the current TLM simulation time. If a synchronization occurs due to an access to a component, then the reference of the request event e_r is passed through the interconnection (5) until it reaches the component that generates the acknowledgement e_a (6), thus building the $e_r \rightarrowtail e_a$ relation. This requires modifying the payload of the transactions so that it includes the reference. Since the acknowledgement event created, thus reaching the last point of a causality chain, all events belonging to it are dumped to disk respecting the causal order (7) (8). This way, the information present in other components, *e.g.* a completion timestamp, can be aggregated to the request events while the chain is still in memory. Since this is done (9), TLM simulation gives back control to the DBT which executes the next instruction.

4.4.3 Advancing Time

We propose two different approaches for advancing time that maintain the $<$, $\rightarrowtail$ and, consequently $\prec$, relations. The first one, called *individual*, is based on the individual clock cycles for each instruction event and its immediate conversion to time. Due to its fine granularity, it gives accurate results, but impacts performance because of the heavy processing it induces at TLM level. The second approach, called *block*, consists of adding all clock cycles for each instruction in the event buffer, converting this total number of

cycles to time, then increasing the simulation time. The TLM time is increased by the number of cycles consumed by the instruction events, and the events are still ordered, thanks to their relative place in the buffer. The *block* approach has a coarser granularity, which improves the performance at the detriment of timing accuracy.

4.5 Experimentations

We detail three experiments. The first one aims to assess the correct generation of instruction traces by checking that the call tree of a program using our traces and an intrusive method gives the same result. The second one aims at verifying the validity of the causality chains. The last one quantifies the runtime slowdown due to the trace generation system and gives some examples of trace sizes, also giving the accuracy over all tested platforms.

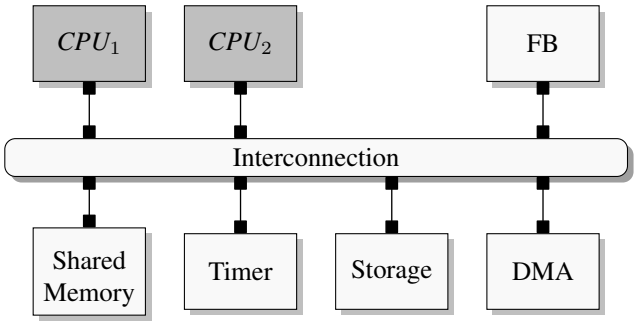


Figure 4.10: Simulation platform mixing DBT and TLM, where gray boxes are DBT based processors and white ones are TLM based models.

Our experimentations make use of the *Rabbits* framework, which itself relies on Qemu [79] for DBT and SystemC for event-driven TLM simulation. The target architecture is composed of n ARM processors (n being a user parameter), a frame buffer, a serial controller, timers, a shared memory and a storage controller. Each processor is a Cortex-A9 with Level 1 Data cache (L1 DCACHE), Level 1 Instruction Cache (L1 ICACHE) and an interrupt controller. The components are interconnected using an abstract Network on Chip (NoC). An example platform with $n = 2$ processors is shown Fig. 4.10. The gray boxes are DBT based processors, whereas the white boxes are TLM models of the other components. The processors and memory are traceable, *i.e.* they have the capability to generate trace events and produce relationships among them. Our experiments aim to produce traces over five different platforms by varying only the number $n = \{1, 2, 4, 8, 12\}$ of processors.

The softwares that run on the platforms are a multi-threaded Motion-JPEG decoder application and a subset of Splash-2 benchmark applications (Ocean, Water-NSquared and Water-Spatial), all of them running on top of a lightweight operating system with SMP capabilities. The interest of having multiple threads, which furthermore can migrate from processor to processor, is that it exercises also the cache-coherency protocol used on the system and thus produces all sorts (instruction, caches misses, invalidations, etc) of events.

4.5.1 Source code coverage

The correctness of the trace generation process needs to be verified first. To that aim, the "well formed" property is firstly checked using a tool developed on purpose which verifies that the relation $<$ considering the same component and the $\hookleftarrow$ relation among different components do follow their definitions. Then, we checked that the call graph of an execution of our benchmark is identical to the one obtained by analysis of the generated trace. Using hook functions[1] which provide a way for the compiler to insert code in functions' prologue and epilogue, we build a reference call graph. Our custom hook functions, which are intrusive since the compiler is adding code for them, simply output the address of the called function as the code is executed, and the address is later converted into the function name. This influences the execution time but does not change the call graph, which is what we are interested in for this experiment.

We can obtain the same information in a non-intrusive way using traces, as the events $\{e \in T | type = instruction\}$ represent the actual execution path of the program, and data $d\langle e\rangle$ contains the program counter value. The DWARF debug symbols of our executable provides the address of each function, so we can follow the flow of execution by tracing subroutine calls, including context switches, and obtain the call graph. To determine the called function, an analysis is made on the data d of each instruction event. When the condition $d\langle e_i\rangle \neq d\langle e_{i-1}\rangle + pc\_inc$ occurs (pc_inc being the increment required to access the next instruction), an analysis to identify if $d\langle e_i\rangle$ is a function prologue, epilogue or a jump is triggered. If it is a prologue or an epilogue, then we mark the access with respectively the *___entry* or *___exit* tag. However, `inline` functions cannot be tracked using this approach, as they have no existence at runtime but are still instrumented, so we removed the `inline` qualifier in application and OS software to compare both results using a simple `diff`.

We ran many programs, some of them with several data sets for long periods of time, and obtained the exact same functions *___entry* and *___exit* tags and the same calling order in both the instrumented and traced software. This experimentally shows that, for the elements of the tuple which are used for this verification, the generated trace is "well formed".

4.5.2 Causality links

This second experiment aims to show that the causality links are built correctly. We generate traces appropriate for a tool which aims at the discovery of scalability bottlenecks, as described in [37]. It relies on a cost function expressed through two parameters, *%_time* and *%_access*, which express, for every accessed address *@n* its contribution to the total (simulated) execution time and total number of accesses respectively.

Our formal trace definition does not rely on accurate time but all events have a time-stamp provided by the simulator indicating the approximative time of their occurrence.

The tool relies on data mining algorithms that are applied to software execution traces T in which only memory accesses events defined as $e = (c, t, d, ts, l)$ are useful. The parameters c, t, d, ts and l are respectively the component that generates the event, the type of event, the data (in this case either program counter or data accessed address), its timestamp and the latency of the access.

[1] As available in `gcc` using the `-finstrument-functions` switch.

Latency l is not directly available in our own traces. When an instruction accesses memory, it activates other components, such as the interconnection and the target memory, thus l is the elapsed time between the beginning of the instruction fetch and the acknowledgement that the access has indeed been performed. Let a platform with hardware components $\{f,g,h\} \in C$ that generate events $\{e_f,e_g,e_h\} \in E$ with relations $e_f \leftarrow\!\!\shortmid \cdots \leftarrow\!\!\shortmid e_g$ and $e_f \leftarrow\!\!\shortmid \cdots \leftarrow\!\!\shortmid e_h$ (as would for instance do a *load multiple word instruction* in which e_f is the instruction event and e_g and e_h are the memory acknowledgement events), then the latency l of event e_f is defined as $l\langle e_f\rangle = \max(ts\langle e_g\rangle, ts\langle e_h\rangle) - ts\langle e_f\rangle$. Using this formula, we were able to reproduce the original results, thus showing that the links that build the causality chains allow reaching all necessary events to compute the missing latency information.

4.5.3 Slowdown and Accuracy

The trace system requires additional code to capture, process and store events. It thus may modify the way the simulation framework counts the elapsed simulated time and will incur execution slowdown. The simulation time advances at each synchronization point using the amount of clock cycles executed since the last synchronization point. Depending on the targeted speed/accuracy trade-off, we can use either the *individual* or *block* strategies defined at the end of section 4.4. To verify how it impacts the accuracy of the overall system, we ran different experiments. The obtained simulated time and execution time are reported Fig. 4.12 for target platforms which include 1, 2, 4, 8 and 12 processors. Graph 4.11 y-axis represents the amount of system clocks spent to decode

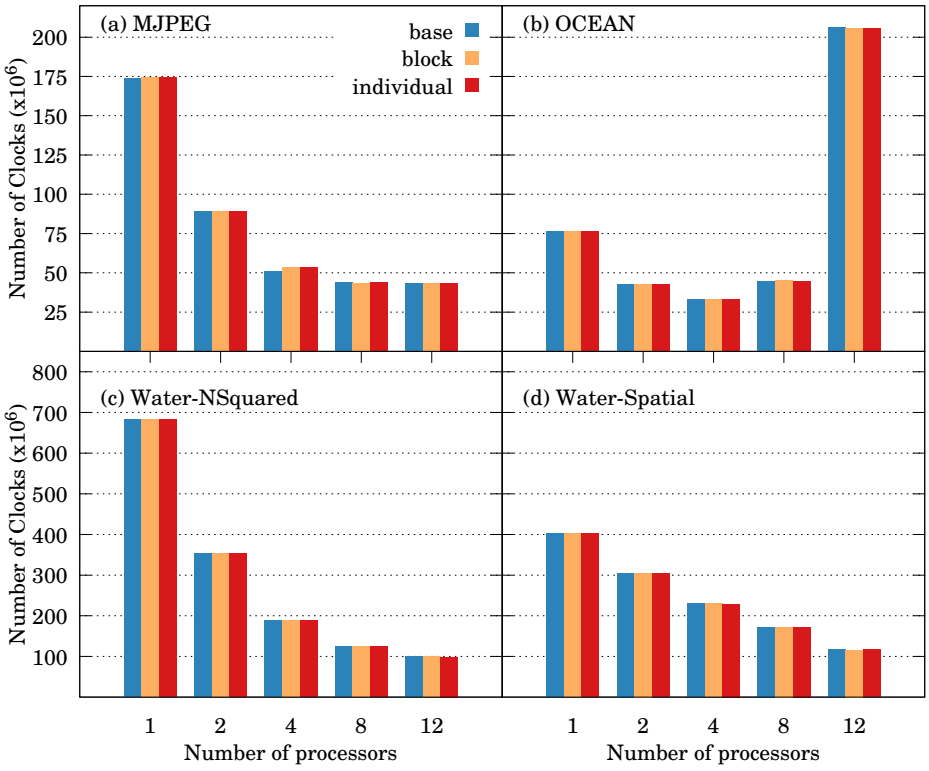


Figure 4.11: Simulated time for 1, 2, 4, 8 and 12 processors.

Number of Processors	MJPEG		Ocean		WaterNSquared		WaterSpatial	
	Block	Indiv	Block	Indiv	Block	Indiv	Block	Indiv
1	0.44%	0.44	0.00%	0.01%	0.00%	0.01%	0.00%	0.00%
2	0.38%	0.38	0.17%	0.17%	0.00%	0.03%	0.01%	0.19%
4	3.89%	4.02	0.08%	0.10%	0.07%	0.24%	0.41%	0.53%
8	1.07%	0.07	0.34%	0.22%	0.05%	0.70%	0.02%	0.23%
12	0.11%	0.14	0.16%	0.23%	0.11%	1.08%	0.74%	0.30%

Table 4.1: Accuracy variation for *block* and *individual* time advancing strategies in comparison to non-traced simulation.

30 frames using MJPEG and to execute Ocean, Water-NSquared and Water-Spatial applications. The system clock represents the number of clocks necessary to completely execute an application. We assume that our simulation platform has just one global clock. The table 4.1 represents the accuracy variation observed in Fig. 4.11 in terms of percentage.

As observed, the number of cycles remains almost the same in the three cases, the difference being mostly due to rounding errors during the conversion of clock ticks to seconds. The simulation time is clearly greatly affected, and the trace-less implementation is a lot faster. This is due firstly to the time required to capture, process and store events, which requires currently at least one function call per target instruction during execution. Storing the events to disk also participates to this large slowdown. And secondly, it is

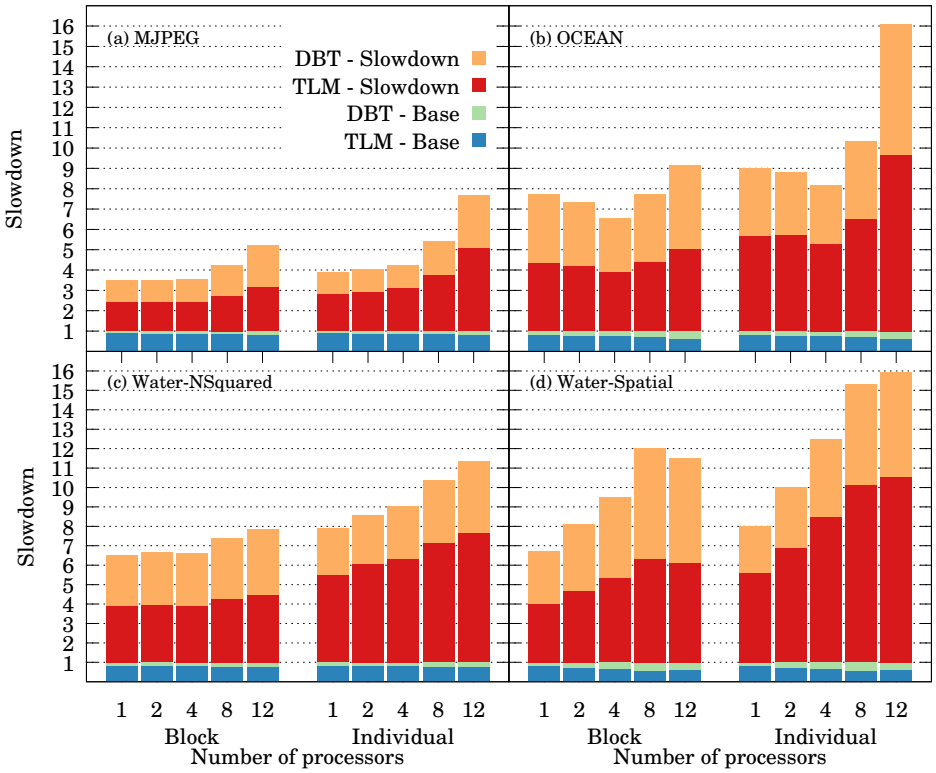


Figure 4.12: Slowdown for 1, 2, 4, 8 and 12 processors.

Size (GB)	# of Processors				
	1	2	4	8	12
MJPEG	19.00	20.00	22.00	32.00	44.00
Ocean	4.06	4.46	6.35	14.82	93.56
Water-NSquared	35.30	36.19	37.81	46.02	52.79
Water-Spatial	20.87	27.78	38.19	53.81	54.49

Table 4.2: Storage cost for some processor configuration.

because of the increasing synchronizations with the TLM simulator, SystemC in our case. Other timestamping strategies have to be defined to minimize slowdown, at the cost of precision.

Graph 4.12 y-axis plots the normalized slowdown induced by our buffering strategies compared to the platform without trace support (base platform) for different applications. Each bar in Graph 4.12 is divided vertically into four pieces: *TLM - base, DBT - base, TLM - slowdown and DBT - slowdown*. The first two represent the normalized base platform execution divided into time spent in TLM and DBT environments. The two last represent the overhead for capturing traces. The combination of all represents the whole. The x-axis is the number of processors in both cases.

According to [39], the Rabbits framework achieves a $\approx 20\times$ simulation speedup compared to statically scheduled cycle accurate simulation, considering a three-processor platform. Then, even with this costly trace generation run-time impact, the simulation with trace, while providing additional information, is still faster than the cycle accurate one.

Table 4.2 gives the sizes, in GByte, of the trace files generated during the simulation of each platform.

4.6 Conclusion

This chapter introduces the trace's formalism and defines what we consider a "well formed" trace. The most interesting outcome is that the "well formed" traces contain causality relations, which open opportunities in reducing the theoretical complexity, and thus increase the practical interest, of many algorithms based on traces aiming at analyzing software and hardware properties.

It also presents a method to produce detailed event traces within simulation environments making use of dynamic binary translation for software code execution. As opposed to "simple" dumps, our traces, thanks to the fact that they are obtained using a virtual platform, allow building and maintaining causality relations between events. The main challenges to overcome were firstly to track causality relations and secondly to maintain the order of events. We have proposed to embed information in non-functional micro-operations to solve both challenges, and have experimentally shown using several examples that this solution was indeed solving the issues.

Chapter 5: Cache Coherence Analysis based on Traces

Contents

This chapter presents our second contribution that aims at making use of the "well-formed" traces to identify cache coherence violations in cache non-coherent architectures. Software cache coherence schemes tend to be the solution of choice in dedicated multi/-many core systems on chip, as they make the hardware much simpler and predictable. However, despite the developers efforts, it is hard to make sure all preventive measurements are taken to ensure coherence. We propose a method to identify the potential cache coherence violations using traces obtained from virtual platforms. These traces contain causality relation among events, which allow firstly to simplify the analysis, and secondly to avoid relying on timestamps. These relations are discussed in details in Chapter 4.

Our method identifies potential violations which may occur during a given execution for write-through and write-back cache policies. Therefore, it is independent of the software coherence protocol.

5.1 Objectives

We propose a method to identify potential cache coherence violations in cache non-coherent multiprocessor architectures to help engineers find and solve them. These platforms rely on software support to maintain cache coherence. The objectives of our contribution takes the above assumptions into account.

- *Detecting potential problems*: the first objective is to identify potential cache coherence problems that are present in a given execution trace. In this context, potential problems mean that there are problems which are not surely pinpointed as a problem but need a special attention. As we use traces as a representation of the system, the proposed method can be classified as a semi-formal approach, we aim at identifying all possible errors detectable based on given execution traces. It does not mean that all errors present in the source code are spotted but just those ones that the execution traces contain. When a cache coherence violation is identified a human readable output is necessary, which shows enough information allowing identification of what caused the error through code source annotation.
- *Coherence Protocol Independence*: The second objective is to define a method independent of the software cache coherence protocol. It means that the definitions of software protocols do not influence the detection method, allowing the detection method to verify also the effectiveness of the protocols themselves. Each software protocol has its own internals, such as main states and intermediate states. Analysing these internals for each protocol will lead to a detection method for each protocol. Thus, we avoid this kind of approach. Consequently, we are interested in approaches that analyse what matters: the order of store and load operations at each memory address. We aim at ignoring "support" operations, such as invalidate and flush which are operations that discard and effectively store the cache value in memory, respectively. However, the consequences of a flush operation, the store operation, is taken into account. Maintaining the method focused on stores and loads decreases the complexity of the analysis, speeding up the overall analysis time.
- *Implicit Architectural Details*: The third objective is to maintain the method independent of memory consistency model and interconnection, however, these factors influence cache coherence directly. The memory consistency model can vary from a simple sequential model to complex out-of-order models, then the order of each actual memory access is determinant to correctly identify the problem. Identically, the interconnection may vary from basic buses to complex Network On Chips (NoC) which also modifies the order of memory accesses. As those factors are important to problem detection, they have somehow to be taken into account, however, they are not the key factor in cache coherence problem identification. Thus, a method independent of such architectural decisions allows exploration of different configuration without modifying the cache coherence detection method.

To reach all these objectives, we propose a method that identifies potential cache coherence violation in MPSoCs. The method relies on execution traces obtained from virtual platforms obeying the formalism described in Chapter 4 and giving the following contributions:

- A novel cache coherence violation detection method based on MPSoC's virtual platform execution traces.
- A detailed set of rules using events' order to detect cache coherence violations in two write policies (*write-through* and *write-back*). Each write policy is covered by a specific set of rules. The proposed rules do not take into account timestamps, being focused on the order of events that can be obtained on a purely functional simulation.
- A functional implementation that provides useful information to engineers to identify and to correct the coherence problems. Our method shows information that indicates the type of problem itself, which events cause the problem and which line of source code represents the event. We believe that all this information is enough to correct the cache coherence violations.

5.2 Trace Definition

In Chapter 4 the trace is defined as $T = (E, <, \hookleftarrow, \prec)$, where E is the set of events, $<$ the component strict total order, $\hookleftarrow$ the causality relation among events and $\prec$ the total system order. To identify potential cache coherence problems, the relation $\prec$ is explored.

The detailed definition of $\prec$ relation is available Chapter 4. Summarizing, it is a relation based on $<$ and $\hookleftarrow$ relying on a shared component to create a total order of events. This relation aims at comparing the cache events and the memory events. We elaborate a set of rules based on this relation to capture errors.

To ease the drawing of the examples, we define the notation $X_{c.o}$ as a X type event generated by component c, and having the system total order o. For example, the event $L_{1.5}$ is a load event that was generated by component 1 with a system total order of 5.

However, for the rules description, the system order attached to the events is omitted to simplify the notation, and replaced by the relation $\prec$ between the events. For example, if $S_{1.7}$ and $L_{2.8}$ are events generated by the components $\{1,2\} \subseteq C$, as their order is respectively 7 and 8, they are ordered by $S_1 \prec S_2$.

5.3 Hierarchy Independence

The number of available architectures and subcomponents variations present in MPSoCs results in a huge space of possible architectures. Our method is architecture independent being applicable to different architectures, like different types of interconnection, out-of-order processors, variations of memory consistency models, so on and so forth.

Our method relies on information that is embedded in traces. Details about interconnection, for example, are reflected by the order relations present in traces. An example of how it is taken into account is found in Chapter 4.

Another point is, our event definitions do not include explicit invalidation events, even though invalidations are used by the software coherence protocols since they help keep the caches updated with the latest modifications. Excluding invalidation events doesn't mean that they are not taken into account. In fact, when a processor requests data, that lie within an invalidated cache line, these data have to be retrieved from main memory, hence, an acknowledgment event is created by the memory. Accordingly, the

memory accesses represented by acknowledgement events can implicitly mean that an invalidation has taken place before a cache request, which explains the reason behind of elimination of invalidate events from traces.

5.4 Detection of Cache Coherence Violations

In this section we discuss the meaning of cache coherence and present the formalism to detect its violation. The first part defines what we mean by cache coherence in the system. After that, we define the violations for the two types of cache write policies, *i.e* write-through and write-back. For each policy, we define a set of rules that once violated, pinpoints a cache coherence problem, relating the source code to the cause of the problem. At the end, we discuss potential identification of what we call *false positives*, which is a situation in which we can not surely assume whether it is a real error or not.

5.4.1 Violation causes

A memory scheme is coherent if the value returned by a load instruction is always the value given by the latest store instruction at the same address [80]. Considering hardware approaches, the goal of cache coherence is to make caches in multicore systems as invisible as caches in single-core systems [23]. However, software protocols for cache coherence are generally intrusive and they can lead to coherence problems when awry implemented. We will use the term "software protocol" as short for "software protocol for cache coherence".

Software protocols depend on cache writing policy. In the *write-through* policy, a violation takes place when a processor stores data in the main memory, and, after that, another processor loads data directly from its cache. This means that the second processor is not aware of the latest data update made by the first processor. The *write-back* policy is more delicate to deal with because there are two situations that can instigate coherence problems. The first situation is when a processor stores data in its cache and after that another processor loads data from its cache or from the main memory. Since the last version of data exists only in the first processor's cache, the second processor will work with the wrong version. The second situation is when a processor stores data in main memory (so here the main memory is up to date) but the second processor loads data directly from its own cache without seeking the latest version from main memory because it is not aware that the data has been modified by the first processor.

5.4.2 Write-through violation

Basically, the write-through violation happens when a load from a cache does not get the latest up-to-date value from the memory, thus leading the processor to load an old value from its own cache. Rules to identify this problem are described below. The premises we have are $\exists\{S_i, L_j\} \subseteq E$ and $\exists\{i, j\} \subseteq C$.

(1)
1. $S_i \prec L_j$ (the store event should happen before the load event);
2. $\nexists S_k$ such that $S_i \prec S_k \prec L_j$, where $k \in C$ (there should not be any other store events after the store in question and before the load in question);
3. $\nexists L_j^*$ such that $S_i \prec L_j^* \prec L_j$, where $L_j^* \neq L_j$ (the load in question should be the first load issued by the processor so it is the first that may cause a problem);

4. $\nexists L_j$ such that $L_j \leftarrowtail A_l$ (the load in question should not be followed by a load from memory);

Fig. 5.1 represents a typical execution containing a violation. In this example, there are three processors accessing the same address. As observed, the second event S_1 and the last one L_3 satisfy the aforementioned rules. So, a coherence problem is spotted. In other words, in this case there is a load without an acknowledgement (a load made directly from the cache without accessing main memory) after a store made by another processor. However, if we consider the second event S_1 and the fourth event L_2 we notice that they do not fulfill condition 4, since $\exists A | L_2 \leftarrowtail A$ then there is no problem caused by these two events.

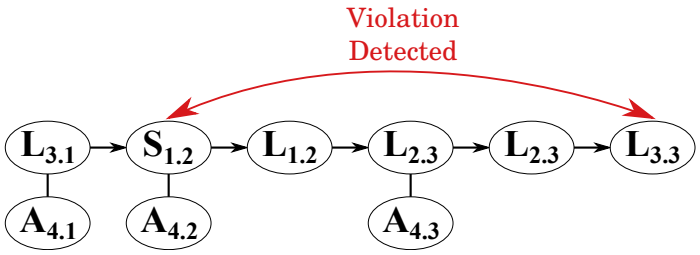


Figure 5.1: Write-through violation detection using system order.

5.4.3 Write-back violation

The write-back mechanism is more complex to analyse due to the fewer memory accesses. Indeed, more than one situation may engender problems. As a consequence, we propose the following set of rules to deal with the write-back policy. Considering the premises: $\exists\{M_i, L_j\} \subseteq E$ and $\exists\{i, j\} \subseteq C$, we have:

(2)
1. $M_i \prec L_j$
2. $\forall k \nexists M_k$ such that $M_i \prec M_k \prec L_j$
3. $\nexists L_j^*$ such that $M_i \prec L_j^* \prec L_j$, where $L_j^* \neq L_j$
4. $\nexists B_i$ such that $M_i \prec B_i \prec L_j$ (there should not be a store B released by the cache of the processor that performed the Modify in question after this Modify and before the load in question).

To illustrate the rules, Fig. 5.2(a) shows an example of violation in a two-processor-based architecture using the write-back policy. It indicates that two events M_2 and L_1 cause a problem since they satisfy the above rules. On the other hand, M_2 and L_2 do not generate a violation because they do not match the condition that requires that two events should belong to two different processors.

The example in Fig. 5.2(b) represents another situation that is detected by rule (2). In this case, the events that cause the violation are M_2 and L_1. Although there is a store into main memory B_1 between the two considered events which means an update took place in main memory, it is not the correct update to be done. The correct one should be M_2 since the last modification of the data has been done by processor 2. As the correct update did not happen, processor 1 will access an erroneous data, resulting in a violation report.

The second rule set deals with stores in main memory and have the same premises as the first set of rules. It is defined as follows:

(3)
1. $M_i \prec L_j$
2. $\forall k \nexists M_k$ such that $M_i \prec M_k \prec L_j$
3. $\nexists L_j^*$ such that $M_j \prec L_j^* \prec L_j$, with $L_j^* \neq L_j$
4. $\exists B_i$ such that $M_i \prec B_i \prec L_j$
5. $\nexists A_j$ such that $L_j \leftarrowtail A_j$

Fig. 5.2(c) shows an example of a situation where rule (3) can be applied. The events M_2 and L_1 satisfy the rule. So a violation is detected.

A write-back operation, as it stores the whole cache line which may include out-of-date address values, can cause violations on neighbor addresses of the actually modified address. Rule (4) is a complement of rule (2) as it handles store operations. It is defined as follows:

(4)
1. $S_i \prec S_j$
2. $\nexists S_k$ such that $S_j \prec S_k \prec S_j$, where $k \in C$
3. $\nexists M_k$ such that $S_j \prec M_k \prec S_j$, where $k \in C$
4. $\exists L_j$ such that $S_i \prec L_j \prec S_j$ and $\exists A_j$ such that $L_j \leftarrowtail A_j$

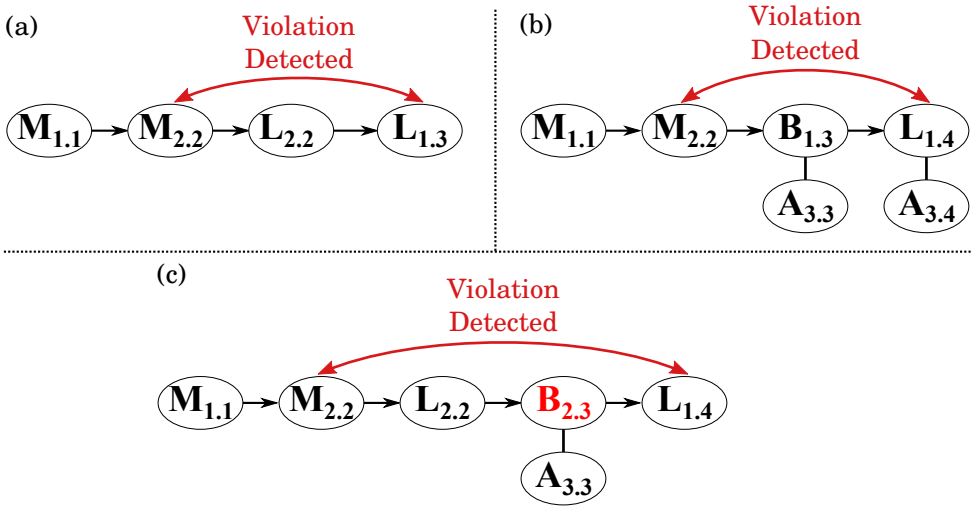


Figure 5.2: Write-back violation detection using system order, where (a) and (b) are violations detected by rule (2) and (c) by rule (3).

5.4.4 Elimination of false positives

The above rules may lead to *false positives*, *i.e.* the actual spotting of a violation while no violation can take place. Indeed, in rare situations, the reordering of cache events we perform using the $\lll$ and $\ggg$ operators to identify the potential violations could stretch too much the events apart. To deal with it, the solution consists of considering

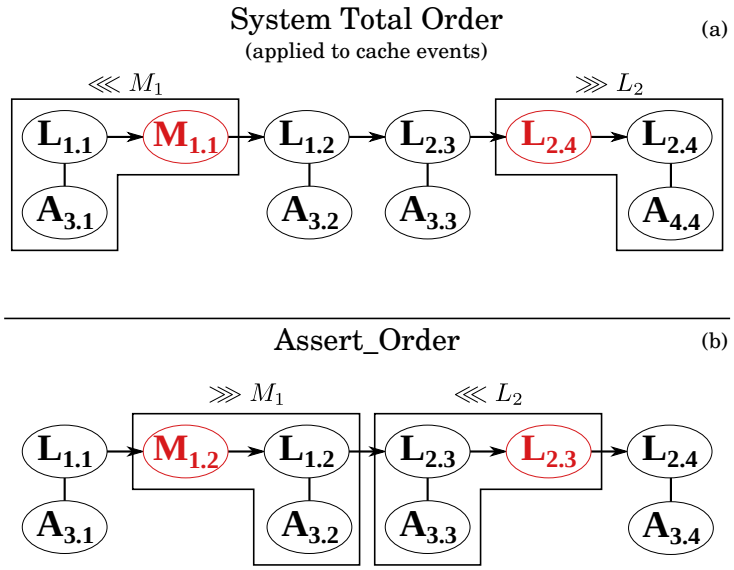


Figure 5.3: Eliminating *false positives* using the ASSERT_ORDER procedure.

also the opposite ordering, *i.e.* moving the stores toward their posterior events and the loads towards their preceding one (inverse operation of the one represented in Fig. 4.7). Therefore, when a violation is pinpointed, this additional reordering is performed. If it contains the same succession of events as the original reordering, which means that the first condition of each set of rules is preserved, then a problem is truly detected.

Operationally, we define the function ASSERT_ORDER(p, e), with $p \prec e$ the relation to be tested. The function performs the operations $\ggg p$ and $\lll e$. If the relation $p \prec e$ is still true, then the global order $p \prec e$ is valid and there is an actual violation. In this case, the violation is reported and the function stops the analysis, otherwise, it signals that it cannot determine whether there is a problem or not, and the analysis goes on.

To illustrate this operation an example is depicted in Fig. 5.3. This example consists of two processors, performing four memory accesses to different addresses in memory, each one performs two read operations. Between these memory accesses each processor performs a cache operation: "a modify" by cpu1 and "a read" by cpu2. The cache operations are both to the same addresses. According to total system order attribution to cache events, the cache-modify access receives the total system order equal to 1 due to the $\lll$ operation and cache-load access receives 4 due to the $\ggg$ operation, as observed in Fig. 5.3.(a). Let's consider a violation between $M_{1.1}$ and $L_{2.4}$, then an ASSERT_ORDER function takes place to evaluate the error. In this case, the opposite operation in both cache operations, observed in Fig. 5.3.(b), changes the global order of both events. However, the relation $M_1 \prec L_2$ remains the same in both cases, then there is no false positive.

5.5 Detection algorithms

Our approach is based on the traversal of acceptor Deterministic Finite Automata (DFA) that checks the rules defined previously. There is one such DFA per address. We use a data structure to maintain relevant information about the access to all memory addresses,

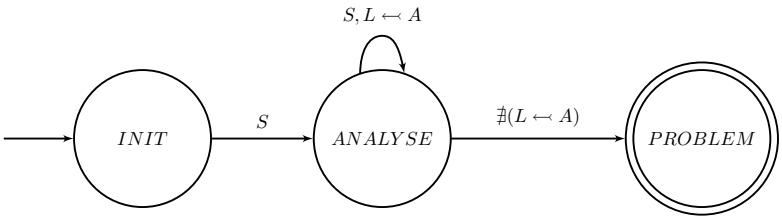


Figure 5.4: Simplified DFA for write-through violation detection

defined as $\{a(0), a(1), ..., a(n-1)\}$, where n is the memory size and $a(x)$ an address tuple which represents the address x. The tuple is defined as $a(x) = (s, m, e, \Theta)$. The meaning of the elements is as follows: s is the last store event at address x; m the last modify event at address x; e the current event occurring at address x; and Θ the set of components which loaded data at address x after s.

Let DFA $M = (Q, \Sigma, \Delta, q_0, F)$, with Q the finite set of states, Σ the input alphabet, Δ a transition function, $q_0 \in Q$ the initial state and $F \subseteq Q$ all accepted final states.

The DFA detecting *write-through* violations implements the rules (1) described in subsection 5.4.2. It is defined as $M_{wt} = (\{\texttt{INIT}, \texttt{ANALYSE}, \texttt{PROBLEM}\}, \{L, S, A\}, \Delta, \texttt{INIT}, \{\texttt{PROBLEM}\})$. Fig. 5.4 is a sketch of M_{wt}.

Its input is the event $e \in E$ such that it obeys the system order relation $\prec$ and there is no other event p to be analysed such that $p \prec e$. We denote $\mathcal{E} \subset E$ the set of events having the same system order as e. The memory address attributes s, m and Θ are updated while new events are received.

The DFA M_{wt} is initialized when an event of type S occurs. This satisfies the analysis of rules (1).1 and (1).2 which are start conditions. The rules (1).3 and (1).4 are analysed in state `ANALYSE`. Algorithm 1 details the DFA. Line 10 guarantees the satisfiability of

Algorithm 1 Write-Through Cache Coherence Violation Detection

Require: e
1: **if** $State = \texttt{INIT}$ **then**
2: **if** $e.t = S$ **then**
3: $s \coloneqq e$; $\Theta \coloneqq \varnothing$
4: $State \coloneqq \texttt{ANALYSE}$
5: **end if**
6: **else if** $State = \texttt{ANALYSE}$ **then**
7: **if** $e.t = S$ **then**
8: $s \coloneqq e$; $\Theta \coloneqq \varnothing$
9: **else if** $e.t = L \wedge \{\exists e_j \in \mathcal{E} | e_j.t = A \wedge e \rightarrowtail e_j\}$ **then**
10: $\Theta \coloneqq \{e.c\} \cup \Theta$
11: **else if** $e.t = L \wedge \{\nexists e_j \in \mathcal{E} | e_j.t = A \wedge e \rightarrowtail e_j\}$ **then**
12: **if** $e.c \notin \Theta$ and $e.c \neq s.c$ **then**
13: ASSERT_ORDER(s, e)
14: $State = \texttt{PROBLEM}$
15: **end if**
16: **end if**
17: **else if** $State = \texttt{PROBLEM}$ **then**
18: Show the violation
19: **end if**

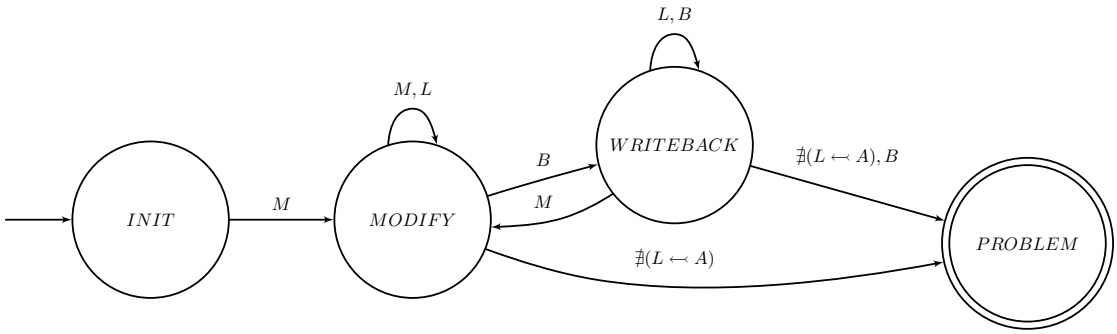


Figure 5.5: Simplified DFA for Write-Back Violation Detection

rule (1).3 while line 12 tests rule (1).4. In this algorithm and the following ones, we will use a dot (.) notation to refer to an element of an event tuple (as defined in Chapter 4) or address tuple.

Write-back verification checks the set of rules (2) and (3). Its DFA is defined as: $M_{wb} = (\{\texttt{INIT}, \texttt{WRITEBACK}, \texttt{MODIFY}, \texttt{PROBLEM}\}, \{M, B, L, A\}, \Delta, \texttt{INIT}, \{\texttt{PROBLEM}\})$. Fig. 5.5 gives a simplified view of M_{wb}. As for the write-through DFA, its input is the most recent cache event $e \in E$. The detailed behaviour of M_{wb} is given in Algorithm 2.

The first three statements of rules (2) and (3) are the initial conditions to analyse. `MODIFY` state tests events when there is at least one modification in cache without an actual update of memory. The verification of possible violations is done at lines 8, 11 and 38. The two first are related to rule (2).4 and the last one to rule (4).4.

The `WRITEBACK` state tests properties similar to the ones verified in write-through.

As it can be easily seen, the complexity of the algorithms is $O(n)$, n being the number of events.

5.6 Implementation and Experimentation

Our experiments aim to show the efficacy of our method to detect coherence violations. We execute applications and operating systems that have initially been conceived to be executed on hardware cache-coherent platforms on cache non-coherent hardware. Then, we identify the potential problems using our method, correct them and re-execute the programs. We repeat this process until no errors are identified. The hardware platforms and software environments are presented hereafter, along with an *ad-hoc* naïve method to correct the violations, the only goal being to show that the execution is finally correct.

Our experimentations use a virtual prototype and an analysis tool, as depicted Fig. 5.6. The virtual prototype is similar to the one presented in Chapter 4, and thus produces "well formed" traces. The traces are sent to the analysis tool through a Unix pipe, avoiding disk usage and allowing parallelism between simulation and analysis. The analysis tool (*Sherlock*) uses the *Trace Reader* to recover the set of events and their relations. They are forwarded to *Sherlock* to verify the rules and pinpoint the cache violations. Finally, *Sherlock* generates a report containing supplementary information recovered from the ELF file using the DWARF data, such as source code lines associated with the error.

The virtual platform is composed of n Cortex-A9 ARM processors (n being a user parameter) with separate L1 data and instruction caches, a frame buffer, a serial controller,

Algorithm 2 Write-Back Cache Coherence Violation Detection

Require: event e

1: **if** $State = \texttt{INIT}$ **then**
2: **if** $e.t = M$ **then**
3: $s \coloneqq \perp$; $m \coloneqq e$; $\Theta \coloneqq \varnothing$
4: $State \coloneqq \texttt{MODIFY}$
5: **end if**
6: **else if** $State = \texttt{MODIFY}$ **then**
7: **if** $e.t = L \wedge \{\nexists e_j \in \mathcal{E} | e_j.t = A \wedge e \leftarrowtail e_j\}$ **then**
8: **if** $m.c \neq e.c$ **then**
9: $State \coloneqq \texttt{PROBLEM}$
10: **end if**
11: **else if** $e.t = L \wedge \{\exists e_j \in \mathcal{E} | e_j.t = A \wedge e \leftarrowtail e_j\}$ **then**
12: $State = \texttt{PROBLEM}$
13: **else if** $e.t = M$ **then**
14: $\text{ASSERT\_ORDER}(m, e)$
15: $s \coloneqq \perp$; $m \coloneqq e$; $\Theta \coloneqq \varnothing$
16: **else if** $e.t = B$ **then**
17: $\text{ASSERT\_ORDER}(m, e)$
18: **if** $m.c = e.c$ **then**
19: $m \coloneqq \perp$; $s \coloneqq e$; $\Theta \coloneqq \varnothing$
20: $State \coloneqq \texttt{WRITEBACK}$
21: **end if**
22: **end if**
23: **else if** $State = \texttt{WRITEBACK}$ **then**
24: **if** $e.t = L \wedge \{\nexists e_j \in \mathcal{E} | e_j.t = A \wedge e \leftarrowtail e_j\}$ **then**
25: **if** $e.c \notin \Theta$ and $e.c \neq s.c$ **then**
26: $\text{ASSERT\_ORDER}(s, e)$
27: $State \coloneqq \texttt{PROBLEM}$
28: **end if**
29: **else if** $e.t = L \wedge \{\exists e_j \in \mathcal{E} | e_j.t = A \wedge e \leftarrowtail e_j\}$ **then**
30: $\Theta \coloneqq e.c \cup \Theta$
31: **else if** $e.t = M$ **then**
32: $\text{ASSERT\_ORDER}(s, e)$
33: $s \coloneqq \perp$; $m \coloneqq e$; $\Theta \coloneqq \varnothing$
34: $State \coloneqq \texttt{MODIFY}$
35: **else if** $e.t = B$ **then**
36: **if** $e.c \in \Theta$ **then**
37: $s \coloneqq e$; $m \coloneqq \perp$; $\Theta \coloneqq \varnothing$
38: **else**
39: $State \coloneqq \texttt{PROBLEM}$
40: **end if**
41: **end if**
42: **else if** $State = \texttt{PROBLEM}$ **then**
43: Show the Violation
44: **end if**

timers, a shared memory and a storage controller. Components are interconnected using a Network on Chip (NoC).

The software benchmark is composed of an operating system part and an application part. On the OS's side, we use a lightweight SMP operating system, called DNAOS.On

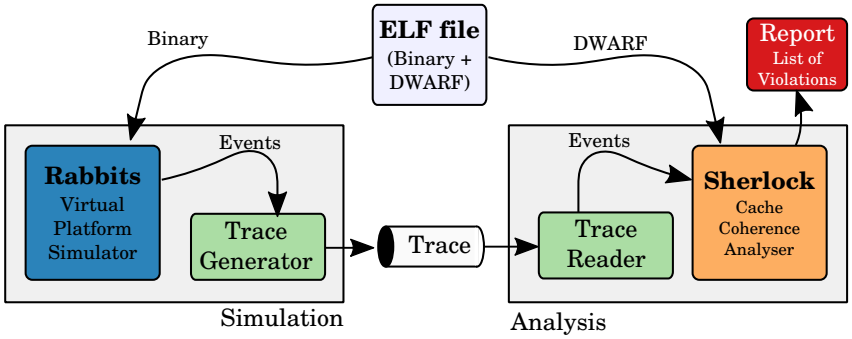


Figure 5.6: Virtual prototyping and analysis flow.

the applications' side, we use a subset of the Splash-2 benchmark and a parallel implementation of a Motion-JPEG decoder, all of them running over DNAOS and *newlibc* as standard C library.

5.6.1 Outputs

Additional information about where and how a violation occurs allows the identification of the problem. The codes given by Listing 5.1 and 5.2 are used as examples. In this example, `lock` operations are performed outside those portions of code. Lines 30 (`queue_add.c`) and 25 (`queue_rem.c`) show the manipulation of the `status` variable. It is there where the examples are focused on. This code is executed in a multiprocessor environment, thus this variable can be accessed concurrently by different processors.

To quickly identify the problem, we categorize the components based on their type of access as: WRITER, READER and MODIFIER. The WRITER is a component that actually requests a write in the main memory. It is usually a cache memory, but a processor can play this role when considering an uncached access. The READER is a component that reads an address content. It is usually a cache memory. MODIFIER is a

```
20  status_t queue_add (queue_t * queue, void * data)
21  {
22      if (queue -> status == 0) {
23        queue -> head = item;
24        queue -> tail = item;
25      }
26      else {
27        queue -> tail -> next = item;
28        queue -> tail = item;
29      }
30      queue -> status += 1;
31
32      return DNA_OK;
33    }
34  }
```

Listing 5.1: queue_add.c source file that reads and modifies the variable `status` at line 30.

```
20  void * queue_rem (queue_t * queue)
21  {
22      queue_link_t * item = NULL;
23
24      item = queue -> head;
25      queue -> status -= 1;
26      queue -> head = item -> next;
27      item -> next = NULL;
28      return item;
29    }
30  }
```

Listing 5.2: queue_rem.c source file that reads and modifies the variable `status` at line 25.

component that modifies a data in a cache without requesting a memory update. In the write-through policy, all write operations are prompted immediately when they arrive, then components are classified as WRITERs and READERs. On the other hand, a write operation in write-back policy can result in a cache modification, then the MODIFIERs classification can be present when this policy is used.

During the analysis of traces, the result of a violation identification is a detailed output. This output aims at explaining why the problem occurred. This information is organized as fields that are shown in Listings 5.3, 5.4 and 5.5.

Most of the fields are intuitively named, for example VIOLATION reports the identified error. However, the order of event is described in a complex way. The main order of the event is called `system_order` that represents the System Order of the event. The two other given orders,`pre_order` and `pos_order`, are the result of operations $\lll$ and $\ggg$ using `system_order`, respectively. Both these operations give further details aiming at comparing correctly the order of two events.

Two examples of violations are shown Listings 5.3 and 5.4, representing both cache write policies. Listing 5.3 shows the output of a violation of rule 1 for a write-through cache. In this case, `CACHE_0` writes a new value in variable `status`. In the write-through policy, write cache operation modifies also the memory. After that, `CACHE_2` needs the

```
1  VIOLATION: [1] Reading an old cache value.
2  ADDRESS  : 0x00002EBC
3  VARIABLE : status
4  WRITER   : component = [0] - CACHE_0
5           system_order = 12344  pre_order = 12344  pos_order = 12344
6                     pc = 0x0002FFAC
7                   file = queue_add.c
8                   line = 30
9  READER   : component = [2] - CACHE_2
10          system_order = 123223 pre_order = 123200 pos_order = 123223
11                    pc = 0x00032EFC
12                  file = queue_rem.c
13                  line = 25
```

Listing 5.3: An output example of rule 1 violation

```
1  VIOLATION: [2] Reading a modified cache value.
2  ADDRESS  : 0x00002ebc
3  VARIABLE : status
4  MODIFIER : component = [3] - CACHE_3
5           system_order = 12232  pre_order = 112232 pos_order = 122314
6                     pc = 0x00003DFC
7                   file = queue_add.c
8                   line = 30
9  READER   : component = [2] - CACHE_2
10          system_order = 221232 pre_order = 221212 pos_order = 221232
11                    pc = 0x000321FF
12                  file = queue_rem.c
13                  line = 25
```

Listing 5.4: An output example of rule 2 violation

value stored in `status`. However, this value is already present in `CACHE_2`, then due to an erroneous coherence behavior, the read value comes from the cache instead of the memory, hence, a violation takes place.

Listing 5.4 shows a violation of rule 2 occurring in a write-back cache policy. In write-back, unlike write-through, the storage of a variable does not imply an immediate write in memory. This modified value can stay in the cache until a triggering action occurs, such as a flush operation. Listing 5.4 details an error due to a read made by `CACHE_2` of the `status` variable that is already modified by `CACHE_3` but remains in its cache. In this case, even if `CACHE_2` reads `status` from memory, it is out-of-date and is, thus, a violation.

However, during the analysis inconclusive results can occur. We define an operation to deal with these events, but in some cases the events happen so close that it is impossible to ensure their relative order. The `ASSERT_ORDER` function processes the `pre_order` and `pos_order` of each event under analysis to infer their relative position. Then, when an inconclusive result occurs, an output showing the results is displayed. In Listing 5.5, we observe that when the operations $\lll$ and $\ggg$ are performed on both events, their order

```
1      VIOLATION: Potential problem reading a modified cache value.
2      ADDRESS  : 0x00002ebc
3      VARIABLE : status
4      MODIFIER : component = [3] - CACHE_3
5               system_order = 12345000  pre_order = 12344999 pos_order
                   = 22123050
6                        pc = 0x00003DFC
7                      file = queue_add.c
8                      line = 30
9      READER   : component = [2] - CACHE_2
10              system_order = 12345001  pre_order = 12344980 pos_order
                   = 22123001
11                       pc = 0x000321FF
12                     file = queue_rem.c
13                     line = 25
```

Listing 5.5: An output example of identification of a potential violation.

in the system is reversed, thus the READER event can occur before the MODIFIER one. This leads to two different situations, hence, a precise conclusion cannot be obtained.

A method guaranteeing the coherence depends on the cache write policy and on the cache coherence protocol. The next section proposes our own method to correct those violations as easily as possible. This proposition does not aim at being efficient. Its goal is to solve the coherence problems.

5.6.2 Correcting the Violations

All modern architectures have commands to control cache line state by software. ARM architectures provide two commands (actually instructions accessing control registers): *invalidate* and *clean*, the latter one being known as *flush* in most other architectures. We use both to correct cache coherence problems.

In *write-through* policy, *invalidate* simply invalidates the line caching a specified address. Thus, an access to an invalidated address fetches the line from memory. Therefore, when a violation is detected, we simply add this command before the instruction that loads the address.

In *write-back* policy, we use both commands (*invalidate* and *clean*). If a violation is detected, then a *clean* command is placed after the store instruction and an *invalidate* command is placed before the load. However, the exclusive load and store instructions, which usually rely on the hardware cache coherence protocol, impose special considerations. Specifically the store must bypass the cache and directly access the memory which is responsible for granting the exclusive access.

These corrections do not claim to be general or efficient, as defining software protocols is out of the scope of this work.

5.6.3 Example usage of the violation detection algorithms

We now focus our discussion on the number of violations detected. As a program with coherence issues will probably crash before its complete execution, we use our method to locate the store and the load that caused a violation. Using this information, we proceed using the methods described in Section 5.6.2. After that, we re-execute the program to identify the next violation point, and then correct it as well. At the end, the "number of violations" represents the number of modifications necessary to eliminate all coherence problems for a given execution.

The number of violations detected per application is given Fig. 5.7. We process iteratively and interactively to identify the violations in all software layers, including DNAOS and *newlibc* code, and correct them. We did not remove the corrections when changing the number of processors or application. Thus, the order in which the applications were processed is important, namely MJPEG, Ocean, Water-nsquared and water-spatial, as the number of violations can only grow in the OS and libraries. Fig. 5.7.a shows the results using write-through policy and Fig. 5.7.b shows the write-back results. Each vertical bar gives the breakdown of the number of violations occurring, from bottom to top, in respectively *newlibc*, DNAOS, and the application, for 2 and 10 processor systems.

Violation points are related to parallel processing control operations (*locks, mutexes and barriers*) and shared variables, as expected. Obviously, most shared variables accesses

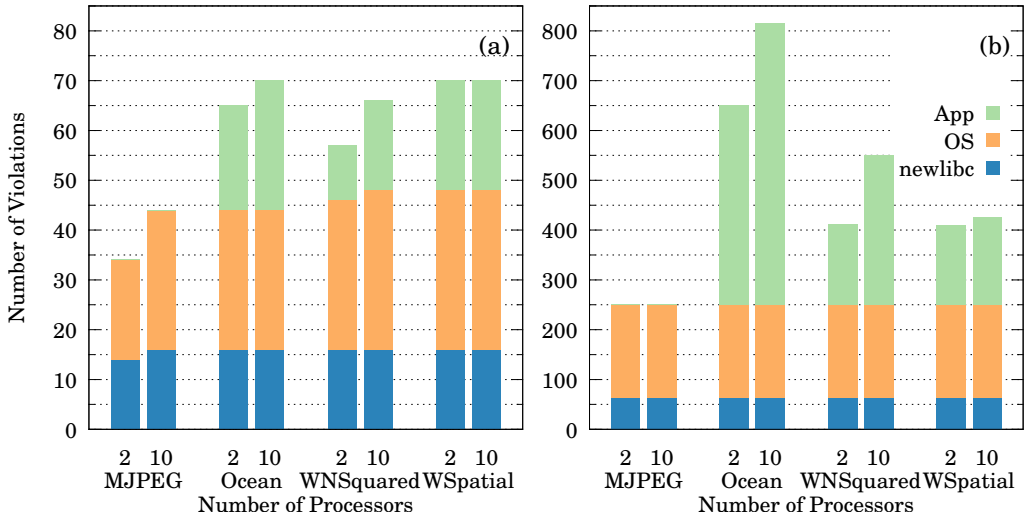


Figure 5.7: Number of violations detected for (a) write-through and (b) write-back policies.

occur in critical sections. Coherence must be maintained to ensure correct values of data upon entering and leaving the sections. Our tool identified the coherence problems occurring at these boundaries, which could be corrected.

Another point to spot is the application model of computation. MJPEG uses tasks communicating through FIFOs implemented in the kernel. In this case no violation was pinpointed in the application code. Splash-2 applications rely on a shared memory model and use parallel processing control operations in their code. We thus pinpointed and corrected several coherence problems. Even though theoretically possible, no violation on shared variables turned out to be a false positive.

5.6.4 Performances

Fig. 5.8 represents the simulation and trace capture time (green and blue bars) and the cache coherence analysis time (orange and red bars) for write-through and write-back policies respectively. The Splash-2 programs use their default parameters, while MJPEG decodes 10 Standard Definition frames.

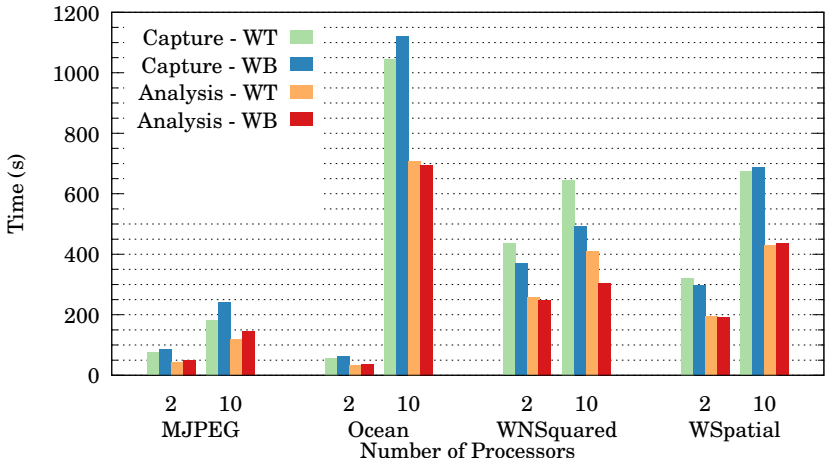


Figure 5.8: Simulation and trace capture time and cache coherence analysis time.

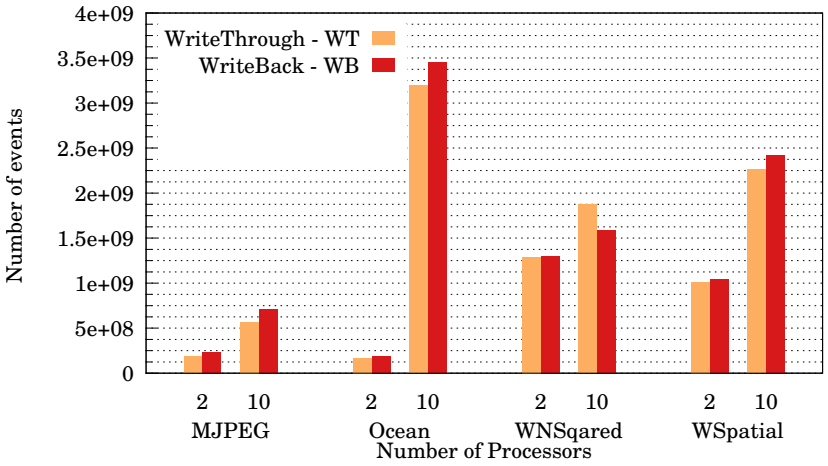


Figure 5.9: Number of events generated during simulation classified by number of processors, application and cache write policy.

As can be seen, simulation and trace capture is always slower than the violation detection algorithm. Because of the parallelism between simulation and analysis, the actual run time on modern machines is bounded by the simulation time.

Fig. 5.9 shows the number of events captured for both write policies. Actually, this number varies according to the cache coherence protocols, which can produce somewhat more or less cache operations. In this case, the data was obtained using the protocols detailed Section 5.6.2. By looking at the data of Fig. 5.8 and 5.9, we observe that the theoretical $O(n)$ complexity of our analysis algorithm, n being the number of events, is experimentally confirmed.

Fig. 5.10 shows the memory usage for the analysis of each write policy. Even though not negligible, is it far less than the number of events, thanks to the fact that the analysis can drop events once they have been used to change state in the DFAs.

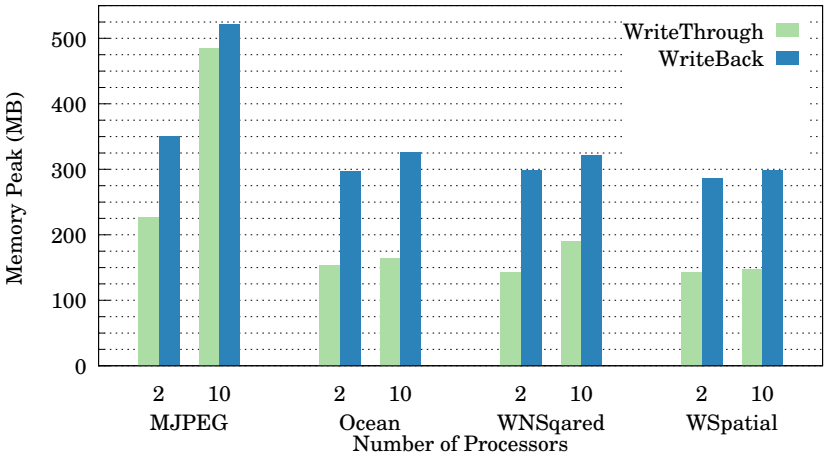


Figure 5.10: Peak memory usage in MBytes

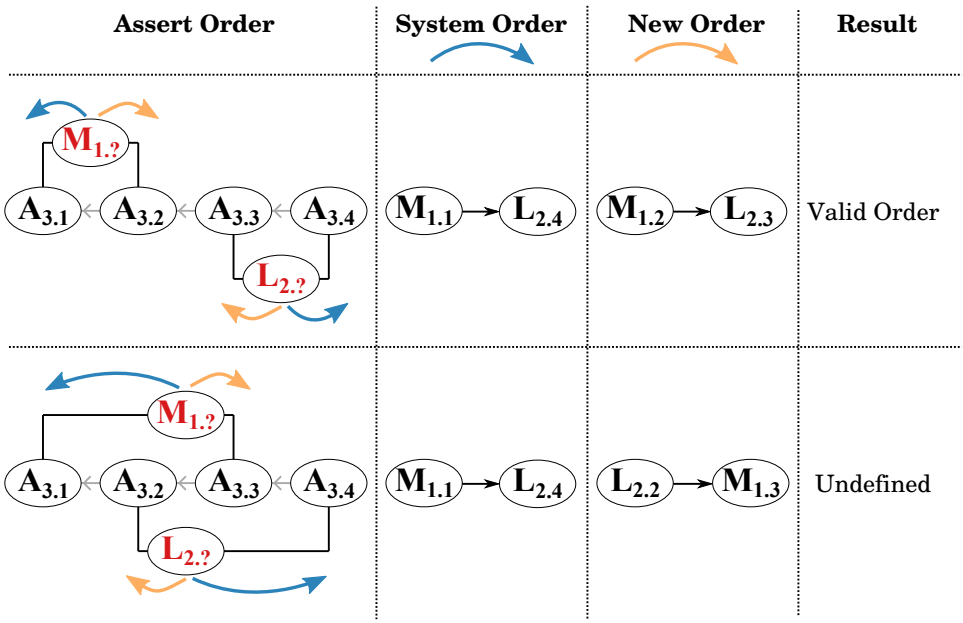


Figure 5.11: Valid and undefined system order examples.

5.7 Limitations

Our approach has some limitations. The first and most obvious one is that the possible violations are detected only for a given run of the programs, thus other executions might trigger other violations. The second shortcoming is due to the fact that false positives are identified only to a certain extent. In fact, some situations are impossible to assess even after conducting the additional reordering as described in Section 5.4.4. Fig. 5.11 represents an example that illustrates a situation where applying both reordering schemes (resulting in two different sequences of events) is not enough to decide whether a problem may or may not take place. Last but not least, we cannot tell if the absence of violation is due to the good programming practice of the developer, who inserts timely invalidations, or is the consequence of previous events.

5.8 Conclusions

To help developers efficiently use emerging cache non-coherent architectures, we have proposed in this chapter a method that analyses traces to pinpoint potential coherence problems. As it takes into account only the cache and memory related events, this method permits checking independently the software cache coherence protocol. It is also non-intrusive, as we capture the traces using a virtual platform, thus no source code modification is necessary. We applied our method to different parallel programs assuming coherent shared memory to identify violations. We correctly identified the events that caused violations and could correct them using appropriate cache commands.

Chapter 6: Deterministic Reverse Debugger based on Traces

Contents

The debugging process is particularly tedious as it involves analyzing parallel execution flows. Executing a program many times is an integral part of the process in conventional debugging, but the non-determinism due to parallel execution often leads to different execution paths and different behaviors. This chapter presents an approach based on simulation, as it is nowadays an integral part of the MPSoC design flow, to ease pinpointing bugs in a parallel execution. To that aim, collecting traces using a virtual platform, in the form presented in Chapter 4, and when an execution fails, re-execute deterministically the program based on such traces, in either forward or reverse direction.

Section 6.1 focuses on the objectives this contribution aims at. Section 6.2 details a strategy for providing forward and reverse execution features to avoid long simulation times during a debugging session. After that, Section 6.3 discusses the initialization and execution of previously mentioned concepts. Then, Section 6.4 shows the architecture of the reverse debugger. The experimental results are presented in Section 6.5. Finally, in Section 6.6, we draw some conclusions.

6.1 Objectives

We propose a method improving the MPSoC debugging process that covers the following objectives:

- *Deterministic Replay*: Our first objective aims at reproducing a deterministic execution based on "well formed" traces. There are two main advantages in reaching this objective. First, our proposed "well formed" execution traces can be exploited to produce a deterministic execution as they provide relations among events that allow the reconstruction of the parallelism. Second, this technique allows the reproducibility of a parallel execution preserving the same behavior of that in simulation, which permits engineers to debug the executions that enclose errors. For example, it is very useful for resolving intermittent bugs because they are often found in parallel software due to concurrence for shared resources and are difficult to reproduce. Thus, replaying deterministically an erroneous execution eases the identification of these bugs and consequently their elimination.

- *Reverse Execution*: Our second objective aims at providing an efficient deterministic reverse debugging method based on "well formed" traces. The reverse execution consists of undoing computations that are performed by a certain instruction, which means recovering previous system state. For example, an engineer configures a breakpoint in the code, thus he executes the code up to this point but he realizes that the good location to insert the breakpoint was already executed. In conventional debugging environments, he has to restart the application, configures the breakpoint to this new location and reexecutes the application until reaching it. Even considering a deterministic replay this process is necessary. The reverse execution help decrease the number of restarts. For this example, backtracking operations until reaching the second breakpoint avoids application restarts. Backtracking an instruction, must be a process simpler than actually executing an instruction during simulation. However, it is not so obvious as it seems to be, because the execution of a program modifies memory locations with values present in the instruction itself, destroying previous memory values that were located there before. Thus reversing an instruction consists of recovering states of memory and/or registers data that can be hard to restore. Maintaining the time complexity as low as possible is part of this objective.

- *Debugger Integration*: The third objective aims at easing the method dissemination. Constructing a debugger from scratch is not desirable for many reasons. First and more relevant is the effort to deal with binary code. The effort of such a development can be considerable, and there is no need to reinvent the wheel. Using an open source, broadly adopted, debugger can ease the development and the adoption of the tools.

Based on these objectives we propose the following contributions:

- Algorithms that allow system replay based on execution traces captured using virtual platforms. These algorithms propose a method for creating an execution graph that represents the behavior of the system during the software execution over a virtual hardware platform.

- Algorithms that perform the walk operations on the execution graph allowing forward and backward operations in linear time complexity. The walking forward operation consists of modifying values based on the instructions without actually executing them. The walking reverse operation consists of recovering values modified by an instruction, *i.e* as they were before the instruction execution.
- A tool that wraps the implementation of these algorithms in *gdb*, which is an open source broadly adopted debugger.

In the next sections, we formalize the algorithms that create the execution graph and perform walk operations on this graph. We also present an overview of our proposed implementation and how it can be used during debugging sessions.

6.2 Execution Traces and Execution Graph

The "well formed" trace is defined in Chapter 4 as $T = (E, <, \leftharpoonup, \prec)$, where E is the set of events, $<$ is the strict component order, $\leftharpoonup$ is the causality chain and $\prec$ is the total system order. We show that, in practice, the trace as previously defined contains enough information to reconstruct the system states exactly as it had been, *e.g.* deterministically. We add to the trace, for replay performance optimization purposes, two new relations to simplify its traversal in both forward and reverse directions.

For debugging purposes the events produced by caches are not taken into account for the execution graph construction. However, they are used for linking the instruction events to memory access through $\leftharpoonup$ relation. Most of commercial and academic debuggers do not maintain cache states. Indeed, caches are flushed or cleaned when a breakpoint is triggered. The caches events are explored in detail in Chapter 5 during analysis of cache coherence using traces.

6.2.1 Forward and Reverse Relations

For simpler trace re-execution, all events are serialized. For that purpose, the forward relation, noted $e_i \curvearrowright e_j$, is built as a total order in E extending $<$. However, all couples of events (e_i, e_j) are not comparable in E with the relation $<$. In these cases, having $e_i \curvearrowright e_j$ or $e_j \curvearrowright e_i$ is equivalent in terms of system state and thus deterministic. In that case, the total order is built using arbitrarily one of these two possibilities. The reverse relation, noted $\curvearrowleft$ is defined as $e_i \curvearrowleft e_j \Leftrightarrow e_i \curvearrowright e_j$.

the The relations $\curvearrowright$ and $\curvearrowleft$ are slightly different from the relation $\prec$. In the former relations, the events are comparable independently of the order of other components. However, using the latter relation, all events that share at least one component are comparable. Therefore, in $\prec$ relation, a shared component is necessary to construct the relation, on the other hand, such requirement is not necessary to construct the relations $\curvearrowright$ and $\curvearrowleft$. The relation $\prec$ is not applied for execution graph creation, relying only on $\curvearrowright$ and $\curvearrowleft$ relations.

6.2.2 Processor and Memory States

The processor state maintains the value of all its internal register states and the information about the instruction currently being executed. More formally, a processor $p \in C$ is

a tuple $p = (\texttt{ce}, \{r_0, r_1, \ldots, r_{k-1}\})$ in which `ce` represents the current instruction and the r_i the values of the k processor architectural registers.

Memories also have an internal state. However, while processors have only few registers, memories have a huge amount of internal values. During software execution, the memory is accessed sequentially and sparsely, so tracking all addresses at once is not necessary. Formally, the memory $m \in C$, in its initial state, is defined by the empty set $m = \varnothing$. It is filled with elements during initialization and while execution progresses. The whole memory may be accessed, in which case $m = \{a_0, a_1, \ldots, a_{l-1}\}$, where l is the memory size. Each element $a \in m$ contains the value at the given address.

The caches are transparent for debugging process, thus their events are not treated.

6.2.3 Execution Graph

We create a directed graph $G(V, A_f \cup A_r)$ for execution. The vertices in V are instances of processor's registers and memory internal elements modifications, denoted v_p and v_m respectively.

The processor vertex v_p is defined as the tuple $v_p = (p_{id}, R)$, where p_{id} indicates the processor it belongs to and R is the set of modified registers, $\forall r \in R \mid r = (id, d, wb)$. The first two elements are intuitively the register identification and its current value. The last one allows representing the *write-before* relation. It is a relation between two write events on the same register such that e_a is the most recent write before e_b, noted $e_a \nsim e_b$. The element wb in r aims to represent this relation, it is thus a reference to the r_{wb} tuple such that $r_{wb} \nsim r$. This relation is a shortcut to access the last modification of a given register. It is mightily used for reverse execution.

For memory vertices, $v_m = (m_{id}, \mathsf{ref}, \mathsf{addr}, d, wb)$, where m_{id} is the memory identifier, `ref` is the element that maintains the relation $\texttt{ref} \leftharpoondown v_m$, which is the causality link to processor vertex v_p that was causing the access, `addr` is the address, d the modified value, and wb has the same meaning as before.

The set A_f contains the arcs representing the $\curvearrowright$ relation. $\forall \alpha \in A_f, (v_i, v_j) \in V \times V \mid \alpha = (v_i, v_j)$, where v_j is the immediate successor of v_i in G. In other words, v_i is the $tail(\alpha)$ and v_j is the $head(\alpha)$. The last vertex reached in G through the arcs in A_f is noted $\top$ and the first one $\bot$. Conversely, the set A_r contains the arcs representing the $v_i \curvearrowleft v_j$ relation, which links a vertex v_j to its immediate predecessor v_i. For this set, $\top$ is the first vertex and $\bot$ the last one.

To exemplify both vertex definitions, let us observe Fig. 6.1 which shows the processor vertices, v_p, on the left and the memory vertices, v_m, on the right, and their possible edges. As explained, both vertex types can be linked through arcs, which permits constructing the execution graph. In this representation, the reverse and forward arcs are denoted respectively α_r (dashed-red) and α_f (blue line). Let us suppose an execution in which v_m had happened after v_p, then there exist arcs $\alpha_f \in A_f \mid \alpha_f = (v_p, v_m)$ and $\alpha_r \in A_r \mid \alpha_r = (v_m, v_p)$. Following these arcs makes it possible to walk ahead and backward, either updating or rolling back operations. To that aim, the data information must be recovered in the current vertex, which requires that each kind of vertex must have the necessary means to recover it. These means are represented by elements r_i for a v_p vertex and by v_m itself for memory accesses. To recover past values quickly, the arcs wb in v_m and r_i have an important role. They link v_m and r_i to the vertex/element that has the previous value of their data. Thus, the rollback operations are performed in

constant time, acting as a shortcut to those values. Besides that, each v_m vertex has an arc pointing to the v_p vertex which caused it. In other words, they preserve the relation $v_p \leftarrowtail v_m$ given by the trace. In Fig. 6.1, it is represented by the dotted-green line labeled *ref*.

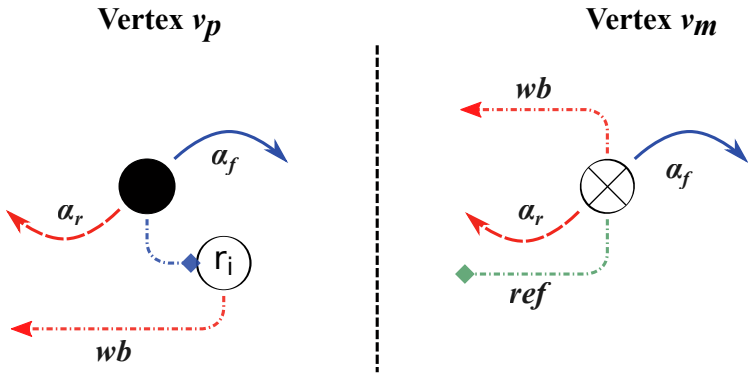


Figure 6.1: Graph elements of vertices v_p and v_m and arcs α_f and α_r in an execution graph.

Component states, current vertex and current processor are maintained in an entity we call *Oracle*, $O = (v_{cur}, p_{cur}, \{p_0, p_1, \ldots, p_n\}, \{m_0, m_1, \ldots, m_k\})$. To simplify notation, we name M and P the set of memories and the set of processors present in O. The currently executed instruction `ce` of processor's states is a vertex of the execution graph.

The instruction and data caches events, although not directly used to update memory or processor state, are used to maintain the causality links.

6.2.4 Practical Example

Creating an execution graph is the process of reading the traces and updating component tuples, consequently the Oracle. To exemplify this process, Fig. 6.2 is used as input trace T. This trace is the same exposed to introduce trace concept in Chapter 4. Therefore, the trace is defined as $T = (E, <, \leftarrowtail, \prec)$, in detail:

- Events: $E = \{e_1, e_2, \ldots, e_{11}\}$;
- Component Strict Total Order ($<$): $<_{CPU_0} = \{(e_1, e_7)\}$; $<_{CPU_1} = \{(e_4, e_9)\}$; $<_{DCache_0} = \{(e_2, e_8)\}$; $<_{DCache_1} = \{(e_5, e_{10})\}$; and $<_{MEM} = \{(e_6, e_3), (e_3, e_{11})\}$;
- Causality ($\leftarrowtail$): $\leftarrowtail_1 = \{(e_1, e_2), (e_2, e_3)\}$, $\leftarrowtail_2 = \{(e_4, e_5), (e_5, e_6)\}$, $\leftarrowtail_3 = \{(e_7, e_8)\}$; and $\leftarrowtail_4 = \{(e_9, e_{10}), (e_{10}, e_{11})\}$; and
- Total System Order ($\prec$): $\varnothing$.

Component Strict Total Order

The events e_4 and e_7 are load instructions, and e_1 and e_9 are stores. For didactic reasons, we consider that all accesses to memory are done at the same address. As discussed before, just the store instructions create a memory node and eventually create register updates, for instance the processor's PC. The load instruction updates the processor's registers, including the PC.

Interruptions are not treated as special events, they are seen through an instruction event that contains the implicitly modified registers. For example, in ARM architectures, when an interruption occurs a branch to the interrupt vector is triggered. This event appears as an instruction that contains some modified registers which include the PC and the status register.

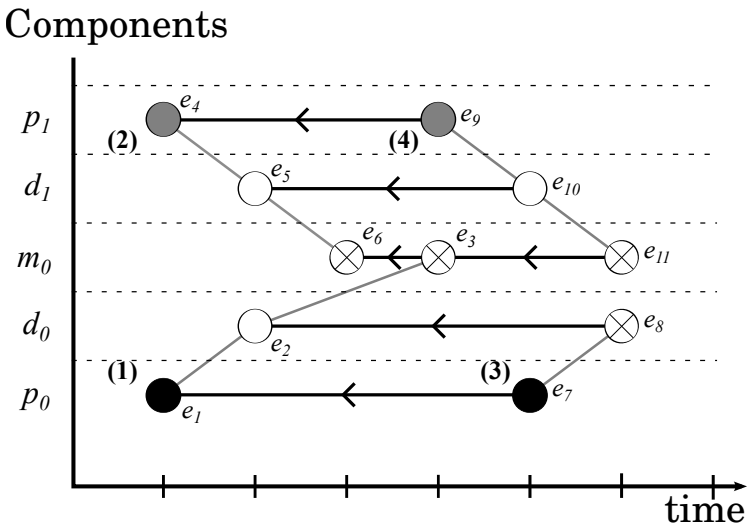


Figure 6.2: Trace example.

To maintain the intuitive nomenclature, each created vertex carries the equivalent event's number. For example, the event e_3 generated the vertex v_3. Based on the example trace exposed in Fig. 6.2, the execution graph is expressed in terms of vertices and arcs as follows:

- The vertices: $V = \{v_1, v_3, v_4, v_6, v_7, v_9, v_{11}\}$;
- The forward flow: $(\bot) \curvearrowright v_1 \curvearrowright v_4 \curvearrowright v_6 \curvearrowright v_3 \curvearrowright v_9 \curvearrowright v_7 \curvearrowright v_{11} \curvearrowright (\top)$; and
- The reverse flow: $(\bot) \curvearrowleft v_1 \curvearrowleft v_4 \curvearrowleft v_6 \curvearrowleft v_3 \curvearrowleft v_9 \curvearrowleft v_7 \curvearrowleft v_{11} \curvearrowright (\top)$.

The resulting sets of arcs are:

- $A_f = \{(\bot, v_1), (v_1, v_4), (v_4, v_6), (v_6, v_3), (v_3, v_9), (v_9, v_7), (v_7, v_{11}), (v_{11}, \top)\}$; and
- $A_r = \{(\top, v_{11}), (v_{11}, v_7), (v_7, v_9), (v_9, v_3), (v_3, v_6), (v_6, v_4), (v_4, v_1), (v_1, \bot)\}$.

Other relations are also created to allow the reverse execution. The causality links are shrunk and represented by $v_1 \leftharpoondown v_3$ and $v_9 \leftharpoondown v_{11}$ and *write-before* relation by $v_1.r_{pc} \leftsquigarrow v_7.r_{pc}$ and $v_4.r_{pc} \leftsquigarrow v_9.r_{pc}$, where r_{pc} is the program counter. The filled blue arrows represent $\curvearrowright$ and dashed red arrows $\curvearrowleft$. Other relations are represented also, $\leftsquigarrow$ is represented by dotted red lines and the relation $\leftharpoondown$ is represented by a dotted green line labeled *ref*. The v_p's register elements are linked to their v_p by dotted blue lines labeled r_i. In this example, we observe that event e_1 modifies the program counter (PC) which is attached to the PC element (r_{pc}) through a r_i link. It also causes a memory access represented by v_3. The same construction holds for v_9 and v_{11}. However, in the latter case, the *wb* link is created due to a previous modification of the processor's PC.

The order of vertices v_1 and v_4 are governed by relation $\curvearrowright$. In this case the v_1 appears before v_4, however, reverting this order does not change the state of the system. As these events modify just the internal processor's register the state of the system is not modified, even if these instructions are "store". In this case, the result of store operations are represented by memory vertices which have to have their order strictly respected. For instance, inverting the order of $v_1 1$ and v_3 may cause false interpretation of the system.

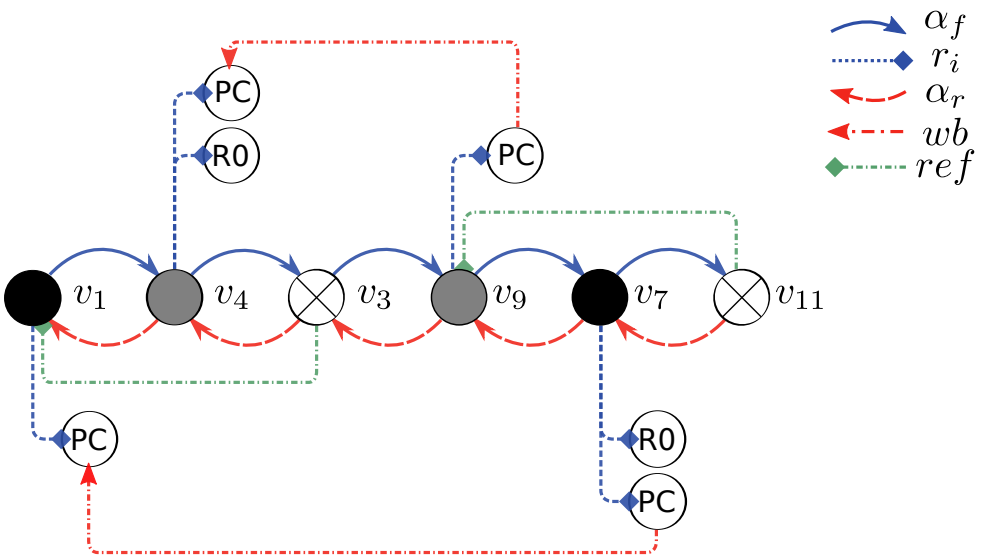


Figure 6.3: Example of execution graph G based on Fig.6.2.

The current state of the system is available by Oracle O.

Considering the execution graph exposed in Fig. 6.3, the Oracle O is filled as follows: $O = (v_{11}, \{p_0, p_1\}, \{m_0\})$; $p_0 = (v_7, \{r0_2, \texttt{pc}_1\})$; $p_1 = (v_9, \{r0_1, \texttt{pc}_3\})$; and $m_0 = \{v_{11}\}$.

The memory value corresponds just to the last store because all previous accesses modify the same address, then the last one is used to represent the current state in Oracle. In the example case, vertex v_{11} has the current state of target address.

6.3 Forward and Reverse Execution

This section presents the algorithms used to perform the conventional execution, *i.e* in forward way, and reverse execution. It starts presenting execution graph and Oracle's initialization, detailing the start values of each tuple internals.

6.3.1 Initialization

The initialization process, which sets up the initial internal resource values, guarantees that trace-based execution always starts with the same initial state of simulation that generates the traces. This information is not available in the traces, and it must be done before the first event is being read.

For processors, the internal registers reset values can be easily obtained in datasheets provided by the manufacturer. The processors' states, $\forall p \in C$, are initialized with them. Considering the memories, they may contain pre-initialized values at certain addresses,

such as static constants or binary code. The non-initialized values are not taken into account during initialization, assuming zero as their initial value both in the simulator and in the debugging tool for determinism. Initialization made by the software execution, such as static and global variables initializations, are explicitly present in the traces, thus taken into account during the next phases. From an implementation point of view, we use a judy array [81] to ensure fast access to the used values. Starting from an empty array, the entries are allocated progressively as addresses are written for the first time.

Formally, we define $r^{\perp}$ and $a^{\perp}$ as the initial elements of register and address state, respectively. Given $\forall p, m \in C$, we define $p_{\perp} = (\varnothing, \{r_0^{\perp}, r_1^{\perp}, \ldots, r_{n-1}^{\perp}\})$, where $r_x^{\perp} = (x, d, \varnothing)$, as the initial processor state, and $m_{\perp} = \{a_i^{\perp}, a_j^{\perp}, a_k^{\perp}\}$, where i, j, k are initialized addresses, as the initial memory state, where $a_i^{\perp}.wb = a_j^{\perp}.wb = a_k^{\perp}.wb = \varnothing$. For memory, all addresses absent from $m_{\perp}$ have implicit zero value. As observed, $p.\texttt{ce} = \varnothing$, because there is no current instruction to execute, and $\forall r^{\perp} \in p_{\perp}$ and $\forall a^{\perp} \in m_{\perp}$ maintains the relation $\varnothing \rightsquigarrow r^{\perp}$ and $\varnothing \rightsquigarrow a^{\perp}$, because there is no previous value in registers or memory.

As example of processor's initialization, we use the ARMv7 architecture. The ARM core provides general-purpose registers (r_0–r_{12}), the stack pointer (SP) r_{13}, the link register (LR) r_{14} which holds return addresses on function calls, the program counter (PC) r_{15}, and a Program Status Register (PSR). Shadow registers, available in various operating modes, are similar to register banks and reduce interrupt latency. For simplification the shadow registers are not represented in this example. All registers in ARM architecture initiates zeroed, except the PSR register which has `0x400001D3`. In this example, the $p = (\varnothing, \{r_0, \ldots, r_{15}, r_{\texttt{PSR}}\})$, we have $r_{0-15} = (0, 0, \varnothing)$ and $r_{\texttt{PSR}} = (16, \texttt{0x400001D3}, \varnothing)$, 16 being the identification number of PSR register.

The binary code and configuration registers addressed in memory have their initial values transfered to memory.

6.3.2 Execution

Execution based on the execution graph can be accomplished in both forward and reverse ways. Each event in the trace can be seen as a delta compared to the previous system state, so there is no need to actually simulate anything, as consistently applying the deltas is enough to determine the state of the system.

6.3.2.1 Forward Execution

While forward executing, if we reach $\top$, we build on the fly the next part of the execution graph from the traces, otherwise execution is directly performed on the graph. Execution from the graph proceeds as follows: instruction vertices update the processors states while memory acknowledgement vertices update the memories ones.

Algorithm 3 describes the on-the-fly building phase of the execution graph. In this algorithm and the following ones, we will use a programming language notation for the tuples. For example, $e.k$ at line 1 of Algorithm 3 refers to member k of the event tuple e. Algorithm 3 basically shows how the information gathered in the events is used to create and update the vertices of the execution graph. The algorithm shows neither components nor oracle updates for simplification. At lines 2 and 11, the vertices are created with the values coming from the event. At line 2, a v_p is created with its processor ID and event ID. The v_m is instantiated with its memory ID ($e.d.\texttt{mem}_{id}$) and address ($e.d.\texttt{addr}$) at line

Algorithm 3 Execution graph creation

Require: event e and (processor $O.p_i$ or memory $O.m_i$)
Ensure: $i = e.c$

1: **if** $e.k =$ "`instruction`" **then** → *Processor Vertex*
2: $v \coloneqq \textsc{Create}(v_p, e.c, e.id)$
3: **for all** `reg` $\in e.d.D_p$ **do**
4: $j \coloneqq$ `reg`$.id$ → *index j used at the next three lines*
5: $v.r_j.d \coloneqq$ `reg.value`
6: $v.r_j.wb \coloneqq p_i.r_j$ → *Register assignment to $\rightsquigarrow$ relation*
7: $p_i.r_j \coloneqq v.r_j$
8: **end for**
9: **else if** $e.k =$ *"ack"* **and** $e.c \in O.M$ **then** → *Memory Vertex*
10: $j \coloneqq e.d.$`addr` → *index j used at lines 14 and 15*
11: $v \coloneqq \textsc{Create}(v_m, \texttt{mem}_{id}, e.d.\texttt{addr})$
12: $v.d \coloneqq e.d.$`value`
13: $v.\texttt{ref} \coloneqq \{v_{\texttt{ref}} | (v_{\texttt{ref}}.id = e_{\texttt{ref}}.id) \wedge (e_{\texttt{ref}} \leftharpoonup e)\}$
14: $v.wb \coloneqq m_i.a_k$ → *Vertex assignment to $\rightsquigarrow$ relation*
15: $m_i.a_k \coloneqq v$
16: **end if**
17: $V \coloneqq V \cup v$
18: $\alpha_a \coloneqq \{\alpha \in A_f \mid tail(\alpha) = \top\}$; $\alpha_b \coloneqq (head(\alpha_a), v)$; $\alpha_c \coloneqq (v, \top)$
19: $A_f \coloneqq (A_f \setminus \{\alpha_a\}) \cup \{\alpha_b, \alpha_c\}$
20: $\alpha_d \coloneqq \{\alpha \in A_r \mid head(\alpha) = \top\}$; $\alpha_e \coloneqq (v, tail(\alpha_a))$; $\alpha_f \coloneqq (\top, v)$
21: $A_r \coloneqq A_r \setminus \{\alpha_d\}) \cup \{\alpha_e, \alpha_f\}$

11. Concerning data and other relations, the `for` loop at line 3 stores the data value and creates the *write-before* relation for each register present in the instruction event. The causality relation present in memory events is transmitted to memory vertex through line 13. Finally, the last algorithm portion (lines 17-21) updates graph elements V, A_f and A_r.

Execution graph creation traverses all events and sequentially creates nodes and arcs, without any later traversal of these elements, thus Algorithm 3 time complexity is $O(n)$, where n is the number of input events.

Algorithm 4 is a simplified view of how forward execution is done. If the next vertex is a processor vertex, then once the processor identified (line 3), line 7 updates its state using the information available in v_p. Identically, the memory state is updated (line 12) using v_m once the memory identified (line 10). As can be seen, there is strictly no computation involved, only the retrieval of the correct values associated to the vertex of the execution graph. As the vertices are traversed one after the other using the forward arcs, it is straightforward to see that it has $O(n)$ time complexity.

6.3.2.2 Reverse Execution

Given a system state and the current executed operation, reverse execution consists of computing the system state as it was before the operation execution. In practice, this consists of reverting the deltas applied in forward execution.

Reverse execution starts from $v \in V$ and walks G following the arcs A_r, restoring the previous system state using the values available in each processed vertex. It stops either when it reaches the last vertex $\perp$, leaving the system in its initial state, or when the user

Algorithm 4 Forward walk in execution graph

Require: Graph $G = (V, A_f)$ and Oracle O
Ensure: $\{\forall v_p, v_m \in V \times V \mid v_p \neq v_m.\texttt{ref}\}$
1: $v \coloneqq O.v_{\texttt{cur}}$
2: **if** $v = v_p$ **then** → ***Processor Vertex***
3: $\quad i \coloneqq v.p_{id}$ → ***index i used at lines 4 and 7***
4: $\quad O.p_{\texttt{cur}} \coloneqq O.p_i$
5: $\quad$ **for all** $\texttt{reg} \in v.R$ **do** → ***Update the registers***
6: $\quad\quad k \coloneqq \texttt{reg}.id$
7: $\quad\quad O.p_i.r_k \coloneqq \texttt{reg}$
8: $\quad$ **end for**
9: **else if** $v = v_m$ **then** → ***Memory Vertex***
10: $\quad i \coloneqq v.m_{id}$ → ***indexes i and j used at lines 12***
11: $\quad j \coloneqq v.\mathsf{addr}$
12: $\quad O.m_i.a_j \coloneqq v$ → ***Update the address***
13: **end if**
14: $\alpha_f \coloneqq \{\alpha \in A_f \mid tail(\alpha) = v\}$ → ***Step Forward Oracle***
15: $O.v_{\texttt{cur}} \coloneqq head(\alpha_f)$

asks to do so using debug facilities (breakpoints, watchpoints, reverse-step and so on).

Reverse execution updates the Oracle and components using information available in G. The reverse operations revert completely the instructions, considering also their memory accesses and any kind of parallel operations that had occurred over other processors. Formally, given a current processor $O.p_{\texttt{cur}}$ and current vertex $O.v_{\texttt{cur}}$, reverting an instruction consists of walking the G graph until reaching $v_p \in V \mid v_p.p_{id} = p_{\texttt{cur}}$, then updating $O.p_{\texttt{cur}} = v_p$. Every vertex operation present in the path between these vertices is recovered, thus reverting each operation done by those vertices, updating component states. Undoing one write operation using $\curvearrowleft$ implies to traverse potentially lots of events. It is accelerated with $\leftsquigarrow$, which recovers the previous value in $O(1)$ time complexity.

Algorithm 5 Reverse walk in execution graph

Require: Graph $G = (V, A_r)$ and Oracle O
Ensure: $\{\forall v_p, v_m \in V \times V \mid v_p \neq v_m.\mathsf{ref}\}$
1: $v \coloneqq O.v_{\mathsf{cur}}$
2: **if** $v = v_p$ **then** → ***Processor Vertex***
3: $\quad i \coloneqq v.p_{id}$ → ***index i used at lines 4 and 7***
4: $\quad O.p_{cur} \coloneqq O.p_i$
5: $\quad$ **for all** $\mathsf{reg} \in v.R$ **do** → ***Rollback the registers***
6: $\quad\quad k \coloneqq \mathsf{reg}.id$
7: $\quad\quad O.p_i.r_k \coloneqq \mathsf{reg}.wb$
8: $\quad$ **end for**
9: **else if** $v = v_m$ **then** → ***Memory Vertex***
10: $\quad i \coloneqq v.m_{id}$ → ***indexes i and j used at line 12***
11: $\quad j \coloneqq v.\mathsf{addr}$
12: $\quad O.m_i.a_j \coloneqq v.wb$ → ***Rollback the address***
13: **end if**
14: $\alpha_r \coloneqq \{\alpha \in A_r \mid tail(\alpha) = v\}$ → ***Step Back Oracle***
15: $O.v_{\mathsf{cur}} \coloneqq head(\alpha_r)$

The simplified algorithm to reverse update processor and memory states is presented in Algorithm 5. It is very similar to forward execution, the differences being that the state values are updated through the *write-before* links, and that the vertex to execute next is the previous one. The last assignments can be observed at lines 14 and 15. Also, note that this algorithm traverses A_r arcs instead of A_f ones. Therefore, the Oracle's element assignments are done using the *wb* element present either in register element or memory vertex, as observed at lines 7 and 12.

As for Algorithm 4, at most each node is traversed back once during reverse execution, thus the asymptotic complexity of Algorithm 5 is $O(n)$.

6.4 Trace Based Debugger Architecture

The present method to create and manipulate the execution graph to obtain forward and reverse executions is implemented in the architecture present in Fig. 6.4.

We call the software that reads and maintains the oracle update as Sherlock. This architecture is extensible by plugin which can extract desirable information from Oracle states or traces. For implementation of deterministic execution based on trace, the Delorean plugin implements the Graph Handler which creates and walks in the graph. This plugin interacts with Oracle Handler to update the processors and memory states. The Oracle Handler is accessed by GDB server to provide information to GDB client, such as processor's register values. The Sherlock implementation is available for general purpose Linux implementations, it does not mean that the debugging process is made for Linux platforms. The debugger uses the captured traces that was generated during simulation. These traces produce the behavior of the target platform, not the one where Sherlock is executed.

On the left hand side, the GDB client for a specific target platform is used without modification, since the reverse command support was introduced in version 7.0. Obviously that reverse command support can vary from one target to another. The sherlock tool implements its own support as described in previous sections. The GDB client uses

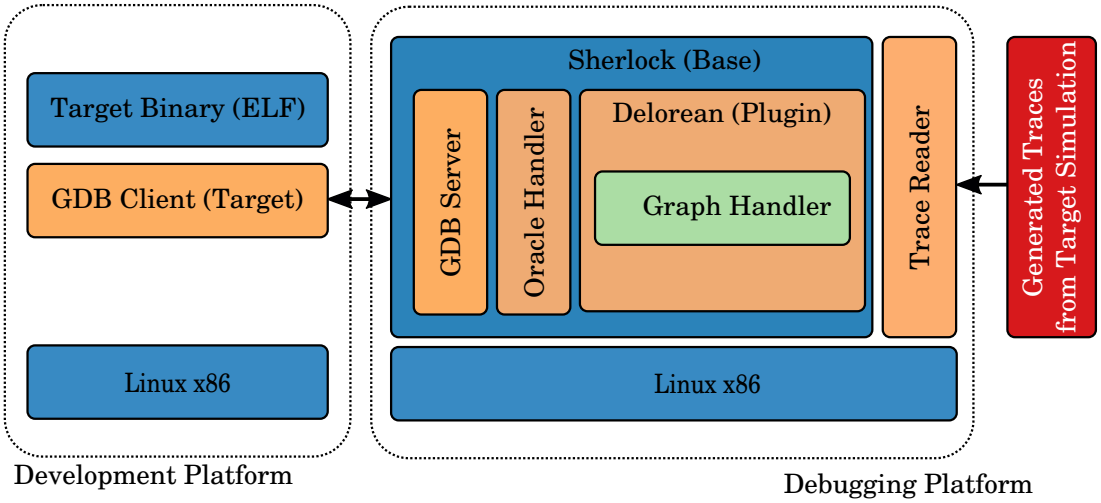


Figure 6.4: Sherlock Tool and Delorean Plugin Architectures.

the target binary code to traverse the source code, ease the processes to configure a breackpoint or a watchpoint. Besides reverse support, the useful step operations are supported.

6.5 Experimentation and Results

We have implemented a tool to replay and backtrack applications deterministically using the traces gathered using MPSoC virtual platforms. This section presents some experiments we did and the results we obtained. Our experiments aim to verify the correctness of the approach and compare its execution speed to a simulation. The questions we want to answer are: (1) how much time the developer wastes when launching a new execution? (2) how much time costs reverse execution in comparison to a new simulation? The gain of our approach in an actual debugging session would be the metric of interest, but as this heavily depends on the bug and the debugging skills of the developers, we cannot provide a fair evaluation for it.

6.5.1 Hardware and Software Platform

We use our virtual platform environment in which we vary the number of processors and produce the corresponding traces. Regarding software, we rely on the GNU cross-development environment, so we support C and C++ through *gcc* and conventional remote debugging, *i.e.* without non-intrusive built-in multiprocessor reverse capabilities that this work adds, using *gdb*.

For our experiments, we use $n = \{1, 2, 4, 8, 12\}$ ARM Cortex-A9 processors models supporting a shared-memory architecture interconnected by a network on chip, which interconnects also timers, a framebuffer, a storage device and serial interfaces. An example of this platform is shown in Fig. 4.10.

The set of software applications we use to evaluate our approach is composed of a multi-threaded Motion-JPEG decoder and a set of parallel programs present in Splash-2 benchmark, namely Ocean, Water-nsquared and Water-spatial.

6.5.2 Tool overview

We implemented the algorithms of Section 6.2 under the form of a *gdb* server. At client side, *gdb* offers a set of commands that performs reverse operation. The main ones are `reverse-continue`, `reverse-next` and `reverse-step`. They trigger the reverse traversal and behave otherwise as their forward counterparts. Our tool support these commands at *gdb* server side sending the correct answer to each command, but the internal behavior has nothing in common with a standard *gdb* server.

We tested the tool implementation to verify that it executes deterministically the traces as follows. We firstly instrument the simulation models so that they produce register dumps after each instruction and dump the value associated to each memory access, thus producing a log of the system. We also instructed our tool to dump the registers, respectively the memory access, when a vertex v_p, respectively v_m, is created. We executed the simulation and obtained both the trace and the log. We ran our tool using the traces and generated our own log, and then compared both logs, which ended up being identical.

6.5.3 Performance - Reverse execution

To proceed with this experiment, a set of applications are used including a Parallel Motion JPEG decoder and a sub-set of applications present in the Splash-2 benchmark. Virtual platforms with $n = \{1, 2, 4, 8, 12\}$ processors are used. To normalize the results, we use as reference a *base* platform, which is the platform without trace generation. The number of target simulated clocks and the host simulation time for the *base* platforms are presented in Fig. 6.5.

We compare the performance of deterministic trace based execution with reverse capabilities as implemented in our tool to the one achieved by simulation without trace generation. We perform the following experiments and measure the time they take to execute the whole applications: raw simulation, simulation with generation of the traces, re-execution of the traces. The re-execution time is split in different phases, following what would occur during a debugging session: execution graph generation, forward and reverse execution on the graph.

For the re-execution, we have the following steps: (1) initialize the debugging session; (2) set breakpoints at start and end of execution flow; (3) forward execution until reaching the end breakpoint; (4) reverse execution until reaching the start breakpoint; and (5) forward execution until reaching the end breakpoint again.

Fig. 6.6 plots execution time ratios, *i.e.* speedups and slowdowns. Both ratios represent respectively the proportional gain and loss in times when compared to the Base platform simulation time, observed in Fig. 6.5. The graph *a* represents the slowdown

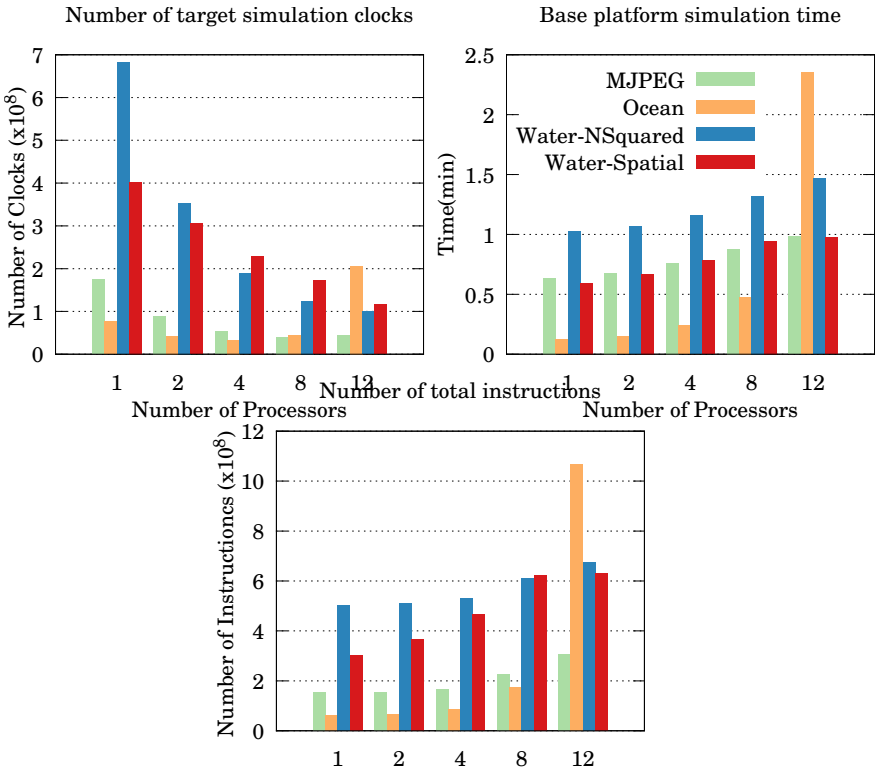


Figure 6.5: Number of target clocks, host simulation time and number of instructions for *base* platforms.

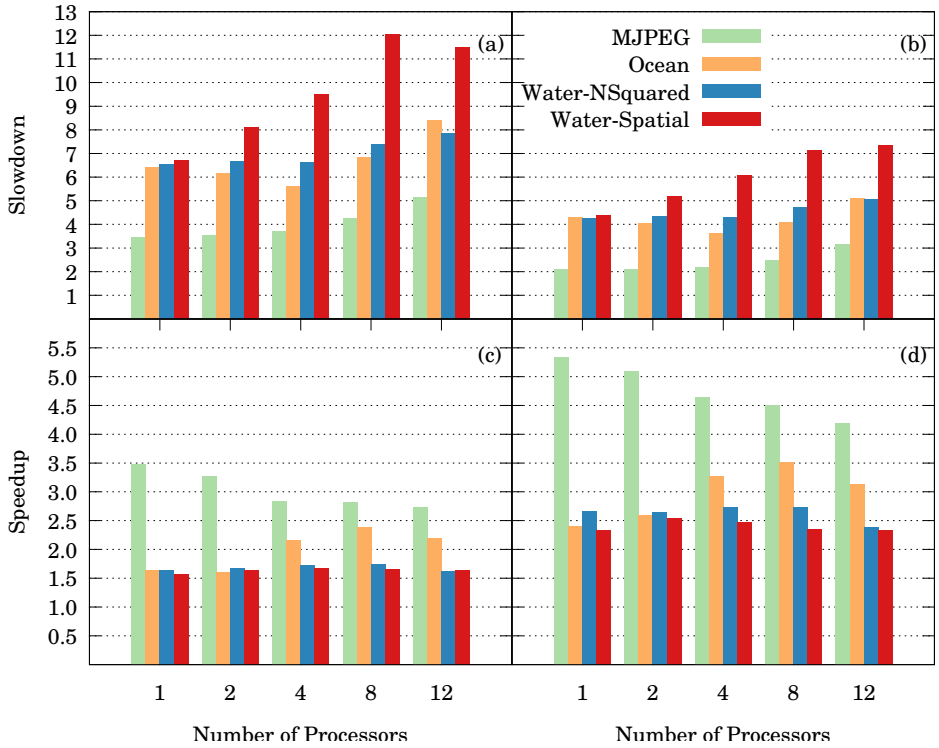


Figure 6.6: Normalized execution time for (*a*) trace generation; (*b*) execution graph creation; (*c*) reverse execution using the execution graph; (*d*) forward execution using the execution graph.

caused by the generation of traces. The graphs *b*,*c*,*d* are obtained during the debug session. Graph *b* represents the slowdown caused by the creation of the execution graph, graph *c* the speedup obtained during reverse execution and graph *d* the speedup obtained by the forward execution, the last two using generated graph. Each graph is grouped by number of processors and each color represents a given application.

To illustrate the potential gain of using this approach, we take the execution of Water-Spatial as an example. Its complete execution takes about 1.5 minutes using 12 processors without trace based debugging support, and we can expect it to be re-executed many times to find the source of a bug. To provide deterministic re-execution support, the same execution takes about 12 minutes, which is clearly much longer. However, the faulty trace is generated only once, and forward and reverse debugging do not need any extra simulations. At worst, the full re-execution from the trace (building the entire execution graph) is 2 to 7 times slower than simulation, but navigating forward and reverse using the graph is 1.5 to 5 times faster.

This is the worst case scenario for our approach. Indeed, first, debugging usually requires only partial execution of the trace, and second, debugging on the simulation platform requires a debugger server that slows down the simulation a lot.

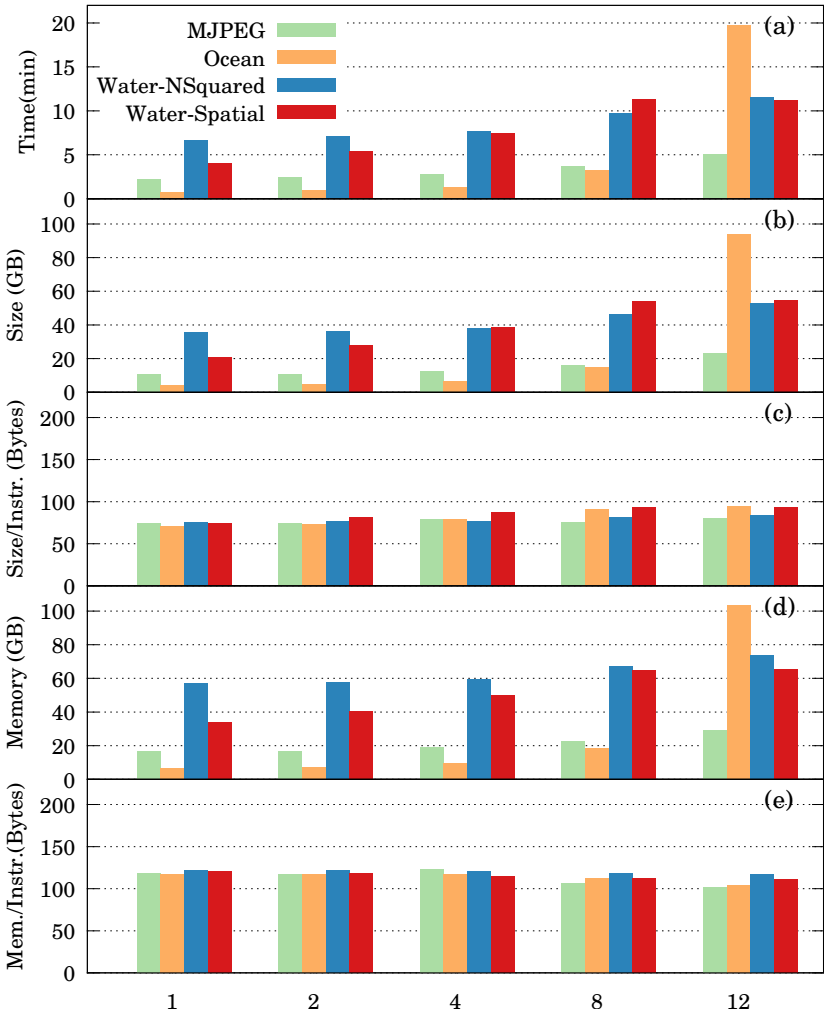


Figure 6.7: *(a)* Time spent to capture traces, *(b)* resources necessary to store traces on disk, *(c)* average disk storage size per instruction, *(d)* creation of the execution graph and *(e)* average memory size per instruction

6.5.4 Resources Consumption

Our experiments generate a huge number of low-level events, consequently disk, memory usage, and the elapsed time have to be evaluated. Fig. 6.7 plots this information. We can see that tracing incurs a non negligible resource consumption. However, we believe this can be amortized thanks to the benefits it brings to the developer.

6.6 Conclusion

Hardware technologies evolve at a dramatic pace. Unfortunately, well accepted techniques to rapidly produce and deploy efficient bug-free software are lacking, thus debug is still a challenging topic in both academia and industry. Advanced industrial works have trust

in reverse execution or virtual platforms to deal with it, *e.g.* startups like Undo [76] for the former and well established companies like Synopsys [82] for the latter. We believe that merging both visions might produce powerful tools.

Indeed debugging is an iterative process, and when parallel programs are executed on parallel hardware, reproducing the conditions of occurrence of a bug is a major issue.

In this chapter, we defined a deterministic forward and reverse execution strategy for debug based on execution traces captured using an MPSoC virtual platform.Execution is supported by the definition of an omniscient entity called *Oracle*, which might be extended to be used for system analysis as well. We implemented the approach and encapsulated it within a *gdb* server, providing the developer with an extension of its usual debugging environment.

Despite its advantages, our method has two main drawbacks. First and foremost, the cost of disk and memory space is not negligible and could impact its applicability. Some propositions could be done to amortize them, but it is still our Achilles' heel. Another limitation is that, unlike conventional debugging, it is not possible to perform on-the-fly value modification. Indeed, our approach is completely based on traces and does not process any instruction, thus manual memory and register modifications during debugging sessions are not feasible.

Chapter 7: Conclusions and Perspectives

Contents

7.1 Conclusions

The global objective of this thesis is to highlight the importance of traces in the development of novel hardware/software systems through methods to capture, analyse and debug these systems. To that aim this work covers the challenges discussed in Chapter 2. We listed there three main questions.

The first question focuses on how execution traces can successfully capture the parallelism present in MPSoCs.

- **Question 1: What is the traces' definition that allows capturing the parallel behavior of applications running in simulated MPSoC platforms?**

The solution presented in Chapter 4 introduces the trace's formalism. Using a simulator allows capturing complex relations which we express as our formal definition of "well formed" traces. We defined relations based on the order of events that appear in each component. The most interesting outcome is that the "well formed" traces contain causality relations, which opens opportunities to reduce the theoretical complexity, and thus increase the practical interest, of many algorithms aiming at analyzing software and hardware properties.

We propose a method to produce traces within simulation environments making use of DBT. Our trace generation, thanks to the simulation environment, does not rely on intrusive methods, such as source code annotation or any kind of compilation modification, resulting in a non-intrusive method. The main challenges to overcome were firstly to track causality relations and secondly to maintain the order of events. We have proposed to embed information in non-functional micro-operations to solve both challenges, and have experimentally shown, using several examples, that this solution did indeed solve the issues.

These execution traces contain relevant information about the behaviour of the system which can be used to identify problems occurring during system execution. The potential

of applicability of these traces raised two more questions and led us to think about their respective answers.

- **Question 2: How to use traces to identify cache coherence violations efficiently in cache non-coherent architectures?**

The solution presented in Chapter 5 helps engineers to efficiently identify cache coherence violations in the emerging cache non-coherent architectures. This non-intrusive method analyses traces to pinpoint potential coherence problems, as it takes into account only the cache and memory related events. It is, first, software-cache-coherence-protocol independent remaining at low level event analysis and, second, it captures the interconnection behavior implicitly, which avoids the analysis of interconnect events to obtain the correct order of events.

We applied our method to different parallel programs assuming coherent shared memory to identify violations. We correctly identified the events that caused violations and could correct them using appropriate cache commands.

We implemented the method as an online parallel analysis tool that allows concurrent simulation and analysis, which can help decrease the development time. We ported a set of parallel applications, that was conceived to run in cache-coherent architectures, to cache non-coherent architecture platforms. This porting process was made with the help of our method that identifies the points in the source code to be modified, which eased the corrections.

The debugging process is necessary to identify hardware/software interactions problems, like those present in software cache coherence, as well as to guarantee the correct behavior of functional parallel software. At this other abstraction level, non-deterministic executions make the debugging process difficult, which led us to the last question.

- **Question 3: How to use traces to provide a deterministic reverse debugging method that can be applied to MPSoC?**

We propose a solution presented in Chapter 6 that defines a deterministic forward and reverse execution strategy for debug. Our method proposes a deterministic debugger system based on traces, where only one simulation is necessary: the one that produces the traces. The solution relies on an execution graph that is constructed using traces. We propose an execution graph specification that allows fast walk in forward and revere directions. We implemented this approach and encapsulated it within a *gdb* server, providing the engineer an extension of its usual debugging environment.

As demonstrated our results, we can obtain considerable development time gain avoiding re-simulation of the entire platform.

7.2 Perspectives

This thesis can be followed by several interesting research directions in debugging techniques based on detailed execution traces. We classify our future works in two time-based categories: *short-term* and *long-term*.

Short-Term perspectives

These perspectives aim at enhancing the method to cover a wide range of components and architectures that were not analysed during this thesis.

- **Explore other architectures**: During this thesis, we focused on exploring a specific architecture composed by L1 caches and shared memory. However, the number of cache levels and memories can vary. Our traces support this increasing number of components, however, due to simulator limitations, it was not explored during the thesis. Then, another future work aims at analysing the traces generated by other configurations of architectures, such as other cache hierarchies and consistency models, to identify problems and to provide the same features as the ones available thanks to this work.
- **Decrease size of traces**: We observe that the size of traces is one of our Achilles' heel. Compressing traces while capturing allows shrinking them which can be one of the possible solutions. The impact of a compression algorithm in the overall simulation time is one of the points to be carefully analysed.
- **Decrease Capture Time**: Our trace capture implementation works sequentially with simulation, which imposes slowdown when the trace system is active. Even considering this slowdown to be acceptable, we believe that we can decrease it applying parallel methods. The approach can be focused on decoupling simulation from trace generation. After that, the trace capture implementation can be parallelized itself. For example, decoupling the filling of events and their actual storage on disk.

Long-Term perspectives

The long term perspectives are related to improvements that we believe will generalize this work:

- **Profiling tools**: Profiling tools aim at identifying the performance bottlenecks of a given application. As our traces capture the behavior of a parallel software, the performance analysis methods can be proposed to identify performance problems, such as hot-spot, race conditions, congestion, code coverage, and so on so forth.
- **Analysis tools**: Analysis tools aim at providing further information to eliminate problems. Most of the problems are related to shared resources, such as shared memories and caches. For example, race condition is a software level issue that is a common source of problems for developers. We believe that the content of our traces permits to address this problem more efficiently. Furthermore, our cache coherence verification method actually does not target any software cache coherence protocol being focused on analysing memory events. Despite its protocol independence, our method can be used to verify whether a protocol implementation works properly or not.
- **Optimization of Execution Graph**: The deterministic reverse debugging method relies on a costly execution graph to perform the walk operations. The graph actually consume a considerable amount of workstation memory, which can be prohibitive

to deploy this method in ordinary development workstations. To minimize its impact, a solution could be to identify an execution window to create the execution graph. This window helps to maintain just the relevant part of execution graph for debugging. For example, since the boot up of a platform is validated, such portion of the code does not need an execution graph composition, avoiding its placement on memory. Other techniques could mix the usage of memory and the disk to alleviate the memory consumption to store this graph.

- **Complex Operating System and MMU**: General OSs rely on MMU to implement the notion of process. Considering MMU operations is necessary to correctly identify the application that causes a problem or if it comes from the kernel. The consideration of MMU related information have to be part of the traces. As a perspective, we intend to formalize the MMU events, which allows an ease mapping among virtual addresses, physical addresses and current process. This method extends the applicability of our approaches to general purpose operating systems.

Résumé en français

Chapitre 1 - Introduction

Les systèmes embarqués pour les applications haut de gamme utilisent des dizaines sinon des centaines de processeurs aujourd'hui, et la croissance du nombre d'éléments de traitement devrait se poursuivre sans modification. Ceci a été rendu possible grâce à la baisse ininterrompue étonnante de la taille du transistor, ce qui permet aujourd'hui d'intégrer non seulement un grand nombre de processeurs, mais aussi beaucoup, sinon tous les autres composants nécessaires sur une seule puce. Le but d'augmenter les performances, diminuer la consommation d'énergie et les insérer dans les plus petits emballages ont été réalisés grâce à ces systèmes multiprocesseurs sur puce (MPSoC). Ils sont maintenant une solution au multimédia, aux applications mobiles et d'à l'automobile au grand public et sont même utilisés pour les serveurs dans les centres de traitement des données.

Le nombre croissant de processeurs intégrés dans une seule puce oblige à repenser le processus de conception numérique complète, enjambant tous les couches matériels et les logiciels. En ce qui concerne le matériel, un exemple caractéristique est la cohérence du cache, dont la mise en œuvre efficace (en terme de performances et de ressources) à grande échelle est encore à démontrer, mais il est seulement un exemple parmi d'autres. En faisant ces constatations, plusieurs architectures *many-cores* ont été conçus pour éviter ces problèmes d'évolutivité matérielle, ce qui conduit à l'évolution des modèles de programmation. Au niveau logiciel, la programmation d'une puce avec un grand nombre de processeurs n'est pas une tâche facile, et il conduit généralement à un grand nombre, difficile à trouver, des erreurs de programmation. Problèmes de synchronisation, les questions de cohérence, les questions de cohérence, etc, sont des sources d'erreurs en raison de la programmation massivement parallèle.

Objectives

Cette thèse a comme objectif fournir des méthodes à base de traces pour déboguer efficacement les systèmes multiprocesseur sur puce. Comme les méthodes basées sur les traces explorent les informations fournies par des traces, la spécification précise des traces est une tâche d'une grande importance. Tout d'abord, nous cherchons à capter des informations utiles lors de la simulation MPSoC pour produire des traces pertinentes. Dans les exécutions parallèles, l'ordre dans lequel les choses se passent est de la plus haute importance, les traces doivent être capables de fournir un moyen pour représenter le parallélisme. L'applicabilité de ces traces est explorée en les utilisant dans deux problèmes de débogage au niveau du système différents. Parmi tous les problèmes au niveau du

système qui arrivent au cours du développement d'un nouveau MPSoC, nous avons choisi de mettre l'accent sur deux d'entre eux qui sont liés à l'intégration matériel/logiciel de bas niveau et de la mise en œuvre d'un logiciel fonctionnel de haut niveau.

Tout d'abord, le principal obstacle à l'évolutivité de MPSoCs est le contrôle de cohérence de cache. Solutions basées sur le matériel ont besoin de beaucoup d'espace et d'imposer l'échange d'un grand nombre de messages par le biais de l'interconnexion pour garantir la cohérence des données de tous les caches dans le système. Solutions logicielles ont été proposées pour éviter le sur coût du matérielle pour garantir la cohérence de cache, mais ils comptent sur le programmateur, et sont donc sensibles aux *bugs* de mise en œuvre. Nous préconisons que des traces bien détaillées peuvent aider à l'identification des problèmes de cohérence des données dans l'analyse de système complet de systèmes de mémoire partagée. Nous proposons une méthode pour identifier ces problèmes sur les systèmes de cache non cohérents. Deuxièmement, le débogage déterministe d'une application parallèle est essentiel de bien identifier les *bugs*. Par exemple, dans le logiciel parallèle, le grand nombre de flux d'exécution entrelace l'accès à la mémoire de façon complexe, alors que le maintien de l'ordre d'accès pour chaque exécution est cruciale pour le processus de débogage. Un dernier point est la méthode de mise au point lui-même. Le processus de débogage classique est généralement constitué d'un nombre excessif des redémarrages de l'application jusqu'à trouver le *bug*. Imaginez la possibilité d'éliminer les redémarres, puis en utilisant une seule exécution pour représenter le comportement du système et donc aller de l'avant et vers l'arrière dans les flux d'exécution. Nous proposons donc une méthode capable de fournir une exécution déterministe des programmes parallèles qui est également capable d'opérations défaire, un retour en arrière dans l'exécution. Nous appelons cette méthode comme une débogage inverse déterministe.

Contributions

Cette thèse présente trois contributions majeurs. La première contribution est composé par un formalisme de traces qui représente le comportement parallèle des programmes. Ces traces sont capturés par une plate-forme virtuelle permettant la spécification de nouvelles relations entre les événements. Le formalisme de trace proposé est basé exclusivement sur l'ordre des événements produites dans les différents composants internes. Par conséquent, ces traces contient une relation de causalité entre les événements, qui évite l'utilisation des horloges. La génération de trace est fait en utilisant une plate-forme virtuelle hybride basé sur les technologies de DBT et TLM.

La deuxième contribution présenté dans cette thèse est basée sur la détection des violations de cohérence des données des plates-formes multiprocesseurs non cohérents. Notre méthode est basée sur l'ordre des événements et les relations de causalité en détail dans notre première contribution. Nous proposons une méthode qui identifie les questions de cohérence de cache en deux politiques d'écriture des caches différents: *write-through* et *write-back*. Un ensemble de règles détaillées à l'aide de l'ordre des événements pour détecter les violations de cohérence de cache dans les deux politiques est formalisé. Chaque stratégie d'écriture est couvert par un ensemble de règles spécifiques. Les règles proposées ne prennent pas en compte les horodateurs, car ils se concentrent sur l'ordre des événements qui peuvent être obtenus sur une simulation purement fonctionnel. Par conséquent, cette méthode d'analyse repose sur le cache et les accès mémoire. Lorsqu'une infraction est identifié, notre méthode suggère des corrections, en fonction du type de la

violation, et il propose également le bon endroit dans le code source où la correction doit être inséré. Expérimentations qui ont été menées preuve l'efficacité de notre méthode. Un prototype fonctionnel a été mis en œuvre pour appliquer les règles proposées et de localiser le point de correction de code source et la cause des violations.

La troisième contribution principale vise à fournir une méthode de mise au point déterministe et réversible basée sur les trace pour les applications parallèles fonctionnant sur des plates-formes virtuelles. Cette contribution inclut également des détails des algorithmes qui créent et maintiennent un graphe d'exécution représentant le comportement du système lors de l'exécution parallèle. Un outil qui enveloppe la mise en œuvre de ces algorithmes dans *gdb*, qui est une source ouverte et débogueur largement adoptée, est également proposé.

Chapitre 2 - Problématique

Les phases de débogage des *SoC* prend la majeure partie des efforts de l'équipe de développement, jusqu'à 40% du temps du développement global. Des outils de débogage nouveaux et efficaces sont nécessaires pour aider à simplifier les concepteurs et les ingénieurs dans la vie quotidienne.

Contexte

Les nouvelles architectures SoC présentent de nombreuses variantes architecturales, comme le nombre de processeurs, organisation de la mémoire, l'interconnexion, etc. Ces différences architecturales posent souvent des soucis au niveaux des interactions matériels et logiciels, qui doivent être pris en compte au cours du processus de mise au point.

Le modèle de programmation définit les règles utilisées pour programmer une plate-forme donnée. Dans les nouvelles MPSoCs et des nombreuses architectures de base, le développement d'un programme concurrent doit être repensé le modèle de programmation adoptée. Le modèle de programmation plus simple est le modèle de mémoire partagée. Ce modèle est fourni par *Portable Operating System Interface* (POSIX) *Threads* (Pthreads) sur les plates-formes cache cohérentes. Cependant, dans le cache des plates-formes non cohérentes, le modèle de programmation le plus adéquat est encore une question ouverte.

L'architecture d'un système multiprocesseur dicte souvent le meilleur modèle de programmation à utiliser. Il y a deux paradigmes largement utilisés: le passage de messages et la mémoire partagée.

Dans cette thèse, nous sommes spécialement intéressés à améliorer le processus de vérification des programmes développés pour les architectures non cohérents. Les systèmes à mémoire partagée sont un cas sensible compte tenu de cache et de l'interaction de la mémoire. Nous nous concentrons sur cette architecture en raison de deux facteurs. Tout d'abord, l'approche de passage de messages ne souffert pas des problèmes de cohérence des donnés. Deuxièmement, des architectures à mémoire partagée utilisent des caches locales qui sont la cause de l'incohérence des données. Nous visons à fournir une méthode pour identifier les violations de cohérence de cache pour les architectures non cohérents. Cette identification a pour but d'être indépendant du modèle de programmation exacte. Pour ce faire, nous nous appuyons sur les traces d'exécution détaillées capturés dans les plates-formes virtuelles.

L'adoption de modèles réduire le coût de développement. Nous représentons le flot de conception en trois *Design Space Exploration* (DSE) niveaux (élevé, moyen et bas niveaux). Le niveau élevé du DSE permet l'exploration à faible coût de plusieurs configurations architecturales, étant ainsi utilisé au début du flot de conception. D'autre part, à la fin de l'écoulement de la conception, le bas niveau DSE permet une vérification précise des interactions matériel/logiciel. Le niveau moyen du DSE offre un bon compromis entre la précision et la flexibilité (représenté par le coût de la modification et de configuration des chances).

Méthodes de Analyse et Débogage

Programmes parallèles sont une confrontation quotidienne pour les ingénieurs. Ces programmes consistent dans l'exécution d'un ensemble d'instructions logiquement liés, également appelés flux d'exécution, en parallèle. Le grand nombre de flux possibles peut être lourde pour les ingénieurs, si elles veulent les suivre dans leur propre chemin. Les techniques de débogage peuvent mettre en évidence des erreurs et aider à trouver leurs causes racines. Pour cet objectif, les techniques largement adoptées sont test non de régression, analyse l'exécution de trace et le débogage classique.

Ces méthodes peuvent être utilisées ensemble pour tirer parti de leurs points forts. Au tout début du développement, les essais non régressives peuvent être utilisées pour vérifier si un problème a eu lieu ou non. Après cela, si une erreur se produit, l'analyse exécution de trace apporte le jeton d'identifier le type d'erreur est produit et donnant à l'endroit approximatif à l'intérieur du code source où le comportement anormal se produit. Enfin, la phase de mise au point classique est nécessaire d'identifier exactement ce qui est la cause du problème, analyser manuellement les effets des instructions précédentes et suivantes, l'exactitude des états de mémoire, etc. Dans débogueurs modernes, l'analyse est intégrée à l'intérieur de l'outil.

Le processus d'analyse est une partie importante du flux de développement. Il peut être divisé en deux catégories principales: statiques et dynamiques.

L'analyse statique est un processus qui effectue l'analyse sans exécuter effectivement le programme. Le code source et le code objet sont généralement son entrée. L'analyse statique est utilisé pour prouver formellement les propriétés d'un programme, par exemple si toutes les variables sont initialisées avant d'être effectivement utilisés, ou l'ensemble des structures de commutation cas sont atteints, etc. En outre, ils peuvent être utilisés pour analyser le flux de données qui contribue la détection des conditions de course, par exemple. Le principal avantage de l'analyse statique est l'examen de tous les chemins d'exécution possibles et les valeurs des variables, et pas seulement celles invoquées lors de l'exécution. Ainsi, l'analyse statique peut révéler des erreurs qui peuvent ne pas se manifester avant plusieurs semaines, des mois ou des années après la libération. Cet aspect de l'analyse statique est particulièrement utile dans l'assurance de la sécurité, parce que les attaques de sécurité exercent souvent une demande de façon imprévue et non testés. Cependant, il est d'une grande complexité de calcul et ne concerne pas le code existant.

D'autre part, l'analyse dynamique est un processus qui examine le programme en cours d'exécution ou en utilisant des objets qui représentent le comportement d'exécution, tels que des traces d'exécution. L'avantage majeur de l'analyse dynamique est sa vitesse, au coût de généralité, par rapport à l'analyse statique, car il ne couvre que les opérations

d'exécution, à l'exclusion de l'analyse des portions de code qui ne sont pas effectivement exécutées dans cette exécution. analyse dynamique peut jouer un rôle dans l'assurance de la sécurité, mais son objectif principal est de trouver et les erreurs de mise au point.

Positionnement

Dans le cadre de cette thèse, les plates-formes virtuelles exécutent le code binaire en générant des traces. Les traces sont exploitées pour analyser l'exécution de ce code binaire. Pendant l'analyse le code binaire est utilisé également pour la récupération des symboles, de sections et d'autres données présentes dans ce fichier. Le code source peut être utilisé pour retourner d'informations à l'ingénieur. Résultant en un rapport contenant, par exemple, les statistiques d'exécution et de la couverture de code source.

Traces d'exécution

Traces sont un ensemble d'événements et un ensemble de relations entre ces événements qui décrivent le comportement d'un système donné. La granularité des traces dépend de quel type d'informations les ingénieurs ont l'intention de capturer. Traces de bas niveau peuvent être applicables pour aider à résoudre les problèmes d'interaction matériel / logiciel, qui sont généralement difficiles à résoudre. Dans le contexte de MPSoC, l'augmentation du nombre d'interactions possibles entre les composantes rendre cette tâche encore plus compliquée.

Les architectures parallèles présentent des complications bien connues. Les accès simultanés faits par deux processeurs différents à un composant partagé au même temps est l'origine du problème de la congestion, par exemple, qui peut avoir un impact direct sur la performance des programmes massivement parallèles. L'identification de ces problèmes peuvent être résolus en utilisant des traces, quand ils transmettent les informations correctes. Autres problèmes parallèles, comme la cohérence, la cohérence et la synchronisation peuvent également être examinées à l'aide des traces. En effet, pour chaque problème l'information à l'intérieur des événements peut changer.

Afin de réduire la complexité d'une analyse, la trace peut être exploité en profitant des informations supplémentaires relatives à l'exécution: descriptions d'événements précis, des événements qui sont par ailleurs difficiles à rassembler, et les relations entre les événements. Par exemple, pour bien comprendre l'impact des accès simultanéités aux données, il faut connaître non seulement le lancement des demandes (exécution des instructions de chargement ou de stockage), mais aussi de leur arrivée à la mémoire, et la relation entre le début et à l'arrivée de chaque demande. Récupération des événements d'initiation est toujours possible, alors que les événements d'arrivée ne peuvent être rassemblés au niveau du matériel. Cependant, en utilisant les plates-formes réels, les relations sont simplement impossible de recueillir dans les deux cas.

Dans les plates-formes réels, pour obtenir les traces, il faut du matériel supplémentaire. Pour obtenir des informations encore plus détaillés, il faut du matériel de plus en plus coûteux. Cependant, plates-formes réels ont des limitations physiques, comme la bande passante des interconnexion. Le conséquence d'avoir un matériel supplémentaire est l'augmentation de puissance consommé. Ces points limitent les capacités de la capture des traces dans les plates-formes réels. D'autre part, les plates-formes virtuelles permettent l'instrumentation des modèles en permettant la construction des chaînes de causalité entre les événements, par exemple. La collecte de traces à travers de ce moyen est non-intrusif

et ne souffre pas de bande passante ou des limitations de la mémoire. Ensuite, la collecte de toutes les informations de propagation dans le système dans une trace structurée est un rôle que les plates-formes virtuelles peuvent jouer honorablement.

L'intrusion est une caractéristique qui décrit la profondeur des modification d'un système pour produire de l'information. Cette modification a plusieurs significations, il peut représenter soit une modification physique ou une modification du code source, par exemple.

Défis à la débogage des MPSoCs

Dans les programmes parallèles, des problèmes communs rencontrés par les ingénieurs, *e.g*, les *race conditions*, la cohérence de cache et la cohérence de la mémoire. MPSoCs sont les moteurs d'exécution parallèles sont maintenant face à ces questions. Chacun d'entre eux ont ses propres paradigmes parallèle associés. Nous croyons que les traces d'exécution peuvent être utilisés pour aider à résoudre ces problèmes en raison de leur représentation valable du comportement du système.

La cohérence de cache est un problème au niveau du matériel / logiciel qui découle de nouvelles architectures. Plusieurs nouvelles architectures n'ont pas de matériel qui garanti la cohérence de cache. Ces plates-formes présentent passent à l'échèle plus facilement sans ce matériel, mais ils sont plus complexe à programmer. Nous croyons que le manque d'outils dédiée à aider l'intégration de systèmes et le débogage est l'une des raisons qui rend l'adoption de telles plates-formes difficiles.

Systèmes de cache non cohérent

Les systèmes avec plusieurs processeurs peuvent avoir accès à des copies multiples d'une donnée dans leurs caches respectifs. Lorsque l'un d'eux fait une *write*, les données présentes dans le cache d'un autre processeur ne doit plus être utilisé. Quand un processeur l'utilise une violation de cohérence de cache se produit. En effet, la cohérence de cache assure qu'un processeur ne attrape jamais une valeur qui n'est pas mis à jour à partir de son propre cache.

Les solutions matérielles classiques ne passent pas facilement à l'échèle. Parmi les problèmes rencontrés par eux, les plus notables sont une forte consommation d'énergie et la complexité de la vérification. Tout d'abord, la plupart des approches classiques, tels que les protocoles *snooping*, entraînera une perte de performance liée à la latence et de la puissance. En effet, les messages d'acquaintance et d'invalidation peuvent consommer plus de bande passante et de la puissance nécessaire. Deuxièmement, la vérification formelle de ces protocoles est complexe en raison de nombreux états transitoires. Le temps consommé par la vérification décourage l'optimisation des protocoles bien établis. Troisièmement, ces protocoles induisent une surcharge de stockage pour suivre la liste de partage.

Cette question a ouvert un domaine de recherche qui vise proposer des nouveaux protocoles logiciel pour la cohérence de cache. Les solutions logicielles sont des alternatives viables, en effet, l'idée est de supprimer la complexité matérielle en avoir un logiciel plus complexe, ce qui peut imprègne l'écosystème, par exemple compilateurs, les langues, l'OS, l'application elle-même, ainsi de suite et ainsi de suite. La cohérence de cache logiciel a été largement étudié dans le passé, mais les architectures many-core bénéficient de

temps de latence beaucoup plus faible et un débit plus élevé que les machines discrètes et plusieurs nouvelles propositions ont été faites récemment sur ce sujet.

Les méthodes de programmation efficaces et des outils manquent pour le débogage des architectures de cache non cohérentes

Débogage reversible

L'obtention d'une exécution déterministe d'un logiciel non-déterministe est un problème qui a été déjà résolu. Une solution bien adoptée est une méthode appelée *replay* déterministe, qui consiste à capter le comportement erroné, puis reproduire de manière déterministe. Après cela, l'ingénieur explore l'exécution déterministe pour trouver le problème. La méthode pour capturer le comportement est basée sur des traces. De toute évidence, la fidélité de la reproduction dépend directement sur les traces enregistrées.

L'exécution à l'envers d'un programme peut aider le développeur à éviter le nombre élevé des redémarrages pendant le processus de mise au point.

La chasse aux *bugs* commence en identifiant le problème, les ingénieurs vont trouver et corriger. Cependant, le manque de fonctionnalités sur les outils de débogage laisse ingénieurs par eux-mêmes. Débogage d'un grand nombre de flux d'exécution sans outils appropriés est une des raisons. Des fonctionnalités alternatives doivent être incorporés à des débogueurs pour les aider. Le *replay* déterministe est apparue comme une solution, mais il peut être nécessaire d'avoir beaucoup de replays jusqu'à repérer la cause. Une autre méthode consiste à intégrer l'exécution à l'envers dans débogueurs. L'exécution inverse consiste à récupérer l'état précédent d'une instruction donnée, à savoir défaire les modifications apportées par ce dernier.

Problématique

Dans ce contexte, cette thèse vise à répondre aux questions suivantes:

Question 1: Comment définir une trace pour capturer le comportement parallèle du logiciel s'exécutant sur une plateforme virtuelle ?

Question 2: Comment exploiter les traces pour identifier efficacement les problèmes de violation de cohérence de cache qui apparaissent dans le MPSoC sans support matériel à la cohérence ?

Question 3: Comment fournir une méthode de débogage déterministe permettant de revenir en arrière, applicable à du logiciel parallèle ?

Chapitre 3 - L'état d'art

Faire le *dumping* des informations du matériel et du logiciel sur l'exécution d'une application est une pratique habituelle. Le but de ces *dumps* est généralement de fournir des informations lisible en ayant une vue sur le comportement du système. La technique la plus connue est l'utilisation des déclarations de *prints*. Par opposition aux *dumps*, des traces contiennent des informations structurées avec une sémantique claire, afin qu'elles puissent être utilisées par les programmes pour le profilage, le débogage et ainsi de suite.

Collecte traces pose plusieurs questions: (1) L'accès aux informations à tracer. Quelle fonction ou une instruction est en cours d'exécution, ce qui est la valeur d'une variable ou d'une case mémoire? (2) La collecte et la transmission des informations. Quelles sont

les ressources nécessaires pour collecter et transmettre les donnés sur le système? (3) Les relations entre les événements. Quels sont les événements ont eu lieu avant un événement donné? Quel événement est la source d'un outre événement donné?

Ces questions ont été abordées par divers approches.

Annotations de code

L'annotation du code est un processus qui consiste à ajouter de nouvelles lignes de code dans le code source. Dans le contexte de la génération de trace, ces nouvelles lignes visent à créer des événements qui seront dans l'ensemble de trace. Ces modifications peuvent modifier le comportement du système. Habituellement, cette méthode implique haute intrusion. Malgré cela, il est largement utilisé dans l'industrie.

Les événements sont liés à des fonctions et des variables, mais ne exposent pas l'exécution des instructions et les accès mémoire en détail. Généralement, ce type de solution repose sur les relations entre les événements qui est maintenue grâce au *timestamping*, que l'exigence d'une horloge de système globale et géré par logiciel. Ces horloges de logiciels sont difficiles à rester fiables sur les plates-formes multiprocesseurs. Cette approche est très intrusive en modifiant ainsi le comportement du logiciel.

Instrumentation Dynamique

Une alternative à la annotation du code est instrumentation dynamique. Elle repose sur l'intégration des *probes* lors de l'exécution de l'application pour analyser son comportement. Les *probes* contiennent des conditions qui, une fois déclenché, produisent un événement. Ces conditions peuvent être décrites dans des langages spécifiques. Leur inclusion et son retrait se produisent dynamiquement. Il implique que lorsque aucune *probe* est présente, l'intrusion de la méthode est presque éliminé. Différemment de la méthode précédente, ces commandes ont besoin d'un *run-time* spécifique.

Cette technique présente quelques inconvénients. La densité de *probes* influe directement sur la performance du système entier. En outre, les interactions matérielles / logicielles ne peuvent pas être examinées à l'aide de cette technique, étant la plupart du temps appliqué à la surveillance des ressources, telles que l'utilisation de la mémoire et le temps de traitement. En plus, il ne peut pas extraire les informations de bas niveau pour l'analyse matérielle / logicielle, tels que les interactions de processeur et de la mémoire cache. Ainsi, cette technique ne peut être appliquée efficacement à l'uniformité et la cohérence de la mémoire cache de l'analyse, par exemple.

Port de traces

Un port de trace est un composant matériel qui capture les interactions de bas niveau entre les composants matériels. Cette approche est différente des deux précédemment présenté, car il repose exclusivement sur le matériel, donc pas de configuration de logiciels et la mise en œuvre sont nécessaires, à l'exception de ceux-ci nécessaires pour configurer le port de trace lui-même. Les ports de traces sont importantes pour l'analyse bas niveau parce que les événements de bas niveau ne peuvent être capturés au niveau du matériel grâce à l'utilisation d'un IP matériel dédié pour les processeurs, ou des composants matériels spécifiques.

Certaines analyses nécessitent des événements de bas niveau, comme il est le cas pour le débogage des programmes concurrents. Ces IPs exportent également des informations via des interfaces physiques, mais dans ce cas, la quantité de données est énorme, que les ressources matérielles étant limitées, il est nécessaire de sélectionner à l'avance les événements à tracer. Même si les techniques de compression peuvent être appliquées, cette quantité de données est toujours un problème.

Plates-formes virtuelles

Une solution alternative est la production de traces issues de plates-formes virtuelles, dans lequel le simulateur et les modèles sont modifiés à cet objectif. Cela donne accès à des événements de bas niveau et reste non-intrusive, donc le seul effet de la production de trace est qu'elle diminue la performance de la simulation. Une analyse peut profiter des relations plus avancées, qui ne peuvent être récupérés à partir des *timestamps*, tels que les relations cause / effet.

Obtenir des informations de ce niveau de détail à partir de plates-formes réels peut être extrêmement coûteux, compte tenu des coûts et de l'impact du logiciel, et peut modifier le comportement du logiciel. En utilisant des techniques de logiciels est tout aussi problématique, car ils sont encore plus intrusive. Ces exemples montrent clairement que pour l'analyse complexe de l'utilisation de plates-formes réels et des techniques logicielles sont impraticables et les plates-formes virtuelles peut être utilisé comme une solution qui n'est pas intrusif en fournissant des traces précises.

La plupart des MPSoC sont modélisés à comme ISS / interprétative, cependant, son applicabilité est intrinsèquement limitée à de petits systèmes. Comme l'interprétation des instructions devient un goulot d'étranglement lorsque le nombre de processeurs grandit, et la simulation native est pas assez souple, les approches alternatives puis ont été proposées, parmi lesquelles DBT apparaît comme la plus prometteuse.

La traduction binaire est un processus qui génère un code binaire hôte exécutable basée sur le code binaire de la cible. Par conséquent, avant le code hôte final, une phase intermédiaire peut être nécessaire en générant un code intermédiaire. Cette phase supplémentaire vise à faciliter l'ajout de nouvelles cibles et de hôtes.

Cependant, au mieux de notre connaissance collecte trace du système dans un cadre de simulation de TLM qui utilise DBT pour l'exécution du logiciel n'a pas été abordée précédemment.

Méthodes formels d'analyse

L'analyse formelle est une méthode mathématique qui est appliqué à la vérification et la spécification des systèmes. Par conséquent, ils ont bésion généralement de calcul coûteux et ils n'échèlent pas bien pour la co-vérification matérielle/logicielle, telles que l'analyse de la cohérence de cache Deux approches sont généralement utilisées pour l'analyse de matériel / logiciel: le *model checking* et la démonstration de théorèmes.

Le *model checking* est une méthode formelle qui permet l'analyse de tous les états accessibles du système. L'objectif est d'analyser chaque état en fonction de règles prédéfinies.

La démonstration de théorème est une méthode formelle qui détermine la validité d'une formule. La logique propositionnelle et temporelle sont des outils utiles pour

la spécification et la vérification de ces formules, et ils sont surtout appliquée dans le domaine du matériel.

Méthodes d'analyse basée sur traces

On peut classer les méthodes basées sur trace comme les méthodes semi-formelles. Les méthodes semi-formelles sont un complément des méthodes formelles. Ils sont utilisés pour diminuer le coût de l'analyse. En général, les méthodes semi-formelles portent un niveau d'abstraction initialement élevée, *i.e.* niveau du système. Après cela, quand un problème est identifié dans un bloc spécifique, les méthodes formels ou la débogage classiques peuvent être appliquées pour les résoudre.

La méthode générale d'analyse semi-formel peut être traduit comme une approche en boucle fermée où le comportement illégal est identifié par un utilisateur ou une méthode automatisée, nommés moniteurs. Lorsqu'un moniteur est déclenchée et un comportement illégal est identifié, une analyse est faite sur des modèles formels, que ce soit une erreur de conception réel ou non. S'il est pas une erreur de conception, il peut être introduit une faut au niveau l'abstraction ou la modélisation de l'environnement. Dans ce cas, les modifications sur le simulateur ou les modèles sont faits et le processus est redémarré. Cette méthode utilise comme information supplémentaire traces générées par le modèle formel pour nourrir le simulateur.

L'efficacité d'une analyse dépend du type d'informations fournies par les traces. L'ordre est l'un de ces paramètres. Nous avons trouvé dans la théorie de l'ordre deux candidats qui sont utiles dans les traces, l'ordre total et les ordres partiels. Strictement parlant, l'ordre total est une relation binaire sur un certain ensemble qui est transitive, antisymétrique et totale. Cette relation permet de comparer tous les événements dans un ensemble. En ayant l'ordre total d'une exécution, il est possible d'analyser ce qui est arrivé avant et après un événement donné. Cependant, les méthodes qui utilisent uniquement l'ordre total pour représenter le système parsèment généralement aucune considération sur l'entrelacement des flux d'exécution. L'ordre partielle est une relation binaire qui indique une certaine paire d'événements que l'une précède l'autre. Dans cette relation ne sont pas tous deux des événements doit être liée.

Certaines méthodes basées sur traces souffrent pas des explosions d'état parce que, intuitivement, ils ne font qu'ajouter des arcs et des sommets à une trace d'entrée donné, par conséquent, il n'y a pas d'espace d'état traversée. POTA utilise traces en appliquant vérification formelle basée sur les expressions de CTL. Il applique une méthode appelée *computation slicing*, qui est une technique d'abstraction pour l'analyse des traces de programmes distribués.

La définition des traces utilisé par le *computation slicing* permettent de capturer les relations entre les composants du système. L'analyse, par l'utilisation de ces relations, permet l'identification des problèmes d'interaction entre le matériel et le logiciel, car il identifie, à travers des événements, le comportement de chaque composant du système.

Ils utilisent des traces pour représenter le comportement du système, délimitant l'analyse. Cependant, l'interaction des composantes peut ne pas être entièrement capturé par des traces ou exhaustivement testé, ce qui peut ouvrir un espace pour les *bugs* opportunistes. A la fin, une compromis performance / couverture est imposée à la vérification du système.

Débogage reversible

Le processus de mise au point dans les systèmes multiprocesseurs est une tâche difficile en raison du grand nombre de flux d'exécution possibles. Un autre facteur compliqué est le non-déterminisme de ces systèmes.

Le débogage déterministe est un processus qui vise à fournir le même ordre d'opérations du système chaque fois qu'une exécution a lieu avec les mêmes intrants. Il est une fonctionnalité très puissante car elle permet la reproductibilité des problèmes intermittents, qui est souvent un problème très difficile à résoudre.

Même soutenu par des solutions de débogage déterministes, le processus classique de débogage interactif peut avoir besoin de beaucoup de redémarrage du système jusqu'à ce que l'identification de *bug* correct. Puis offrant des fonctionnalités supplémentaires qui permettent d'éviter un nombre élevé de redémarrages du système, tels que la fonctionnalité de débogage réversible a comme objectif faciliter le processus de débogage. Le débogage réversible est un processus qui fournit aux développeurs les moyens pour exécuter un programme dans un sens à l'envers à celle de l'exécution normal d'un programme.

Log d'exécution et *structured backtrack*

Le *log* d'exécution est un fichier qui contient le *change-set* du système. Le *change-set* se compose de toutes les variables dont les valeurs sont modifiées par les déclarations du programme.

L'approche *structured backtrack* est une méthode qui intègre plus d'informations sur les structures de contrôle à l'intérieur de *log* d'exécution. Il observe les variables qui pourraient être modifiés à l'intérieur de ces structures de contrôle (`for` et `while`). A partir de cette information, les valeurs de ces variables sont stockées avant d'entrer dans la boucle, par conséquent, étant donné que l'affectation suivante à traiter est située après une boucle.

Checkpointing

Les techniques de *checkpointing* visent à créer périodiquement un fichier contenant l'état du système. Les points de contrôle peuvent être créés soit lorsqu'une condition est déclenchée, par exemple, un appel à une fonction est exécutée, ou soit simplement après un intervalle de temps donné. Ainsi, lorsque l'exécution à l'envers est lancé, le débogueur récupère l'état du système précédent plus proche de l'état souhaité, puis re-exécuter le code en avance jusqu'à ce qu'il atteigne le point souhaitable.

Cependant, le débogage à l'envers base de points de contrôle a besoin d'améliorations pour permettre l'exécution déterministe.

Instrumentation du code source

Instrumentation du code source est une autre méthode pour fournir la fonctionnalité de débogage à l'envers. Elle consiste à ajouter des déclarations non fonctionnelles à le code source fonctionnel. Dans le cas du débogueur réversible, ces déclarations non fonctionnelles visent à produire un *log* d'exécution, ainsi, ils sont utilisés pour capturer les valeurs modifiées causés par chaque déclaration ou instruction exécutée, à savoir,

quels registres ont été modifiés ou quelles sont les nouvelles valeurs ont été écrits dans la mémoire.

Cependant, les méthodes basées sur l'instrumentation du code source ont tendance à être très intrusifs, en adoptant même instrumentation partielle. Un exécution non déterministe peut être produit en raison de l'intrusion de l'instrumentation, qui peut changer l'ordre des opérations parallèles. Une autre limite est liée à la disponibilité de code source. Seules les fonctions présentes dans le code source peut être fait marche en arrière, ainsi bibliothèques binaires propriétaires ne sont pas pris en charge.

Génération reverse du code

Il est une méthode qui consiste à générer un exécutable inverse qui annule les opérations effectuées par une déclaration donnée. Il utilise la notion de déclaration *self-defined*. Il est une expression où les opérandes et les résultats finaux se trouvent dans la même variable.

La génération de code dynamique inversé consiste à produire le code inverse lors de l'exécution au cours d'une session de débogage. Cette méthode combine les techniques de calcul inverse mathématiques avec des techniques *saving-states*. Il permet le débogage à l'envers déterministe de programmes parallèles.

Ils consomment de temps de traitement importante par rapport leurs paires, grâce aux calculs nécessaires pour obtenir les états inverses. Le déterminisme de ces méthodes est également pas clair. Comme l'instrumentation du code source, des calculs supplémentaires sont exécutées pour fournir la fonctionnalité inverse, tels que les états sauvegardés ou le calcul des déclarations inverses. Ainsi, l'ordre des différents *threads* peut varier. Ensuite, il est très intrusive, à ce stade, qui peut influer sur le déterminisme de l'exécution.

GDB

Le débogueur *open source* bien connu *gdb*, depuis la version 7.0 sorti en Septembre 2009, prend en charge les commandes d'exécution inverse. Pour chaque plate-forme cible doit être disponible pour soutenir l'exécution inverse. Une cible est l'environnement d'exécution occupé par un programme. Il peut être utilisé pour mettre en œuvre des fonctionnalités spécifiques, augmentant le nombre de commandes prises en charge par une plate-forme donnée, travaillant aussi comme un plug-in. C'est le cas de l'appui de débogage inverse.

L'utilisation d'un débogueur bien connu comme base pour démontrer la faisabilité d'une méthode aider à son adoption en académie et l'industrie. En outre, les recherches se concentrent sur la méthode, puisque les protocoles d'infrastructure de l'utilisateur et de la communication sont déjà largement adopté

Concernant *gdb*, *record target* qui est le responsable pour l'implantation du débogage réversible est très intrusive. Ainsi, mieux appliqué à debogage d'application séquentielle, il est difficilement applicable pour le débogage des couches complètes de logiciels parallèles. Heureusement, l'architecture de *gdb* est extensible et de nouvelles méthodes peut utiliser son infrastructure pour déployer des approches novatrices.

Chapitre 4 - Définition de la trace et méthode de capture

Nous travaillons avec l'hypothèse que les traces doivent représenter le comportement d'un multiprocesseur exécutions, qui se composent d'un grand nombre d'actions qui se déroulent en parallèle. Ainsi, pour obtenir une trace cohérente, nous répartissons nos objectifs dans quatre catégories: *niveau d'abstraction*, *parallélisme*, *intrusion* et *vitesse*.

Niveau d'abstraction: le premier objectif vise à capturer des traces qui permettent la représentation au niveau du système, étant applicable au processus de débogage. Modèles au niveau du système se concentrent sur le comportement fonctionnel, ce qui simplifie les détails d'implémentation, qui aident à produire des résultats fonctionnels en utilisant la simulation au lieu d'utiliser le matériel réel.

Parallélisme: Le deuxième objectif vise à capturer le comportement des logiciels parallèles fonctionnant sur MPSoCs. Une mauvaise mis en œvre de la programmation parallèle et d'approche ou leur absence est la source de nombreux *bugs*. Ainsi, une méthode qui préserve l'ordre total et relie ces événements aux conséquences que chaque événement provoque dans le système permettent de reproduire le parallélisme des événements.

Intrusif: Notre troisième objectif est de proposer une méthode non intrusive pour capturer des traces. L'intrusion peut modifier l'ordre dans lequel un programme est exécuté. Dans les environnements d'exécution parallèles, cette modification peut être encore pire, car il peut modifier l'ordre d'une ressource partagée est accessible masquage ou conduisant à des problèmes comme la congestion parallèle d'interconnexion.

Vitesse: Le quatrième objectif est d'obtenir une méthode rapide pour capturer des traces. La simulation de systèmes complexes prend généralement beaucoup de temps pour être terminé en raison de l'augmentation du nombre de composants, de leurs interactions et les limites de la technologie. Simulateurs de plate-forme virtuelle rapides sont donc un bon choix pour capturer des traces.

Nous décrivons comment ces objectifs sont couverts par la définition formelle des traces et détaillant la façon dont nous les capturer.

Définition des événements et de la trace

Compte tenu de C un ensemble de composants matériels, nous définissons un événement comme un tuple $e = (c, t, d)$, où c est le composant, avec $c \in C$, t le type d'événement, d les données supplémentaires. En outre, l'attribut $t \in \{I, S, M, B, L, A\}$, avec I signifie l'instruction d'un processeur, S une écriture dans lune case mémoire ou dans le cache, L une lecture dans le cache, M une modification dans le cache, B une écriture dans la mémoire sans modification du cache (ce qui signifie une reprise) et a une acquaintance fait par la mémoire ou le cache. Le complément de données d contient des informations spécifiques instruction. Pour les charges et les magasins d contient l'adresse de l'accès, par exemple. D'autre part, l'instruction `add` a d contenant valeur de registre. Cet attribut est défini autant générale que possible parce que les différents événements rassemblent des données différentes.

Par conséquent, en notant E l'ensemble des événements que le système produit lors d'une exécution et E_C l'ensemble des événements d'un composant c, alors $E = \bigcup_{c \in C} E_C$ l'union disjointe de E_C. L'ordre dans E est aussi l'union disjointe de tous les ordres d'origine, qui ne forment pas un ordre total de E.

Les interactions entre les composants sont également recueillies lors de la simula-

tion. Prenons une architecture simple composée d'un processeur, mémoire cache L1 et la mémoire. Les événement de écriture et de lecture conduisent soit à un événement d'acquaitance,soit à la création d'autres événements d'écriture ou de lecture, respectivement. Dans le premier cas, les données sont déjà dans la mémoire cache et il faut demander des données à partir de cette mémoire, alors le cache produit l'acquaitance. Dans le second cas, le cache n'a pas les données demandées forçant une demande à la mémoire, la demande est créé par le cache, et l'acquaitance est généré par la mémoire. Sur la base de cette connaissance, nous définissons deux relations entre les événements.

Component Strict Total Order (<): il est de l'ordre entre les deux événements différents qui se sont produits dans le même composant. Nous définissons $<_c$, un ordre total strict sur E_c, tel que $\forall(e_i, e_j) \in E_c \times E_i | E_i <_c e_j$ lorsque l'occurrence de E_i précède l'apparition de e_j.

Causalité ($\hookleftarrow$): il est la relation entre deux événements e_i et e_j, où e_j est une conséquence de e_i, représentés par $e_i \hookleftarrow e_j$. Dans le contexte de traces, e_i et e_j se produisent en différents composants. Nous notons $(e_i, e_j) \in E_k \times E_l$avec$(k, l) \in C \times C$ et $k \neq l | e_i \hookleftarrow e_j$ le fait que l'événement e_j "est en raison de« l'événement e_i.

Les deux relations entretiennent les propriétés ci-dessous:

Réflexive: $e_a \hookleftarrow e_b \implies e_a \neq e_b$ et $e_g < e_h \implies e_g \neq e_h$.

Transitive: $e_a \hookleftarrow e_b \wedge e_b \hookleftarrow e_c \implies e_a \hookleftarrow e_c$ et $e_g < e_h \wedge e_h < e_i \implies e_g < e_i$.

Antisymétrique: $e_a \hookleftarrow e_b \wedge e_b \hookleftarrow e_a \implies e_a = e_b$ et $e_g < e_h \wedge e_h < e_g \implies e_g = e_i$.

Deux événements ne sont pas comparables si elles sont générées dans les différents composants et ne pas avoir une relation de cause à effet. Cependant, cette comparaison est crucial pour l'analyse de la cohérence de cache où il est important de comparer les événements qui se produisent dans les deux caches différents, par exemple. Souvent, les événements qui se produisent dans des caches L1 ne déclenchent pas les événements dans d'autres caches L1, sauf pour le contrôle de la cohérence du matériel. Cependant, dans les systèmes qui ne possèdent pas un tel contrôle sur le matériel pris en charge, une telle comparaison est simplement impossible. Nous utilisons les définitions précédentes pour introduire l'ordre totale du système et le composant partagé.

Composants partagés: Il est un composant $c \in C$ qui génère les événements $e \in E_c$ et qui maintient la relation de causalité avec au moins deux autres composantes différentes. Notant $E_S = E_a \cup E_b \cup E_s$ l'union des ensembles d'événements générés par $\{a, b, s\} \in C$ tel que $e_i \in E_a$, $e_j \in E_b$ et $\{e_k, e_l\} \in E_s$, alors le composant s est un composant partagé si et seulement si $e_i \hookleftarrow e_k$ et $e_j \hookleftarrow e_l$.

Ordre totale du système ($\prec$): il est une relation qui permet de comparer les événements générés dans différents composants. Il est obligatoire que les deux composants ont généré des événements qui ont eu des conséquences dans au moins un autre élément commun, c'est-à-dire une *composant partagé*. Ainsi, l'ordre du système est basé sur l'ordre strict d'un *composant partagé*. Nous notons cette relation $\prec$. Il a les mêmes propriétés que $<$. Nous utilisons le terme ordre du système à court pour l'ordre total du système.

Nous définissons une *trace* que $T = \{E, <, \hookleftarrow, \prec\}$, avec $<$ défini comme $\{ lt_c | \forall c \in C\}$. L'union des ordres définie par $<$, $\hookleftarrow$ et $\prec$ rend un ordre total strict dans E.

L'ordre du système d'événements qui ne disposent pas la relation $\hookleftarrow$ d'un événement généré par un *composant partagé* est déterminé par leur type et leur relation $<$. L'événement *modify-cache M* est du même ordre de système que l'événement précédent qui a partagé l'accès. Formellement, étant donné deux composants $\{a, b\} \in C$, où b est un *composant partagé*: l'ensemble des événements générés sont $E_a = \{e_1, e_2\}$ et $E_b = \{e_3\}$,

où le e_2 est un *modify-cache*. Ces événements ont les relations suivantes: $e_1 \leftharpoonup e_3$ et $e_1 < e_2$. Ainsi, l'ordre du système de e_1 est de l'ordre de e_3 et nous attribuons à e_2 le même ordre de système e_1. Nous définissons cette opération comme *rewind* et de la représenter à l'aide du symbole $\lll$. Un autre type d'événements est le lecture de la cache L. Dans ce cas, l'ordre du système est attribuée compte tenu de l'événement successeur qui a la relation $\leftharpoonup$ avec le *composant partagé*. Par exemple, en considérant les mêmes composants, mais avec différents ensembles d'événements et de relations, $E_a = \{e_1, e_2\}$ et $E_b = \{e_3\}$, où e_1 est un *cache hit*. Si les relations $e_1 < e_2$ et $e_2 \leftharpoonup e_3$ sont construits, puis nous attribuons à e_1 le même ordre de système e_2. Nous représentons cette opération en utilisant $\ggg$ et appelons *forward*. Les deux opérations peuvent être utilisés sur tout type d'événements.

Les opérations *forward* et *rewind* sont principalement utilisés pour attribuer l'ordre du système à des événements de cache.

Génération de traces en utilisant le plateforme virtuelle DBT/TLM

La production de la trace a besoin des modifications dans la DBT en couvrant la traduction, la génération et l'exécution du code. Ces modifications ont comme objectif recueillir et transmettre des informations des modèles. Ces modèles doivent être modifiés pour incorporer la création des liens de causalité entre les événements et l'obtention des informations en provenance de la DBT et les intégrer dans les nouveaux événements qu'ils créent. Cela ne perturbe pas le comportement fonctionnel et est donc non intrusive la fois en fonction et le calendrier.

Comme la DBT repose sur une micro-opération intermédiaire, l'idée est d'ajouter deux nouveaux instructions non fonctionnelles *event_insn* et *event_commit*. Un *event_insn* alloue l'événement et attribue les éléments de tuple connus au moment de l'exécution de l'instruction. Un *event_commit* indique que tous les éléments de l'événement sont mises à jour et délivre l'événement.

Lors de la traduction du code, pour chaque instruction cible, un *event_insn* est ajouté en tant que première instruction de micro-opération de la représentation intermédiaire. Ceci est valable pour toutes les instructions, même si l'instruction n'a pas été réellement exécuté. Le PC et l'*opcode* sont données statiques passés comme paramètres de l'opération *event_insn*. Suivant le même principe, à la fin de chaque instruction traduite, une micro-opération *event_commit* est insérée.

Pendant l'exécution du code, le *event_insn* crée un événement d'instruction. Comme l'exécution produit une *fetch* d'instruction et éventuellement un accès aux données en cache, les demandes de cache sont produites. Le résumé des propriétés remplis à des événements est présenté ci-dessous:

Exécution conditionnelle: instructions qui reposeraient sont analysés lors de l'exécution et ils sont exécutés que si les conditions sont remplies.

Accès alignés: accès non alignée vers la mémoire déclenchent plusieurs accès à la mémoire par rapport les accès normales.

Accès multiples: Certaines instructions peuvent déclencher plusieurs accès à la mémoire.

Nombre de cycles d'instruction: nombre effectif de cycles utilisés pour traiter l'instruction cible, ou un *nop* si le prédicat est fausse.

Accès de écriture et lecture exclusifs: informations nécessaires pour une analyse ultérieure.

Cache de données, cache d'instructions, et accès non-cache: classe l'accès effectué, le cas échéant.

Expérimentations

Nous détaillons trois expériences. La première vise à évaluer la génération correcte de traces d'instruction en vérifiant que l'arbre d'appel d'un programme en utilisant nos traces et une méthode intrusive donne le même résultat. La seconde vise à vérifier la validité des liens de causalité. Le dernier quantifie le ralentissement d'exécution causé par du système de génération de trace et donne quelques exemples de tailles de trace, ce qui donne également la précision sur l'ensemble des plates-formes testées.

Nos expérimentations font usage du *framework Rabbits*, qui lui-même repose sur Qemu pour DBT et SystemC pour la simulation de TLM. L'architecture cible est composé de n processeurs ARM (n est un paramètre de l'utilisateur), un *frame buffer*, un contrôleur UART, `timers`, une mémoire partagée et un contrôleur de stockage. Chaque processeur est un Cortex-A9 avec cache de niveau 1 de données (L1 DCACHE), d'instructions (L1 ICACHE) et un contrôleur d'interruption. Les composants sont interconnectés par un réseau abstrait sur puce (NoC). Les processeurs et la mémoire sont traçables, *i.e.* ils ont la capacité de générer des événements et de produire des relations entre eux. Nos expériences visent à produire des traces plus de cinq plates-formes différentes en faisant varier seulement le nombre $n = \{1, 2, 4, 8, 12\}$ de processeurs.

Les logiciels qui fonctionnent sur les plates-formes sont une application de décodage MJPEG *multi-thread* et un sous-ensemble du *benchmark* Splash-2 (océan, Water-NSquared et Water-Spatial), chacun d'eux sont exécutés sur un système d'exploitation léger avec capacités SMP. L'intérêt d'avoir plusieurs threads, ce qui peut en outre migrer d'un processeur à d'autre, est qu'il exerce également le protocole cache-cohérence sur le système et produit donc toutes sortes d'événements (instruction, *cache misses*, invalidations, etc.).

Couverture de code

La rectitude du processus de la génération de trace doit être vérifié en premier. Pour cet objectif, la propriété «bien formé» est d'abord vérifiée à l'aide d'un outil mis au point sur le but qui vérifie que la relation $<$ compte tenu de la même composante et la relation $\hookleftarrow$ entre les différents composants suivent leurs définitions. Ensuite, nous avons vérifié que le graphe d'appels d'une exécution de notre référence est identique à celui obtenu par l'analyse de la trace générée. En utilisant fonctions *hook* [1] qui fournissent un moyen pour le compilateur d'insérer du code dans le prologue et l'épilogue de fonctions, nous construisons une graphe d'appel de référence. Nos fonctions *hook* personnalisés, qui sont intrusive car le compilateur ajoute le code pour eux, il suffit de sortie l'adresse de la fonction appelée comme le code est exécuté et puis l'adresse est la suite convertis en le nom de fonction. Cette situation influe sur le temps d'exécution, mais ne change pas le graphe d'appel, qui est ce que nous sommes intéressés à cette expérience.

Nous pouvons obtenir les mêmes informations d'une manière non-intrusive en utilisant les traces, avec les événements $\{e \in T | type = instruction\}$ en représentant le chemin

[1] Disponible dans `gcc` utilisant l'option `-finstrument-fonctions`.

d'exécution réelle du programme, et data $d\langle e\rangle$ contient la valeur du PC.

Nous avons exécuté de nombreux programmes, certains d'entre eux avec plusieurs ensembles de données pour de longues périodes de temps, et nous avons obtenu exactement les mêmes fonctions *___entry* et *___exit* et le même appel pour à la fois le logiciel et instrumenté tracé. Cela montre expérimentalement que, pour les éléments du tuple qui sont utilisés pour cette vérification, la trace générée est "bien formé".

Chaines de causalité

Nous générons des traces appropriées pour un outil qui vise à la découverte des goulets d'étranglement à cause de l'augmentation du nombre de processeurs. Elle repose sur une fonction de coût exprimée à travers deux paramètres, *%_time* et *%_access*, qui expriment, pour chaque accès adresse @n sa contribution au total (simulé) le temps d'exécution et le nombre total de accès respectivement.

L'outil repose sur des algorithmes de fouille de données qui sont appliquées à les traces d'exécution T dans lequel seul les événements d'accès à la mémoire définis comme $e = (c, t, d, ts, l)$ sont utiles.

La latence l n'est pas directement disponibles dans nos propres traces. Soit une plate-forme avec des composants matériels $\{f, g, h\} \in C$ qui génèrent des événements $\{e_f, e_g, e_h\} \in E$ avec les relations $e_f \leftarrow\!\!\prec \cdots \leftarrow\!\!\prec e_g$ et $e_f \leftarrow\!\!\prec \cdots \leftarrow\!\!\prec e_h$ (comme ce serait par exemple faire une instructions de multiples accès où e_f est l'événement de l'instruction et e_g et e_h sont les événements d'acquaitance de la mémoire), puis le temps de latence l de l'événement e_f est défini comme $l\langle e_f\rangle = \max(ts\langle e_g\rangle, ts\langle e_h\rangle) - ts\langle e_f\rangle$. En utilisant cette formule, nous avons eu reproduire les résultats originaux, montrant ainsi que les liens qui construisent les chaînes de causalité permettent d'atteindre tous les événements nécessaires pour calculer les informations de latence manquant.

Ralentissement et précision

Il faut des modifications dans le simulateur pour implanter le système de traçage pour capturer, traiter et stocker les événements. Pour vérifier comment elle influe sur la précision de l'ensemble du système, nous avons exécuter différentes expériences.

Le nombre de cycles reste à peu près le même dans les trois cas, la différence étant principalement due à des erreurs d'arrondi lors de la conversion de *ticks* d'horloge à des secondes. Le temps de simulation est évidemment grandement affectée, et la mise en œuvre du simulateur sans traces est beaucoup plus rapide. Par exemple, pour une simulation avec 12 processeurs et l'application MJPEG le ralentissement est de 5 fois. Pour une simulation avec les même nombre de processeurs et pour l'application WaterStatial le ralentissement est de 12 fois. Cela est dû d'une part au temps nécessaire pour capturer, traiter et stocker les événements, ce qui nécessite actuellement au moins un appel de fonction par instruction cible lors de l'exécution. Stocker les événements sur le disque participe également à cette grande ralentissement. Et d'autre part, il est à cause des synchronisations croissantes avec le simulateur de TLM, SystemC dans notre cas.

Les tailles du fichier de traces pour une simulation avec 12 processeurs et les application MJPEG et Ocean sont 44Go et 93Go, respectivement.

Chapitre 5 - Analyse de la cohérence de cache basée sur les traces

Nous proposons une méthode pour identifier les violations de cohérence de cache potentielles en architectures multiprocesseurs non cache cohérentes pour aider les ingénieurs à les trouver et à les résoudre. Ces plates-formes reposent sur un support logiciel pour maintenir la cohérence de cache. Les objectifs de notre contribution prend en compte les hypothèses ci-dessus.

Détecter les problèmes potentiels: le premier objectif est d'identifier les problèmes de cohérence de cache potentiels qui sont présents dans une trace d'exécution donné. Comme nous utilisons des traces comme une représentation du système, la méthode proposée peut être classée comme une approche semi-formel, nous cherchons à identifier toutes les erreurs possibles détectable basées sur les traces d'exécution donnés. Cela ne signifie pas que toutes les erreurs présentes dans le code source sont repérés, mais seulement ceux-là que les traces d'exécution contiennent.

Indépendance du protocole de cohérence: Le deuxième objectif est de définir une méthode indépendante du protocole logiciel cache de cohérence. Cela signifie que les définitions de protocoles logiciels ne influencent pas sur le processus de détection, ce qui permet le procédé de détection pour vérifier également l'efficacité des protocoles eux-mêmes. Par conséquent, nous sommes intéressés à des approches qui analysent l'ordre écriture et lecture à chaque case mémoire.

Détails implicite de l'architecture: le troisième objectif est de maintenir la méthode indépendante du modèle de cohérence et de l'interconnexion, cependant, ces facteurs influencent la cohérence de cache directement. Le modèle de cohérence de la mémoire peut varier d'un modèle séquentiel simple au modèles complexes *out-of-order*, alors l'ordre de chaque accès à la mémoire réelle est déterminant pour identifier correctement le problème. Comme ces facteurs sont importants pour la détection des problèmes, ils ont en quelque sorte à être pris en compte, cependant, ils ne sont pas le facteur clé dans la cohérence de cache l'identification des problèmes. Ainsi, un procédé indépendant de ces décisions d'architecture permet l'exploration d'une configuration différente, sans modifier le procédé de détection de la cohérence.

Pour atteindre tous ces objectifs, nous proposons une méthode qui identifie la violation de cohérence de cache potentielle sur les MPSoC. La méthode repose sur les traces d'exécution obtenus à partir de plates-formes virtuelles obéissant le formalisme décrit précédemment et donnant les contributions suivantes:

(1) Une méthode de détection de violation de la cohérence de cache roman basé sur des traces d'exécution des MPSOC issues de la plate-forme virtuelle. (2) Un ensemble de règles détaillées en utilisant l'ordre des événements pour détecter les violations de cohérence de cache en deux règles d'écriture (*write-through* et *write-back*). (3) Une mise en œuvre fonctionnelle qui fournit des informations utiles aux ingénieurs pour identifier et corriger les problèmes de cohérence. Notre méthode montre l'information qui indique le type de problème lui-même, dont les événements causent le problème et la ligne de code source représente l'événement.

Détections des violations de cohérence de cache

Un système de mémoire est cohérente si la valeur reçu par une instruction de lecture est toujours la valeur donnée par la dernière instruction de écriture à la même adresse.

Considérant les approches de matériel, l'objectif de cohérence de cache est de faire de caches dans les systèmes multicœurs aussi invisible que caches dans les systèmes single-core. Cependant, les protocoles logiciels pour la cohérence de cache sont généralement intrusive et ils peuvent conduire à des problèmes de cohérence lors de la mise en œuvre.

Protocoles logiciels dépendent de la politique de l'écriture de la mémoire cache. Dans la politique *write-through*, une violation a lieu lorsque des données écrit par un processeur dans la mémoire principale, et, après cela, une autre processeur faire une opérations de lecture de données directement à partir de son cache. Cela signifie que le second processeur n'a pas connaissance de la dernière mise à jour de données faite par le premier processeur. La politique *write-back* est plus délicate à traiter parce qu'il y a deux situations qui peuvent susciter des problèmes de cohérence. La première situation est quand un processeur écrit des données dans son cache et après un autre processeur lire les données de son cache ou à partir de la mémoire principale. La dernière version des données existe seulement dans la mémoire cache du premier processeur, le second processeur travaillera avec la mauvaise valeur. La deuxième situation est quand un processeur écrit des données dans la mémoire principale (alors voici la mémoire principale est à jour) mais le seconde processeur lire les données directement à partir de son propre cache sans chercher la dernière version de la mémoire principale, car il ne sait pas que les données a été modifié par le premier processeur.

Politique Write-through

Fondamentalement, la violation de *write-through* qui se passe quand une charge à partir d'un cache ne soit pas la dernière valeur mise à jour de la mémoire, ce qui conduit le processeur pour charger une ancienne valeur de son propre cache. Les règles pour identifier ce problème sont décrites ci-dessous. Les prémisse que nous avons sont $\exists\{S_i, S_j\} \subseteq E$ et $\exists\{i, j\} \subseteq C$.

(1)
1. $S_i \prec L_j$ (il faut que le écriture soit avant le lecture);
2. $\nexists S_k$ tel que $S_i \prec S_k \prec L_j$, dont $k \in C$ (il ne devrait pas y avoir d'autres événements en magasin après le magasin en question et avant la charge en question);
3. $\nexists L_j^*$ tel que $S_i \prec L_j^* \prec L_j$, dont $L_j^* \neq L_j$ (le lecture en question devrait être la première charge délivré par le processeur de sorte qu'il est le premier qui peut causer un problème.);
4. $\nexists L_j$ such that $L_j \leftarrow\!\!\prec A_l$ (le lecture en question ne devrait pas être suivie par un lecture à partir de la mémoire);

Politique Write-back

Le mécanisme *writeback* est plus complexe à analyser en raison de le numéro faible d'accès vers la mémoire. En conséquence, nous vous proposons ci-dessous l'ensemble des règles pour faire face à la politique *write-back*. Considérant les prémisses $\exists\{M_i, L_j\} \subseteq E$ and $\exists\{i, j\} \subseteq C$, nous avons:

(2)
1. $M_i \prec L_j$
2. $\forall k \nexists M_k$ tel que $M_i \prec M_k \prec L_j$

3. $\nexists L_j^*$ tel que $M_i \prec L_j^* \prec L_j$, dont $L_j^* \neq L_j$
4. $\nexists B_i$ such that $M_i \prec B_i \prec L_j$ (Il ne devrait pas exister une écriture B fait par le cache du processeur qui effectue le *modify* en question après cette *modifier* et avant la lecture en question).

Le deuxième ensemble de règles traite des écritures dans la mémoire principale et ont les mêmes prémisses que le premier ensemble de règles. Elle est définie comme suit:

(3)
1. $M_i \prec L_j$
2. $\forall k \nexists M_k$ such that $M_i \prec M_k \prec L_j$
3. $\nexists L_j^*$ such that $M_j \prec L_j^* \prec L_j$, with $L_j^* \neq L_j$
4. $\exists B_i$ such that $M_i \prec B_i \prec L_j$
5. $\nexists A_j$ such that $L_j \leftarrowtail A_j$

Une opération de *writeback*, car il écrit la ligne de cache entier pouvant inclure des valeurs d'adresses pas mis à jour, peut provoquer des violations sur les adresses voisines de l'adresse effectivement modifiée. Règle (4) est un complément de la règle (2) car il gère les opérations d'écritures. Elle est définie comme suit:

(4)
1. $S_i \prec S_j$
2. $\nexists S_k$ such that $S_j \prec S_k \prec S_j$, where $k \in C$
3. $\nexists M_k$ such that $S_j \prec M_k \prec S_j$, where $k \in C$
4. $\exists L_j$ such that $S_i \prec L_j \prec S_j$ and $\exists A_j$ such that $L_j \leftarrowtail A_j$

Élimination des faux positifs

Les règles ci-dessus peuvent conduire à faux positifs, *i.e.* l'identification d'une violation alors qu'aucune violation ne peut avoir lieu. En effet, dans de rares cas, la remise en ordre des événements de cache que nous effectuons en utilisant les opérateurs $\lll$ et $\ggg$ pour identifier les violations potentielles pourrait étirer trop les événements. Pour y faire face, la solution consiste à considérer également l'ordre inverse, *i.e.* déplacer les écritures vers leurs événements postérieurs et les lectures à leur opérations précédente. Par conséquent, lorsqu'une violation est identifié, cette remise en ordre supplémentaire est effectué. Si elle contient la même succession d'événements que la réorganisation d'origine, ce qui signifie que la première condition de chaque ensemble de règles est préservée, alors un problème est détecté vraiment.

Sur le plan opérationnel, nous définissons la fonction ASSERT_ORDER(p,e), avec $p \prec e$ la relation à tester. La fonction effectue les opérations $\ggg p$ et $\lll e$. Si la relation $p \prec e$ est toujours vrai, alors l'ordre global pour $p \prec e$ est valide et il y a une violation effective. Dans ce cas, la violation est signalée et la fonction arrête l'analyse, sinon, il signale qu'il ne peut pas déterminer s'il y a un problème ou non, et l'analyse se poursuit.

Algorithmes de détection

Notre approche est basée sur le parcours d'accepteur DFA qui vérifie les règles définies précédemment. Il est un de ces DFA par adresse. Nous utilisons une structure de données pour conserver les informations pertinentes sur l'accès à toutes les adresses de la mémoire, défini comme $\{a(0), a(1), \ldots, a(n-1)\}$, dont n est la taille de la mémoire et $a(x)$ un tuple d'adresse qui représente l'adresse x. La tuple est défini comme $a(x) = (s, m, e, \theta)$. La signification des éléments est la suivante: s est le dernier événement d'écriture à l'adresse x; m le dernier événement *modify* à l'adresse x; e l'événement courant se produisant à l'adresse x; et θ l'ensemble des composants qui a lu les données à l'adresse x après s.

Le DFA $M = (Q, \Sigma, \Delta, q_0, F)$, avec Q l'ensemble fini d'états, Σ l'alphabet d'entrée, Δ une fonction de transissions, $q_0 \in Q$ l'état initial et $F \subseteq Q$ tous les états finaux acceptés.

Le DFA de détection pour les violations *write-through* met en œuvre les règles (1). Il est défini comme $M_{wt} = (\{\texttt{INIT}, \texttt{ANALYSE}, \texttt{PROBLEM}\}, \{L, S, A\}, \Delta, \texttt{INIT}, \{\texttt{PROBLEM}\})$.

Son entrée est l'événement $e \in E$ tel qu'il obéit à la relation d'ordre global $\prec$ et il n'y a aucun autre événement p à analyser tel que $p \prec e$. On note $\mathcal{E} \subset E$ l'ensemble des événements ayant le même ordre global que e. Les attributs de l'adresse mémoire sont mis à jour tandis que de nouveaux événements sont reçus s, m et θ.

La vérification *write-back* vérifie l'ensemble des règles (2), (3), (4). Ses DFA est défini comme: $M_{wb} = (\{\texttt{INIT}, \texttt{WRITEBACK}, \texttt{MODIFY}, \texttt{PROBLEM}\}, \{M, B, L, A\}, \Delta, \texttt{INIT}, \{\texttt{PROBLEM}\})$. En ce qui concerne la radiation à travers DFA, son entrée est le plus récent événement de cache $e \in E$.

Expérimentations

Nos expériences visent à montrer l'efficacité de notre méthode pour détecter les violations de cohérence. Nous exécutons des applications et un système d'exploitation qui ont été initialement conçus pour être exécuté sur des plateformes de cache cohérente supporté par le matériels sur les MPSoCs sans cette supporte matériel à la cohérence de cache. Ensuite, nous identifions les problèmes potentiels en utilisant notre méthode, nous les corrigeons et nous vérifions en ré-exécutant les programmes. Nous répétons ce processus jusqu'à l'élimination des d'erreurs.

Nous nous concentrons la discussion sur le nombre d'infractions relevées. Comme un programme avec les problèmes de cohérence sera probablement planter avant son exécution complète, nous utilisons notre méthode pour localiser l'écriture et la lecture qui a provoqué une violation. En utilisant cette information, nous procédons les corrections nécessaires. Après cela, nous ré-exécutons le programme pour identifier le prochain point de violation, puis corriger ainsi. A la fin, le "nombre de violations" représente le nombre de modifications nécessaires pour éliminer tous les problèmes de cohérence pour une exécution donnée.

Nous traitons de manière itérative l'identification des violations de toutes les couches logicielles, y compris DNAOS et *newlibc*. Nous ne supprimons les corrections lors de la modification du nombre de processeurs ou de l'application. Ainsi, l'ordre dans lequel les demandes ont été traitées est important, à savoir MJPEG, océan, water-nsquared et water-spatial, que le nombre de violations ne peut se développer dans le système d'exploitation et les bibliothèques. Pour une simulation avec 10 processeurs et la politique *write-through*, nous avons 44 corrections de violations concernant l'application MJPEG. Ces nombre de corrections est divisé entre la newlib et DNAOS avec 16 et 28, respectivement. Dans

les mêmes configurations l'application *waterspatial* a eu besoin 70 corrections partagé entre la *newlib*, DNAOS et l'application avec 16, 32 et 22, respectivement. Le nombre des corrections pour la politique *writeback* est plus importante. Le MJPEG a eu besoin de 60 corrections dans la *newlib*, 190 dans le DNAOS, en totalisant 250 corrections. L'application Ocean a eu encore plus modifications (810) partagés entre *newlib* (60), DNAOS (190) et l'application (560).

Les points de violation sont liés à des opérations de contrôle du traitement parallèle (*locks, mutex et les barriers*) et des variables partagées, comme prévu. De toute évidence, la plupart des variables partagées accède se produisent dans les sections critiques. La cohérence doit être maintenue pour assurer que les valeurs correctes de données à l'entrée et à la sortie des sections. Notre outil a identifié les problèmes de cohérence qui se produisent à ces limites, ce qui pourrait être corrigées.

Limitations

Notre approche a des limites. La première et la plus évidente est que les éventuelles violations sont détectées seulement pour un exécution donnée, ainsi d'autres exécutions pourraient déclencher d'autres violations. Le deuxième inconvénient est dû au fait que les résultats faussement positifs sont identifiés seulement dans une certaine mesure. En fait, certaines situations sont impossibles à évaluer, même après avoir procédé à la réorganisation supplémentaires comme décrit précédemment. Nous ne pouvons pas dire si l'absence de violation est due à la bonne pratique de programmation de le développeur, qui insère invalidations en temps opportun, ou est la conséquence des événements précédents.

Chapitre 6 - Le débogage déterministe réversible basée sur traces

Nous proposons une méthode d'amélioration processus de débogage des MPSoCs qui couvre les objectifs suivants:

Réjeu déterministe: Notre premier objectif vise à reproduire une exécution déterministe basé sur des traces "bien formés". Il y a deux avantages principaux à atteindre cet objectif. Tout d'abord, nos traces d'exécution "bien formés" proposées peuvent être exploitées pour produire une exécution déterministe car ils fournissent les relations entre les événements qui permettent la reconstruction du parallélisme. En second lieu, cette technique permet la reproductibilité d'une exécution en parallèle en conservant la même comportement de la simulation, ce qui permet obtenir la même exécution a chaque redémarrage.

Exécution à l'envers: Notre deuxième objectif vise à fournir une méthode de débogage à l'envers déterministe efficace basée sur des traces "bien formés". L'exécution inverse se compose de calculs défaisant qui sont effectuées par une instruction donnée, ce qui signifie la récupération de l'état précédent du système. Dans les environnements de débogage classiques, le développeur doit redémarrer l'application, configurer les *breakpoints* à ce nouvel emplacement et redémarrer l'application jusqu'à l'atteindre. Même en tenant compte d'un réjeu déterministe ce processus au-dessus est nécessaire. L'aide de l'exécution inverse diminue le nombre de redémarrages.

Intégration au débogueur: Le troisième objectif vise à faciliter la diffusion de la méthode. La construction d'un débogueur à partir de zéro n'est pas souhaitable pour de nombreuses raisons. Premier et plus pertinent est l'effort pour faire face à la manipulation

du code binaire. L'effort d'une telle manipulation peut être considérable, et il n'y a pas besoin de réinventer la roue. Utilisation d'une débogueur open source largement adopté peut faciliter le développement et l'adoption des outils.

Sur la base de ces objectifs, nous proposons les contributions suivantes:

(1) Algorithmes qui permettent réjeu de système basé sur les traces d'exécution issues des plates-formes virtuelles. Ces algorithmes proposent un procédé pour la création d'un graphe d'exécution qui représente le comportement du système pendant l'exécution de logiciels sur une plate-forme matérielle virtuelle. (2) Algorithmes qui effectuent les opérations sur le graphe d'exécution permettant marcher en avance et en arrière et des opérations en ayant la complexité algorithmique au temps linéaire. L'opération marche en avant consiste à modifier des valeurs basées sur les instructions sans vraiment les exécuter. L'opération inverse marche consiste à récupérer les valeurs modifiées par une instruction, emph i.e comme ils étaient avant l'exécution de l'instruction. (3) Un outil qui enveloppe la mise en œuvre de ces algorithmes dans le *gdb*, qui est un débogueur open source largement adopté.

Relations "forward" et "reverse"

Pour simplifier le réjeu basée sur la trace, tous les événements sont sérialisés. A cet effet, la relation de *forward*, noté $e_i \curvearrowright e_j$, est construit comme un ordre total de E étendant $\ll$. Cependant, tous les couples d'événements (e_i, e_j) ne sont pas comparables dans E avec la relation $\ll$. Dans ces cas, ayant $e_i \curvearrowright e_j$ ou $e_j \curvearrowright e_i$ est équivalent en termes de l'état du système et donc déterministe. Dans ce cas, l'ordre total est construit en utilisant arbitrairement une de ces deux possibilités. La relation *reverse*, noté $\curvearrowleft$ est défini comme $e_i \curvearrowleft e_j \leftrightarrow e_i \curvearrowright e_j$.

États des processeurs et mémoires

L'état du processeur maintient la valeur de l'ensemble de ses états de registres internes et les informations sur l'instruction en cours d'exécution. Plus formellement, un processeur $p \in C$ est un tuple $p = (\mathtt{CE}, \{r_0, r_1, \ldots, r_{k-1}\})$ dont $\mathbb{CE}$ représente l'instruction en cours et le r_i les valeurs des k registres de l'architecture du processeur.

Les mémoires ont aussi un état interne. Cependant, tandis que les processeurs ont seulement quelques registres, mémoires ont une énorme quantité de valeurs internes. Pendant l'exécution du logiciel, la mémoire est accessible séquentiellement et à faible densité, de sorte que le suivi de toutes les adresses à la fois est pas nécessaire. Formellement, la mémoire $m \in C$, dans son état initial, est définie par l'ensemble vide $m = \varnothing$. Il est rempli d'éléments lors de l'initialisation et tandis que l'exécution progresse. La totalité de la mémoire peut être consulté, dans ce cas, $m = \{a_0, a_1, \ldots, a_{l-1}\}$, dont l est la taille de la mémoire. Chaque élément $a \in m$ contient la valeur à l'adresse indiquée. Les caches sont transparentes pour le débogage, ainsi leurs événements ne sont pas traités.

Graphe d'exécution

Nous créons un graphe orienté G $(V, A_f\ cup A_r)$ pour l'exécution. Les sommets de V sont des instances de registres du processeur et les modification des éléments internes de la mémoire, notée V_p et V_m respectivement.

Le sommet du processeur V_p est défini comme le tuple $V_P = (p_{id}, R)$ dont p_{id} indique le processeur auquel il appartient et R est l'ensemble des registres modifiés, $\forall r \in R \mid r = (id, d, wb)$. Les deux premiers éléments sont intuitivement l'identification de registre et sa valeur actuelle. Le dernier permet représentant la relation *write before*. Il existe une relation entre deux événements d'écriture sur le même registre tel que e_a est la plus récente écriture avant e_b, a noté $e_a \leftsquigarrow e_b$. L'élément wb en r vise à représenter cette relation, il est donc une référence à le tuple r_{wb} tel que $r_{wb} \leftsquigarrow r$. Cette relation est un raccourci pour accéder à la dernière modification d'un registre donné. Il est utilisé pour l'exécution à l'envers.

Pour les sommets de la mémoire, $V_m = (m_{id}, \mathtt{ref}, \mathtt{addr}, d, wb)$ dont m_{id} est l'identifiant de la mémoire, `ref` est l'élément qui maintient la relation $\mathtt{ref} \leftharpoonup V_m$, ce qui est le lien de causalité au processeur vertex V_p qui a été à l'origine de l'accès, `addr` est l'adresse, d la valeur modifiée et wb a la même signification que précédemment.

L'ensemble A_f contient les arcs représentant la relation $\curvearrowright$. $\forall \alpha \in A_f, (v_i, v_j) \in V \times V \mid \alpha = (v_i, v_j)$ dont v_j est le successeur immédiat de v_i dans G. En d'autres termes, v_i est le $tail(\alpha)$ et v_j est le $head(\alpha)$. Le dernier sommet atteint en G à travers les arcs à a_f est noté $\top$ et le premier $\bot$. A l'inverse, l'ensemble A_r contient les arcs représentant la relation $v_i \curvearrowleft v_j$ qui relie un sommet v_j à son prédécesseur immédiat v_i. Pour cet ensemble, $\top$ est le premier sommet et $\bot$ la dernière.

Les états des composants, le sommet courant et le processeur actuel sont maintenues dans une entité que nous appelons *Oracle*, $O = (v_{cur}, p_{cur}, \{p_0, p_1, \ldots, p_n\}, \{m_0, m_1, \ldots, m_k\})$. Pour simplifier la notation, nous notons m et p l'ensemble de mémoires et de l'ensemble des processeurs présentent dans O. Le instruction actuel `ce` est un sommet du graphe d'exécution.

Les événements de cache d'instruction et de donnée, mais ils ne sont pas directement utilisé pour mettre à jour le état de la mémoire ou du processeur, sont utilisés pour maintenir les liens de causalité.

Exécution en avance et en arrière

Initialisation

Le processus d'initialisation qui met en place les valeurs des ressources internes initiales garantit que l'exécution basée sur les traces commence toujours avec le même état initial que celui de la simulation qui génère les traces. Ces informations ne sont pas disponibles dans les traces, et cela doit être fait avant la lecture du première événement dans le fichier de la trace.

Pour les processeurs, les valeurs initiales des registres internes peuvent être facilement obtenues dans les fiches techniques fournies par le fabricant. Les états des processeurs, $\forall p \in C$, sont initialisés avec eux. Compte tenu les mémoires, ils peuvent contenir des valeurs pré-initialisées à certaines adresses, comme constantes statiques ou code binaire. Les valeurs non initialisées ne sont pas prises en compte lors de l'initialisation, en supposant qu'elles sont nulles à la fois dans le simulateur et dans l'outil de débogage pour garantir le déterminisme. L'initialisation faite par l'exécution de logiciels, tels que les initialisations de variables statiques et globales, sont explicitement présente dans les traces, ainsi pris en compte lors des prochaines phases. D'un point de vue de la mise en œuvre, nous utilisons un tableau Judy pour assurer un accès rapide aux valeurs utilisées.

À partir d'un tableau vide, les entrées des adresses sont affectées au fur et à mesure pendant l'exécution de l'application.

Formellement, nous définissons $r^{\perp}$ et $a^{\perp}$ comme les premiers éléments de registre et l'état d'adresse, respectivement. En ayant $\forall p, m \in C$, nous définissons $p_{\perp} = (\varnothing, \{r_0^{\perp}, r_1^{\perp}, \ldots, r_{n-1}^{\perp}\})$ dont $r_x^{\perp} = (x, d, \varnothing)$, comme l'état du processeur initiale, et $m_{\perp} = \{a_i^{\perp}, a_j^{\perp}, a_k^{\perp}\}$ dont i, j, k sont des adresses initialisées, comme l'état de la mémoire initiale, dont $a_i^{\perp}.wb = a_j^{\perp}.wb = a_k^{\perp}.wb = \varnothing$. Pour mémoire, toutes les adresses absentes de $m_{\perp}$ ont la valeur implicite égal à zéro. Comme observé, $p.\mathtt{ce} = \varnothing$, car il n'existe pas d'instruction en cours d'exécution, et $\forall r^{\perp} \in p_{\perp}$ et $\forall a^{\perp} \in m_{\perp}$ maintient la relation $\varnothing \leftsquigarrow r^{\perp}$ et $\varnothing \leftsquigarrow a^{\perp}$, car il n'existe pas de valeur précédente dans les registres ou la mémoire .

Le code binaire et le configuration des registres adressé dans la mémoire ont leurs valeurs initiales transférées à la mémoire.

Exécution en avance

Pendant l'exécution en avance, si nous parvenons à $\top$, nous construisons sur le la prochaine partie du graphe d'exécution à partir des traces, sinon l'exécution est effectuée directement sur le graphe. Cette exécution sur le graphe se produit comme suit: sommets d'instruction de mettre à jour les processeurs, tandis que les sommets d'acquaitance de la mémoire met à jour la mémoire.

La création du graphe d'exécution traverse tous les événements et crée séquentiellement des sommets et des arcs, sans simuler les composants, ainsi la complexité algorithmique temps est $O(n)$ dont n est le nombre d'entrée événements.

Exécution en arrière

Compte tenu l'état du système et l'opération exécutée actuelle, inverser un exécution consiste à calculer l'état du système tel qu'il était avant l'exécution de l'opération. Dans la pratique, cela consiste à revenir les deltas appliquées dans l'exécution en avant.

L'exécution à l'envers commence à partir de $v \in V$ et marche sur G en suivant les arcs A_r, en restaurant l'état antérieur du système en utilisant les valeurs disponibles dans chaque sommet traitées. Ce processus s'arrête soit quand il atteint le dernier sommet $\perp$, en laissant le système dans son état initial, ou lorsque l'utilisateur demande de le faire en utilisant les installations de débogage (*breakpoints*, *watchpoints*, inverser un seul instruction et ainsi de suite).

L'exécution à l'envers met à jour le Oracle O et les composants en utilisant les informations disponibles dans G. Les opérations d'inversion reviennent complètement les instructions, en tenant compte également de leur accès à la mémoire et tout type d'opérations parallèles qui ont eu lieu sur les autres processeurs. Formellement, étant donné un processeur actuel $O.p_{\approx\diagdown}$ et le sommet courant $O.v_{\approx\diagdown}$, revenant une instruction consiste à marcher sur le graphe G jusqu'à atteindre $v_p \in V \mid v_p.p_{id} = p_{\mathtt{cur}}$, puis la mise à jour $O.p_{\mathtt{cur}} = v_p$. Chaque opération de sommet présente dans le chemin entre ces sommets est récupéré, en revenant ainsi chaque opération effectuée par ces sommets, en mettant à jour des états de composant. Annulation d'une opération d'écriture en utilisant $\curvearrowleft$ implique de traverser beaucoup d'événements potentiellement. Il est accéléré avec $\leftsquigarrow$, qui récupère la valeur précédente en complexité en temps $O(1)$.

Experimentations

Nous avons mis en place un outil de rejouer et de revenir en arrière des applications de façon déterministe en utilisant les traces issue de plates-formes virtuelles des MPSoCs. Nos expériences ont pour objectif de vérifier l'exactitude de l'approche et de comparer sa vitesse d'exécution d'une simulation. Le gain de notre approche dans une session de débogage réelle serait la métrique d'intérêt, mais que cela dépend en grande partie sur le bogue et les compétences de débogage des développeurs, nous ne pouvons pas fournir une évaluation juste pour elle.

Pour illustrer le gain potentiel de l'utilisation de cette approche, nous prenons l'exécution de Water-Spatial, par exemple. Son exécution complète prend environ 1,5 minutes en utilisant 12 processeurs sans générer les traces, et nous pouvons nous attendre qu'elle soit ré-exécuté plusieurs fois pour trouver la source d'un *bug*. Pour fournir un soutien à ré-exécution déterministe, la même exécution prend environ 12 minutes, ce qui est nettement plus longue. Cependant, la trace défectueux est généré une seule fois, et la débogage en avant et en arrière ne nécessite pas de simulations supplémentaires. Au pire, la ré-exécution complète de la trace (la construction de l'ensemble du graphe d'exécution) est 2 à 7 fois plus lent que la simulation, mais la navigation avant et arrière à l'aide du graphique est 1,5 à 5 fois plus rapide.

Nos expériences générer un grand nombre d'événements de bas niveau, par conséquent disque, utilisation de la mémoire et le temps écoulé être évaluée. La taille de la trace pour une simulation avec 12 processeurs est de 20 Go pour l'application MJPEG et 90 Go pour l'Ocean.

Les ressources nécessaires à la création du graphe d'exécution n'est pas négligeable. Le débogage de l'application MJPEG s'exécutant sur 12 processeurs consomme 30 Go de mémoire vif. L'application Ocean nécessite de 110 Go sur les même conditions.

Nous pouvons voir que le traçage encourt une consommation non négligeable des ressources. Cependant, nous pensons que cela peut être amorti grâce aux avantages qu'il apporte au développeur.

Chapitre 7 - Conclusions et Perspectives

L'objectif global de cette thèse est de mettre en évidence l'importance de traces dans le développement de systèmes matériels/logiciels nouveaux grâce à des méthodes de capturer, analyser et déboguer ces systèmes.

La première question porte sur la façon dont les traces d'exécution peuvent réussir à capturer le parallélisme présent dans acs MPSoCs.

Question 1: Comment définir une trace pour capturer le comportement parallèle du logiciel s'exécutant sur une plateforme virtuelle ?

L'utilisation d'un simulateur permet de capturer les relations complexes que nous exprimons comme une définition formelle de traces que nous appelons "bien formés". Nous avons défini des relations fondées sur l'ordre des événements qui apparaissent dans chaque composant. Le résultat le plus intéressant est que les traces "bien formés" contiennent relations de causalité, ce qui ouvre des possibilités de réduire la complexité théorique, et donc d'accroître l'intérêt pratique, de nombreux algorithmes visant à analyser les propriétés des logicielles et matérielles.

Nous proposons une méthode pour produire des traces dans l'utilisation des environnements de simulation DBT. Notre génération de trace, grâce à l'environnement de simulation, ne repose pas sur des méthodes intrusives, comme le code source annotation ou tout type de modification de compilation, résultant en une méthode non intrusive.

Le potentiel de l'applicabilité de ces traces a soulevé deux autres questions et nous a amené à réfléchir à leurs réponses respectives.

Question 2: Comment exploiter les traces pour identifier efficacement les problèmes de violation de cohérence de cache qui apparaissent dans le MPSoC sans support matériel à la cohérence ?

La solution permet aux ingénieurs d'identifier efficacement les violations de cohérence des données dans le cache sur les architectures non cache cohérents. Cette méthode d'analyse non intrusive basée sur traces identifie les problèmes potentiels de cohérence, car il prend en compte que les événements de cache et de la mémoire liés. Elle reste indépendant du protocole logiciel qui garanti la cohérence de données de cache. L'analyse est basée sur les événements bas niveau "bien formé".

Nous avons appliqué notre méthode à différents programmes parallèles supposant mémoire partagée cohérente pour identifier les violations. Nous avons identifié correctement les événements qui ont causé des violations et pourrait les corriger en utilisant les commandes de cache appropriées.

À cet autre niveau d'abstraction, les exécutions non déterministes de rendre le processus de mise au point difficile, ce qui nous a conduit à la dernière question.

Question 3: Comment fournir une méthode de débogage déterministe permettant de revenir en arrière, applicable à du logiciel parallèle ?

Nous proposons une solution qui définit une stratégie d'exécution déterministe en avant et en arrière pour le débogage. Notre méthode propose un débogueur déterministe basé sur des traces, où une seule simulation est nécessaire: celle qui produit les traces. La solution repose sur un graphe d'exécution qui est construit en utilisant des traces. Nous vous proposons une spécification graphique d'exécution qui permet marche rapide vers l'avant et vénèrent directions. Nous avons mis cette approche encapsulé dans un serveur *gdb*, fournissant à l'ingénieur une extension de son environnement de débogage d'habitude.

Comme l'ont démontré nos résultats, nous pouvons obtenir un gain de temps de développement considérable en évitant la re-simulation de l'ensemble de la plateforme.

Perspectives

Cette thèse peut être suivie par plusieurs directions de recherche intéressants dans le débogage des techniques basées sur les traces d'exécution détaillées. Nous classons nos futurs travaux dans deux catégories basées sur le temps: *court terme* et *long terme*.

Ces perspectives visent à améliorer la méthode pour couvrir une large gamme de composants et des architectures qui ne sont pas analysés au cours de cette thèse.

Explorer d'autres architectures: notre méthodes peux être exploiter en considérant d'autres aspects architectural comme le nombre de niveau de caches, d'autres processeurs et d'autres modèles de cohérence.

Diminuer la taille des traces: Nous observons que la taille des traces est l'un des notre talon d'Achille. La compression de traces lors de la capture permet de les rétrécissant qui peut être l'une des solutions possibles.

Diminuer le temps de capture: notre mise en œuvre de la capture de trace fonctionne avec une simulation séquentielle qui impose ralentissement lorsque le système de trace est active. Découpler le remplissage des événements et leur stockage réelle sur le disque peut diminuer le temps de capture.

Les perspectives à long terme sont liés à des améliorations que nous aiderons à généraliser ce travail:

Outils de profilage: les outils de profilage visent à identifier les goulots d'étranglement de performance d'une application donnée. Comme nos traces capturent le comportement d'un logiciel parallèle, les méthodes d'analyse de performance peuvent être proposées pour identifier les problèmes de performance, tels que les *hot-spot*, les *race conditions*, la congestion, la couverture de code, *etc.*

Les outils d'analyse: les outils d'analyse visent à fournir plus d'informations pour éliminer les problèmes. La plupart des problèmes sont liés à des ressources partagées, comme les mémoires partagées et les caches. Par exemple, les *race conditions* est une question au niveau de logiciel qui est une source connue de problèmes pour les développeurs. Nous croyons que le contenu de nos traces permet de résoudre ce problème de manière plus efficace.

Optimisation du graphe d'exécution: La méthode de débogage à l'envers déterministe repose sur un graphe d'exécution coûteuses pour effectuer les opérations. Le graphe consomme réellement une quantité considérable de mémoire vif qui peut être prohibitif pour déployer cette méthode dans les postes de travail de développement ordinaires. Afin de minimiser son impact, une solution pourrait consister à identifier une fenêtre d'exécution pour créer ce graphe d'exécution. D'autres techniques pourraient mélanger l'utilisation de la mémoire vif et le disque pour alléger cette consommation pour stocker ce graphe.

Système d'exploitation complexe et MMU: général OSs comptent sur MMU pour mettre en œuvre la notion de processus. Les informations sur le MMU peuvent faire partie des traces. Comme une perspective, nous avons l'intention de formaliser les événements de MMU, ce qui permet mappé les adresses virtuelles, les adresses physiques et processus en cours. Cette méthode étend l'applicabilité de nos approches de systèmes d'exploitation à usage général.

Glossary

ARM	Advanced RISC Machine
CABA	Cycle Accurate Bit Accurate
CTL	Computation Tree Logic
DBT	Dynamic Binary Translation
DFA	Deterministic Finite Automata
DMA	Direct Memory Access
DSE	Design Space Exploration
DVFS	Dynamic Voltage/Frequency Scaling
ETB	Embedded Trace Buffer
IP	Intellectual Property
IPC	Inter Processus Communication
ISS	Instruction Set Simulator
LTTng	Linux Trace Toolkit: next generation
MSI	Modified-Shared-Invalid
MMU	Memory Management Unit
MPI	Message Passing Interface
MPPA	Massively Parallel Processor Array
MPSoC	Multi-Processor System On Chip
NoC	Network On Chip
OS	Operating System
PC	Program Counter
PSR	Program Status Register
POSIX	Portable Operating System Interface
PV	Programmer's View
PVT	Programmer's View with Time
POTA	Partial Order Simulation Traces
RTL	Register Transfer Level
SCC	Single-chip Cloud Computer
SMP	Symmetric Multiple Processor
SoC	System On Chip
TA	Transaction Accurate
TAP	Test Access Port
TPA	Trace Port Analyzer
TB	Translation Block
TLB	Translation Lookaside Buffer
TLM	Transaction Level Modeling

List of Publications

International Journals

[1] **Marcos Aurélio Pinto Cunha**, Nicolas Fournel and Frédéric Pétrot. Deterministic reversible MPSoC debugger based on virtual platform execution traces, In Design Automation for Embedded Systems, pages 1–17, 2015, Springer ***[Published]***

[2] **Marcos Aurélio Pinto Cunha**, Omayma Matoussi and Frédéric Pétrot. Detecting software cache coherence violations in MPSoC using traces captured on virtual platforms, In ACM Transactions on Embedded Computing Systems, special issue on Languages, Compilers, Tools, and Applied Theory for Embedded Systems. ***[Submited]***

International Workshops

[3] **Marcos Aurélio Pinto Cunha**, Nicolas Fournel, and Frédéric Pétrot. Collecting traces in dynamic binary translation based virtual prototyping platforms, In Proceedings of the 2015 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools, RAPIDO'15, Amsterdan, Netherlands, pages 4, 2015. ***[Published]***

Posters

[4] **Marcos Aurélio Pinto Cunha** and Frédéric Pétrot, Simulation Multiprocesseur rapide avec support de trace pour le rejeu, l'analyse et la rétro annotation du code In Journées scientifiques SEmba, Domaine des Hautannes, Saint Germain au Mont d'Or, 2013 ***[Published]***

References

[1] A. Sangiovanni-Vincentelli et al. “Benefits and challenges for platform-based design”. In: *Proceedings of the 41st annual Design Automation Conference*. ACM, 2004, pp. 409–414. URL: http://dl.acm.org/citation.cfm?id=996684 (visited on 15/07/2015).

[2] H. Goldstein. “Checking the play in plug-and-play”. In: *IEEE Spectrum* 39.6 (June 2002), pp. 50–55. ISSN: 0018-9235. DOI: 10.1109/MSPEC.2002.1005639.

[3] I. Molnar. “Kill the "Big Kernel Lock (BKL)" tree”. In: *Linux kernel mailing list*. May 2008. URL: http://article.gmane.org/gmane.linux.kernel/680445 (visited on 05/07/2013).

[4] I. Molnar and A. van de Ven. “Lockdep Design”. In: *Linux Kernel Documentation*. Aug. 2013. URL: https://www.kernel.org/doc/Documentation/lockdep-design.txt (visited on 05/10/2014).

[5] Mentor Graphics. “Evolution of Debug”. Oct. 2015. URL: https://www.mentor.com/products/fv/multimedia/verification-and-debug--old-school-meets-new-school?cmpid=8626 (visited on 25/10/2015).

[6] A. S. Tanenbaum and H. Bos. *Modern operating systems*. Prentice Hall Press, 2014.

[7] A. Kohler et al. “Low-latency collectives for the intel SCC”. In: *IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE. 2012, pp. 346–354.

[8] F. Dubois. “Une méthodologie de conception de modèles analytiques de surface et de puissance de réseaux sur puce hautement paramétriques basée sur une méthode d’apprentissage automatique”. PhD thesis. Université de Grenoble, 2013.

[9] L. Cai and D. Gajski. “Transaction level modeling: an overview”. In: *First IEEE / ACM / IFIP International Conference on Hardware/Software Codesign and System Synthesis, 2003.* 2003, pp. 19–24. DOI: 10.1109/CODESS.2003.1275250.

[10] G. Latu et al. *Non regression testing for the JOREK code*. en. report. INRIA, Nov. 2012, p. 17. URL: https://hal.inria.fr/hal-00752270/document (visited on 25/01/2016).

[11] D. Hoffman. “Non-Regression Test Automation”. In: *Proceedings of the 26th Pacific Northwest Software Quality Conference*. 2008, pp. 361–370.

[12] L. G. Murillo et al. “Automatic detection of concurrency bugs through event ordering constraints”. In: *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*. Mar. 2014, pp. 1–6. DOI: 10.7873/DATE2014.295.

[13] K. Georgiev, V. Martin and V. Marangozova-Martin. “MPSoC Zoom Debugging: A Deterministic Record-Partial Replay Approach”. In: *2014 12th IEEE International Conference on Embedded and Ubiquitous Computing (EUC)*. Aug. 2014, pp. 73–80. DOI: 10.1109/EUC.2014.20.

[14] L. G. Murillo et al. “Deterministic event-based control of Virtual Platforms for MPSoC software debugging”. In: *2015 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. July 2015, pp. 348–353. DOI: 10.1109/SAMOS.2015.7363697.

[15] K. Pouget et al. “Interactive Debugging of Dynamic Dataflow Embedded Applications”. In: *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International*. May 2013, pp. 345–354. DOI: 10.1109/IPDPSW.2013.23.

[16] J. Zheng et al. “On the value of static analysis for fault detection in software”. In: *IEEE Transactions on Software Engineering* 32.4 (Apr. 2006), pp. 240–253. ISSN: 0098-5589. DOI: 10.1109/TSE.2006.38.

[17] H. AlBreiki and Q. Mahmoud. “Evaluation of static analysis tools for software security”. In: *2014 10th International Conference on Innovations in Information Technology (INNOVATIONS)*. Nov. 2014, pp. 93–98. DOI: 10.1109/INNOVATIONS.2014.6987569.

[18] S. Friederich, J. Heisswolf and J. Becker. “Hardware/software debugging of large scale many-core architectures”. In: *2014 27th Symposium on Integrated Circuits and Systems Design (SBCCI)*. Sept. 2014, pp. 1–7. DOI: 10.1145/2660540.2661013.

[19] A. Sen and V. Garg. “Formal Verification of Simulation Traces Using Computation Slicing”. In: *IEEE Transactions on Computers* 56.4 (2007), pp. 511–527. ISSN: 0018-9340. DOI: 10.1109/TC.2007.1011.

[20] D. Hedde and F. Pétrot. “A non intrusive simulation-based trace system to analyse Multiprocessor Systems-on-Chip software”. In: *22nd IEEE International Symposium on Rapid System Prototyping (RSP)*. May 2011, pp. 106 –112. DOI: 10.1109/RSP.2011.5929983.

[21] M. Loghi and M. Poncino. “Exploring energy/performance tradeoffs in shared memory MPSoCs: Snoop-based cache coherence vs. software solutions”. In: *Proceedings of the conference on Design, Automation and Test in Europe*. Vol. 1. IEEE Computer Society. 2005, pp. 508–513.

[22] M. Loghi, M. Poncino and L. Benini. “Cache coherence tradeoffs in shared-memory MPSoCs”. In: *ACM Transactions on Embedded Computing Systems (TECS)* 5.2 (2006), pp. 383–407.

[23] D. J. Sorin, M. D. Hill and D. A. Wood. “A Primer on Memory Consistency and Cache Coherence”. In: *Synthesis Lectures on Computer Architecture* 6.3 (May 2011), pp. 1–212. ISSN: 1935-3235. DOI: 10.2200/S00346ED1V01Y201104CAC016. URL: http://www.morganclaypool.com/doi/abs/10.2200/S00346ED1V01Y201104CAC016 (visited on 16/09/2014).

[24] Q. Fu et al. “Where do developers log? an empirical study on logging practices in industry”. In: *Companion Proceedings of the 36th International Conference on Software Engineering*. ACM. 2014, pp. 24–33.

[25] D. Domingo and W. Cohen. “SystemTap Beginners Guide”. July 2009. URL: https://sourceware.org/systemtap/SystemTap_Beginners_Guide/index.html (visited on 28/10/2009).

[26] G. Matter. "About DTrace". URL: http://dtrace.org/blogs/about/ (visited on 28/10/2013).

[27] W. Sun and J. Geng. "DTrace the Solaris Doors in Inter-process Communication". In: *International Symposium on Computer Science and Computational Technology, 2008. ISCSCT '08.* Vol. 1. Dec. 2008, pp. 551–553. DOI: 10.1109/ISCSCT.2008.103.

[28] B. Gregg and J. Mauro. *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X, and FreeBSD.* en. Prentice Hall Professional, Mar. 2011. ISBN: 9780137061877.

[29] ARM Limited. "Embedded trace macrocell ETMv1.0 to ETMv3.5: Architecture Specification". Sept. 2011. URL: http://infocenter.arm.com/help/topic/com.arm.doc.ihi0014q/IHI0014Q_etm_architecture_spec.pdf (visited on 05/06/2015).

[30] MIPS Technologies. "EJTAG Trace Control Block Specification". Mar. 2002. URL: http://www.t-es-t.hu/download/mips/md00148a.pdf (visited on 10/06/2015).

[31] A. B. Hopkins and K. D. McDonald-Maier. "Debug Support Strategy for Systems-on-Chips with Multiple Processor Cores". In: *IEEE Transactions on Computers* 55.2 (2006), pp. 174–184.

[32] B. Vermeulen, K. Gooseens and S. Umrani. "Debugging Distributed-Shared-Memory Communication at Multiple Granularities in Networks on Chip". In: *Proceedings of ACM/IEEE International Symposium on Networks-on-Chip.* 2008, pp. 3–12.

[33] B. Vermeulen et al. "Overview of Debug Standardization Activities". In: *IEEE Design Test of Computers* 25.3 (2008), pp. 258–267. ISSN: 0740-7475. DOI: 10.1109/MDT.2008.81.

[34] A. Merkle. "State-of-the-art Multicore Debugging and Tracing concepts". In: *MAD - Multicore Application Debugging Workshop.* 2013.

[35] J. F. Cantin, M. H. Lipasti and J. E. Smith. "The complexity of verifying memory coherence". In: *ACM Symp. on Parallel Algorithms and Architectures.* 2003.

[36] L. Liu et al. "Diagnosing root causes of system level performance violations". In: *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD).* Nov. 2013, pp. 295–302. DOI: 10.1109/ICCAD.2013.6691135.

[37] S. Lagraa, A. Termier and F. Pétrot. "Scalability Bottlenecks Discovery in MPSoC Platforms Using Data Mining on Simulation Traces". In: *Design, Automation Test in Europe Conference Exhibition (DATE).* 2014. DOI: 10.7873/DATE.2014.199.

[38] S. Lagraa, A. Termier and F. Pétrot. "Data mining MPSoC simulation traces to identify concurrent memory access patterns". In: *Design, Automation Test in Europe Conference Exhibition (DATE), 2013.* 2013, pp. 755–760. DOI: 10.7873/DATE.2013.161.

[39] M. Gligor, N. Fournel and F. Pétrot. "Using binary translation in event driven simulation for fast and flexible MPSoC simulation". In: *Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis.* ACM, 2009, pp. 71–80. ISBN: 978-1-60558-628-1. DOI: 10.1145/1629435.1629446. URL: http://doi.acm.org/10.1145/1629435.1629446 (visited on 03/12/2012).

[40] M. Monton et al. "Mixed SW/SystemC SoC Emulation Framework". In: *IEEE International Symposium on Industrial Electronics, 2007. ISIE 2007.* June 2007, pp. 2338–2341. DOI: 10.1109/ISIE.2007.4374971.

[41] T.-C. Yeh, G.-F. Tseng and M.-C. Chiang. "A fast cycle-accurate instruction set simulator based on QEMU and SystemC for SoC development". In: *Proceeding of 15th IEEE Mediterranean Electrotechnical Conference MELECON*. Apr. 2010, pp. 1033–1038. DOI: 10.1109/MELCON.2010.5475901.

[42] H. Guan et al. "SINOF: A dynamic-static combined framework for dynamic binary translation". In: *Journal of Systems Architecture* 58.8 (Sept. 2012), pp. 305–317. ISSN: 1383-7621. DOI: 10.1016/j.sysarc.2012.05.002. URL: http://www.sciencedirect.com/science/article/pii/S1383762112000422 (visited on 04/12/2012).

[43] M. F. Atig et al. "On the verification problem for weak memory models". In: *ACM Sigplan Notices*. Vol. 45. ACM, 2010, pp. 7–18. URL: http://dl.acm.org/citation.cfm?id=1706303 (visited on 13/07/2015).

[44] F. Pong and M. Dubois. "Formal automatic verification of cache coherence in multiprocessors with relaxed memory models". In: *IEEE Transactions on Parallel and Distributed Systems* 11.9 (2000), pp. 989–1006. ISSN: 1045-9219. DOI: 10.1109/71.879780.

[45] R. Komuravelli, S. V. Adve and C.-T. Chou. "Revisiting the Complexity of Hardware Cache Coherence and Some Implications". In: *ACM Trans. Archit. Code Optim.* 11.4 (Dec. 2014), 37:1–37:22. ISSN: 1544-3566. DOI: 10.1145/2663345. URL: http://doi.acm.org.gate6.inist.fr/10.1145/2663345 (visited on 16/12/2014).

[46] K. L. McMillan. "Parameterized verification of the FLASH cache coherence protocol by compositional model checking". In: *Correct hardware design and verification methods*. Springer, 2001, pp. 179–195. URL: http://link.springer.com/chapter/10.1007/3-540-44798-9_17 (visited on 13/07/2015).

[47] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. en. Springer Science & Business Media, Mar. 2013. ISBN: 978-3-662-07964-5.

[48] S. Owre, J. M. Rushby and N. Shankar. "PVS: A prototype verification system". In: *Automated Deduction—CADE-11*. Springer, 1992, pp. 748–752. URL: http://link.springer.com/content/pdf/10.1007/3-540-55602-8_217.pdf (visited on 14/07/2015).

[49] L. Beringer. *LEGO Proof Development System: User's Manual*. May 2006. URL: http://www.lfcs.inf.ed.ac.uk/reports/92/ECS-LFCS-92-211/ (visited on 15/07/2015).

[50] M. Saaltink. "The z/eves system". In: *ZUM'97: The Z Formal Specification Notation*. Springer, 1997, pp. 72–85. URL: http://link.springer.com/chapter/10.1007/BFb0027284 (visited on 14/07/2015).

[51] L. C. Paulson. *Isabelle: A generic theorem prover*. Vol. 828. Springer Science & Business Media, 1994. URL: https://books.google.fr/books?hl=fr&lr=&id=RxlhqG0-cGwC&oi=fnd&pg=PA1&dq=isabelle+theorem+proving&ots=zXINR0nvgl&sig=NeK0D4nn46_Qzjek02OeefYgvn4 (visited on 15/07/2015).

[52] M. Daum et al. "Integration of a software model checker into Isabelle". In: *Logic for Programming, Artificial Intelligence, and Reasoning*. Springer, 2005, pp. 381–395. URL: http://link.springer.com/chapter/10.1007/11591191_27 (visited on 15/07/2015).

[53] S. Berezin. "Model checking and theorem proving: a unified framework". PhD thesis. SRI International, 2002. URL: http://reports-archive.adm.cs.cmu.edu/anon/anon/home/ftp/usr/ftp/2002/CMU-CS-02-100.pdf (visited on 15/07/2015).

[54] C. Prada-Rojas et al. "Observation tools for debugging and performance analysis of embedded linux applications". In: *Conference on System Software, SoC and Silicon Debug-S4D*. 2009.

[55] C. Prada-Rojas et al. *Summarizing Embedded Execution Traces Through a Compact View*. English. 2010. URL: http://mescal.imag.fr/membres/carlos.prada/publications/Paper_S4D-2010.pdf (visited on 05/06/2013).

[56] Y. Lu and W. Li. "A semi-formal verification methodology". In: *4th International Conference on ASIC, 2001. Proceedings*. 2001, pp. 33–37. DOI: 10.1109/ICASIC.2001.982491.

[57] G. J. Holzmann. *The SPIN model checker: Primer and reference manual*. Vol. 1003. Addison-Wesley Reading, 2004. URL: http://www.cin.ufpe.br/~acm/esd/intranet/spinPrimer.pdf (visited on 05/10/2015).

[58] D. Hedde. "Analyse de la consistance mémoire dans les MPSoCs à l'aide du prototypage virtuel". Français. PhD thesis. Université de Grenoble, June 2013.

[59] G. Pokam et al. "QuickRec: Prototyping an Intel Architecture Extension for Record and Replay of Multithreaded Programs". In: *Proceedings of the 40th Annual International Symposium on Computer Architecture*. 2013, pp. 643–654. ISBN: 978-1-4503-2079-5. DOI: 10.1145/2485922.2485977. URL: http://doi.acm.org/10.1145/2485922.2485977 (visited on 07/11/2014).

[60] G. Altekar and I. Stoica. "ODR: Output-deterministic Replay for Multicore Debugging". In: *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*. ACM, 2009, pp. 193–206. ISBN: 978-1-60558-752-3. DOI: 10.1145/1629575.1629594. URL: http://doi.acm.org/10.1145/1629575.1629594 (visited on 07/11/2014).

[61] G. W. Dunlap et al. "Execution Replay of Multiprocessor Virtual Machines". In: *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. 2008, pp. 121–130. ISBN: 978-1-59593-796-4. DOI: 10.1145/1346256.1346273. URL: http://doi.acm.org/10.1145/1346256.1346273 (visited on 07/11/2014).

[62] S. I. Feldman and C. B. Brown. "IGOR: A System for Program Debugging via Reversible Execution". In: *Proceedings of the 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*. 1988, pp. 112–123. ISBN: 0-89791-296-9. DOI: 10.1145/68210.69226. URL: http://doi.acm.org.gate6.inist.fr/10.1145/68210.69226 (visited on 14/08/2014).

[63] H. Agrawal, R. De Millo and E. Spafford. "An execution-backtracking approach to debugging". In: *IEEE Software* 8.3 (May 1991), pp. 21–26. ISSN: 0740-7459. DOI: 10.1109/52.88940.

[64] K. S. Perumalla. *Introduction to Reversible Computing*. en. CRC Press, Sept. 2013. ISBN: 978-1-4398-7340-3.

[65] G. Pothier and E. Tanter. "Back to the Future: Omniscient Debugging". In: *IEEE Software* 26.6 (Nov. 2009), pp. 78–85. ISSN: 0740-7459. DOI: 10.1109/MS.2009.169.

[66] S. Rydh, P. S. Magnusson and B. Werner. *Devices, methods and computer program products for reverse execution of a simulation*. US Patent 7,849,450. 2010.

[67] A. Tolmach and A. W. Appel. "A debugger for Standard ML". In: *Journal of Functional Programming* 5.02 (1995), pp. 155–200. URL: http://journals.cambridge.org/abstract_S0956796800001313 (visited on 07/07/2015).

[68] A. L. Coetzee. "Combining reverse debugging and live programming towards visual thinking in computer programming". PhD thesis. Stellenbosch University, 2015.

[69] S. M. Srinivasan et al. "Flashback: A Lightweight Extension for Rollback and Deterministic Replay for Software Debugging." In: *USENIX Annual Technical Conference, General Track*. Boston, MA, USA, 2004, pp. 29–44. URL: https://www.usenix.org/legacy/events/usenix04/tech/general/full_papers/srinivasan/srinivasan_html/paper.html (visited on 08/07/2015).

[70] S.-K. Chen, W. Fuchs and J. Y Chung. "Reversible debugging using program instrumentation". In: *IEEE Transactions on Software Engineering* 27.8 (Aug. 2001), pp. 715–727. ISSN: 0098-5589. DOI: 10.1109/32.940726.

[71] T. Akgul and V. J. Mooney III. "Instruction-level Reverse Execution for Debugging". In: *Proceedings of the 2002 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. ACM, 2002, pp. 18–25. ISBN: 1-58113-479-7. DOI: 10.1145/586094.586101. URL: http://doi.acm.org/10.1145/586094.586101 (visited on 14/08/2014).

[72] J. Lee. "Dynamic Reverse Code Generation for Backward Execution". In: *Electronic Notes in Theoretical Computer Science* 174.4 (May 2007), pp. 37–54. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2006.12.028. URL: http://www.sciencedirect.com/science/article/pii/S1571066107001946 (visited on 09/07/2014).

[73] J. Yi. "A Case for Dynamic Reverse-code Generation to Debug Non-deterministic Programs". In: *Electronic Proceedings in Theoretical Computer Science* 129 (Sept. 2013). arXiv: 1309.5152, pp. 419–428. ISSN: 2075-2180. DOI: 10.4204/EPTCS.129.27. URL: http://arxiv.org/abs/1309.5152 (visited on 08/07/2015).

[74] Free Software Foundation, Inc. *GDB and Reverse Debugging*. Nov. 2012. URL: http://www.gnu.org/software/gdb/news/reversible.html (visited on 09/07/2015).

[75] Free Software Foundation. *ProccessRecord - GDB Wiki*. June 2013. URL: https://sourceware.org/gdb/wiki/ProcessRecord (visited on 09/07/2015).

[76] G. E. W. Law and J. P. Smith. *System and method for bi-directional debugging of computer*. US Patent 8,090,989, http://undo-software.com, 2012.

[77] Consortium, SoCLib and others. *Projet SoCLib: Plate-forme de modélisation et de simulation de systèmes integrés sur puce*. Technical report, CNRS, 2003. http://www.soclib. fr.

[78] Synopsys Inc. "DesignWare TLM Library". 2015. URL: http://www.synopsys.com/Prototyping/VirtualPrototyping/VPModels/Pages/DW-TLM-Library.aspx (visited on 27/07/2015).

[79] F. Bellard. "QEMU, a Fast and Portable Dynamic Translator". In: *FREENIX*. 2005.

[80] L. Censier and P. Feautrier. "A New Solution to Coherence Problems in Multicache Systems". In: *IEEE Transactions on Computers* C-27 (12 1978), pp. 1112–1118. ISSN: 0018-9340. DOI: 10.1109/TC.1978.1675013.

[81] A. Silverstein. *Judy IV Shop Manual - High Performance Dynamic Array*. http://judy.sourceforge.net/doc/shop_interm.pdf. Jan. 2002.

[82] Synopsys Inc. "Synopsys Virtual Prototyping Solutions for Embedded Software Development". URL: http://www.synopsys.com/Systems/VirtualPrototyping/Pages/default.aspx (visited on 27/02/2014).