Parallel Standard-Compliant SystemC Simulation of Loosely-Timed Transaction Level Models

Gabriel Busnot

N°d'ordre NNT : 2020LYSE1315

THESE de DOCTORAT DE L'UNIVERSITE DE LYON
opérée au sein de
l'Université Claude Bernard Lyon 1

Ecole Doctorale N° 512
InfoMaths

Spécialité de doctorat : Informatique

Soutenue publiquement le 18/12/2020, par :
Gabriel Busnot

Simulation parallèle et conforme au standard SystemC de modèles transactionnels à temps relâché

Devant le jury composé de :

Guérin Lassous, Isabelle	Professeur des Universités	Université Claude Bernard Lyon 1	Présidente du jury
Maraninchi, Florence	Professeur des Universités	Université Grenobles Alpes	Rapporteure
Pêcheux, François	Professeur des Universités	Sorbonne Université	Rapporteur
Andrade, Liliana	Maître de Conférences	Université Grenobles Alpes	Examinatrice
Dömer, Rainer	Professeur des Universités	Université de Californie	Examinateur
Maillet-Contoz, Laurent	Responsable Industriel	STMicroelectronics	Examinateur
Moy, Matthieu	Maître de Conférences	Université Claude Bernard Lyon 1	Directeur de thèse
Sassolas, Tanguy	Ingénieur-chercheur	CEA	Co-Encadrant
Ventroux, Nicolas	Responsable Industriel	Thales	Co-Encadrant, invité

Remerciements

Il me sera difficile de nommer toutes celles et tous ceux qui m'ont soutenu au cours de cette thèse. J'espère néanmoins que chacun se reconnaitra dans l'une des quelques lignes qui suivent.

Je tiens tout d'abord à remercier Matthieu Moy, Maître de conférence à l'Université Claude Bernard Lyon 1, pour avoir dirigé ma thèse avec attention malgré la distance qui le séparait de la capitale. Je remercie également chaleureusement Tanguy Sassolas et Nicolas Ventroux du CEA, List, pour leur encadrement au quotidien et leur optimisme permanent sans lequel je n'aurais sans doute pas mené ces travaux à leur terme.

Un grand merci aussi à Florence Maraninchi de l'Université Grenobles Alpes et à François Pêcheux de Sorbonne Université pour avoir accepté d'être rapporteurs et de consacrer de nombreuses heures à une lecture critique de mon manuscrit. Je remercie par la même occasion le reste de mon jury de soutenance : Liliana Andrade, Rainer Dömer, Isabelle Guérin Lassous et Laurent Maillet-Contoz.

Mais cette section serait largement incomplète sans y remercier tout mon entourage : mes collègues de travail avec qui j'ai pu échanger sur toutes sortes de sujets, du plus technique au plus informel, et qui ont pu m'apporter leur aide à plusieurs reprises ; mes amis, ceux du CEA et d'ailleurs, qui m'ont patiement écouté me plaindre avec plus ou moins de théâtralité trois années durant ; Jason qui a bénévolement relu intégralement mon manuscrit à quelques jours du rendu final ; et enfin ma famille et à ma compagne qui, bien que n'ayant pas la moindre idée de ce que je faisais chaque jour devant mon écran, ont toujours cru en moi et ont eu raison puisque j'écris ces lignes aujourd'hui !

Merci enfin au CEA pour avoir financé ces trois années de thèses de doctorat.

Résumé détaillé

Présentation générales

Cette thèse porte sur la simulation parallèle SystemC. Spécifiquement, elle vise une famille de modèles dit transactionnels à temps relâché. La contribution principale de cette thèse porte sur la conception et le développement d'un noyau de simulation SystemC parallel. Ce noyau permet l'exécution parallèle de modèles SystemC standard après ajout d'annotations spécifiques.

Le manuscrit se divise en 5 chapitres. Le premier chapitre présente des notions générales relatives à SystemC ainsi qu'au parallélisme et à la concurrence en informatique. Le second chapitre présente la problématique de la thèse ainsi qu'un état de l'art des techniques de simulation SystemC parallèle. Le troisième chapitre est le cœur du manuscrit. Il explique les procédés principaux mis en œuvre pour permettre la simulation SystemC parallèle de modèles transactionnels à temps relâché. Le quatrième présente les expérimentation du procédé expliqué chapitre 3. Ces expérimentations mettent en valeur des limitations dans le cas de simulation de logiciels exécutés sous un système d'exploitation tel que Linux. Le cinquième et dernier chapitre analyse les causes de ces observations et propose des solutions spécifiques ainsi que leur évaluation expérimentale.

Chapitre 1

Dans ce premier chapitre, les notions générales de concurrence et de parallélisme qui cohabitent étroitement dans le contexte de la simulation SystemC parallèle sont introduites. Les similitudes et différences parfois subtiles entre ces deux notions y sont illustrées à l'aide d'exemples. Le langage de description matérielle SystemC est ensuite présenté. Le niveau de description RTL, cible initiale de SystemC, est utilisé pour illustrer les bases de SystemC que sont la simulaiton à événements discrets, les modules, les ports, les « channels » ainsi que les processus. Enfin le plus récent niveau de description transactionnel est introduit. Les deux niveaux de précision de gestion du temps « approximatif » et « relâché » sont décrits ainsi que des techniques de modélisation modernes standards telles que l'interface mémoire directe (DMI) et le découplage temporel. Les procédés développés dans ce manuscrit s'emploient à supporter l'ensemble des techniques de modélisation proposées.

Chapitre 2

Le deuxième chapitre débute par un bref tour d'horizon des techniques d'accélération usuelles applicables à la simulation de modèles SystemC. Le choix de la parallélisation entre les processus SystemC au niveau du noyau de simulation est ainsi justifié. Les obstacles qui s'opposent spécifiquement à l'évaluation parallèle de processus SystemC sont alors exposés. En particulier, les problématiques de « thread-safety »[1] et d'atomicité des processus SystemC sont introduites au travers d'exemples. Une évaluation non atomique de processus est ici désignée sous le terme de *conflit*.

La seconde partie du deuxième chapitre est un état de l'art des techniques de simulation SystemC parallèle. De nombreuses approches y sont rapportées : simulation à temps synchronisé ou découplé, simulation accélérée sur GPU ou sur matériel dédié, simulation distribuée ou centralisée, exécution reproductible ou non ou encore simulation RTL ou TLM. Il ressort de cette étude qu'une unique approche supporte efficacement les modèles transactionnels à temps relâché, spécialement ceux faisant usage du découplage temporel et du protocole DMI.

Il s'agit de SCale 1.0, un noyau de simulation SystemC parallèle qui a servi de point de départ aux travaux présentés ici. SCale 1.0 est donc présenté de manière plus détaillée dans la fin du chapitre 2 et ses limitations font l'objet d'une analyse méticuleuse. Il en ressort que bien que présentant des mécanismes théoriquement efficaces, ces derniers inapplicables avec de nombreux logiciels simulés, en particulier ceux exécutés sous Linux. Des problèmes de performance importants se posent également.

Chapitre 3

Ce chapitre présente SCale 2.0, une refonte majeure de SCale 1.0. La plupart des mécanismes de SCale 1.0 y sont remplacés et seule l'approche générale reste comparable : l'instrumentation des accès mémoire simulés dans le but de contrôler les interactions entre processus et de préserver leur atomicité. L'idée principale consiste en effet toujours à garantir que les interactions par accès mémoire entre les processus décrivent un ordre partiel entre ces derniers, c'est à dire qu'il existe un évaluation séquentielle produisant le même résultat que l'évaluation parallèle en cours.

Le chapitre s'ouvre sur la description du flot d'exécution globale d'une phase d'évaluation sous SCale 2.0 : évaluation parallèle de processus suivie d'une évaluation séquentielle dans le cas où certains on été suspendus ; analyse des dépendances entre processus en cas de phase séquentielle et récupération d'erreur par retour en arrière.

Les procédés appliqués lors de la phase parallèle spécifiquement sont ensuite détaillés en commençant par la garantie de non dépendance entre processus lors de la phase parallèle, une propriété fondamentale autorisant de nombreuses optimisations. Une

[1] La *thread-safety* désigne le fait qu'une portion de code peut être exécutée simultanément par plusieurs fils d'exécution sans causer d'incohérence.

politique d'autorisation d'accès mémoire offrant la garantie en question ainsi qu'une implémentation efficace sont ensuite définies.

La phase parallèle peut être suivie d'une phase séquentielle qui est alors décrite : ordre d'évaluation des processus suspendus lors de la phase parallèle, vérification des dépendances entre processus à l'aide de l'enregistrement préalable des accès mémoire, génération de la trace permettant le reproduction de simulation et retour en arrière en cas de violation d'atomicité des processus.

Enfin, le système de gestion des interactions via mémoire partagée est généralisé à tout type d'interaction en considérant chaque partie du modèle simulé comme une ressource pouvant être, au sens large, soit lue, soit écrite, à l'instar d'une adresse mémoire. Cela permet, par exemple, de supporter les interactions par interruptions (timer, inter-processeur, etc.).

Chapitre 4

Le quatrième chapitre est une analyse expérimentale du procédé décrit chapitre 3. Le serveur pourvu de 36 cœurs hébergeant les expérimentations est décrit ainsi que le modèle simulé : une architecture symétrique à mémoire partagée pourvue de 1 à 32 cœurs RISC-V simulés par QEMU. Les benchmarks baremetal[2] de multiplication de matrice parallèle, de détection de contours par filtrage de Deriche et de réseau de neurones convolutif Mobilenet sont présentés, ainsi que leurs versions sous Linux additionnées des benchmarks Blackscholes et Swaptions issues de la suite Parsec.

Un protocole de validation fonctionnelle expérimentatale de SCale 2.0 basé sur un benchmark hautement aléatoire est également exposé. Il illustre la capacité de SCale 2.0 à produire toujours le même résultat lors de l'utilisation de la fonction de reproduction de simulation. Lorsque cette dernière n'est pas activée, le résultat du benchmark de test présente une forte variabilité.

La suite du chapitre présente les performances de simulation offerte par SCale 2.0. Des accélérations atteignant $\times 15$ face au noyau SystemC de référence et de $\times 110$ face à SCale 1.0[3] sont observés.

Les accélérations concernant les applications sous Linux sont en revanche nettement moins satisfaisants puisque toujours inférieurs à 4. Ces résultats sont analysés en détail dans la suite du chapitre. Cette analyse met en valeur divers facteurs tels que le mauvais potentiel de parallélisation lors de la simulation du boot et de l'extinction de Linux. La raison principale réside cependant surtout dans la présence d'un grand nombre de

[2]Logiciel s'exécutant à même le matériel, sans le support d'un système d'exploitation.

[3]Les performances très en retrait de SCale 1.0 sont expliquées par son utilisation dans un contexte très différent de celui dans lequel il a été développé. La fréquence des accès mémoire, notamment est plusieurs ordres de grandeur supérieur dans le cas présent, saturant totalement sont système d'instrumentation et causant cette régression.

conflits qui nécessitent autant de retours en arrière coûteux. Ce constat révèle que malgré la grande efficacité de SCale 2.0 dans la majorité des situations, il existe des schémas d'accès mémoire notamment qui causent de nombreux conflits.

Chapitre 5

Le cinquième et dernier chapitre propose des solutions spécifiques aux problèmes identifiés dans la dernière partie du chapitre 4. La première concerne les partie de la simulation faiblement parallélisables telles que le boot de Linux. Ces parties sont aussi souvent peu intéressantes pour l'utilisateur de SCale 2.0 qui cherche surtout à tester le logiciel qu'il a développé et qui s'exécute sous Linux. Un système de variation de la précision de simulation est proposé. Il permet de choisir dynamiquement le mode de simulation des accès mémoire, permettant notamment de les effectuer directement dans QEMU sans recourir au modèle SystemC. En contrepartie, la simulation parallèle n'est plus possible mais les vitesses de simulations sont alors comparables. Ce mode de simulation séquentielle peu précise mais rapide et déterministe est particulièrement adapté à la simulation des parties du code sans intérêt directe pour l'utilisateur.

La seconde cause majeure de ralentissement est le grand nombre de conflits concentrés dans certaines parties de la simulation et responsables à eux seuls de la détérioration globale des performances. L'étude de l'origine de ces conflits montre que le code provenant du noyau Linux est responsable de la majeure partie des conflits (e.g., support des fautes mémoire et gestion du système de fichiers). Ces conflits sont par nature difficile à anticiper et donc à éviter avec fiabilité. La solution retenue consiste alors à ne pas paralléliser la simulation du code appartenant au noyau Linux. Le niveau de privilège des processeurs simulés est utilisé à cet effet : dès qu'un processeur qui le niveau de privilège minimal *utilisateur*, le processus simulant ce processeur est exécuté séquentiellement jusqu'à ce que son niveau de privilège retombe au niveau utilisateur. La quasi totalité des conflits est alors évitée au prix d'un ralentissement à peine mesurable, la majorité du code simulé étant exécuté en mode utilisateur.

Une nouvelle mesure des performances de SCale 2.0 est enfin réalisée avec ces dernières techniques. Des accélérations d'environ $\times 9$ face au noyau séquentiel de référence ont obtenus pour l'intégralité des benchmarks. La comparaison avec SCale 1.0 n'est pas possible, ce dernier ne supportant pas la simulation de systèmes exécutant Linux.

Publications

- Articles:
  - Gabriel Busnot, Tanguy Sassolas, Nicolas Ventroux, Matthieu Moy. *Standard-compliant Parallel SystemC simulation of Loosely-Timed Transaction Level Models.* ASP-DAC 2020 - 25th Asia and South Pacific Design Automation Conference, Jan 2020, Beijing, China. pp.1-6. <hal-02416253>
  - Gabriel Busnot, Tanguy Sassolas, Matthieu Moy, Nicolas Ventroux. *Standard-compliant Parallel SystemC simulation of Loosely-Timed Transaction Level Models: from baremetal to Linux-based applications support.* VLSI
  - Amir Charif, Gabriel Busnot, Rania Mameesh, Tanguy Sassolas, Nicolas Ventroux. *Fast Virtual Prototyping for Embedded Computing Systems Design and Exploration.* 11th Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools, Jan 2019, Valence, Spain. <10.1145/3300189.3300192>.<hal-02023805>
- Patents:
  - Gabriel Busnot, Tanguy Sassolas, Nicolas Ventroux. *Procédé de simulation parallèle reproductible de niveau système électronique mis en œuvre au moyen d'un système informatique multicœurs de simulation à événements discrets.* FR1911332
  - Gabriel Busnot, Tanguy Sassolas, Matthieu Moy. *Procédé de simulation parallèle reproductible de niveau système électronique mis en œuvre au moyen d'un système informatique multicœurs de simulation à événements discrets.* FR2012150
- Poster:
  - Gabriel Busnot, Tanguy Sassolas, Nicolas Ventroux, Matthieu Moy. *Parallel SystemC Simulation Of Multicore Platforms Running Linux.* ACACES Poster Session, Jul 2018, Fiuggi, Italy.

Contents

Introduction **13**

1 Parallel Computing and SystemC Simulation Background **17**
1.1 Introduction . . . 18
1.2 Concurrency and Parallelism . . . 18
1.2.1 Concurrency . . . 18
1.2.2 Fine-Grained Control Over Concurrency with Coroutines . . . 19
1.2.3 Parallelism . . . 21
1.2.4 Enabling Parallel Computation with Threads . . . 23
1.3 SystemC Modeling and Simulation Overview . . . 26
1.3.1 Virtual Prototyping in the SoC Design Flow . . . 26
1.3.2 The SystemC Modeling Language . . . 27
1.3.3 Discrete Event Simulation of SystemC Models . . . 29
1.4 TLM-2.0 . . . 33
1.4.1 TLM: Abstraction of the Communication Layer . . . 33
1.4.2 Coding Styles in TLM . . . 34

2 Parallel SystemC Simulation: Challenges and Existing Solutions **39**
2.1 Parallel SystemC-TLM Simulation: Problem Statement . . . 40
2.1.1 SystemC Acceleration Strategies . . . 40
2.1.2 Parallelizing SystemC . . . 41
2.2 Existing Approaches . . . 44
2.2.1 Synchronous SystemC Parallelization . . . 47
2.2.2 Time Decoupling . . . 50
2.2.3 SCale 1.0: Runtime Processes Interactions Monitoring . . . 55

3 Proposed Solution for LT-TLM Parallel Simulation **63**
3.1 Overview . . . 64
3.1.1 Simplified Model . . . 64
3.1.2 General Execution Flow . . . 65
3.1.3 `mem_instr` Outline . . . 67
3.2 The Parallel Evaluation Phase . . . 68
3.2.1 Advantages of Zero Dependencies Parallel Phase . . . 68
3.2.2 The Address Monitoring Finite State Machine (FSM) . . . 69
3.2.3 Correct Memory Access Recording Order . . . 74

3.2.4 Efficient FSMs Reset . . . 75
3.2.5 Fast Scalable FSM Storage . . . 77
3.3 The Sequential Evaluation Phase . . . 79
3.3.1 Choosing the Sequential Evaluation Order . . . 80
3.3.2 Asynchronous Dependencies Analysis . . . 81
3.3.3 Simulation Replay . . . 85
3.3.4 Rollback-Based Conflict Recovery . . . 87
3.4 Generalization to Any Shared Resources . . . 94

4 Evaluation of the Proposed Simulation Technique 97
4.1 Experimental Setup and Use Cases . . . 98
4.1.1 The Host Computer . . . 98
4.1.2 Simulated Architecture . . . 99
4.1.3 Simulated Software . . . 100
4.1.4 Metrics and Measurement Protocol . . . 102
4.2 Functional Validation . . . 104
4.2.1 Case Study: the Spinlock-Based Barrier . . . 104
4.2.2 Experimental Functional Validation . . . 108
4.3 Performance on Baremetal and Linux-Based Use Cases . . . 109
4.3.1 Baremetal Performance Evaluation . . . 109
4.3.2 Linux Performance Evaluation . . . 113

5 Full Software Stack Simulation Challenges and Solutions 115
5.1 Introduction . . . 116
5.2 Investigating the Performance of Linux-Based Benchmark Simulation . 116
5.3 Fast Sequential Mode . . . 119
5.3.1 Region of Interest . . . 119
5.3.2 Variable Accuracy . . . 120
5.3.3 Dynamic Scheduling Policy . . . 122
5.4 CPU-Mode-Based Unscheduling . . . 123
5.4.1 Conflicts Study . . . 123
5.4.2 Executing OS Kernel Code Sequentially . . . 128
5.5 Final SCale 2.0 Performance Evaluation . . . 130

Conclusion 135

Bibliography 143

Introduction

This manuscript presents the work that I have conducted at the *Commissariat à l'Énergie Atomique et aux Énergies Alternatives (CEA)* in Saclay (France) during my Ph.D. thesis. I was a member of the *Design Automation & Architecture Laboratory (LECA)*, a laboratory working in the virtual prototyping field, as well as a member of the *Laboratoire de l'Informatique du Parallélisme (LIP)* for the University Claude Bernard Lyon 1. My thesis is the sequel of my *Projet de Fin d'Études (PFE)* — a 6-month internship that concludes masters and engineering degrees — on variable accuracy SystemC simulation. I have carried on my work on this topic during the first months of my thesis and integrated the developed features in the final version of the presented work.

System on a Chip

Electronic System Level (ESL) design and verification is increasingly challenging due to the soaring complexity of Systems on Chips (SoCs) and time-to-market constraints. An SoC is an advanced electronic component that gathers on a single chip a complete computing system. SoCs are to be found in all places where a conventional computer is not suited like, for instance, automotive, aeronautics & aerospace, smartphones, USB devices or Internet of Things (IoT) devices. All these applications present strong integration constraints with respect, for example, to power consumption, area, real-time features, efficiency, reliability, or even electro-magnetic compatibility.

While a regular computer Central Processing Unit (CPU) is composed of a set of identical processing cores connected to external main memory and peripherals, an SoC embeds on a single chip various regular processing cores (e.g., ARM cortex A7 and A15 in the big.LITTLE architecture), memory, communication modules (e.g., Bluetooth, Wi-Fi and cellular networking), accelerators (e.g., Graphical Processing Unit (GPU) and codecs) or security devices (e.g., secured biometric identification). The only limit to SoC architecture is the manufacturing process used to engrave these Intellectual Properties (IPs) and the complexity that the hardware designer can embrace.

Following the path set by Moore's law, SoCs complexity reaches unprecedented levels year after year. With multi-billion-transistor chips integrated even in entry-level devices, new design techniques must help architects grasp this level of intricacy. Thus,

Electronic Design Automation (EDA) tools strive to enable architects to fully exploit the new possibilities offered by each leap in manufacturing techniques. In that respect, abstraction is the sinews of war. SoCs are now provided with configurable IPs viewed as black boxes which can be dropped in larger designs with minimal effort. This helps significantly reducing hardware design cycles.

However, hardware is only one side of a coin that has the software on the other. Software complexity is growing even faster than hardware complexity with the democratization of multi-core architectures and hardware heterogeneity that brings new challenges to software developers. Also, when power and efficiency requirements get too high, complexity tends to shift to software with the use of simpler hardware like non-coherent cache architectures. As a result, EDA tools must also enable faster and easier software development tools.

Simulation in The Design Flow

Developing complex software requires systematic testing which in turn necessitates to run this software on the SoC it targets. Waiting for validated hardware to be available before starting software development is not an option, especially considering that hardware development also takes advantage of feedbacks from software teams. As a result, both processes must take place concurrently through the adoption of the hardware and software co-design workflow.

The industry standard solution to that necessity is Virtual Prototyping (VP). It consists in assembling a software model of the SoC under design to build a simulator able to run the software targeting this SoC. For this technique to be viable, VP must fulfill the following requirements:

Cost: The Virtual Prototype (VP) must be cheap and fast to develop.

Speed: The VP must execute fast enough in order not to slow down the build-test-fix software development cycle.

Accuracy: The VP must provide accurate-enough information needed by the software developer in the early development phases.

Debug: The VP must provide useful information relative to errors caused by the software under development.

Repeatability: Bugs often need to be reproducible to be fixed efficiently.

The Transaction-Level Modeling (TLM) [Ayn09] standard has been developed to fulfill these requirements. It is part of the C++-based SystemC [IEE12] Hardware Description Language (HDL). TLM first enables interoperability, allowing independent actors to provide their IP in the form of a standard black-box model that can be easily

integrated into a bigger VP. In a transaction-level model, only components behavior is modeled as opposed to a Register Transfer Level (RTL) model where components internals are also simulated. This difference gives a significant edge to TLM models when it comes to development cost and speed. It also drastically increases simulation speed at the cost of a moderate yet inevitable loss in accuracy (i.e., speed-accuracy trade-off). In addition to the possibility of hooking up a debugger to the simulated processors, all regular C++ debugging techniques can be applied to a TLM model simulation. Finally, a TLM simulation relies on the co-routine semantics enforced by the SystemC standard as for most HDL, thus providing repeatability.

Still, state-of-the-art simulation techniques are now struggling to keep up with modern hardware complexity and simulation speed tends to shrink inexorably. This is a direct consequence of the standard SystemC simulation kernel provided by Accellera [ScR] being single threaded as a direct implementation of co-routines, thus exploiting a single core of the host machine. At a time where computation power increase relies on more and more cores being fitted into a single chip, simulation techniques can no longer rely on single core performance improvement to keep up with the increasing SoC complexity.

Contributions and Outline Of The Manuscript

This thesis tackles precisely parallel and standard-compliant simulation of TLM models. In particular, parallel simulation must not give up on the co-routine semantics as it would imply harder modeling and potentially non-reproducible bugs. This work especially focuses on a specific type of TLM models: Loosely-Timed models. These are the most abstract but also fastest models by up to two orders of magnitude when compared to the other types of SystemC models (RTL or even Approximately-Timed TLM models). LT-TLM models are particularly challenging to simulate in parallel, which this work strives to achieve.

We propose SCale 2.0, a standard-compliant parallel SystemC kernel that guarantees co-routine semantics preservation and simulation reproducibility as a direct consequence. This work was started after SCale 1.0 [VS16]. We support any TLM model including the Loosely-Timed coding style with the use of the Direct Memory Interface (DMI) protocol. Our technique based on lightweight shared-resources access monitoring has a limited overhead even when used with the fastest Instruction Set Simulators (ISS's) available.

This manuscript is organized as follows: Chapter 1 lays down the bases necessary to understanding the present work: concurrency and parallelism concepts, parallel programming challenges and SystemC simulation principles.

Chapter 2 exposes the challenges of parallel SystemC simulation and the existing solutions described in the literature. In particular, SCale 1.0, the starting point of the present work is introduced in detail in this chapter.

Chapter 3 exposes the core mechanisms implemented in SCale 2.0 that have been published at ASP-DAC 2020 conference in [BSV+20]. These mechanisms include: a lightweight FSM-based shared-resources access granting policy that prevents most process atomicity violations during parallel evaluation; a fast and highly scalable data-structure for FSM instances storage; a general process-level rollback system to recover from process atomicity violations; and a deterministic simulation replay mechanism for bug reproduction.

Chapter 4 describes the experimental setup including the host platform, the simulated platform, the functional validation benchmark, and the performance benchmarks. Experimental results are then analyzed: baremetal benchmarks show very good scaling on a 36-core host but Linux-based benchmark are not efficiently supported at that point.

Consequently, Chapter 5 further analysis the challenges brought by the simulation of complex guest OS simulation and introduces additional solutions including: a variable accuracy system coupled with adaptive parallel/sequential evaluation for simulation fast-forwarding; and a simulated-CPU-mode-based process scheduling algorithm for improved conflict avoidance before giving the final experimental results. The solutions and results presented in this chapter have been submitted to the ASP-DAC 2020 journal and are under second review.

Chapter 1

Parallel Computing and SystemC Simulation Background

1.1 Introduction 18
1.2 Concurrency and Parallelism 18
1.2.1 Concurrency 18
1.2.2 Fine-Grained Control Over Concurrency with Coroutines 19
1.2.3 Parallelism 21
1.2.4 Enabling Parallel Computation with Threads 23
1.3 SystemC Modeling and Simulation Overview 26
1.3.1 Virtual Prototyping in the SoC Design Flow 26
1.3.2 The SystemC Modeling Language 27
1.3.3 Discrete Event Simulation of SystemC Models 29
1.4 TLM-2.0 33
1.4.1 TLM: Abstraction of the Communication Layer 33
1.4.2 Coding Styles in TLM 34

1.1 Introduction

In this chapter, concepts that are necessary to the understanding of the rest of the manuscript are exposed. The primary goal is to illustrate these concepts and to help build an intuition of them. This chapter does not aim at being an exhaustive lecture on these topic but instead attempts to hide the complexity of some aspects of these topics when possible.

First, the general concepts of concurrency and parallelism as used in this document are explained: what are they, what are their similarities but above all, what makes them different and often incompatible? An algorithmic approach is first used to introduce these concepts before presenting how they can be used when programming actual applications. Then the SystemC HDL is introduced through the industrial needs that motivate its use. Next, the fundamentals of SystemC are exposed through a simple RTL use case. Modules, ports, channels, processes, and simulation kernel are defined at that point. Finally, the TLM level of abstraction, the target of this work, is explained as well as DMI and the global quantum, two standard acceleration techniques that we strive to support with our parallel SystemC simulation kernel.

1.2 Concurrency and Parallelism

Concurrency and parallelism are two closely related concepts yet very different. This section aims at defining and differentiating them to better understand the standard SystemC mechanisms and the contribution of this thesis: SystemC parallelization. It should be noticed that SystemC-specific concepts are not used in this section.

1.2.1 Concurrency

In this section, we define a *process*[1] as a succession of related instructions being executed. In the context of computer science, a process consists in a processor fetching and executing coded instructions (e.g., x86 or ARMv8 [Int19; ARM20]) often located in memory to perform the corresponding register manipulations and memory accesses.

It is often assumed that a process is executed *in isolation*, that is without interacting with other processes. As a result, when a process reads several times a given memory location without writing to it in between, it can expect to read the same value again and again. On the opposite, if a process is not running in isolation, unexpected state mutations (e.g., register content modification) can compromise the process validity.

One such case where processes no longer are in isolation occurs when several processes run *concurrently*, that is if they are *in progress* simultaneously. Formally, with process A (respectively process B) executing during the time interval T_A (respectively T_B), then A and B are concurrent if and only if $T_A \cap T_B \neq \emptyset$. In particular, if two

[1] Process is not used in the sens of a SystemC process in this section.

processes A and B are running concurrently, it does not imply that A and B are *doing progress* simultaneously at any time. As a result, concurrency can be achieved on singlecore processors by doing process multiplexing, that is by executing several processes alternately, each for a small period of time.

In practice, concurrency can be observed at many places in a conventional computer system. At the Operating System (OS) level, thousands of processes can coexist simultaneously. One of the major OS roles is then to keep all of these processes isolated from each other while letting them share the available processing time as fairly as possible. At the program level, the developer can exploit the concurrency capabilities of its system and language to implement control flows that rely on multiple call stacks as detailed in Section 1.2.2. At the hardware level, multiple processing units can execute multiple processes concurrently and even simultaneously as developed in Section 1.2.3.

1.2.2 Fine-Grained Control Over Concurrency with Coroutines

Concurrency can be exploited and controlled at the program level using *coroutines* to implement *cooperative multi-tasking*. A coroutine is an independent control flow (i.e. an execution stack) that is resumed and suspended by the application programmer. In each thread, these control flows are active one at a time as they get suspended and resumed by the programmer. When a coroutine suspends itself, it is said that it *yields*.

Coroutines effectively allow a program to be composed of multiple tasks interacting with each-other in a cooperative manner, that is, all tasks are aware of what each other do and of when they do it. Because a coroutine has its own call stack, that is its own context of execution, the terms *context* and *coroutine* can often be substituted.

Coroutine usage is illustrated by Algorithms 1 and 2 describe two different implementations of the Fibonacci sequence generation. The goal is to generate the numbers of the Fibonacci sequence one at a time and to print each value before generating the next: the values cannot be stored in an array before being all printed at once, that is the values generation and printing must happen concurrently. We also assume that the Fibonacci sequence generator is provided by an external source, which means that the printing cannot be inserted in the middle of the Fibonacci generation function. Such apparently artificial constraints are representative of typical SystemC use cases: several complex IP models provided by various vendors as black boxes and assembled together by the end user.

In Algorithm 1, the generator is a classic function that returns a value and takes no argument. It implies that a specific mechanism is needed to memorize the last computed value in order not to return the same value over and over. Here, two values of the Fibonacci sequence are memorized instead of one to speedup the computation of the next value. The printing function is then responsible for printing the successive values returned by the generator.

On the opposite, in Algorithm 2, the generator function does not need to specify what data needs to be persistent as it never returns, and its stack frame is never cleared. Instead, it stores the successive computed values in a global buffer accessible to the printing coroutine. Once a new value is computed, the generator *self-suspends* to pass control to the printing loop which later also self-suspends to pass control back to the generator, and so on.

Algorithm 1 Iterative computation of the n first values of the Fibonacci sequence

```
1:  procedure PRINT(n)
2:      for i in [0..n[ do
3:          print(FIBONACCIGENERATOR())
4:      end for
5:  end procedure
6:  procedure FIBONACCIGENERATOR
7:      static fib0 = 0                          ▷ Local persistent state
8:      static fib1 = 1                          ▷ Local persistent state
9:      ret = fib0
10:     next = fib0 + fib1
11:     fib0 = fib1
12:     fib1 = next
13:     return ret
14: end procedure
```

Algorithm 2 Coroutine-based computation of the n first values of the Fibonacci sequence

```
1:  fib = 0                                  ▷ Global state shared by all coroutines
2:  procedure PRINT(n)
3:      for i in [0..n[ do
4:          print(fib)
5:          resume FIBONACCIGENERATOR
6:      end for
7:  end procedure
8:  procedure FIBONACCIGENERATOR             ▷ Runs in a dedicated coroutine
9:      fib1 = 1                             ▷ Local state persistent across yields
10:     while true do
11:         next = fib + fib1
12:         fib = fib1          ▷ Set global variable to pass the new computed value
13:         fib1 = next
14:         resume PRINT
15:     end while
16: end procedure
```

In that simple example, the coroutine based version seems needlessly complex but it quickly takes the advantage when the persistent state gets more complex, for instance,

in the presence of a lot of data or when the suspend and resume points can change. Without co-routines, the entire state of each task needs to be explicitly saved and restored each time the task suspends and resumes, which is cumbersome, error prone and less flexible. Therefore, coroutines are well suited to programs where several mostly unrelated complex tasks need to run concurrently. Coroutines are typically used in graphical user interface engines. Hardware modeling happens to fall right under this category and SystemC, as most hardware modeling languages, relies on coroutines to model hardware behavior as detailed in Section 1.3.

However, while coroutines are well suited to program concurrent tasks, they are often bad candidates for parallel execution as they specifically expect to run in alternance with one another but never in parallel. The next sections explain what parallelism is, how it is achieved on regular computers and why it does not combine well with coroutines.

1.2.3 Parallelism

While concurrency describes several tasks *in progress* at the same time, parallelism describes several tasks *doing progress* at the same time.

Despite being semantically close, parallelization sets strong additional constraints on the tasks involved. Indeed, with cooperative multitasking, coroutines suspend and resume at deterministic points in the program, making all interactions happen in a well-defined and easily predictable order. When enabling parallelism between tasks, they will inevitably desynchronize without the programmer introducing extra synchronization. In other words, the tasks will progress at different speeds and interactions will happen in an undefined and unpredictable order. When the order of a sequence of interactions between several parallel tasks is not well defined and can cause variations in the program output, this is called a *race condition*.

Race conditions sometimes have no impact on the validity of the program but can also compromise it, as in Algorithm 3. Here, the printing loop is running while the Fibonacci generator loop computes the successive values of the Fibonacci sequence. Unless the printing loop iterates at exactly the same speed as the generator loop, they will desynchronize, and the printed values will not be those of the Fibonacci sequence like on the example Figure 1.1.

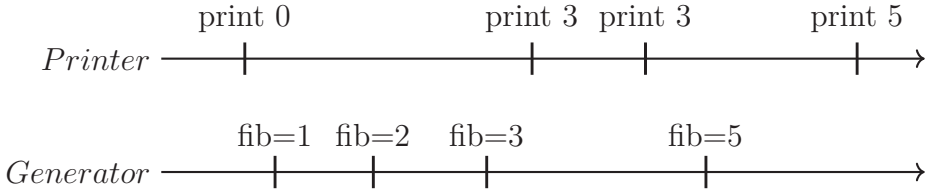

Figure 1.1 – *Possible interleaving when executing the printing and the Fibonacci generator loops in parallel without synchronization like in Algorithm 3.*

Algorithm 3 Parallel computation and printing of the n first values of the Fibonacci sequence with race conditions

```
1: fib = 0
2: Start PRINT and FIBONACCIGENERATOR in parallel
3: procedure PRINT(n)
4:     for i in [0..n[ do
5:         print(fib)
6:     end for
7: end procedure
8: procedure FIBONACCIGENERATOR
9:     fib1 = 1
10:    while true do
11:        next = fib + fib1
12:        fib = fib1
13:        fib1 = next
14:    end while
15: end procedure
```

In order for tasks to run in parallel and interact in a well-defined manner, synchronization must be added by the programmer. Synchronization can take several forms like:

1. *waiting*: A task waits for one or more tasks to complete some specific operations before continuing. Mutual exclusion, barriers or condition variables are common primitives used to wait for various types of events.

2. *atomic operations*: the state on which an atomic operation is performed can either be observed before the operation starts or after it finishes but never while it is happening. Atomic instructions like compare-and-swap or transactional memories [HLR10] enable atomic operations on memory.

In Algorithm 3, additional synchronization is required. A simple approach implemented in Listing 1 would be for example to add a flag specifying if the generator is allowed to write *fib*, the only shared variable: if the flag is true, the generator is allowed to write to *fib* but the printer cannot read it safely and *vice versa* if the flag is false. Both tasks then wait for the flag to have the appropriate value before doing an access to *fib* and flips it once done before continuing to release the other task. In that case, the generator can compute the next value while the printer prints the current one and only needs to wait before updating *fib*. The program is now not only concurrent but also parallel.

Parallelizing a program in practice however involves several technicalities at the software and hardware level that are introduced in the next section.

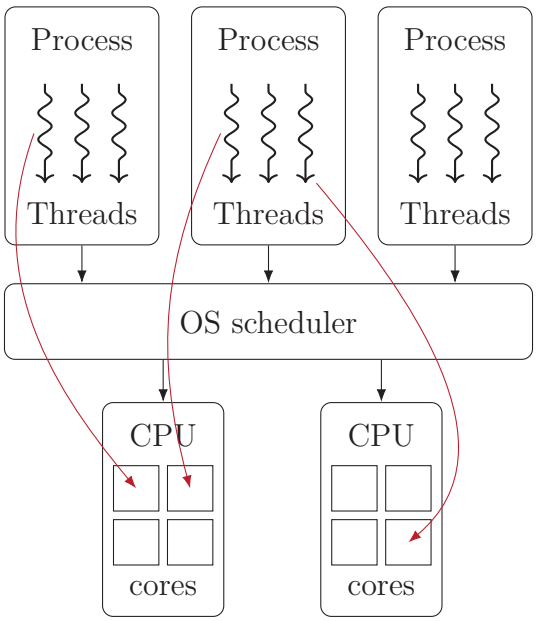

Figure 1.2 – *Organization of a multi-core dual-socket system executing multithreaded processes*

1.2.4 Enabling Parallel Computation with Threads

The notion of thread starts at the language level and percolates down to the hardware layer through the operating systems as illustrated in Figure 1.2. While the semantics of a thread is completely defined at the programming language level, the performance of a multi-threaded program is conditioned by the hardware behavior. This section describes the most important of these aspects regarding the developments of this work.

Threads at the Language Level

Threads of execution are the fundamental concept behind parallel programming. As a coroutine, a thread is an independent control flow in an application but as opposed to a coroutine, threads *can* execute simultaneously. A program that makes use of multiple threads is said to be multithreaded.

In order to program a multithreaded application, the language must first provide the thread construct either natively or via a library. For instance, in C++11 [C] or later, a thread can be spawned using the `std::thread` class. The Listing 1 is a parallel C++ implementation of Algorithm 3 with a flag used to order the accesses to *fib* done by each thread.

Functions `fibonacciGenerator` and `print` are both called in a dedicated thread at lines 28 and 30. The generator thread is then *detached*, that is it becomes independent from the main thread. It allows the program to exit normally without waiting for the generator thread to finish as it will never finish. The printer thread is *joined*, that is the main thread waits for it to finish (i.e., the `print` function returns) before proceeding

Listing 1 C++ parallel implementation of the printing of the Fibonacci sequence.

```
1  #include <thread>
2  #include <atomic>
3  #include <iostream>
4
5  int fib = 0;
6  std::atomic_bool write_fib(false);
7
8  void print(int n){
9      for(int i = 0 ; i < n ; ++i){
10         while(write_fib){} // Wait
11         std::cout << fib << std::endl;
12         write_fib = !write_fib; // Let the generator resume
13     }
14 }
15 void fibonacciGenerator(){
16     int fib1 = 1;
17     while(true){
18         int next = fib + fib1;
19         int next1 = fib1;
20         fib1 = next;
21         while(!write_fib){} // Wait
22         fib = next1;
23         write_fib = !write_fib; // Let the printer resume
24     }
25 }
26 int main(){
27     // Start the generator and let it run independently
28     std::thread(fibonacciGenerator).detach();
29     // Start the printer and wait for it to finish
30     std::thread(print, 8).join();
31 }
```

to the end of the program.

It can be noticed that the `write_fib` flag is of type `std::atomic_bool` instead of `bool`. This is mandatory to avoid *data races*, that is concurrent accesses to a same memory location with at least one write. A data race would here occur between lines 11 and 22: the printer reads `fib` and the generator writes to it. Another data race would also occur on the `write_fib` flag itself. Defining the flag as an atomic boolean instead of a regular boolean solves the issue by constraining the order of execution of the program statements surrounding the atomic memory accesses. Incidentally, when defining `write_fib` as a regular `bool` instead, depending on the optimization level and

on whether gcc or clang is used to compile the program, the printer thread will either print the Fibonacci sequence, print only zeros or fall into an infinite empty loop. This is the result of a data race making the program ill-formed, which basically lets the compiler produce whatever behavior it wants.

Atomic variables are the first step toward the complex and sometimes remarkably counter intuitive C++ memory model defined starting from version 11 upwards. Some brief incursions into the C++ memory model will be required to demonstrate some key aspects of the contributions of this thesis like in Section 3.2.2. Without getting into details, a memory model describes the semantics of memory accesses: it constrains how memory accesses can be reordered and in which order memory writes should appear to the other threads. Indeed, all CPU — or more accurately all Instruction Set Architectures (ISA's) — have their own memory model according to which instructions and specifically memory accesses can be reordered. Fortunately, as long as the program is well defined and the compiler is standard-compliant, the programmer does not need to think about the CPU memory model but only to the (usually simpler) language memory model.

Threads at the System Level

Threads at the program level are supported at the OS level using OS threads. OS threads are basically what the OS scheduler can manage. When a thread is spawned in a process, it leads to the creation of an OS thread that can be scheduled independently from other threads. As a result, all threads become independent tasks executed whenever the OS scheduler decides to. While the scheduler strives to be fair amongst all threads, significant execution time variations between threads can still be observed.

At the hardware level, OS threads are executed by the CPU, specifically, one CPU core can execute one thread at a time[2]. The OS scheduler then decides which thread is executed by which core at what time.

Now that everything is setup to program and run multithreaded parallel applications, one should wonder whether it will be faster than its equivalent sequential version. The answer is far from being always *yes*. Several algorithmic and hardware considerations can make a parallel program slower than its sequential counterpart, among them being:

- Overloading the CPU by spawning many more active threads than there are available cores will result in numerous context switches as a result of the OS scheduler attempting to be fair to all threads.
- Poorly decoupled threads, i.e., threads that constantly wait for each other.
- Frequent non-read-only data sharing causing systematic cache line invalidation on coherent architectures like all mainstream computers.

[2]Simultaneous MultiThreading (SMT) can raise this limit to 2 or more

- False sharing, that is two or more pieces of data that are not shared between several threads but happen to live in the same cache line[3], causing frequent cache line invalidation despite no data being shared in the program.
- Use of inappropriate synchronization primitives.
- Non-Uniform Memory Access (NUMA), which describes a configuration where some memory locations are slower to access than others depending on the core being considered.
- SMT where all logical cores mapped onto the same physical core must share most of its resources. Thus, scheduling two threads on the same core is often not advisable for best performance.

Luckily, the OS scheduler is usually decent enough at dealing with NUMA and SMT to dispense the programmer from thinking about it. The rest is left to the programmer and has been carefully refined in the present work in order for the proposed parallel SystemC kernel to be faster than the sequential reference version.

1.3 SystemC Modeling and Simulation Overview

Now that the general computer science notions useful to the understanding of SystemC and its parallelization are laid down, this section explains the fundamental concepts of SystemC and TLM-2.0 modeling and simulation, as well as their role in the SoC industry.

1.3.1 Virtual Prototyping in the SoC Design Flow

An SoC is an integrated circuit that gathers on a single chip one or more processing units, memory and optionally a GPU, external communication modules and various accelerators. The electronic design industry refers to these components as IPs that are interconnected with either a bus or more recently a Network on a Chip (NoC). SoCs are most often tailored to specific applications requiring various sets of features from computing power to wireless communication capabilities including security, efficient codec decoding or other specialized data processing. In addition, the SoC design industry is increasingly competitive with rapidly evolving technologies and needs. The ability to design an SoC in a reduced amount of time is key to fulfill the ever-tighter time-to-market constraints: typically, a new high-end smartphone chip is released every year with strong timing constraints involving huge market shares.

Designing a new SoC involves two major tasks: the hardware design and validation and the software development. These two tasks are strongly interdependent: the software needs the hardware to be developed and tested and the hardware can require an

[3]Even two adjacent cache lines can cause false sharing on Intel processors for instance.

upgrade if, for instance, the performance of the software cannot meet the requirements on a given hardware.

In order for software to be developed independently from hardware availability, the industry relies on VP. A VP is a software model of the hardware under development able to run the software targeting this hardware. Thanks to VP, the hardware design and software development are decoupled and can happen simultaneously in a process called hardware/software co-design, cutting down on the overall SoC design cycle duration.

However, for a VP to be profitable, it must be fast to develop. This is achieved mostly through IP model reuse. Most IP vendors license there products together with models that can be used to develop VPs. But for models to be compatible with each other out of the box, they must conform to a same standard. One of them is SystemC/TLM-2 [IEE12; Ayn09], a C++ based hardware modeling library and simulation engine. SystemC/TLM-2 offers a collection of constructs suitable for hardware modeling. But more importantly, it specifies the interfaces that IP models must expose so that they can integrate conveniently and reliably with other models inside full SoC models.

Also, VPs must run fast enough not to slow down the *build-test-fix* software development cycle. The main variable of adjustment is the speed-accuracy trade-off: the more details are simulated, the slower the simulation. This thesis aims at providing solutions to accelerate simulations without sacrificing accuracy.

1.3.2 The SystemC Modeling Language

SystemC is a C++-based HDL standardized in 2005 as IEEE Std 1666™-2005 and updated in 2011 as IEEE Std 1666™-2011 [IEE12]. It initially aimed at competing with other HDL such as VHDL [VHD97] or Verilog [Ver91]. It quickly became a popular VP tool by enabling system-level modeling at various and possibly mixed levels of abstraction ranging from cycle-accurate-bit-accurate to loosely-timed TLM and including RTL. This thesis being focused on TLM model simulation, RTL models will only be used in this section to illustrate the general principles of SystemC modeling and simulation.

An SoC is composed of hierarchical hardware blocks that communicate together through various communication channels such a wires, buses or NoCs. SystemC being based on the object-oriented language C++, such modular hardware design naturally translates into class-based modeling. A collection of SystemC classes provide the basic interfaces and features to model each building block of an SoC:

- `sc_module` is the base class for all hardware blocks.
- `sc_port` and `sc_export` are the base classes for block ports, that is interfaces for `sc_modules` to connect to the outer world.

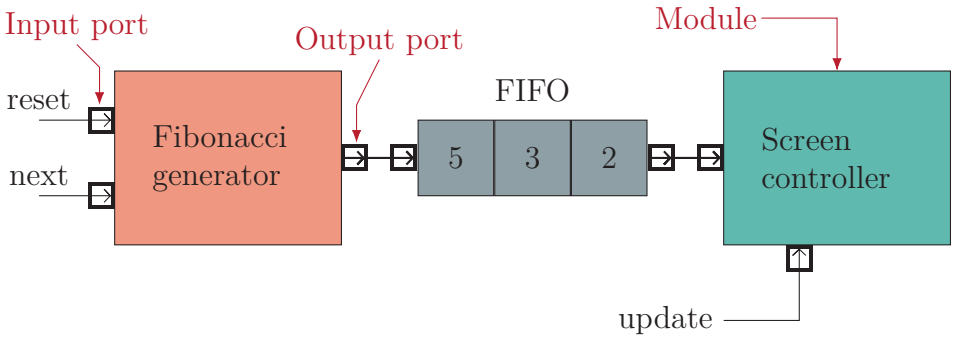

Figure 1.3 – *A simple SystemC model of a hardware implementation of Algorithm 2 with the addition of the control signals reset, next and update.*

- `sc_prim_channel` and `sc_channel` are the base classes for communication channels between blocks and are meant to connect instances of `sc_port` and/or `sc_export` together.

A variety of specializations of these classes are provided to model the most common type of components. For instance, `sc_in`, `sc_out` and `sc_inout` are specialization of `sc_port` that provide specialized interfaces used to read and/or write from/to a channel. Also, `sc_signal` models a simple wire and `sc_fifo` models a fixed capacity queue. Both are specializations of `sc_prim_channel` and can carry a wide variety of data types thanks to template parameterization[4].

The aforementioned classes allow to describe the architecture of a design. The internal logic of the components can then be defined using SystemC *processes*. Processes are special methods in the `sc_module` they belong to. A process can then access and modify all the data contained in its `sc_module` just as a regular method can. SystemC processes can be of two kinds:

- SC_METHOD: A process that always runs from start to end without persistent context from an execution to the next (i.e., a regular function).
- SC_THREAD: A process with persistent context that can suspend and resume (i.e., a coroutine). SC_THREAD processes present a slight memory and speed overhead over SC_METHOD.

To illustrate these concepts, the Fibonacci sequence generator and printer example is modeled in Figure 1.3 using a seven-segment display controller as a printer. If there is some room left in the FIFO and the next signal rises, the generator computes a new value and stores it in the FIFO. The display controller fetches a new value from the FIFO whenever the update signal rises and the fifo is not empty. The Fibonacci

[4]Specific requirements can be imposed on the types used as template arguments like being trivially copiable.

generator module is connected to the reset and next signal using two input ports. Its output port is connected to the head of the FIFO channel whose tail is connected to an input port on the display controller. The display controller also has an input port used to trigger an update of the screen.

The generator behavior must be implemented as an SC_THREAD to preserve the context from one iteration to the next, just as in the Algorithm 2. The display controller only requires an SC_METHOD sensitive to the update signal as it only fetches data from the FIFO and computes the command of the display.

The next section will now describe when and how the processes in a SystemC simulation are executed to simulate the behavior of a real hardware system.

1.3.3 Discrete Event Simulation of SystemC Models

Just as with most HDL, a SystemC model can be simulated using Discrete Event Simulation (DES) [Nan93]. DES models a system as a succession of events occurring at discrete successive time points. As a broad outline, the simulator advances time to the next closest scheduled event and runs the processes *sensitive* to this event in a sequential fashion. These processes may then schedule new events to occur in the future. This sequence repeats until the end time is reached or no more events are scheduled.

SystemC applies this scheme following the execution flow chart Figure 1.4. The platform is first initialized by running all processes once and then starts the main simulation loop. It is composed of these three steps:

1. The *evaluation phase*: all runnable processes are evaluated in a sequential unspecified order. Please note that though unspecified the order is deterministic for a given SystemC kernel with a given SystemC model to ensure reproducible simulation

2. The *update phase*: when accessing an `sc_prim_channel` like an `sc_signal` during the evaluation phase, any modification of the state of the channel is postponed to the following update phase. That way, the visible state of all channels is immutable during each evaluation phase.

3. The *delta or timed notification phase*: the earliest notification time (which is the current time in case of a delta notification) is computed and the corresponding events are triggered, making the sensitive processes runnable during the next evaluation phase.

Processes can be made sensitive to an event either using *static sensitivity* or *dynamic sensitivity*. Static sensitivity is defined during the initialization of the platform using the `sensitive` SystemC command right after declaring a process. Also, an SC_METHOD

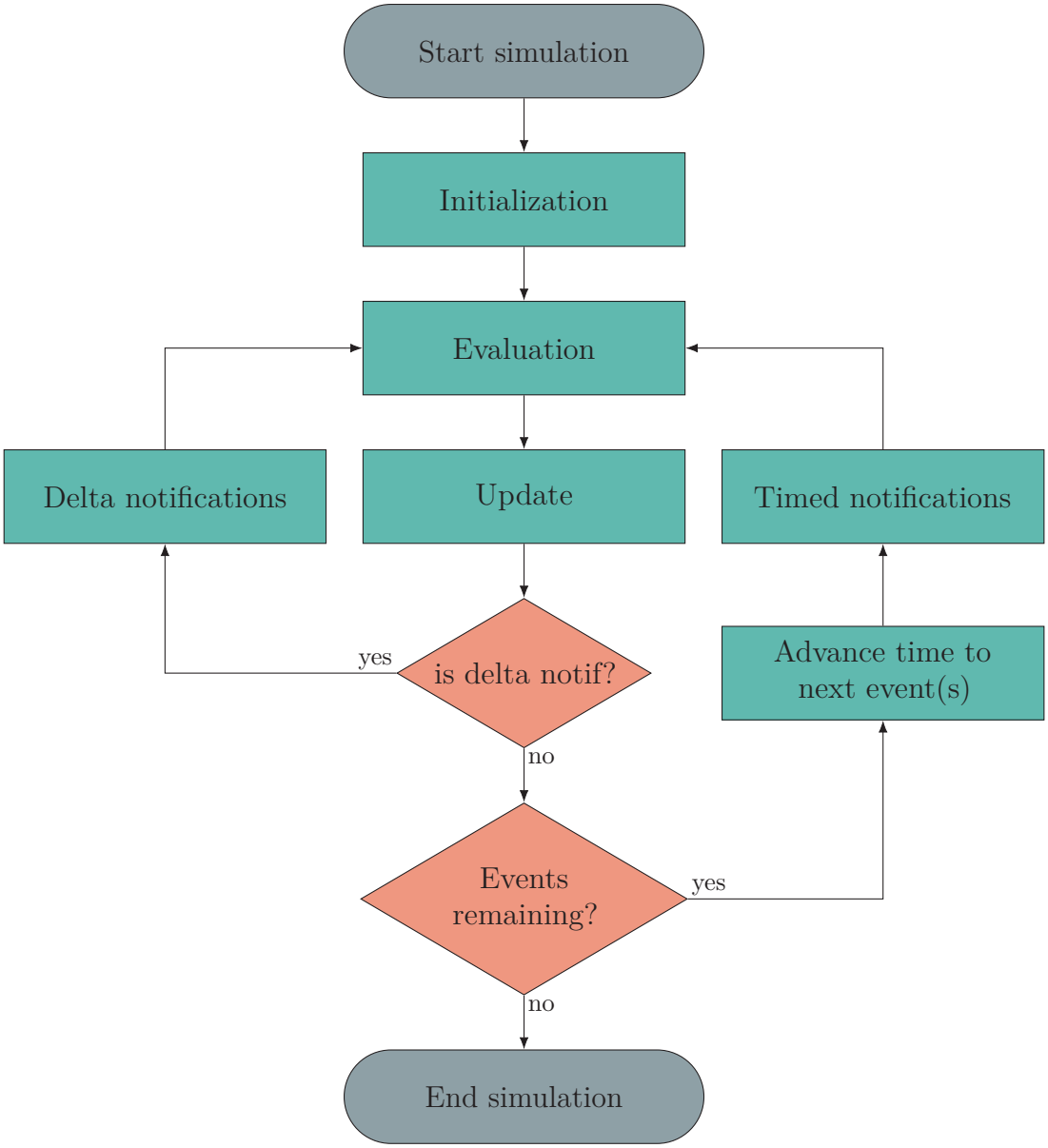

Figure 1.4 – *Flow chart of the reference Accellera SystemC simulation kernel.*

(resp. an SC_THREAD) can be made dynamically sensitive to an event by calling `next_trigger()` (resp. `wait()`) with an event and/or a delay as parameters. For instance, in Figure 1.3, the generator SC_THREAD is statically sensitive to a rising edge on the reset signal and on the next signal and the display controller SC_METHOD is statically sensitive to a rising edge on the update signal.

Let us now assume that we want to simulate the time required for the generator to compute a new value and for the controller to convert an input into a command. In DES, processes evaluation always takes place in a null simulated time. Thus, the simulation accounts for the duration of the simulated tasks between processes evaluation, during the timed notification phase. This is achieved by adding timing information in the processes in the form of `wait(`*`time`*`)` or `next_trigger(`*`time`*`)` statements. These statements will trigger the continuation of the process evaluation after the timing statement.

A complete SystemC model of the generator in the example Figure 1.3 could then be like on Listing 2. It can be noticed that `genThread` is registered as an SC_THREAD and thus is executed as a coroutine with a persistent state (fib0 and fib1 in that case). The events triggered by a rising edge of reset or next are registered in the static sensitivity list of the `genThread` process in the constructor. The time taken by either the reset or the next value computation is accounted for by calling `wait(sc_time)` with the desired amount of time before calling `wait()` without arguments to wait for an input signal rising edge.

Though, modeling at this level of abstraction (RTL) is time consuming and simulations are very slow at a few thousand simulated instructions per second for full system simulation. For fast virtual prototyping and (much) higher simulation speed, the TLM level of abstraction is a better choice.

Listing 2 SystemC model of the Fibonacci sequence generator.

```
#include <systemc>
using namespace sc_core;

// Macro for sc_module definition
SC_MODULE(fibo_gen){
  sc_in<bool> reset, next;
  sc_out<int> out;

  void genThread(){
    // Initialization
    int fib0 = 0, fib1 = 1;
    wait();
    // Infinite loop
    for(;;){
      if(reset){
        fib0 = 0;
        fib1 = 1;
        // Account for reset duration (2ns)
        wait(sc_time(2, SC_NS));
      } else if(next) {
        out = fib0;
        int fib2 = fib1 + fib0;
        fib0 = fib1;
        fib1 = fib2;
        // Account for computation duration (10ns)
        wait(sc_time(10, SC_NS));
      }
      wait(); // wait for reset or next signal
    }
  }
  // Macro for sc_module constructor declaration
  SC_CTOR(fibo_gen){
    // Register genThread as an SC_THREAD
    SC_THREAD(genThread);
    // Register reset and next rising edges in
    // the static sensitivity list of genThread
    sensitive << reset.pos() << next.pos();
  }
};
```

1.4 TLM-2.0

TLM-2.0 [Ayn09] (which will now be referred to as TLM) is a standard built on top of SystemC and released in 2008. It provides an interoperability layer together with utility features to facilitate model development and improve cross compatibility for model reuse. The main goal of TLM is to speed up VP development and simulation speed of memory-mapped bus-based platforms. The present manuscript describes a new parallelization technique targeted at the loosely-timed TLM models.

1.4.1 TLM: Abstraction of the Communication Layer

TLM introduces new constructs for communication modeling. While in classic SystemC, communication between models usually involves signals and protocol simulation, TLM communications rely on interface method calls. When in classic SystemC a module writes data on a channel in order for the module on the other end to receive this data, a TLM-enabled module sends data to another module calling an interface method directly exposed by the targeted module.

This form of memory access simulation is referred to as a *transaction*. It consists in the transmission of a data structure called the *payload* that packs together all the information needed to describe a memory access request. It includes (among other things) the type of memory access (read or write), the targeted address, the length of the access, the status of the transaction and the data being either read or written.

A simple TLM model is illustrated in Figure 1.5. Three types of modules can be

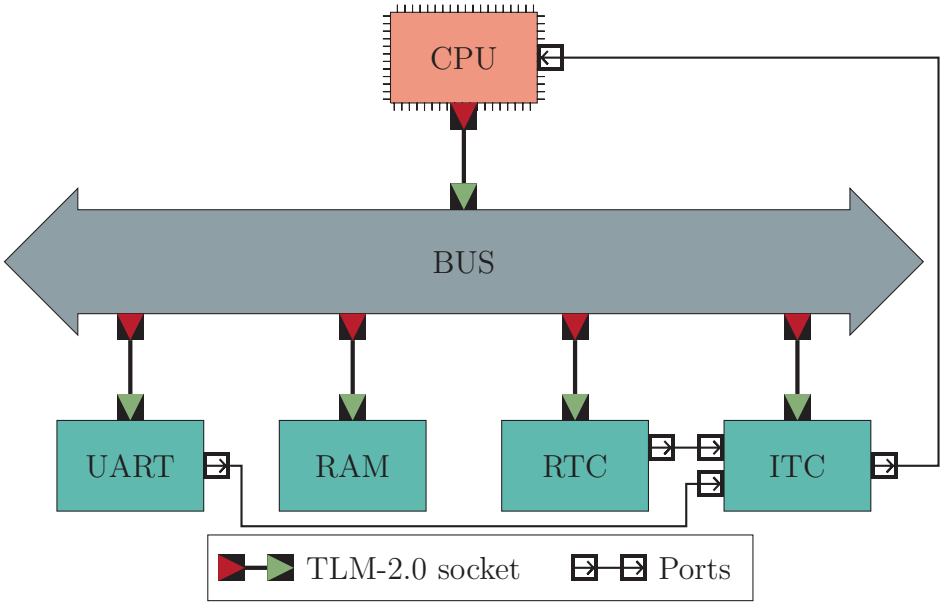

Figure 1.5 – *A basic platform modeled at the TLM abstraction level. A single CPU is connected to a RAM memory, a UART, a real time clock and an interrupt controller. Address based communications rely on TLM while interrupt signals use regular ports and channels.*

distinguished on this example:

- Initiator: a module that actively initiates transactions targeted at other modules. A processor is a typical example of initiator.
- Target: a module that passively responds to transactions initiated by other modules. Memories, peripherals, and coprocessors are often target modules.
- Interconnect: a module that transmits transactions initiated and targeted by and to other modules. Buses and NoCs are classic interconnects.

In order for modules to be compatible with each other, the TLM standard defines a set of standard interfaces called *sockets*. An initiator module exposes *initiator sockets*, a target module exposes *target sockets*, and an interconnect exposes both target and initiator sockets.

As a result, a TLM transaction consists in a generic payload being passed from a module to the next through a succession of interface method calls. Each module on the path of a transaction can alter the payload according to its intended behavior. Once the payload has reached its target, it follows the backward path to the initiator and each module can check and update the transaction status.

1.4.2 Coding Styles in TLM

TLM defines two coding styles: *approximately timed* (TLM-AT) and *loosely timed* (TLM-LT). The former provides greater timing accuracy at the cost of a slower simulation speed. On the opposite, the later gives access to several standard acceleration techniques like the *global quantum* and the *DMI*. Both techniques provide consequent speedup at the cost of a coarser grain timing. The coding styles defined in the TLM standard are only guidelines that the user is strongly encouraged to use. They enforce no specific modeling rules. It is however assumed that these guidelines are followed by the user in the rest of this manuscript.

This work focuses on TLM-LT as it provides the greatest base speed. However, TLM-AT will also be presented as many of the work introduced in Chapter 2 target the TLM-AT coding style.

TLM-AT

Let us begin with the most accurate but also most complex coding style: TLM-AT. Using the TLM-AT coding style, transactions can be split into several phases to better represent the real communication protocol and timing of the simulated system. In particular, the query and answer phases of a transaction, called *forward* and *backward* paths, are modeled in two distinct steps. Two different socket interfaces are used for the two phases of a transaction: `nb_transport_fw` (for non-blocking transport forward) of

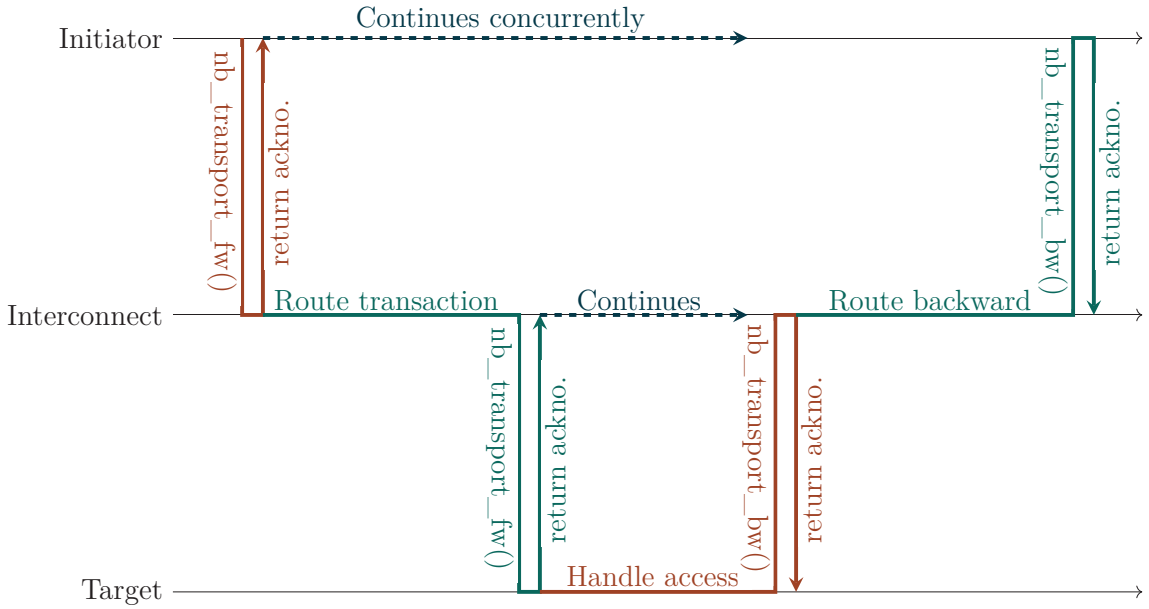

Figure 1.6 – *Chronogram of a non-blocking transaction from initiator to target routed through interconnect.*

target sockets and `nb_transport_bw` (for non-blocking transport backward) of initiator sockets. Such transactions are said to be *non-blocking* as an initiator does not have to wait for a transaction to complete to continue.

A TLM-AT transaction would then follow the lines in Figure 1.6: The initiator starts the transaction using the `nb_transport_fw` interface of the interconnect target socket. The interconnect acknowledges the transaction to the initiator which can then continue to run concurrently. In the meantime, the interconnect routes the payload to the target following the same protocol. Once the target has processed the request, it calls the `nb_transport_bw` of the interconnect and the transaction is routed back to the initiator similarly to the way it got to the target in the first place. After every step of the transaction, i.e., roughly after every acknowledgment, the corresponding simulation time required can be consumed by calling `wait()`. It is important to note that the transaction is handed over from a process to the next by transferring the payload. It allows each process to continue running concurrently to the transaction, thus better representing the real hardware concurrency.

This coding style allows for modeling complex phenomenons such as interconnect contention but limits the simulation speed to only a few Million Simulated Instructions Per Second (MIPS). Also, the complexity of TLM-AT models makes them a rather unusual choices for VP as ease of development is a key feature.

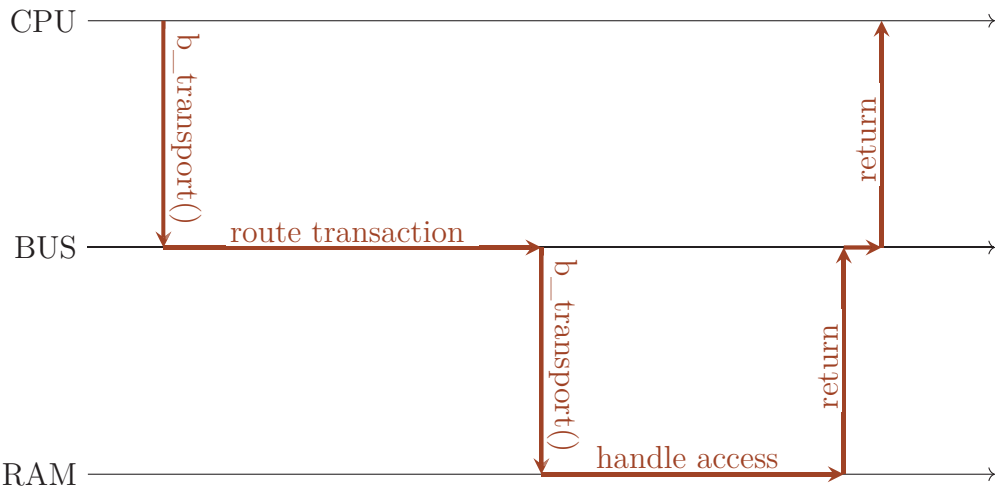

Figure 1.7 – *Chronogram of a blocking transaction from CPU to RAM routed through BUS in Figure 1.5.*

TLM-LT

When using the TLM-LT coding style, a process is allowed to perform several operations and account for the total amount of time they took with a single call to `wait()`. For instance, a complete memory access can be simulated in a single evaluation phase. This helps reducing the number of context switches between processes to accelerate the simulation.

In TLM-LT, transactions are said to be *blocking* and thus use the `b_transport` (for blocking transport) target socket interface. Blocking means that a transaction completes in a single function call and a single evaluation phase. As a result, interconnect and target modules are not allowed to call `wait()` and the entire transaction is executed in the context of the initiator process.

In order to account for the simulated processing time of the various modules on the path of a blocking transaction, a delay variable follows the payload and is updated at every step of the transaction. On the example Figure 1.5, if the CPU performs a blocking transaction toward the RAM, the access will be simulated according to the chronogram Figure 1.7.

Direct Memory Interface

TLM enables DMI accesses. It consists for an initiator to directly access the underlying memory buffer of a target. This protocol is exclusively used for memory components as their only function is to store data in a buffer. It is also much more common in combination with the TLM-LT coding style as it would seriously degrade TLM-AT timing accuracy.

In order for an initiator to access a target with DMI, the target must first provide

a pointer to its internal memory buffer. An initiator knows if a target supports DMI by looking at the *DMI hint* field in the payload after completing a transaction. If the target supports DMI, it should have set it to `true`. The initiator can then query the DMI pointer by sending a specific request to the target using the `get_direct_mem_ptr` target socket interface. The initiator can finally access the memory of the target as a classic C-style array. A rough access time estimation also provided by the target can be used to preserve decent timing accuracy when using DMI.

A target module can revoke DMI permissions by calling the initiator socket interface called `invalidate_direct_mem_ptr`. It can happen in case the target configuration has changed or if the target is an accelerator only providing DMI between computational phases for instance.

Temporal Decoupling

Temporal decoupling[5] is the last standard feature of TLM that aims at speeding up simulations. It is also reserved to the TLM-LT coding style as it favors speed against timing accuracy, too. It allows an initiator to run ahead of global simulation time by a certain amount of time called the *global quantum*.

Temporal decoupling is illustrated Figure 1.8. Without temporal decoupling, all three initiator processes are evaluated alternately for short periods of time usually corresponding to a single transaction. It results in numerous context switches between processes but also with the SystemC simulation kernel. On the contrary, with temporal decoupling, each initiator process evaluates for much longer periods of time (typically 100 to 10.000 times longer depending on the quantum and average transaction durations) until their local time gets greater than the global quantum.

As a result, the number of context switches is reduced by several orders of magnitude which provides consequent speedups. Combined with DMI, simulation speed can reach hundreds of MIPS.

[5]Not to be confused with *time decoupling* which relates to Parallel Discrete Event Simulation (PDES) as explained in Chapter 2

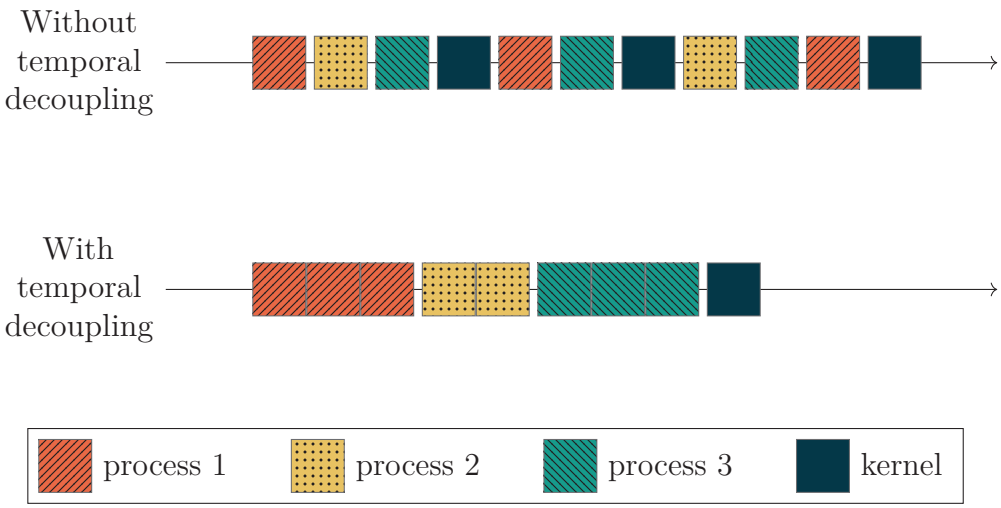

Figure 1.8 – *Illustration of the effect of temporal decoupling on a three process simulation. Context switches happen at every color change and are symbolized by a small gap. Simulation time is only updated during kernel phases.*

Chapter 2

Parallel SystemC Simulation: Challenges and Existing Solutions

2.1 Parallel SystemC-TLM Simulation: Problem Statement 40
2.1.1 SystemC Acceleration Strategies 40
2.1.2 Parallelizing SystemC 41
2.2 Existing Approaches 44
2.2.1 Synchronous SystemC Parallelization 47
2.2.2 Time Decoupling 50
2.2.3 SCale 1.0: Runtime Processes Interactions Monitoring 55

2.1 Parallel SystemC-TLM Simulation: Problem Statement

2.1.1 SystemC Acceleration Strategies

Just as for any software, there are many ways of accelerating SystemC simulations. The most classic way would consist in optimizing the existing code. It most often relies on using better algorithms for expensive computations, reducing the number of dynamic memory allocation, caching the result of heavy computations if they need to be reused, improving memory access patterns, reducing the number of conditional branching, etc. On the model side, such optimizations are different for every single simulator and must be carried out on every model independently. On the kernel side, the reference Accellera SystemC kernel is already well optimized for the general case and there is not much room for improvement.

Simulation speedup can also be achieved through hardware acceleration. Offloading some computations to dedicated resources such as a GPU can potentially yield great accelerations. The most straightforward application would consist in finding processes in the simulation that perform expensive computations and to accelerate them if possible. For instance, the logic of a video decoding IP would probably present a lot of data level parallelism and run much faster on a GPU. However, a general approach to hardware accelerated SystemC simulation is much more difficult. We discuss some of them in Section 2.2.

The two previous approaches do not change the behavior of the model they are applied to. On the opposite, other acceleration techniques require to change the model behavior, usually sacrificing accuracy for speed. The use of TLM, temporal decoupling and DMI, presented in Section 1.4.2, are examples of model transformations that result in significant speedups in the orders of magnitude of up to 10× each. Because these two techniques are now largely adopted in the industry, we want the approach presented in this manuscript to be compatible with them and provide an additional speedup. This, by itself, is a big differentiation factor from most of the other approaches that typically do not support efficiently DMI or temporal decoupling.

Finally, the last classic way of accelerating a program is to parallelize it at the thread level. SystemC, as a Discrete Event Simulator (DES), natively uses a single thread to run the simulation and thus does not take advantage of parallelization. As a result, SystemC parallelization is the most common approach to SystemC acceleration. This is also the approach chosen in this manuscript. However, parallelizing SystemC presents a variety of obstacles that are addressed in the next section.

2.1.2 Parallelizing SystemC

Parallelization opportunities

Not all programs can be parallelized, at least not in a profitable way. Parallelizing a program requires that some of the steps in this program be independent from each other so that they can run simultaneously. Specifically, two tasks can run in parallel if they do not have to *wait* for each other. *Wait* must be understood in every sense. For instance, if a task A needs a result being computed by a task B, A is said to be waiting for B. But also, if A and B attempt to perform two actions that compete on a same resource, either A or B must wait for the other task to complete the action before proceeding. For instance, in C/C++, `{x++;}` evaluated by two threads without additional synchronization causes a data race, leading to undefined program behavior. Whether `x++` is performed under mutex protection or using an atomic `fetch_add` operation, one task will wait for the other to complete at some point. If a task A waits for another task B, A must stop until B has done the expected action, thus eliminating parallelism during that wait.

Also, running a set of tasks in parallel is only beneficial if the tasks are long enough. Indeed, spawning a thread, waking it up or simply passing data to it takes some time that can be significantly longer than the time required to run the task itself. In the case of SystemC, the simulation process can be split between three alternating tasks: the notification phase during which the processes that must run during the next evaluation phase are setup, the evaluation phase during which the model is simulated and the update phase during which channels that have been accessed perform extra actions. Depending on the type of model, the relative length of these tasks varies greatly. For instance, the evaluation and update phases can be of comparable length on a detailed RTL model while the evaluation phase can take up for close to 100% of the time on a TLM-LT model with temporal decoupling enabled.

The main focus of this manuscript being on TLM-LT models simulation, the evaluation phase represents the vast majority of the simulation process and thus is the logical target of parallelization. Though, we will see in Chapter 4 that once the evaluation phase gets fast enough, the other phases represent a non-negligible part of the total simulation. Accelerating them is however not the topic of this manuscript and belongs to the future work.

Parallelizing the evaluation phase could consist in the introduction of parallelism inside SystemC processes themselves. But this would not be a general approach as it would be specific to each model that is not known from the simulation kernel. Indeed, from its perspective, the evaluation phase only is a succession of process evaluations: SC_METHODs and SC_THREADs. The content of these processes is then unknown from the kernel which sees them as black boxes.

Because we want to propose a SystemC acceleration thanks to parallelization, we cannot make assumptions on the content of SystemC processes. As a result, the only

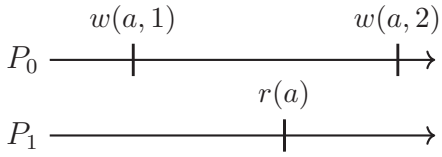

Figure 2.1 – *An example of process atomicity violation caused by concurrent accesses to a same memory location: a is initially 0, P_1 reads a as 1, which would be an impossible value in sequential evaluation.*

parallelism that we can introduce in a SystemC simulation is between the processes of the evaluation phase themselves. Adding intra-process parallelism is more of the user responsibility from our perspective but remains compatible with the proposed approach.

Process Atomicity Violation And thread-safety

The vast majority of SystemC parallelization approaches focus on the evaluation phase for the reasons exposed in Section 2.1.2. Whether it is applied to RTL or TLM models, parallelizing the evaluation phase presents several challenges described in [Döm16; BMC16]. The first and biggest of them is the co-routine semantics of SystemC. It requires that all processes scheduled during a given evaluation cycle be evaluated atomically. The SystemC standard however allows parallel evaluation as long as this co-routine semantics is preserved but it does not give hints about how to achieve it.

TLM is, in itself, very challenging for parallel simulation as it replaces the channel-based (e.g., `sc_signal` or `sc_fifo`) communication between modules with interface method calls, that is classic C++ function calls. When using channels to communicate, the various modules' states and processes tend to remain mostly isolated during the evaluation phase. Each process reads its inputs on a channel and writes its output on another one. The SystemC primitive channels provide a good isolation between the reading and writing processes (constant values are read until next update phase). On the other side, TLM causes much more state sharing between processes because of the shared slave TLM modules, especially the shared memories. Indeed, if, for instance, two processors simulated by two different SC_THREADs access the same shared memory, they might, during a given evaluation phase, both access the same address inside this memory and cause an atomicity violation like in Figure 2.1. In that example, the process P_1 reads a value that only exists in the middle of the evaluation of P_0. If the evaluation of P_0 was atomic, like required by the SystemC standard, P_1 would not have been able to read this value.

Most solutions presented in Section 2.2 try to preserve the co-routine semantics at the cost of various restrictive assumptions. The main assumption is that the different SystemC processes are isolated, that is they do not share data while they are being evaluated or they only use the allowed communication media. For instance, only signals or TLM sockets can be used for cross-process communication depending on the

approach. The event notification policy can also be constrained to provide guarantees that can be exploited by the parallel simulator.

Also, when dealing with classic TLM models, thread-safety is usually not guaranteed as these models are designed with a sequential mindset, that is expecting they will never run simultaneously with other processes. This implies that evaluating the processes of a TLM model in parallel will most likely cause various data races making the outcome of the simulation undefined. For instance, all processes that use a same DMI pointer are susceptible to cause data-races when using it if evaluated in parallel. The fact that all C++ constructs like global variables are allowed in SystemC makes it even harder to ensure even only thread-safety, let alone processes atomicity. Protecting shared resources with critical sections (e.g., mutexes) helps with thread-safety but does not ensure process atomicity and severely hampers performance in most cases.

To make matters worse, TLM-2.0 enables temporal decoupling, allowing processes to run for much longer periods of time during each evaluation phase. This multiplies the risk of process atomicity violation like in Figure 2.1. Indeed, a single process can typically perform hundreds of memory accesses per evaluation phase under temporal decoupling.

Synchronized Process Scheduling

Another threat to parallelization especially present in TLM models is the lack of natural synchronization between processes, that is the fact that very few processes are scheduled at the same date during a simulation. As a result, little parallelism seems achievable in a TLM simulation as opposed to an RTL simulation where most processes are synchronized on a clock.

Indeed, at first sight, only processes scheduled during the same evaluation cycle are good candidates for parallel evaluation. For instance, if a SystemC scheduler was to evaluate in the same evaluation phase two processes P_1 and P_2 scheduled at two different time points $T_1 < T_2$, an event at time $T_3 \in [T_1, T_2[$ could be notified by P_1. A process P_3 sensitive to that event would then be evaluated later than P_2 with respect to the wall clock time, despite being notified by an earlier event with respect to the simulation time. If the evaluation of P_3 influences the evaluation of P_2, that is if P_2 depends on P_3, then this scenario is a causality violation, which is of course prohibited by DES principles. In [BMC16], the authors show that most of the time, less than two processes can be scheduled simultaneously in a classic TLM model. This constraint, however, is tackled by PDES (Section 2.2.2). Because the speedup is bounded by the number of processes that can run in parallel, a way must be found to raise this number.

Fast CPU Simulation

Except maybe for some embarrassingly parallel problems, parallelizing a program always incurs some overhead, that is each part of the problem individually takes longer when processed in parallel to others. This overhead must then be compensated by a

good enough parallel implementation to achieve an overall speedup. The very first source of overhead lies in synchronization, that is when two threads access a shared resource. For instance, due to hardware optimizing features such as memory caching, the more threads one uses to increment a same variable N times, the longer it takes. Also, the higher the access frequency to a shared variable, the slower each access tends to get.

Anticipating a bit on the core of this manuscript, we parallelize SystemC processes by monitoring their interactions to preserve their atomicity. If such interactions monitoring incurs a lot of cross-thread communication and synchronization, the speedup will suffer for the reason previously mentioned. In the case of process interactions monitoring, the amount of cross-thread communications is correlated with the number of simulated CPU instructions since the SystemC model activity is mostly driven by loads and stores initiated by the simulated CPU and the more there are, the more interactions between processes occur.

CPU are simulated using ISS's which have gotten very fast in the recent years, executing millions of simulated memory accesses per second. For instance, QEMU [Bel05] reaches speeds above 1000 MIPS using a single core of the host machine [CBM+19]. The memory access simulation on the SystemC side then must be extremely fast and typically involves a couple of very short function calls thanks to DMI. Monitoring such a short and frequent event could rapidly lead to huge parallelization overhead. The instrumentation technique must then be extremely optimized in this context compared, for instance, to a model that do not use DMI.

Having highlighted the major challenges of SystemC parallelization, Section 2.2 presents the existing work related to parallel SystemC simulation and shows that many of these issues remain unsolved.

2.2 Existing Approaches

To this day, all attempts to parallelize SystemC simulations have made some restrictive assumptions. They are usually related to the abstraction level of the models that can be efficiently simulated in compliance with the SystemC semantics and, by extension, to the type of communications used inside these models. A summary of the presented solution is listed in Table 2.1 as well as a general classification in Figure 2.2. Some solutions require `sc_prim_channel`-based communications only between processes (most commonly `sc_signal`) to preserve decoupling. Other solutions targeting higher levels of abstraction prefer the message-passing paradigm enabled by channels like TLM sockets. Communication using shared variables is rarely supported in a standard-compliant way and often compromises simulation repeatability if not the simulation validity altogether.

Channel-based communication often implies that the simulated design can be split in subsystems called *partitions* and that will run in parallel like in Figure 2.3. Reducing

Table 2.1 – *List of the related works presented in Section 2.2 with pointers to the pages where they are detailed.*

References	Authors	Tool name	Main characteristics	Page
[CCZ06]	Chopard et al.	N/A	Distributed synchronous simulation and centralized time computation.	47
[SLP+10; SWL+14]	Schumacher et al.	parSC, legaSCi	Processes mapped to workers and evaluated in parallel. Resource protection through containment zones.	48 and 49
[VPS+14]	Ventroux et al.	RAVES	Dedicated manycore for process evaluation and SystemC kernel hardware acceleration.	49
[VCB+12]	Vinco et al.	SAGA	GPU acceleration with process duplication for better thread decoupling.	50
[HLH+09]	Hao et al.	ArchSC	Distributed simulation using the ArchSim framework. Enables time decoupling.	51
[VPG06; MMG+10]	Viaud et al. & Mello et al.	TLM-DT	Each partition has its local time that it updates upon communication with other partitions.	52
[WML+16; WLA+16]	Weinstock et al.	SystemC-Link	Communication delay between partitions is constrained to enable PDES.	52 and 53
[CHD12; LSD16; SLD17; SCD18; CAD20]	Chen et al. & Liu et al. & Schmidt et al. & Cheng et al.	RISC	Compiler based processes dependencies analysis to issue parallel process evaluation scheduling.	53 and 54
[Bec17]	Becker et al.	DystemC	Multi kernel simulation relying on Kahn process networks for communication.	54
[VSV+16]	Virtanen et al.	IPTLM	TLM protocol adapted to use shared memory for multi-process simulation.	54
[VS16; BSV+20]	Ventroux, Sassolas & Busnot et al.	SCale	Shared resource access monitoring for process atomicity enforcement.	55

the amount of communication between these partitions is very important as it limits the frequency of time consuming synchronizations. However, not all models can be partitioned in a sensible manner. For instance, systems based on a central shared memory will see the partition containing the memory permanently interacting with the others and act as a bottleneck, slowing down the whole simulation as a result.

Despite these common characteristics, parallel SystemC solutions differ by many aspects: multi-thread, multi-process, distributed on several hosts, hardware accelerated, single kernel, multi kernel, static code-base analysis driven or even introduction of new semantics on top of SystemC. Yet, a fundamental difference between two solutions lies in the way they manipulate the simulation time: synchronously or in a decoupled fashion.

Synchronous simulation (Section 2.2.1) is well suited to RTL models as they tend to be synchronized with a central clock, thus offering a lot of parallelism in each evaluation phase. However, with TLM, time decoupling[1] (Section 2.2.2) is better suited to allow processes scheduled at close-enough time points to run in parallel. Similarly, temporal decoupling can also be exploited to restore process simultaneity but only SCale 1.0 [VS16] presented in Section 2.2.3 and the work presented in this manuscript

[1]Not to be confused with *temporal decoupling*.

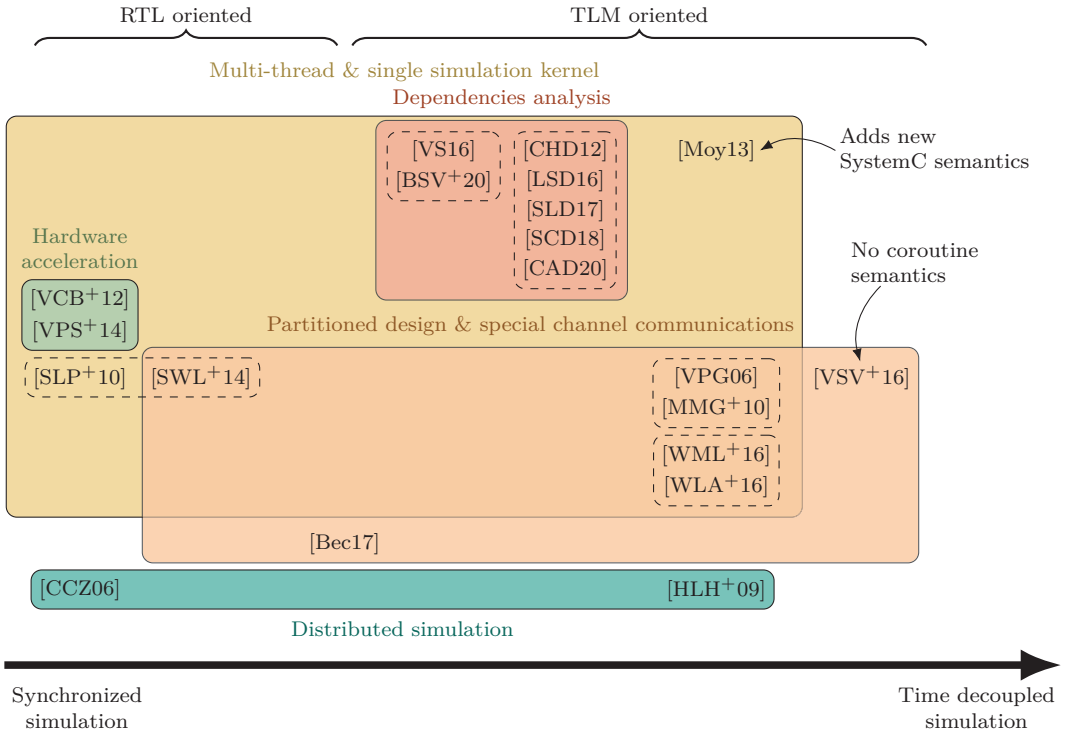

Figure 2.2 – *Positioning of existing works on parallel SystemC simulation. All references in a same dashed frame relate to a same tool or to tools developed by the same research team. The main discriminant represented on the horizontal axis is the level of synchronization of the simulation that mostly correlates with the targeted level of abstraction. The resulting order is not strict but rather represents a general progression as relaxed synchronization is implemented in different manners by the various papers.*

that has been first introduced in [BSV+20] deal with some TLM-LT specific issues like the tight coupling of processes.

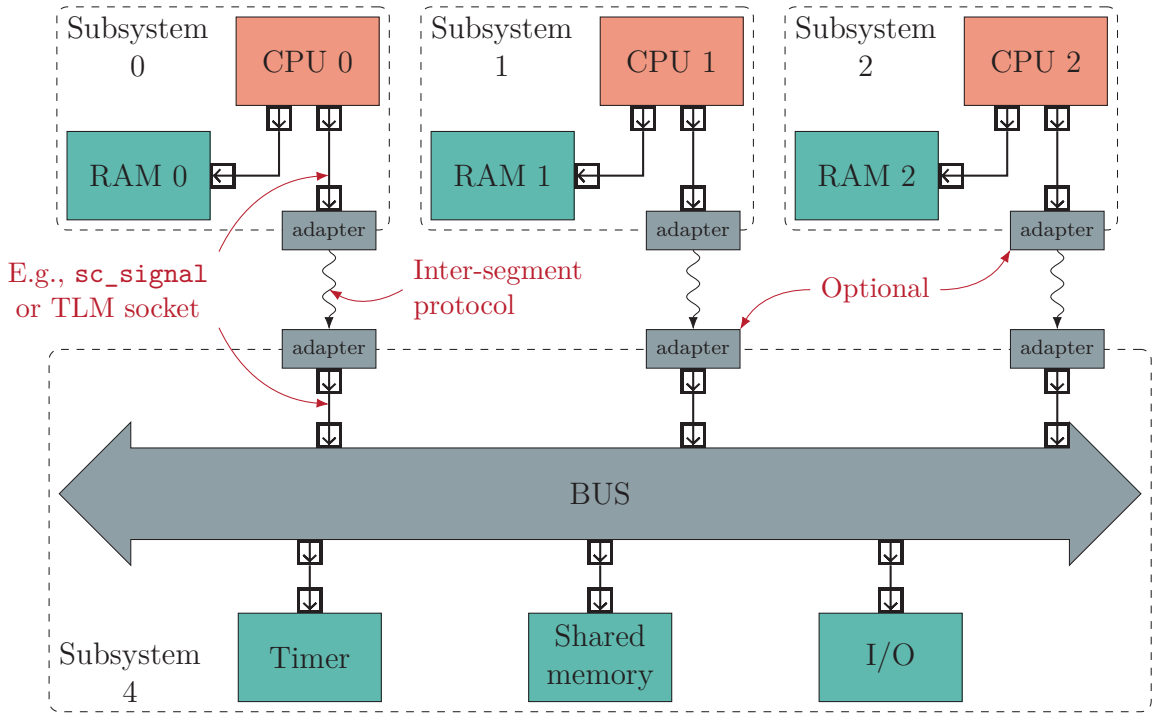

Figure 2.3 – *Classic example of a model decomposed into several mostly independent subsystems communicating through channels. Each CPU is assumed to mostly communicate with its local memory and seldom with the peripherals and the shared memory. Adapters can optionally be used at the interface between subsystems if the decoupling offered by the SystemC channel is not sufficient or if some protocol conversion is required like in a multi-OS-processes simulator for instance.*

2.2.1 Synchronous SystemC Parallelization

In synchronous parallel SystemC simulation, all processes use the global simulation time and stay synchronized with it during the whole simulation. As illustrated in Figure 2.4, only processes that are scheduled during a same evaluation phase can be evaluated in parallel. As a general principle, the SystemC processes are assigned to several units of execution (e.g., OS threads or processes) that we call *workers*[2] and that run in parallel. After every evaluation phase, all workers must synchronize with each other and the next simulation time is computed in a centralized way before stepping to the next evaluation phase. Thus, synchronous parallel SystemC simulation tends to be well suited to clocked RTL models.

Among synchronous parallelization solutions, a conservative approach is presented in [CCZ06]. The design to be simulated must be split into several subsystems chosen to be as independent as possible as in Figure 2.3. These subsystems will be distributed on several nodes connected via ethernet and simulated in parallel using a dedicated sequential SystemC kernel for each of them. Processes from different subsystems

[2]for the sake of uniformity of this thesis presentation, though sometimes named otherwise by their authors.

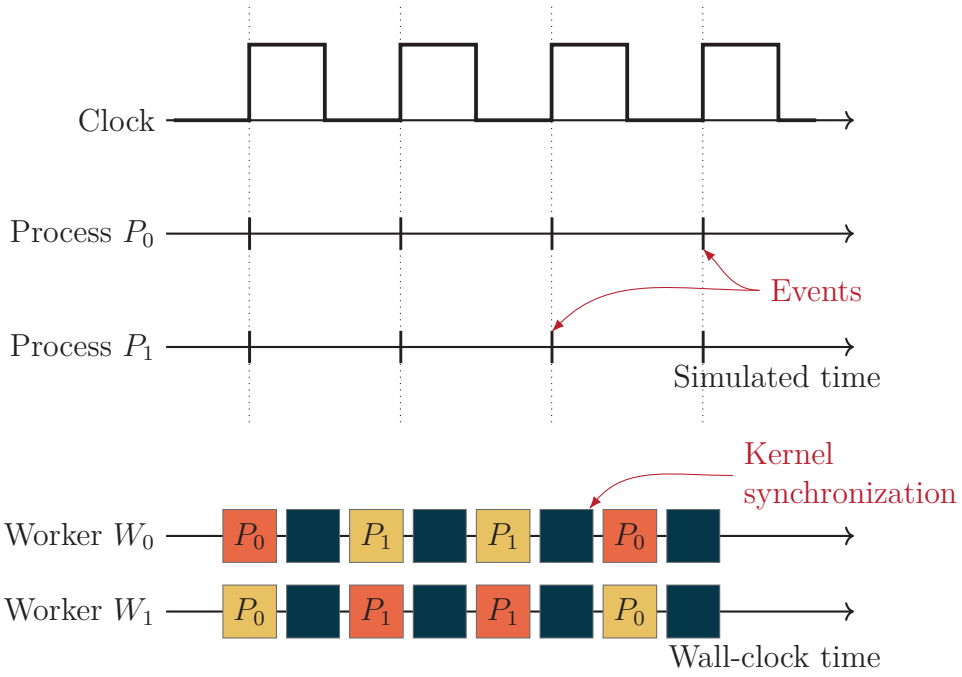

Figure 2.4 – *Example of synchronous parallel SystemC simulation. All processes are scheduled at every rising edge of the simulated system clock.*

communicate using regular channels (with host network adapters in the middle) to ensure decoupling during each evaluation phase. Indeed, the values written on SystemC channels are not available to readers before the next evaluation phase. The value presented to the reader is then constant for the whole duration of the evaluation phase. One of the simulation nodes, called the master node, is in charge of computing the next simulation time after each evaluation phase. This process can become a bottleneck when the number of processes grows as it must gather information from all sequential kernels and broadcast back the next simulation time between each evaluation phase. Shared variables and remote timed event notifications are not supported by this solution. This simulator architecture is shared by many other solutions, including TLM-oriented ones. Constraining communications between processes evaluated in parallel to use a certain type of channel gives control to the simulation kernel(s) to ensure process decoupling.

In [SLP+10], parSC, a centralized parallel SystemC scheduler uses multiple workers to run the evaluation phase. SystemC processes are mapped to these workers and the evaluation phase is bounded with barriers to synchronize the workers. The rest of the kernel logic remains sequential, including requests to the kernel that are buffered by each worker and processed sequentially after each evaluation phase. It is however the responsibility of the user to protect all resources shared between concurrently running processes using conventional synchronization mechanisms like mutexes. It is likely that in case of resource sharing, process atomicity is compromised together with simulation reproducibility.

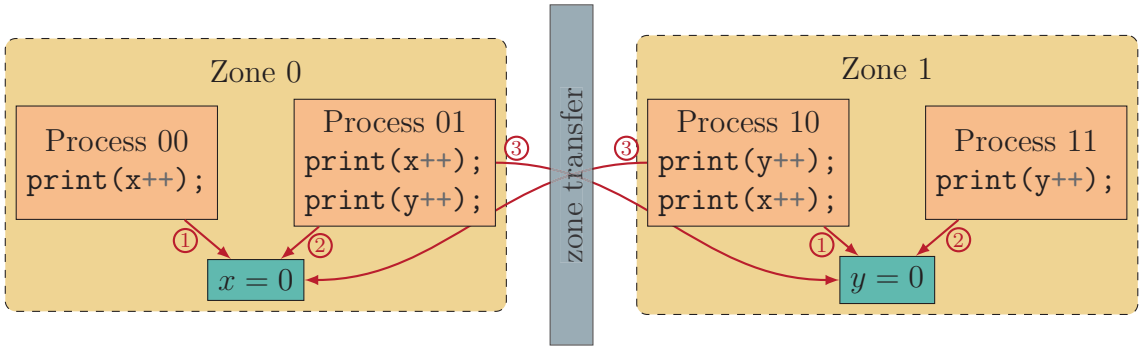

Figure 2.5 – *Example of atomicity violation in a simulation using [SWL+14]. Circled figures give the order in which processes access each variable. We assume that processes access x and y only to print their current value and increment them. For Process 01 and Process 10 to access y and x respectively, they must first undergo a zone transfer.*

LegaSCi [SWL+14], the sequel to parSC with better support for TLM, addresses this issue with the introduction of containment zones. All resources (data and processes) belong to a given zone. Only processes of a given zone can access resources of the same zone and processes of a zone run sequentially, thus preventing race conditions from happening. Data from a remote zone must only be accessed through a TLM Interface Method Call (IMC). In particular, shared variables between processes are forbidden. In case a process crosses a zone boundary through an IMC, it is migrated to the accessed zone and blocked until all processes of this zone are evaluated. However, the user must ensure that a migrated process does not access data from its original zone after migration unless no other process of its original zone accesses this data again. Otherwise, data races could occur between the migrated process and a process from its original zone that might be evaluated in parallel. However, this mechanism does not prevent process atomicity violation as illustrated on Figure 2.5. In that case, two processes (Process 01 and process 10) migrate in each other's zone. It allows them to read data previously written by the other process. Process 01 will see $y = 2$ and Process 10 will see $x = 2$, which would be impossible in a sequential evaluation. Yet, this should not threaten simulation reproducibility in most cases. However, the order in which migrated processes are evaluated relatively to each other constitutes a race condition and can compromise reproducibility.

Hardware acceleration has also been explored, especially for synchronous RTL models. For instance, RAVES [VPS+14] is a specialized multicore shared memory SoC coupled with a SystemC kernel accelerator. While the processes of a same evaluation phase are evaluated in parallel by the numerous available cores, the update and notification phases are hardware accelerated. The update phase relies on a parity register telling which value is to be read or written on each signal. Only `sc_signals` update can be accelerated that way, but they represent the majority of the channels in most RTL models. The notification phase uses a parallel search to find the processes sensitive to each triggered event. Communication between processes is obviously limited to signals as race and processes atomicity violations are not considered.

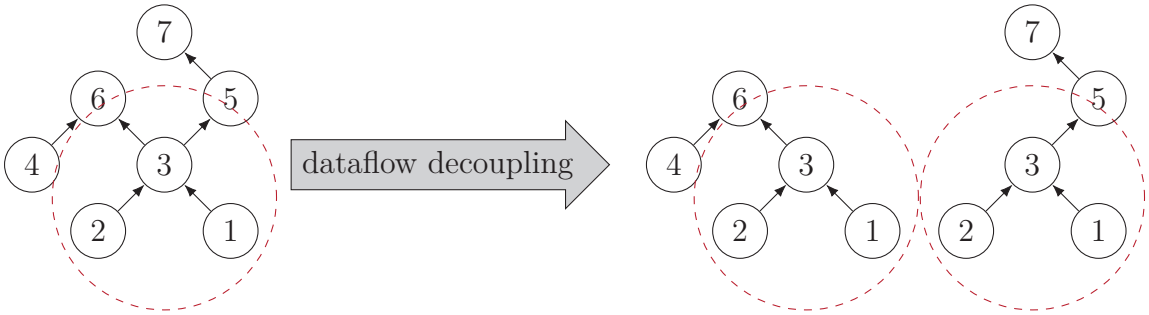

Figure 2.6 – *Dataflow decoupling using process duplication as in [VCB+12]. Nodes are processes and edges indicate the data flow between processes. Processes 5 and 6 initially depend on 1, 2 and 3. After dataflow decoupling, 6 depends on 1, 2 and 3 and 5 depends on duplicates of 1, 2 and 3. The two resulting flows are independent and can be evaluated in parallel.*

GP-GPU computing is another approach to hardware acceleration adopted with SAGA [VCB+12]. An RTL model is statically analyzed to construct the process dependency graph based on the model signals. Independent dataflows between processes are then extracted. Some processes can be duplicated to reduce dataflow coupling as illustrated in Figure 2.6. Indeed, when one or more processes take part in two different dataflows, it can be advantageous to duplicate them to have two independent dataflows, each with its version of the duplicated processes. This is especially true when targetting GPUs as they offer good parallel performance as long as all tasks do not interact with each other. Each dataflow is then executed on a different GPU warp in a sequential order respecting the dependency between processes. Warps synchronize after all processes of each dataflow have been evaluated, that is at least before every timed notification. While a GP-GPU can support great amounts of parallelism, this solution will only be the most efficient on designs with a lot of data level parallelism across processes. Also, not all of C++ is compatible with CUDA kernels, thus limiting the expressiveness of SystemC when simulated on a GP-GPU. In particular, TLM processes are unlikely to support GP-GPU evaluation.

2.2.2 Time Decoupling

At the root of all the approaches presented in this section is Parallel Discrete Event Simulation (PDES) [Fuj90]. This technique essentially allows to evaluate in parallel processes scheduled at different times while maintaining timing consistency. This principle is illustrated in Figure 2.7. A time window called *lookahead time* defines at every instant the processes that can be evaluated: all events scheduled to be triggered inside the time window can be notified. The sensitive processes can then be evaluated in parallel to each other. The size of this window depends on the simulated model and the guarantees it provides. It is usually defined as the longest period of time during which it can be guaranteed that no event will occur.

PDES can then be either conservative or optimistic. The former guarantees that

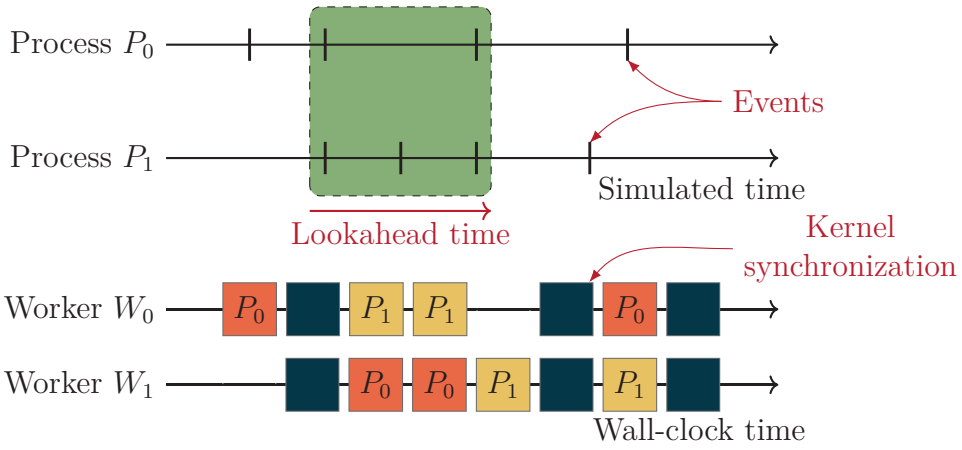

Figure 2.7 – *Overall principle of PDES. Processes that are scheduled in a same time window can run in parallel on several workers.*

the local time of each part of the design will never be greater than any event received by this part (i.e., timing violations never occur). In other word, if the local time of a part P of the design is t_P, a conservative model must ensure that no other part will send an event to P with a timestamp lower than t_P. This guarantee can be extremely difficult in dynamic systems that include, for instance, CPU. If not, a timing violation occurs and only optimistic PDES allows that. Indeed, in optimistic PDES, the risk is taken that sometimes timing violations occur, but a rollback mechanism is provided to recover from such an error. Optimistic PDES allows for larger lookahead time and increased parallelism but rollback incurs potentially large overheads. Optimistic PDES is considered too hard to apply to SystemC due to the complex state of a simulation preventing a general and efficient approach to rollback and thus is never used in practice. Yet, the approach proposed in this work could be perceived as a derivative of PDES in that sense that it speculates on the absence of illegal interactions between processes before checking it and doing a rollback if an error occurred.

ArchSC, a distributed SystemC simulation framework is described in [HLH+09]. It is based on ArchSim [HLX+09], a distributed simulation platform for system-level High Performance Computer (HPC) design. It mainly requires that processes communicate only through channels and that the design can be partitioned into relatively independent subsystems. These subsystems are then mapped to several host computers using the ArchSim [HLX+09] parallel simulation framework[3]. Each subsystem has its own SystemC scheduler. Communications between subsystems are achieved using ArchSim channels that wrap and multiplex the behavior of conventional SystemC channels in a distributed context. Time synchronization between processes is achieved by waiting on all remote input channels of each node to determine the next earliest timestamped message. According to this and to its own internal events, each subsystem can compute its next simulation time. As required by PDES, timestamped messages must be delivered in order by the remote processes to avoid timing violations. Remote timed

[3]ArchSim is not specific to SystemC.

event notifications are not supported either. Distributed simulation can scale up to hundreds of nodes but synchronization between these nodes relies on networking whose latency is orders of magnitude higher than shared memory synchronization. Also, the amount of parallelism will often be limited by the number of relatively independent parts in the simulated system. For instance, with the advent of Uniform Memory Access (UMA) chips reaching up to 64 cores to this day, distributed simulation might become less attractive in these cases.

While the previous approach is more oriented toward RTL simulations, [VPG06] tackles TLM simulations with TLM-DT for *Distributed Time*. This solution is explicitly targeted at shared memory SoC simulation. Thus, three types of components are defined (initiator, interconnect, and target), as well as three types of communications (request from an initiator to a target going through an interconnect, the associated response and an interrupt from a target to an initiator). Here, each component of the design has its own local time and there is no more global simulation time. Initiators are free to run until they send a request to a target or they reach the lookahead time set by the user before the simulation. Interconnects wait for a packet to be present on all their inputs to make sure to process the earliest one. They can then compute its new local time before forwarding the earliest request to the correct target. The target updates its local time in turn using the transaction timestamp before sending back the answer. The transaction delay is added to the request timestamp at each processing step so that the initiator can update its own local time at the end of the transaction. However, interrupts cannot be handled in the same way as it would cause deadlock. For instance, an interrupt could be sent by a target to an initiator which is waiting for an answer from this same target, the initiator and the target would end up waiting for each other. Thus, interrupt requests are polled by initiators between transactions and handled as soon as their local time is greater than the interrupt timestamp, causing small timing errors and non-determinism. The timing accuracy of a TLM-DT model is comparable to a TLM-AT model. SystemC-SMP, a parallel simulator dedicated to these types of models is proposed in [MMG+10]. Processes are grouped to favor internal communications and then mapped to different CPU, each running its own local scheduler.

Both [HLH+09; VPG06] could be subject to deadlock if they had not used *null messages* as suggested by the PDES algorithm. Null messages are timestamped messages that require no action from their recipient. They are only sent to let the recipient know the current time of the sender as on the example Figure 2.8. Indeed, with conservative PDES, an actor is not allowed to advance past the earliest timestamped message it *might* receive. As a result, all actors must send null messages at bounded time intervals to let the recipients know they will not send messages with an earlier timestamp than the null message.

The lookahead time is exploited differently in [WML+16]. While it was only a limit to the amount of time a process can run without synchronizing in [VPG06], the lookahead time t_{la} also defines the minimum amount of time that a remote event

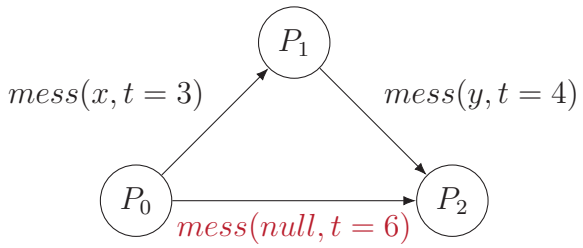

Figure 2.8 – *Usage of null messages to prevent deadlocks. P_2 must wait to be sure that P_0 will not send a message earlier than P_1 before processing the message from P_1. However, P_0 is waiting for P_1 to process its message and before answering, P_1 needs P_2 to answer its own message. Without the null message from P_0 to P_2, both would be indirectly waiting for each other.*

notification must be triggered in advance. For instance, if the local time of a process is t_p, then it must not notify events to process in other time zones with a timestamp earlier than $t_p + t_{la}$. This requirement guarantees that the remote process will not see the event in the past as it must not run in advance of t_p by more than t_{la}. This is a direct application of conservative PDES. However, this constraint can be hard to honor in a real-world model so the authors introduced *flexible time decoupling*. It consists in automatically adjusting all remote event notification delays according to the chosen policy. The *accurate* policy forbids any adjustment and raises errors if the lookahead is not respected. The *deterministic* mode increases the notification delay just as much as required for the lookahead to be respected. Finally, the *fast* mode increases the delay to be just ahead of the targeted thread time, which sacrifices determinism but avoids timing violations. The deterministic and fast modes do modify the behavior of the system compared to a standard sequential SystemC evaluation. This solution can take advantage of temporal decoupling as the local time advance guarantees that events will be notified ahead of time of remote processes.

SystemC-Link [WLA+16] brings an additional refinement defining delays for each channel linking two time zones. This delay becomes a kind of local lookahead time that is applied only to processes in time zones connected by the channel. Two scheduling policies are also provided: *as-soon-as-possible* and *as-late-as-possible*. The former makes each process yield whenever it wants to advance its local time while the later lets processes run until they reach the maximum lookahead allowed by the delays of the neighbor channels. Choosing between these two options depends on whether speed or accuracy respectively is prioritized.

Imposing no constraint on the simulated design, [CHD12] presents an approach based on compiler-driven static analysis. A standard model can be analyzed by a SystemC-semantics-aware compiler to detect the dependencies between code *segments* (i.e., the code between two scheduling points). Based on this analysis, segments can run in parallel if they do not have dependencies like accessing a same variable. But also, if the compiler can prove that a given segment will not receive any event before its next scheduling time, this segment can be issued in advance. This is called out-of-order

parallel evaluation. Load balancing based on the compiler-estimated run time of each segment is added in [LSD16]. A major limitation of this approach, however, is its lack of support for programmatically constructed platforms (e.g., CPU instantiated in a for loop). This is addressed in [SLD17] where the previously compile-time information can now be completed at run-time after platform elaboration. Also, closed source libraries can be manually annotated to be handled appropriately by the scheduling algorithm. In order to reduce false positives when analysing inter-process dependencies, the module interconnections are taken into account in what is called port-call-path-sensitive analysis [SCD18]: two segments of modules that are not connected are guaranteed not to have dependencies. Finally, [CAD20] adds event delivery prediction so that processes can be scheduled even before any of their sensitive events is triggered. The major limitation of this approach, however, is the drastic pessimization caused by pointer dereferencing: two segments that dereference a pointer are systematically conflicting if the content of the pointer is not statically known. As a result, a Symmetric Multiprocessing (SMP) TLM-LT model will always run sequentially because of the dynamically defined address of transactions targeting the shared memory component.

In [Bec17], a modeling technique based on Kahn networks [Kah74] is proposed. The parallel SystemC simulation infrastructure, called DistemC, requires splitting the simulated design into several partitions connected using single-producer single-consumer blocking fifo queues. Each partition is then simulated by a dedicated SystemC kernel. Kahn's networks guarantee determinism as long as all processes only rely on data read from these fifo queues. By providing an efficient implementation of a lockless fifo queue called FOFIFON, the authors accelerated a complex industrial design that includes both TLM components and RTL hardware accelerators like a video decoding IP generated from High Level Synthesis (HLS).

Some sort of time decoupling is also provided by the `sc_during` semantics defined in [Moy13]. While in classic DES, a task always run instantaneously before catching up by waiting for the amount of time it would have taken on the real system, `sc_during` allows to start a task with a duration associated. The explicit use of a duration helps determining which tasks are independent so that they can run in parallel: two tasks whose durations overlap are independent as they do not need the result from the other one to start. Additional functionalities are provided to control running `sc_during` tasks and interact with the simulation kernel from such task. Tasks with duration is implemented as an independent library and thus can be used with any standard-compliant SystemC simulation kernel. This approach introduces parallelism in a very simple way. However, this is the responsibility of the user to guarantee that tasks are running in isolation, otherwise race conditions and non-determinism could occur.

Finally, with raw simulation speed and ease of use in mind, [VSV+16] proposes a multiprocess and multi-kernel simulation engine designed around IPTLM, an inter-process adaptation of the TLM protocol. Inter-process communications are then strictly restricted to blocking messages going through IPTLM sockets implemented

using POSIX shared memories. In particular, shared variables and events are strongly discouraged as both are only visible from inside a same process. Determinism is not guaranteed anymore and there are no references to time synchronization between processes, indicating that very loosely-timed models are targeted. Also, inter-process communications must be avoided as much as possible to keep performance high due to the latency of inter-process message-passing compared to regular memory accesses.

2.2.3 SCale 1.0: Runtime Processes Interactions Monitoring

SCale 1.0 Positioning

As they require frequent enough synchronization most of the aforementioned standard-compliant approaches target at most TLM-AT models, which are rather slow at a few MIPS, or TLM-LT models offering sufficient architectural decoupling. On the opposite, TLM-LT models can reach hundreds of MIPS thanks to temporal decoupling but present several additional obstacles to parallelization as they tend to make extensive use of shared host resources. It is especially true when considering the DMI interface which bypasses transactions altogether and is not efficiently supported by any of the aforementioned approaches in the classic case of an SMP model with a single shared memory for instance. Also, as the time between two synchronizations increases, so does the risk of atomicity violations. As a result, quantum-based temporal decoupling is an additional obstacle to the use of all previously described approaches.

These problems are addressed by SCale 1.0, the parallel SystemC kernel described in [VS16]. SCale 1.0 principally aims at simulating in parallel TLM-LT models that use temporal decoupling, while respecting the co-routine semantics of processes. As the name suggests, SCale 1.0 is the starting point for SCale 2.0, the work presented in this manuscript, which shares the same objectives. Both SCale 1.0 and SCale 2.0 rely on the same general principle: memory accesses monitoring. However, aside from the original infrastructure of SCale 1.0 that provides SystemC primitives support and worker-based process evaluation, SCale 2.0 is a major upgrade of the memory access monitoring system. In particular, it provides both more flexibility, more compatibility and a lot more performance at the same time while improving on subtle points. This section will describe in details the aspects of SCale 1.0 that have been reused in SCale 2.0 but also highlight the major components that have been reworked from scratch in SCale 2.0.

SCale 1.0 Execution Model

Both SCale versions are synchronous simulation kernels. They rely on the use of the global quantum to restore simultaneity between processes. While in an RTL model, processes are synchronized by a clock, in a temporally decoupled TLM-LT model, processes must synchronize by the quantum as illustrated Figure 2.9. While the standard use of the quantum preconizes to wait for the local time offset as soon as it gets bigger than the quantum, SCale recommends waiting for the quantum size instead. This does not affect the overall model accuracy but is mandatory to achieve good speedups. As a result, SCale parallelizes simulation at the delta cycle level.

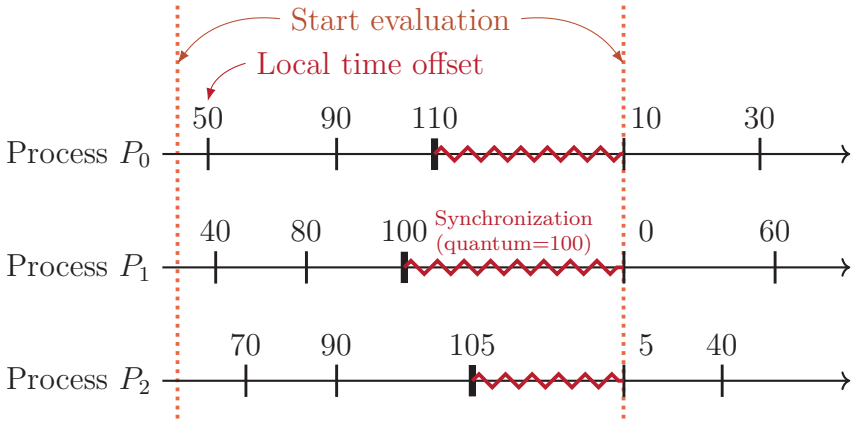

Figure 2.9 – *Three processes synchronized thanks to the use of a temporal decoupling. When the local offset of each process gets higher to the global quantum, the process wait for the size of the length of the global quantum. As a result, all three processes waits for 100 and get scheduled together in the same evaluation phase. The local time offset exceeding the global quantum is transferred to the next evaluation phase.*

During the elaboration phase, SystemC processes are grouped by the user to be assigned to different workers. Workers run in parallel to each other and each worker is responsible for evaluating sequentially its processes that are scheduled in the current evaluation phase. It results in the alternance of parallel evaluation phases surrounded by sequential update and notification phases which we refer to as *kernel phases*. Also, the kernel phase is handled by a dedicated thread. As such, this model of execution is very similar to [SWL+14].

However, workers can be unscheduled during their evaluation in case there is a risk of process atomicity violation. Such risk is explained in Section 2.2.3. Interrupted workers will then complete during the sequential evaluation phase that follows the parallel evaluation phase. As the name suggests, workers run one at a time during the sequential phase. It allows to safely access the shared memory locations that caused the workers to be unscheduled in the first place.

Preventing Process Atomicity Violations

The main objective of SCale is to prevent process atomicity violations like in Figure 2.1. It uses *simulated memory access monitoring* as its central mechanism to prevent and detect such hazard. In order to monitor memory accesses, each of them must be preceded by a call to the instrumentation function `mem_instr` provided by SCale. This function takes the targeted address, the burst length and the type of access (read or write) as argument.

Everything that happens inside `mem_instr` differs from SCale 1.0 to SCale 2.0. In Scale 1.0, the addresses touched by the memory access are first compared to two lists:

- The safe memory ranges, that is the memory ranges that are only read by processes or that are never accessed concurrently.

- The list of shared memory ranges, that is the memory ranges that are accessed concurrently using reads and writes.

These two lists are defined by the user at the beginning of the simulation according to its knowledge of the application running on the simulated platform. For instance, synchronization variables such as mutexes are good candidates to be declared as shared while the input data, if it is statically linked inside the application, will likely be safe.

If calls to `mem_instr` only concern addresses declared as safe, `mem_instr` returns immediately. Safe addresses thus are used only to reduce instrumentation overhead. If calls to `mem_instr` concern at least one address declared as shared, SCale checks if another worker has already accessed a shared address during the current evaluation phase. If not, then `mem_instr` registers the access and returns. If so, then the calling worker is unscheduled to complete its execution during the sequential phase. This mechanism is illustrated in Figure 2.10. Finally, if a worker accesses at least one address that is not declared as safe nor shared, the access is registered for process atomicity verification after the evaluation phase as described in Section 2.2.3.

In order to reduce the instrumentation overhead, an *address resolution* is defined by the user at the begining of the simulation. The address resolution is used to group adjacent addresses under the same address number. For instance, if the address resolution is 4, then every access to an aligned 4-byte `int` is considered as a single access while an access to an aligned 8-byte `double` is considered as two accesses to two adjacent addresses. The address resolution can be seen as the byte size for SCale, that is the smallest addressable chunk of memory for SCale. The lower the address resolution value, the more accurate and the slower the instrumentation. The higher the resolution, the more likely false conflict may be detected amongst adjacent addresses.

Process Atomicity Verification

During each evaluation phase, all memory accesses except those targeted at safe addresses are registered. It allows for building a dependency graph for each evaluation phase. This dependency graph reflects dependencies between workers instead of dependencies between processes. This is a valid simplification of the dependencies analysis because if workers have been evaluated atomically, then processes also have been evaluated atomically. However, it is also a slight pessimization as registering a dependency between two workers means that all processes of the first worker are considered to depend on all workers of the second one, even if only one process of the first worker depends on one process of the second worker. This simplification has been preserved in SCale 2.0 as it has implications in the lowest layers of SCale 1.0 and would have required extensive refactoring that is out of the scope of this thesis.

Processes P_{2i} and P_{2i+1} are assigned to worker $W_i, i \in \{0, 1, 2\}$.
x is a memory location written by P_0 and read by P_2 and P_5.
This is assumed to be the only shared access among all processes.

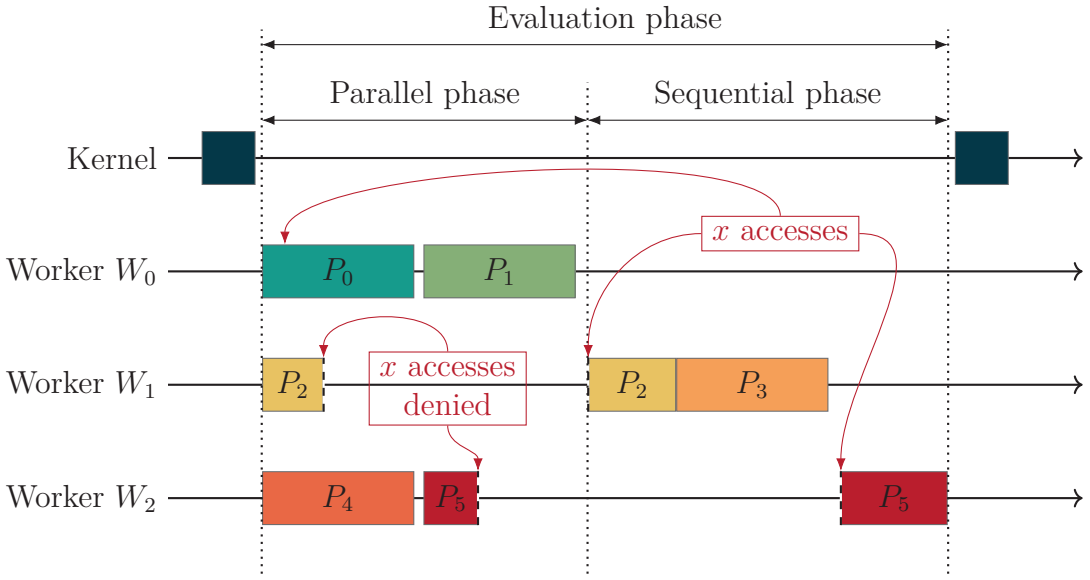

Figure 2.10 – *Example of SCale evaluation phase. P_2 and P_5 attempt to access x which is a memory location declared as shared and first accessed by P_0. It results in workers W_1 and W_2 being unscheduled to complete during the sequential phase. The kernel thread is idle during the evaluation phase while workers are idle during kernel phases. The resulting evaluation is equivalent to the purely sequential evaluation $W_0 \rightarrow W_1 \rightarrow W_2$.*

However, there is no theoretical limit to applying SCale concepts to SystemC process dependencies instead of worker dependencies.

A dependency between workers W_a and W_b written as $W_a \rightarrow W_b$ is created when a pair of accesses to a same memory location implies that an equivalent sequential schedule of the current evaluation phase must evaluate W_a before W_b to yield the same result. Specifically, a dependency exists when one of the following pair of accesses to a given memory location involves two different workers:

- Read After Write (RAW): If W_b reads a value after W_a wrote it during a parallel evaluation, a sequential schedule where W_b comes before W_a would make W_b read a possibly different value.
- Write After Read (WAR): The reason is similar to the RAW case.
- Write After Write (WAW): If W_b overwrites a value previously written by W_a, the final value differs from a sequential schedule where W_b comes before W_a.

In order to register dependencies efficiently, the memory space is split into pages

of a few kilobytes lazily allocated and stored in a map. A page is an array that contains for each one of the addresses in the page (taking the address resolution into account) a dependency graph. During the simulation, each memory access leads to the corresponding dependency graph to be updated according to the RAW, WAR and WAW rules. A dependency graph in SCale 1.0 and SCale 2.0 is a directed graph with the workers as vertices and an edge from W_a to W_b ($W_a \rightarrow W_b$) if W_b depends on W_a.

Once the evaluation phase is completed, it can be checked whether workers have been evaluated atomically by checking if there exists a sequential schedule that yields the same state at the end of the evaluation phase, that is if and only if the global dependency graph of the evaluation phase defines a partial order on the workers, that is if and only if it is acyclic.

This global dependency graph is obtained by combining the dependency graphs of all addresses in a page into an intermediate dependency graph, and then combining these intermediate dependency graphs into the global dependency graph. Combining two graphs simply consists in cumulating their edges. If there is a circular dependency in one of these graphs, this is called a *conflict* and the simulation is not compliant to the SystemC standard anymore. Also, data races might have occurred in the simulated memory, making the simulation invalid altogether. In that case, the user is suggested to declare the lacking shared addresses and run a new simulation. The graphs acyclicity is checked for each address, for each page and for the global graph to indicate potential conflict as accurately as possible (address level, page level or global level).

This check is performed after every evaluation phase. If the graph is acyclic, then it can be used to define an equivalent sequential schedule of workers for simulation replay and debug purpose. Only the workers involved in dependencies need to be scheduled sequentially during the simulation replay.

Simulation Replay

Many processes are implemented such that they could be evaluated without enforcing strict atomicity. For instance, every time a process checks if its local time has reached the global quantum, it expects to yield. Thus, processes are usually coded as if they were atomic only between two local time checks against the global quantum.

However, preserving global process atomicity is useful to provide simulation replay for debug purposes for instance. Indeed, if processes were interleaved, no partial order could be defined between them at the end of some evaluation phases. Without this partial order, there is no way to guarantee that a given evaluation phase during a new simulation is equivalent to this same evaluation phase in the original simulation, all inputs being equal.

Because SCale guarantees worker atomicity (and so processes atomicity as well), it can define a partial order on the workers that guarantees an equivalent replay. This partial order is defined by the dependency graph as: if W_1 depends on W_0 ($W_0 \rightarrow W_1$)

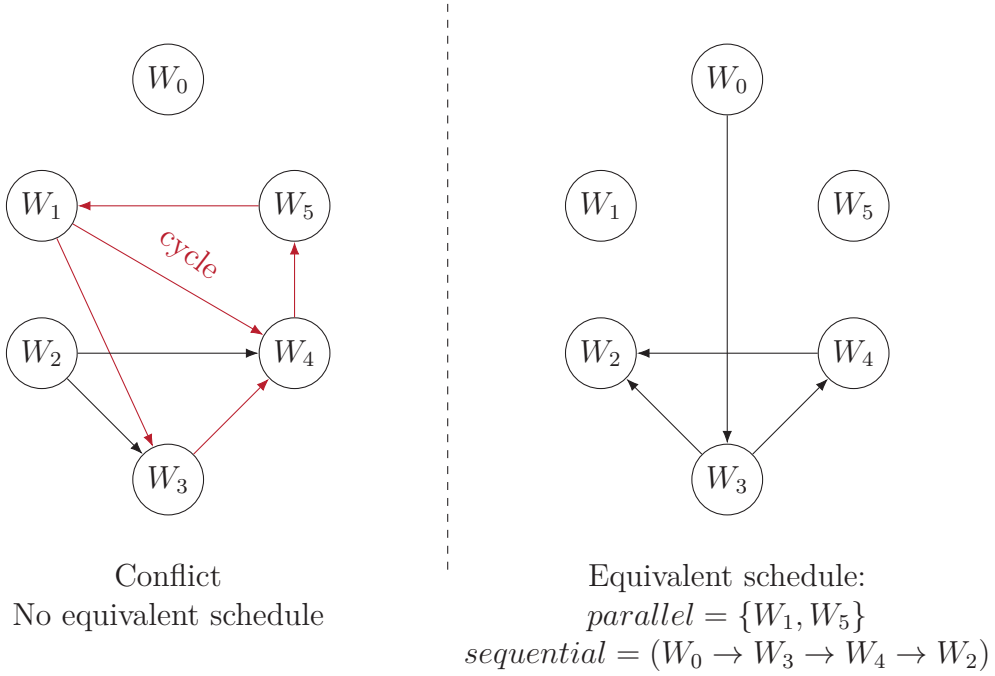

Figure 2.11 – *Extraction of an equivalent schedule from a dependency graph. The left graph present a conflict (cycle) so no equivalent schedule can be extracted as opposed to the right graph.*

then W_1 must be scheduled after W_0 in case of replay (hence the "$\rightarrow$" notation). All workers that take part in no dependencies can still run in parallel. An example is given in Figure 2.11.

The replay trace in SCale 1.0 then consists in serializing to a file a map that associates to every evaluation phase identified by a pair $(time, delta)$ the ordered list of workers that must be evaluated sequentially (the other being evaluated in parallel). This file can be loaded at the beginning of a simulation to instruct SCale 1.0 to perform a replay. SCale 1.0 will then check before every evaluation phase if the current time and delta cycle number is present in the replay map. If so, all workers that are not present in the associated list will first run in parallel. Then the workers in the list will be evaluated sequentially according to the list order. In SCale 2.0, the replay mechanism has been redesigned and is now used in additional situations like rollback as exposed in Chapter 3.

SCale 1.0 Limitations

First, it is often very hard to predict which memory regions are going to be shared during the simulation, especially in the presence of dynamic memory allocation which prevents static code analysis from providing the addresses of the shared variables. Also, SCale manipulates only physical addresses. If the simulated software is targeted at an OS such a Linux, then it only manipulates virtual addresses. Physical addresses will then be allocated at runtime, making it impossible for the user to anticipate the shared

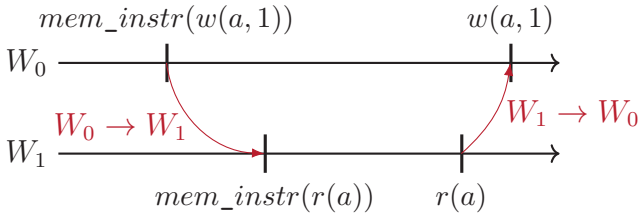

Figure 2.12 – *Example of bad memory access recording order. The dependency $W_0 \rightarrow W_1$ is recorded while the dependency caused by the actual memory accesses is $W_1 \rightarrow W_0$.*

addresses (without looking at the virtual memory management implementation in the OS kernel, at least).

Also, shared memory regions often move during a simulation. For instance, in the Deriche filter [Der87], the processed image is first copied in a working buffer. Then the buffer is processed in lines, making each line accessed only by the thread in charge of this line. However, the buffer is then processed in columns, making the columns accessed only by one thread each. Overall, almost all pixels are accessed by two different threads: one during the horizontal pass and another one during the vertical pass. Should the whole buffer be considered as shared? The Deriche filter, as simple as it is, could not be simulated in parallel by SCale 1.0. However, it is required to do so to guarantee that no conflict will occur when switching from horizontal to vertical processing and vice versa. In the context of dynamic virtual memory allocation, memory reuse gets even more frequent and unpredictable. A more flexible and powerful mechanism was thus required to tackle this limitation overcome by SCale 2.0.

Another limitation comes from the fact that memory accesses must be recorded in the same order as they are simulated in order for the conflict checking to be correct. If, like in Figure 2.12, the accesses are recorded in a different order than the actual memory accesses, the recorded dependency is the opposite of the real dependency. This would then lead to incorrect conflict detection and prevent correct simulation replay. However, SCale 1.0 leaves to the user to ensure that memory accesses are recorded in the correct order and provides no helping mechanism. The simplest solution consists in putting the instrumentation and the access together inside a critical section protected by a mutex, but this solution hampers performances and does not accommodate higher worker counts starting from about 4 after our experience.

Performance-wise, the shared data structures used to record dependencies often do not support concurrent accesses, requiring several mutex. This can result in a very high instrumentation overhead and a poor scaling when the number of workers increases. Also, every evaluation phase must be checked for conflicts before starting the next evaluation phase. This can greatly extend the duration of the kernel phase, which can significantly reduce overall parallelism.

Finally, SCale only supports processes that interact through shared memory exclu-

sively. If they interact using non-addressable resources like interrupt lines or shared variables in the model, SCale cannot monitor these interactions.

SCale 2.0, the work presented in this manuscript is inspired from SCale 1.0 and reuses a part of its infrastructure but tackles its main functional limitations while providing significant speed and scaling improvements through a major redesign. In particular, shared and safe address ranges no longer need to be manually declared by the user but are detected at runtime instead. Instrumentation and accesses are always performed in the correct order without requiring any additional synchronization from the user. Finally, the approach has been generalized to all types of interactions instead of only shared memory related ones. SCale 2.0 thus drastically improves the speed of baremetal applications (c.f., Section 4.3.1) and enables the simulation of Linux-based applications (c.f., Section 5.5) while greatly simplifying the simulator usage.

Chapter 3

Proposed Solution for LT-TLM Parallel Simulation

3.1 Overview 64
3.1.1 Simplified Model 64
3.1.2 General Execution Flow 65
3.1.3 `mem_instr` Outline 67
3.2 The Parallel Evaluation Phase 68
3.2.1 Advantages of Zero Dependencies Parallel Phase 68
3.2.2 The Address Monitoring FSM 69
3.2.3 Correct Memory Access Recording Order 74
3.2.4 Efficient FSMs Reset 75
3.2.5 Fast Scalable FSM Storage 77
3.3 The Sequential Evaluation Phase 79
3.3.1 Choosing the Sequential Evaluation Order 80
3.3.2 Asynchronous Dependencies Analysis 81
3.3.3 Simulation Replay 85
3.3.4 Rollback-Based Conflict Recovery 87
3.4 Generalization to Any Shared Resources 94

This chapter is the core of this manuscript. It presents the mechanisms developed in SCale 2.0 to allow SystemC standard-compliant parallel simulation of TLM-LT models. The first part of the chapter gives an overview of SCale 2.0 architecture before diving into each component's details.

3.1 Overview

3.1.1 Simplified Model

For the sake of simplicity, it is assumed in this chapter that the only shared resource of the simulation is the model shared memory. We explain in Section 3.4 how the presented system is easily generalized to any form of shared resources like peripherals, interrupt lines, etc. Thus, let us consider the simplified model Figure 3.1. In this model, processors are considered as pure initiators that can only read and write a shared memory. We also assume that each processor is simulated by a single SC_THREAD and no other processes are defined. The memory is a simple wrapper around a big contiguous memory buffer, and it provides as many input ports as there are processors.

Processors access memory atomically, that is the order between memory accesses is always well defined and there is no risk of data race. Memory accesses also have no side effects on other accesses aside from changing the written value if the access is a write. As a result, processors can only interact by accessing a common memory location. Any other action is guaranteed to be interaction free between processors.

With such model in mind, processes are evaluated atomically if and only if memory accesses to all location are performed in an order equivalent to a given sequential evaluation of processes with except for reads that can reordered relatively to each other. This is obviously not guaranteed if processes are evaluated in parallel. Hence, the simplified problem that we are trying to solve requires to control the order of memory accesses in this context of parallel execution.

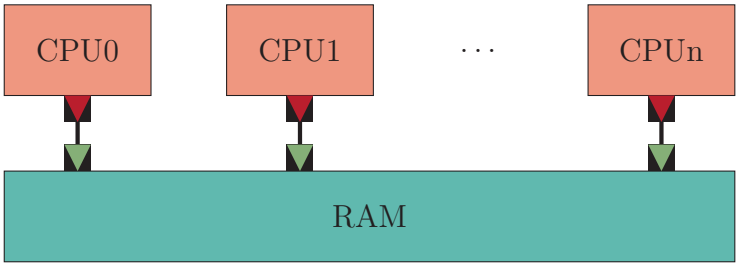

Figure 3.1 – *A simplified representation of a simulated model. Processors can only interact through shared memory. Other peripherals, interconnects or caches are ignored.*

3.1.2 General Execution Flow

The flow chart of the parallel evaluation phase of SCale 2.0 is represented Figure 3.2. Like SCale 1.0, SCale 2.0 parallelizes the evaluation phases at the delta cycle level: all processes scheduled in a same delta cycle are launched in parallel. The evaluation phase is also split into the parallel phase and the (possibly empty) sequential phase.

We also assume here that the user has correctly instrumented all memory accesses through a call to the provided function `mem_instr` before each memory access in our simplified platform model, as required by SCale 1.0. In this simplified model, `mem_instr` must be inserted only once in the ISS SystemC wrapper, anywhere before the memory access initiation point. In SCale 2.0 too, `mem_instr` is responsible for unscheduling workers trying to access a memory location considered hazardous. In SCale 2.0, an access is considered hazardous as soon as it can introduce a dependency between two workers. This criterion is further developed in Section 3.2.1. In case some workers are unscheduled during the parallel phase, the sequential phase takes place.

One major difference with SCale 1.0 is the way SCale 2.0 checks for conflicting memory accesses, that is circular dependencies between workers. This check is necessary to ensure that process atomicity has not been violated at the end of each evaluation phase. This aspect is detailed in Section 3.3.2. Instead of performing the check at the end of every evaluation phase, SCale 2.0 only does it when the sequential phase is not empty, that is when at least one worker has been unscheduled. The reason for that is explained in Section 3.2.1. Also, conflict checks are performed asynchronously while the simulation continues, thus exploiting more parallelism by greatly reducing the kernel phase duration. Conflict check results are then collected by the kernel thread when it is waiting for the worker to complete the parallel phase.

Because SCale 2.0 does not rely on preliminary declaration of all shared memory regions, it is expected that conflict may occur even when simulating a perfectly valid software on a correctly instrumented model. Such expected errors are then recovered automatically through a rollback to the last valid checkpointed state. To that extent, the simulation is periodically checkpointed. The rollback mechanism is explained in Section 3.3.4.

Finally, as in SCale 1.0, the collected dependencies can be stored to disk to replay the simulation. SCale 2.0 does so in a more efficient way, though (c.f. Section 3.3.3).

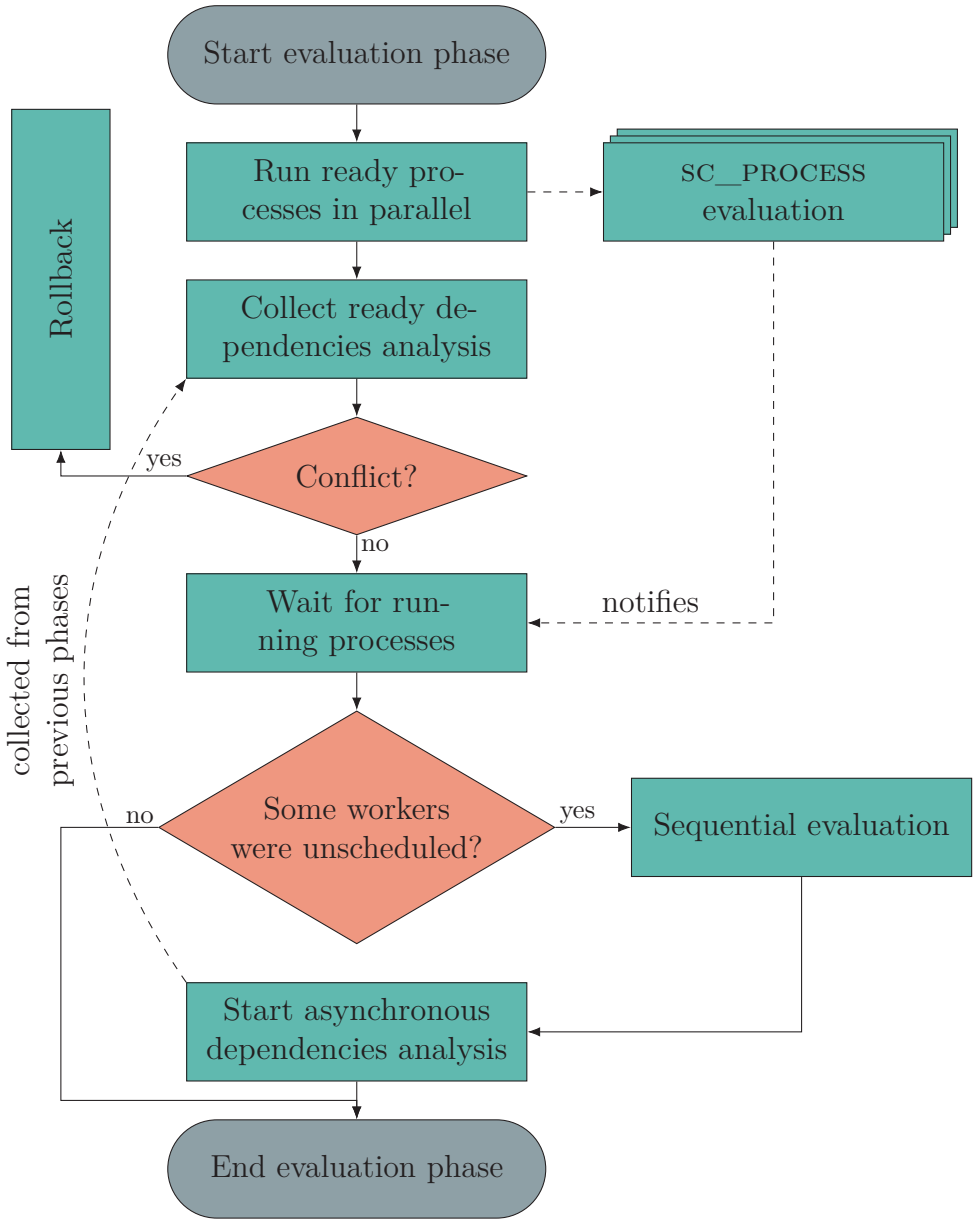

Figure 3.2 – *Evaluation phase flow chart of SCale 2.0.*

3.1.3 `mem_instr` Outline

Aside from functions used to configure SCale 2.0 during simulation initialization, the memory access instrumentation function `mem_instr` is the only function the user needs to adapt our simplified model to SCale 2.0. It is only required that every memory access is preceded by a call to this function.

`mem_instr` is outlined on Algorithm 4. It takes three arguments: the accessed address, the number of bytes accessed and the type of access (read or write). `mem_instr` does the following main operations:

1. Compute the reduced address according to the address resolution (Line 4). The reduced address is the quotient of the address by the address resolution. Two addresses that correspond to the same reduced address are associated to the same monitoring state.
2. Update the monitoring state associated to the reduced address (Line 6). This monitoring state is used to determine if the access to the targeted address can be performed.
3. Wait until the sequential phase if the access cannot be performed during the parallel phase (Line 8).
4. Record the memory access characteristics (Line 11) in case a conflict check is later needed. It can be noticed that the full address is used here, not the reduced one. This is done to reduce the risk of false positives during the dependencies analysis as the extra memory and computation required is not significant.

Algorithm 4 `mem_instr` function outline

```
1:  procedure MEM_INSTR(addr, nBytes, isWrite)
2:      if not in sequential phase then
3:          p ← getCurrentWorkerId()
4:          a ← reducedAddr(addr)                          ▷ c.f. 3.2.2
5:          m ← FSMarray[a]                                ▷ c.f. 3.2.5
6:          go ← m.updateFSM(p, isWrite)                   ▷ c.f. 3.2.2
7:          if not go then
8:              wait sequential phase                      ▷ c.f. 3.3
9:          end if
10:     end if
11:     recordAccess(addr, nBytes, isWrite, p)             ▷ c.f. 3.3.2
12: end procedure
```

One fundamental enhancement of SCale 2.0 over SCale 1.0 is that `mem_instr` does not require extra synchronization to record the correct memory access order and avoid the problem presented in Figure 2.12 where memory accesses are recorded in a different

order than they are performed. The only requirement is that `mem_instr` is called before the access it instruments. This is one of the many benefits of the *zero dependencies guarantee* introduced in Section 3.2.1.

3.2 The Parallel Evaluation Phase

All evaluation phases start with a parallel phase where ready processes are evaluated by the worker they are attached to. Workers run in parallel and evaluate their processes sequentially.

3.2.1 Advantages of Zero Dependencies Parallel Phase

During a simulation, the vast majority of the evaluation phases do not present any dependencies. In other words, all processes access only independent or read-only data during most of the evaluation phases. This can be explained by the fact that accesses to shared variables tend to be rare in real world applications to preserve execution speed. For that reason, we have made design choices that take advantage of this assessment. In particular, *we enforce that no dependency occurs during the parallel evaluation phase, postponing all the complex logic to the sequential phase.*

Assuming that zero dependencies can occur during the parallel phase, several major properties are obtained and detailed in the following sections:

1. "Instrumentation + memory access" need not be atomic as long as instrumentation comes first.

2. Memory accesses during the parallel phase can be recorded in parallel as they never depend on each other, so their order is not important.

3. If no worker is unscheduled during the parallel phase, then no dependencies exist which implies that no conflicts have occurred, and the check is not required.

Hence, the "zero dependencies during parallel phase" property is a strict prerequisite to enable vital optimizations across the whole parallel simulation system.

A straightforward way to provide this guarantee could be to record, for each address, all accesses of the evaluation phase and determine for each new access if it incurs a dependency with the previous accesses. If so, the offending worker would get unscheduled. It is optimal in the sense that there is no false unscheduling, that is no worker is unscheduled unless it is about to cause a dependency. Still, this solution is slow to implement for several reasons. Among them, it requires an expensive lookup at every memory access, and it relies on a variable-size container which requires synchronized accesses as a consequence.

Instead, we provide the zero dependencies guarantee using an FSM-based access granting policy detailed in the next section. While being heuristic and (slightly) sub-optimal, it allows extremely fast implementation.

3.2.2 The Address Monitoring FSM

Generality

Keeping in mind that zero dependencies must occur during the parallel phase, we must define a memory access granting policy to determine which accesses can be performed during the parallel phase and which must be postponed to the sequential phase.

As introduced in the previous section, the optimal policy described in the previous section does not perform well enough in practice. Thus, we must define a memory access policy with a much faster decision time, even if false positive dependency detection could occur. Let us go through the process of designing an efficient policy starting from a simplistic one and improving on it. This will help in understanding why the chosen policy has no simple alternative under the zero dependencies guarantee constraints.

A preliminary optimization consists in grouping addresses into blocks of size S, defined by the user and called the *address resolution* as in SCale 1.0. An appropriate size typically is the largest number of bytes accessible with a single CPU instruction (e.g., 8 bytes on RISC-V 64 bits or 16 bytes on AMD64 without Single Instruction Multiple Data (SIMD) extensions). With such address resolution, it is guaranteed that each regular memory access can be instrumented in a single call to `mem_instr`.

When calling `mem_instr` with address a as argument, the *reduced address* a' is computed first (Algorithm 4 line 4) as $a' = a/S$. If the worker is not unscheduled, it can then safely access any address in the range $[a', a' + S)$. Choosing an appropriate address resolution is key to reducing monitoring cost. Grouping addresses is a conservative approximation: we may unschedule a worker that could have been granted the access, but we never grant an access to a non-protected address.

In case a memory access spreads among several aligned S-bytes intervals because it is either misaligned or it accesses more than S bytes, a helper function `mem_instr_slow`, that calls `mem_instr` several times to protect all the accessed memory, must be used instead of `mem_instr`. We have chosen to have two functions so that memory accesses that require a single call to `mem_instr` are instrumented as fast as possible.

Toward an Access-Granting Policy

In this section, the design process of the final address monitoring FSM is exposed. The detailed presentation of the final FSM takes place in the next section.

One of the simplest access policy one could use to guarantee no dependency between workers could be to provide exclusive access to a single worker on each reduced address.

The simplest approach would be to chose the first worker to access each address to become the *owner* of this address for the current quantum. The address is said to be in the OWNED state. An algorithmic representation of such policy is given on Algorithm 5. Notice that, as is, this algorithm presents a data race between Lines 4 and 5 that must be carefully handled in a real implementation. We only focus on the logic for now, though.

Algorithm 5 Memory access granting policy #1

```
1: procedure canDoAccess1(address, workerID)
2:     if wasAccessed(address) AND workerID ≠ owner(address) then
3:         return false
4:     else if NOT wasAccessed(address) then
5:         registerOwner(address, workerID)
6:     end if
7:     return true
8: end procedure
```

Policy #1 is good if all workers almost never access shared memory locations during the whole evaluation phase. However, shared read-only data — which is extremely common — is very badly handled by such policy as a single worker can access each address during each parallel phase, whether it is for writing or reading. Thus, a refinement is needed to allow either a single worker to access each address in both read and write mode or to allow only reads during the parallel phase: a read-only state is needed. Exclusive ownership and read-only are the only two ways a variable can be accessed during a parallel phase while respecting the zero dependencies guarantee.

But it must now be decided whether the first access of the evaluation phase to an address should set it in the OWNED state or the READ_ONLY state. If the first access is a write, then READ_ONLY is not an option and the address necessarily becomes OWNED. If the first access is a read, both OWNED and READ_ONLY are valid options. Though, if OWNED is chosen, this new policy would be equivalent to policy #1. In order to differentiate from policy #1, READ_ONLY is the only option in case the first access is a read. It translates into the Algorithm 6 which must also be implemented with data race hazards in mind.

But policy #2 also has a major drawback. Let us assume an address a where a loop index i used by a process evaluated by worker W is stored. This loop index i is being used for a few evaluation phases already and a new evaluation phase begins. Will the first access of W at address a be a read or a write? It is likely to be a read (e.g., testing for the end condition of the loop). With policy #2, a would in that case be classified as READ_ONLY, which will then prevent W to write i during the whole parallel phase. This will apply to most addresses in the working set of W, resulting in its quasi systematic unscheduling. It would in the end lead to a quasi-exclusively sequential evaluation of all workers.

Algorithm 6 Memory access granting policy #2

```
1: procedure canDoAccess2(address, isRead, workerID)
2:     if isRead AND isReadOnly(address) then
3:         return true
4:     else if isWrite AND isReadOnly(address) then
5:         return false
6:     else if isRead AND NOT wasAccessed(address) then
7:         setReadOnly(address)
8:         return true
9:     else if isWrite AND NOT wasAccessed(address) then
10:        registerOwner(address, workerID)
11:        return true
12:    else
13:        return canDoAccess1(address, workerID)          ▷ c.f. Algorithm 5
14:    end if
15: end procedure
```

Proposed Access-Granting Policy

The actual access granting policy used in the rest of this manuscript is defined by the FSM described Figure 3.3. One instance of this FSM is associated to each reduced address using a custom data structure described in Section 3.2.5. Each reduced address can independently be in one of these 4 states:

1. NO_ACCESS: Before the first access of the evaluation phase.
2. OWNED: When an address has been accessed by *only one worker* and with *at least one write*. This worker is called the *owner* of the address.
3. READ_EXCLUSIVE: When an address has been *only read* by a *single worker*. This worker is also called the *owner* of the address.
4. READ_SHARED: When an address has been *only read* and by *at least two workers* since last reset.

The main idea behind this FSM is the same as for policy #2. It aims at allowing workers to freely access the memory they are not sharing (the OWNED state) and to allow read-only memory to be accessed concurrently (the READ_SHARED state) to unschedule as few workers as possible. But as explained, these two states are not sufficient and the READ_EXCLUSIVE state is crucial to make the FSM efficient: it allows to wait until it is possible to choose between OWNED and READ_SHARED based on more than only reads from a single worker. This state corresponds to the case where an address is both read-only and accessed by a single worker. The address will leave this state as soon as either the owner writes to it or another worker reads it.

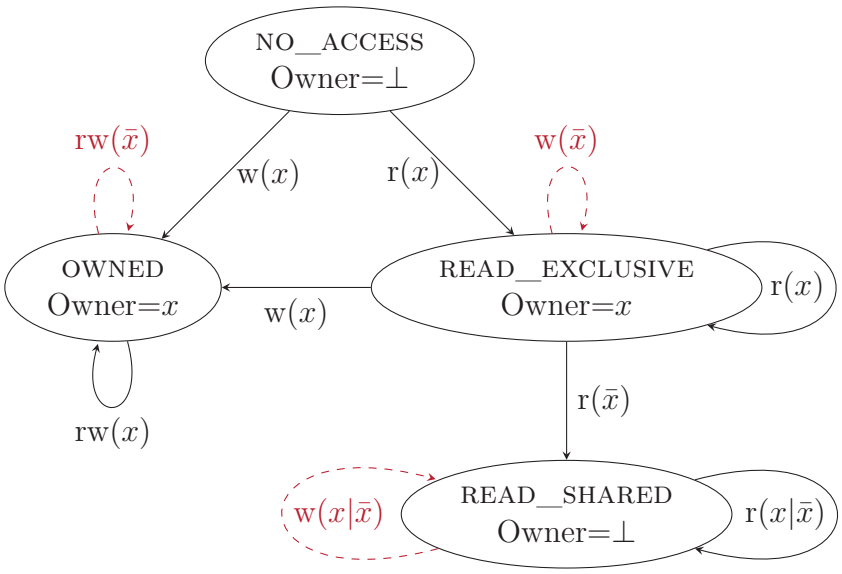

Figure 3.3 – *Memory access monitoring FSM. x is the worker identifier (WID) of the worker doing the access from* NO_ACCESS*; $\bar{x}$ designates any worker other than x; r and w stand for read and write. Workers are unscheduled on ⇢ transitions.*

It can be noticed that any access causing a dependency corresponds to a "⇢" transition, causing the offending worker to be unscheduled. Thus, the zero dependencies guarantee is provided by this FSM.

If the policy based on the FSM Figure 3.3 is proven efficient in Chapters 4 and 5, it is also due to the fast implementation it enables. Indeed, the FSM state is composed of the following fields packed in 4 bytes to allow for a fast atomic Compare And Swap (CAS): the state id (4 possible values), the owner's WID and a generation counter for a fast reset (see Section 3.2.4).

Upon memory access, Algorithm 7 is used as the updateFSM function of `mem_instr` (i.e., at Line 6 of Algorithm 4 for a fast and wait-free[1] FSM update. The transition computation getNewS is defined by Figure 3.3 and consists in a 4-case switch statement with a couple of ternary expressions per case. getNewS returns the new state and a boolean which is true if and only if the transition is not a "⇢", that is if the access is granted.

It can be noticed that the transition application requires an atomic CAS as two or more workers might attempt to update the FSM concurrently. However, doing the CAS is required only if changing the state of the FSM as shown by the test at Line 4 of Algorithm 7. This optimization brings speedups ranging from ×1.2 to ×2.2 on baremetal benchmarks.

[1] *wait-free* means that the number of steps required to perform the FSM update and the time required for each step are bounded, no matter what other workers do and how many of them there are.

Algorithm 7 FSM update algorithm

```
1:  procedure UPDATEFSM(PID, accessType)
2:      S_old ← S                                   ▷ S is the FSM 4-byte state
3:      S_new, go ← getNewS(S_old, PID, accessType)
4:      if S_new != S_old then
5:          S.CAS(expected=S_old, new=S_new)
6:          if CAS failed then
7:              return updateFSM(PID, accessType)
8:          end if
9:      end if
10:     return go
11: end procedure
```

In order to prove this optimization, the semantics of the FSM must be reminded: the FSM arbitrates which access can be performed during the parallel phase and which cannot. It receives a non-ordered set of inputs and processes them in sequence, thus introducing an order between the memory accesses. For a total order to exist between successive transitions, each transition must be atomic and *sequentially consistent*. It is equivalent to saying that each transition takes place instantly at a given point in time called the *serialization point* of the transition [Zak17]. In the case where the transition is not fixed (i.e., it changes the state), the serialization point is the CAS when it is successful. If the CAS fails, then it has no effect and the whole FSM update procedure is restarted from the state loading. If the transition is fixed (i.e., it does not change the state), the serialization point is the initial atomic state load, which has the same effect as loading the state, computing the transition, and doing a successful CAS, all atomically.

Thanks to this implementation, once an FSM has reached its final state (in one or two actual CAS), all subsequent granted accesses do not need to write a single byte of data in the FSM. Because the vast majority of memory accesses fall into that category, the instrumentation mechanism provides great scalability with the number of workers.

In order to maximize the benefits of the optimization Line 4, the state of the FSM should be preserved across several evaluation phases to minimize the number of state changes. Indeed, addresses tend to be used in the same way for relatively long periods of time (c.f. temporal reference locality principle [Den05]). But just as an address is not used for a single purpose for the whole power-on time of a computer, keeping the same state for an address during the whole simulation is not efficient either. Thus, FSMs still need to be reset sometimes but not at every evaluation phase. Choosing a good reset policy is a difficult question by itself and is discussed in Section 3.2.4.

3.2.3 Correct Memory Access Recording Order

One crucial simulation correctness condition is that the memory accesses are recorded in an order that is equivalent to the order in which they are simulated. More precisely, they must be recorded in an order that exhibits the same dependencies as their real order[2]. In this section, a first counter example is given to illustrate the issue. Then, the zero dependencies guarantee provided by the address monitoring FSM will be used to relax the ordering constraints to the point where equivalent recording order is guaranteed at no additional cost.

Let us assume an alternate FSM that does not provide the zero-dependencies guarantee applied to the example Figure 2.12 page 61. The instrumentation records the write of P_0 to address a before the read of P_1 while P_1 reads a before P_0 writes it. The dependencies analysis would then consider $P_0 \rightarrow P_1$ because of a RAW on a while the real dependency is $P_1 \rightarrow P_0$ because of a WAR. Such error could lead to undetected conflicts which would not be acceptable from a correctness standpoint. An obvious workaround would be to include the instrumentation and the access in a critical section prohibiting at least the other memory accesses to the same variable to happen simultaneously, but that would result in severe performance degradation.

However, enforcing strictly identical order of accesses and recorded accesses is not necessary if the correct dependencies are detected. In particular, if the recorded order of independent memory accesses differs from their real order, no dependencies are missed as theses accesses cannot introduce dependencies at all. The zero dependencies guarantee provided by our FSM implies that no two dependent memory accesses can occur concurrently during the parallel phase. In particular, the scenario in Figure 2.12 cannot happen as P_1 would be unscheduled before the read to a is both recorded and performed. In general, the order in which memory accesses are recorded during the parallel phase is not significant under the zero dependencies guarantee. Indeed, any reordering of the recorded accesses would produce no dependencies.

Eventually, we do not guarantee strict memory access recording order. Instead, we provide a sufficient guarantee so that the instrumentation and the corresponding access need not be protected by any sort of additional synchronization for the recorded order to be correct: if the access is granted, then it can be recorded in any order relative to the other accesses of the parallel phase. In other words, the accesses recorded during the parallel phase form a set without specific ordering between its elements. This has a major impact on performance as it allows each worker to record accesses in its own independent container, enabling perfect algorithmic scaling with the number of workers running in parallel. During the sequential phase, preserving ordering of recorded accesses is trivial as workers run one after the other.

To put the performance enabled by this feature in perspective, it can be compared to the necessary use of a per-address reader-writer lock to provide atomic instrumentation

[2]A well-defined total order is not guaranteed to exist unless sequential consistency is enforced on all memory accesses, which is not the default on AMD64.

and access together. It has been tried in a first attempt to improving SCale 1.0 and resulted in high overhead and, above all, poor scaling above 2 workers.

3.2.4 Efficient FSMs Reset

In the FSM depicted in Figure 3.3 no transitions leave OWNED and READ_SHARED states: they are end points. As a Therefore, once reached, such state would last during the entire simulation. This accommodates programs whose memory accesses pattern is constant over its execution. For instance, if the simulated program only consists in multiplying two squared matrices of size n and storing the result in a third one: $C = A \times B$. Each one of the N threads is responsible for computing $\frac{n}{N}$ consecutive lines of C. As depicted in Figure 3.4, A would stay in the READ_EXCLUSIVE state as each line is only read by a single worker. B would stay in the READ_SHARED state as all workers read the entire matrix. C would stay in the OWNED state as each line is written by a single worker.

However, addresses are often used by a worker for a certain amount of time and then by another like in the Deriche filter. In this algorithm, horizontal and vertical passes on an image are alternated. In a naive parallel implementation, the image is split horizontally for the horizontal pass and vertically for the vertical pass. As a consequence, as shown in Figure 3.5 pixels are all in the OWNED state after the horizontal pass and need to change owner before the vertical pass in order not to unschedule workers after every memory access in the image buffer. However, setting the owner of an address is only possible from the NO_ACCESS state, hence the need for FSMs reset at some points during the simulation.

Efficient simulation of the Deriche filter (and of most algorithms), hence requires to be able to reset the FSMs at well-chosen instants. We will discuss here two aspects of the problem:

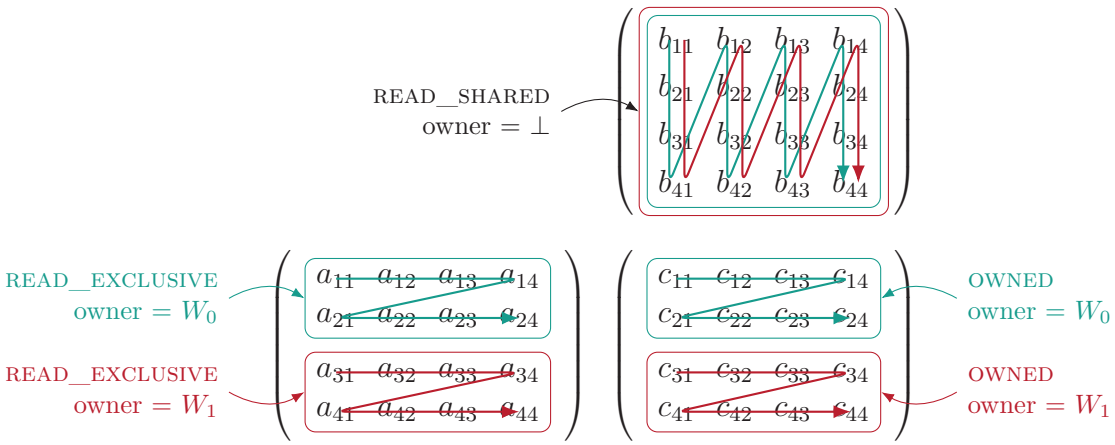

Figure 3.4 – *Classification of the memory addresses during a* 4×4 *matrix multiplication using 2 processes assigned to 2 workers. Traversals are represented by zigzaging arrows.*

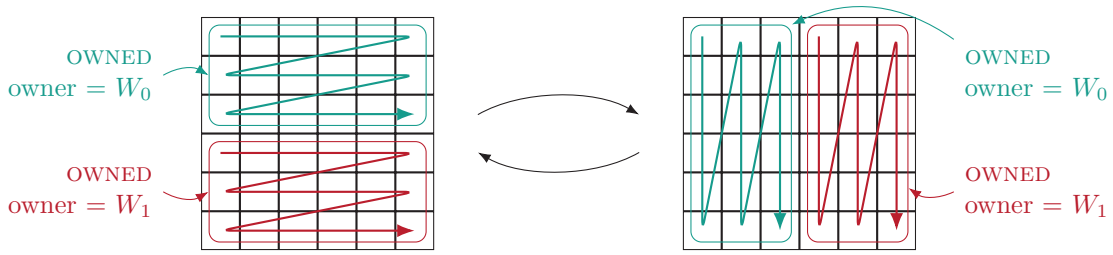

Figure 3.5 – *Classification of the memory addresses during a Deriche image filtering.*

1. When to reset the FSMs?

2. How to reset the FSMs efficiently?

Ideally, an FSM should be reset whenever its state does not reflect the actual usage of the associated address (e.g., between two passes in the Deriche filter). This is very hard to detect accurately from the SystemC model standpoint, but an incorrectly classified address will cause quasi systematic unscheduling, which is easy to measure. A first approach could be to reset all addresses that caused some unscheduling during the previous quantum. However, a worker can only be unscheduled once per quantum so no more addresses than there are workers can be reset in each quantum with this approach. This is inefficient as large amounts of addresses usually have to change owner simultaneously (e.g., the whole image in the Deriche filter, again).

Though, setting apart the addresses that need to be reset from those that do not is impossible as it depends on the future of the simulation. As a result, we have chosen to reset all addresses every time a worker is unscheduled. That way, all addresses can transit to their most appropriate state from that point. Addresses that did not need to be reset will only incur at most a couple of additional CAS that are relatively inexpensive and performed in parallel by all workers. Resetting all addresses is a good strategy to avoid further workers unscheduling caused by some memory changing owner.

It must be noticed that a truly shared address such as a mutex will, when accessed, systematically cause except the first to be unscheduled. It will each time result in the reset of all FSM with the small performance costs that could add up. However, truly shared addresses are usually accessed relatively infrequently in a well written parallel program. This potential weakness of this reset policy thus is not a concern. The small amount of sequential evaluation is demonstrated in Figure 5.1.

Choosing to reset either none or all FSMs also enables $O(1)$ reset implementation using a *generation-based reset*. As mentioned in Section 3.2.2, a generation counter is part of the state of the FSM. To virtually reset all FSM, a single counter called `fsm_gen` needs to be incremented before starting the next evaluation phase. The value of `fsm_gen` is then passed to updateFSM as third argument in Algorithm 7 which

is updated as in Algorithm 8 to perform *lazy reset*. This way, FSMs are only reset if they are accessed and directly by the very first worker accessing them. The zero dependencies guarantee is maintained as the reset of an FSM can only occur right before the very first access of the quantum.

Algorithm 8 FSM update algorithm with lazy reset

```
1:  procedure UPDATEFSM(PID, accessType, gen)
2:      S_old ← S
3:      curGen ← S_old.gen
4:      if curGen ≠ gen then                                   ▷ reset
5:          S_tmp.state ← NO_ACCESS
6:          S_tmp.owner ← ⊥
7:          S_tmp.gen ← gen
8:      else S_tmp ← S_old
9:      end if
10:     S_new, go ← getNewS(S_tmp, PID, accessType)
11:     if S_new ≠ S_old then
12:         S.CAS(expected=S_old, new=S_new)
13:         if CAS failed then
14:             return updateFSM(PID, accessType)
15:         end if
16:     end if
17:     return go
18: end procedure
```

3.2.5 Fast Scalable FSM Storage

On the one hand, in order for memory accesses instrumentation to be fast and scalable with the number of workers, FSMs must be stored in a container that supports concurrent accesses while requiring little to no synchronization upon lookups. Yet, the only mandatory operation is constant-time random access to rapidly get the FSM instance associated to an address. Most of the classic operations such as iteration or deletion are not required thanks to the use of a lazy-reset approach.

On the other hand, the memory map of the simulated platform can be shaped in arbitrary ways. It can span over huge memory ranges, be sparse or even runtime defined (e.g., the PCI-e protocol). The memory usage of our map must also remain contained as it affects rollback performance (c.f. Section 3.3.4).

Let us first briefly discuss the most common containers and why they do not fulfill our requirements:

- A statically allocated contiguous array of FSMs would potentially guzzle huge amounts of memory to cover the whole address map of the simulated platform.

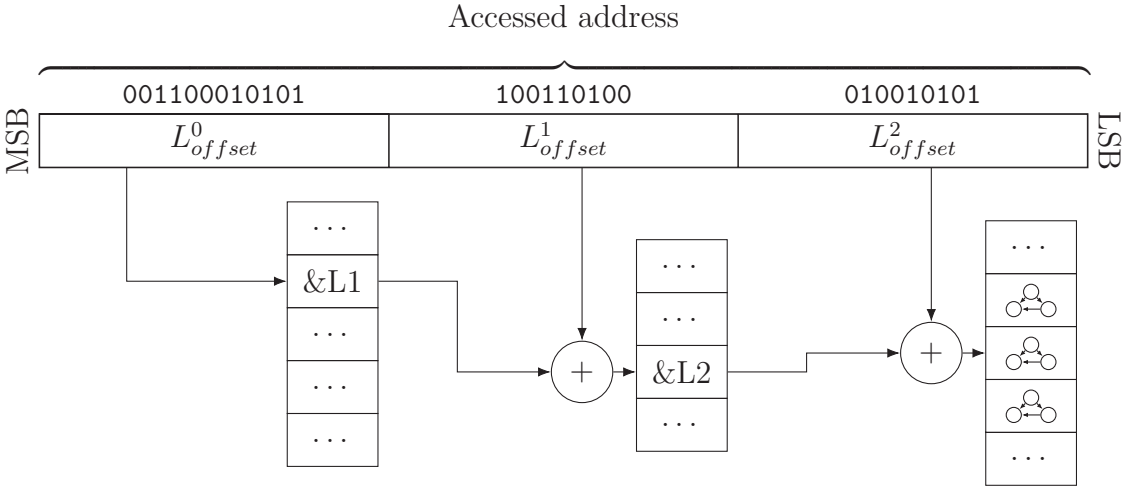

Figure 3.6 – *Multi-level FSM table. Only tables hit by the access are drawn.*

- Splitting it into a list of arrays to cover only the used memory ranges is still quite memory intensive, requires additional target-dependent parameterization from the user, is algorithmically inefficient if there are many holes in the memory map and does not support dynamically defined memory maps.
- `std::map` and `std::unordered_map` both require external synchronization for concurrent accesses, which would compromise performances.

Instead, we designed a custom container inspired from multilevel page tables as illustrated Figure 3.6. Accessing an element is done using the successive fields of the address to compute offsets in successive levels of nested tables. If the table is defined with N levels, the $N-1$ first levels (the *intermediate* levels) contain pointers to the next level tables. The last level contains the elements themselves (i.e., the FSMs in our case).

When an intermediate table is allocated, all pointers are set to `nullptr`. When a worker W reaches a `nullptr` on the path to an FSM, the next levels of the table need to be allocated. To that extent, W first allocates the next level of the table before attempting an atomic CAS on the null pointer to make it point to the freshly allocated next level of the table. If the CAS fails, it means that another worker has concurrently allocated the next level so W can free the allocated next level and proceed. This wastes an allocation and initialization but happens extremely rarely, though. When allocating the last level of the table, all FSMs must be initialized to the NO_ACCESS state. Non-null pointers in the intermediate levels of the table being constant, no synchronization is required for concurrent access once initialized.

This container supports constant-time wait-free concurrent random accesses. It is also compatible out of the box with a 64-bit physical memory map with a memory usage dedicated to FSM storage close to the real memory usage of the simulated software.

Indeed, each reduced address (typically each 8-bytes block) is associated to a 4-bytes FSM instance and contiguous addresses have contiguous FSM except at the last level arrays boundaries. As a result, and due to the principle of spatial memory locality, last level tables tend to be well used, that is a good proportion of the allocated FSM instances are accessed.

A software caching system that memorizes the lastly accessed page(s) of FSM could also be implemented to reduce the number of table traversals, but the additional logic cancels the theoretical benefit with a 64-bit simulated address space. Though, this could be profitable in a 128-bit context as the depth of the page table would roughly double to keep a good sparsity. The lookup would then take twice as long on average and is likely slower than checking if the targeted page is software-cached.

We have implemented a template-parameterized page table allowing us to experiment several configurations. Such structure presents a speed-memory trade-off as increasing the number of levels tends to improve the allocation granularity but increases at the same time the number of pointer indirections required to access an element. The best configuration can depend slightly on the simulated platform but usually lies between 3 and 4 levels. Using a larger first level (allocated only once) and successive levels of decreasing size seems to give the best compromise on the experiments conducted but the impact is small. Also stepping down to 2 levels in case of a 32-bit target platform does not show any significant speedup ($\sim 1\%$).

We eventually chose a 3-level table with the first level array containing 2^{23} pointers to second level arrays, each containing 2^{21} pointers to third level arrays, each containing 2^{20} FSM instances. This configuration can cover all individual bytes in a 64-bit address space ($23 + 21 + 20 = 64$). The table geometry is statically configured to generate optimized compiled code. As a result, the full 64-bit space is covered, even in case the user choses to run SCale with a 1-byte address resolution. It should be noted that the simulated platform has a 32-bit address space, but we chose to configure SCale 2.0 for the more general 64-bit case.

3.3 The Sequential Evaluation Phase

When one or more workers have been unscheduled during the parallel phase, they must complete their execution during the sequential phase. First, we need to choose an order in which to resume them (Section 3.3.1). Also, dependencies can appear during the sequential phase, so we perform dependencies analysis at the end of each sequential phase (Section 3.3.2). In case no conflict is detected but dependencies exist, they are recorded to be used during simulation replay (Section 3.3.3). However, if a conflict did occur, then the simulation is no longer valid so we rollback to start over from the last checkpointed valid state (Section 3.3.4).

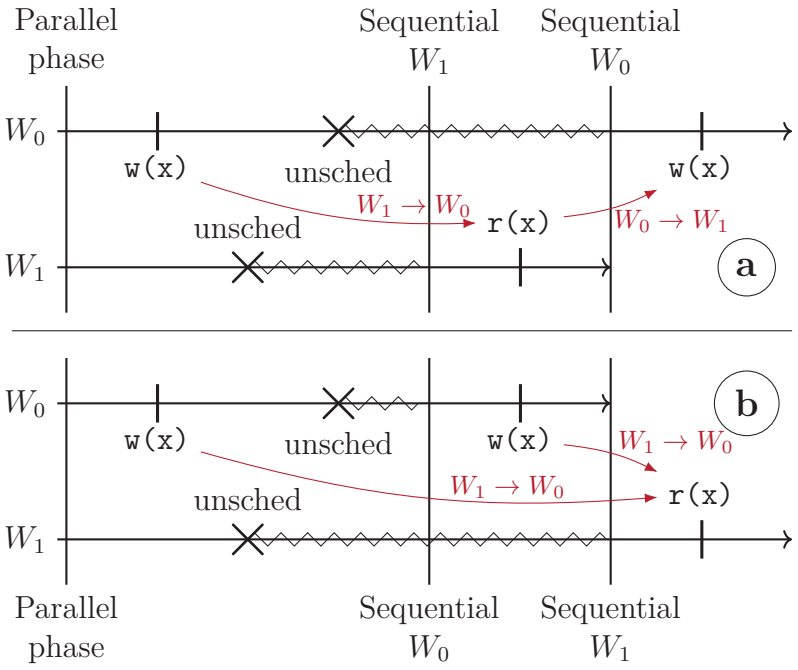

Figure 3.7 – *Two sequential worker scheduling leading to different dependencies.*

3.3.1 Choosing the Sequential Evaluation Order

The workers that have been unscheduled during the parallel phase are resumed during the sequential phase to complete their evaluation. A priori, any worker resuming order is equally valid, but some orders are more likely to trigger conflicts than others. Indeed, the dependencies formed during the sequential phase depend on the order in which workers are resumed. Figure 3.7 illustrates how the sequential scheduling can cause a conflict or avoid it. In this example, we assume W_0 and W_1 are unscheduled because of an access to an already OWNED address not part of the illustration. Independently from this unscheduling access, they both access the shared variable x during the evaluation phase. If W_1 is scheduled before W_0 like in case **(a)** a conflict is formed as W_1 reads between the two writes of W_0. However, if W_0 is scheduled before W_1 like in case **(b)**, both writes happen before the read from W_1, avoiding the conflict.

This simple example illustrates the need for an efficient sequential scheduling policy. Having no existing dependency at the beginning of the sequential phase, these cannot be used to constrain the scheduling order. Ideally, the memory accesses to be performed by each worker would be known in advance and the sequential phase could be scheduled according to that. In that case, all avoidable conflicts could be avoided. However, the memory accesses performed by each worker cannot be anticipated as they result from the simulated software. Also, workers can interact with each other and influence each other's execution in a different manner depending on the sequential phase scheduling order.

In fact, there is a single memory access per worker that is known in advance of

the sequential phase: the memory access that caused the worker to be unscheduled. Specifically, when a worker W_0 is unscheduled due to an access to an address a owned by another worker W_1, it is likely that W_1 has accessed a during the current parallel phase. As a result, it is likely that when W_0 will resume and do the access to a, the dependency $W_1 \rightarrow W_0$ will be created, whatever the chosen sequential scheduling. A reasonable choice, then, is to schedule W_1 before W_0 in the sequential phase to minimize the risk of dependency $W_0 \rightarrow W_1$, which would result in a conflict. If the address a that caused W_0 to be unscheduled has no owner (i.e., it is in the READ_ONLY state), then W_0 might create a dependency during the sequential phase with all the readers of a which are unknown. As a result, the best that can be done is to schedule W_0 as late as possible to increase the odds that it runs after the readers of a.

Based on this principle, to determine the complete scheduling order, every time a worker is unscheduled, a dependency with the owner of the accessed address (if it exists) is registered in a temporary dependency graph G. At the end of the parallel phase, we attempt a topological sort on G to obtain an optimized scheduling order. The workers that have been unscheduled but are not in G have tried to write a READ_SHARED address, thus have no identified predecessor. Thus, these are added at the end of the sequential schedule.

In case G is cyclic, it does not necessarily mean that a conflict will occur. Indeed, the owner of an address that caused another worker to be unscheduled might not have accessed the address during the current evaluation phase. Thus, dependencies in G are likely but not certain. For that reason, we schedule all workers in ascending process identifier (PID) order in case G is cyclic.

We have observed about 40% more conflicts when always scheduling workers ascending PID order compared to using the intermediate dependency graph in Linux benchmarks simulation. An improvement to the ascending PID order when G is cyclic could be to break cycles in G removing specific edges and do the topological sort not to discard all registered dependencies. This has not been implemented and is left to future work.

One thing to keep in mind, though, is that the sequential phase might or might not issue a conflict, independently from the shape of the intermediate dependency graph G. G is only a heuristic which has proven to give significantly better results than random sequential phase scheduling on most applications. Because a conflict might occur anyway, a strict dependencies analysis is always required at the end of sequential phases as explained in the next section.

3.3.2 Asynchronous Dependencies Analysis

Dependencies Analysis algorithm

We have demonstrated in Section 3.2 that dependencies cannot occur during the parallel phase thanks to our memory access granting policy and worker unscheduling. In case

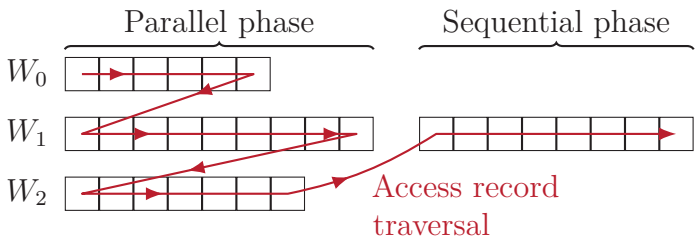

Figure 3.8 – *Structure used to record memory accesses with a valid traversal order used for dependencies analysis.*

there is no sequential phase because no workers have been unscheduled — which is the case more than 99% of the time for most applications — we can simply proceed to the next evaluation phase without any extra precaution.

In case a worker has been unscheduled during the parallel phase, a conflict check must take place to detect either a conflict or determine the dependencies between worker for a later simulation replay. To that extent, during both the parallel and the sequential phases, all memory accesses are recorded to construct the worker dependency graph at the end, if need be (line 11 of Algorithm 4). The stored information for each access is the accessed address, the number of bytes, the type of access and the ID of the accessing worker.

Recording all accesses during the parallel phase in a conventional ordered container such as a `std::vector` would create huge contention as pushing elements requires exclusive access to the container. Fortunately, thanks again to the zero dependencies guarantee, the recording order of memory accesses during the parallel phase does not change the final dependencies. We only need that all accesses recorded during the parallel phase are recorded before the accesses of the sequential phase. The sequential phase then must be recorded in correct order.

Recording the sequential phase in the real order is achieved using an `std::vector` shared by all workers in the sequential phase. For the parallel phase where the order does not matter, we use one vector per worker to guarantee maximum decoupling of memory accesses recording. Accesses can then be enumerated one vector at a time finishing with the vector of the sequential phase like in Figure 3.8. We call the global structure composed of one vector per worker for the parallel phase and a shared vector for the sequential phase an *access record*.

Dependencies are analyzed at the byte level, meaning that accesses that hit the same reduced address but different bytes inside the reduced address will not cause dependencies. For instance, if an access hits 4 bytes, it will be treated as 1 access for each one of the 4 bytes. Doing so reduces false positives at the cost of a slower dependencies analysis.

Specifically, the dependencies analysis consists in enumerating all accesses of the

Step	Worker	Access
1	0	w
2	0	r
3	2	r
4	1	r
5	0	r
6	1	w

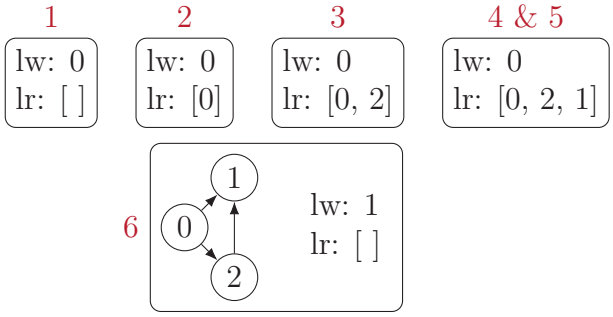

Figure 3.9 – *Algorithm used to record dependencies caused by accesses to one byte. The array describes an example sequence of accesses done at this byte. r and w respectively designate read and write accesses. The global dependency graph and both the last write (lw) and the last readers (lr) are assumed to be initially empty. At the* 6^{th} *step, dependencies since previous write are registered all at once in the global dependency graph.*

access record in a valid order such as in Figure 3.8. For each access, it updates a small data structure associated with each byte and represented on figure Figure 3.9. This structure contains the ID of the last writer and of all readers of the address since the last write. Upon read, the set of last readers only is updated with the ID of the current reader. Upon write, three sets of dependencies are recorded in the *global dependency graph*:

- A dependency between the previous writer and all the previous readers already recorded in the structure.
- A dependency between all the previous readers and the current writer.
- A dependency between the previous writer and the current writer (in case there is no previous reader).

The previous readers are then cleared, and the previous writer is replaced with the current one. Once all accesses have been enumerated, the dependencies between the last writer and the following readers are recorded to avoid missing dependencies due to reads not followed by a write, as the dependencies are normally recorded upon writes.

A topological sort is then attempted on the global dependency graph. If there is no conflict, a linear ordering of the workers involved in the dependencies is produced to be saved in the *output replay vector* alongside the analyzed evaluation phase index. It can later be used to replay the simulation (c.f. Section 3.3.3). If the topological sort fails because the graph is cyclic, there has been a conflict and simulation must rollback to the previous checkpoint (c.f. Section 3.3.4).

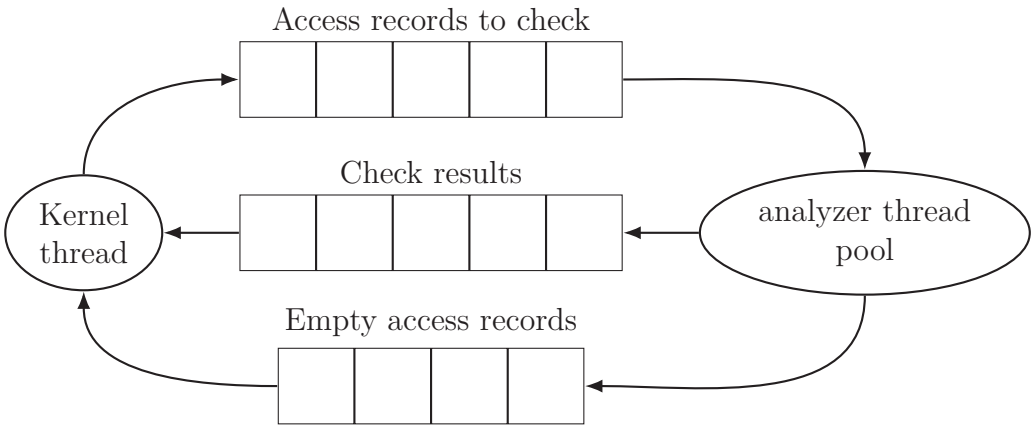

Figure 3.10 – *Asynchronous dependencies analysis infrastructure.*

Asynchronous Dependencies Analysis

While the dependencies analysis could be performed by the kernel thread before the update phase, it is performed asynchronously by another OS thread. The simulation can then proceed without waiting for the check to complete. This optimization can be done as the result of the dependencies analysis is not required for the future of the simulation. Indeed, if no conflict was detected, then the workers sequential ordering is only used in a later simulation replay. If there was a conflict, then the simulation must rollback to the previous checkpoint, and continuing the simulation was useless but also harmless. As there is usually no conflict, continuing the simulation speculatively brings an important speedup that can be higher than $\times 2.2$ on the MobileNet Benchmark for instance.

In order to perform the conflict analysis asynchronously, a pool of OS threads called *analyzer threads* is created at the begininng of the simulation. These threads wait for a new memory access record to be submitted for a check. If a dependencies analysis is required, the access record is submitted by the kernel thread in a queue to be fetched by an analyzer thread as represented in Figure 3.10. When the analyzer thread is done with an access record, it pushes the result in another queue to be fetched back by the kernel thread. The kernel thread fetches pending dependencies analysis results when it is waiting for the workers to finish a later evaluation phase.

Another optimization consists in recycling the access records once they are analyzed. Indeed, growing newly constructed vectors causes numerous memory reallocations and initializations that are expensive compared to the rest of the memory accesses monitoring procedure. For that reason, the vectors that compose the access records are only cleared to preserve their internal buffer. Indeed, the C++ 11 (and newer) standard [C] specifies that the capacity of an `std::vector`, that is the size of the underlying buffer, shall not change upon calls to `std::vector::clear`, effectively guaranteeing that the underlying buffer is not freed.

The impact of this optimization varies greatly with the number of interactions between processes, that is with the number of dependencies analysis, but a $\times 1.3$ speedup has been observed on Matmul when recycling memory access records.

3.3.3 Simulation Replay

SCale 2.0 allows for simulation replay. That is, the first run of a simulation, which we call the *recording run* generates an execution trace that can be used during subsequent runs of the same simulation to reproduce the same behavior. We call such run a *replay run*. Notice that the recording run, while being non-deterministic in the sense that the equivalent sequential schedule of each phase is not predetermined, respects the co-routine semantics so it is standard-compliant.

However, the SystemC standard also requires that for a given model without intrinsic non-determinism (e.g., keyboard inputs, wall-clock-time dependent actions, etc.), all simulation runs that use the same input must produce the same output. Thus, simulation replay is a required feature which is usefull to efficient debugging. Indeed, Simulation replay is useful for debugging purpose to reproduce a faulty behavior, identify its cause and fix it. However, replaying a simulation requires determinism, which is our first motivation in achieving standard-compliant parallel simulation, that is having each evaluation phase be equivalent to a sequential evaluation.

Replay Trace Generation

In order to be able to replay a simulation, it is needed to know which sequential schedule each evaluation phase is equivalent to. This is achieved during the recording run using worker dependencies analysis yielding an equivalent partially ordered sequential schedule after each sequential evaluation phase. Only the workers involved in dependencies appear in this sequential schedule. For instance, if there are 5 workers with IDs ranging from 0 to 4 and the dependency graph after evaluation phase E is

$$2 \rightarrow 0, 2 \rightarrow 3$$

then the equivalent sequential schedule provided by the dependencies analysis can be $2 \rightarrow 0 \rightarrow 3$ or $2 \rightarrow 3 \rightarrow 0$. As 1 and 4 are not part of it, it implicitly means that they are completely independent from the other workers and that they can be scheduled at any time in the equivalent sequential schedule of E. In practice, they are evaluated simultaneously to the sequential evaluation of the other workers.

Instead of a sequential evaluation of the workers involved in dependencies, they could be evaluated in a succession of parallel sub-phases. In the case above, only 2 would be evaluated in the first sub-phase while 0 and 3 would be evaluated in the second sub-phase.down in the used benchmarks, this optimization has not been implemented yet.

To enable simulation replay, the recording run needs to store in a file all the equivalent partial sequential schedules returned by the dependencies analysis. Each

of them is then associated to the index of the evaluation phase they relate to. This identifier is simply an integer incremented before each evaluation phase.

Replay Scheduling

When replay is activated by passing the trace file to the simulator, an *input replay vector* is initialized from the replay file content. The input replay vector then is a list of pairs $(eval\_phase\_index, eval\_order)$. These pairs are sorted in decreasing order so that the earliest phase is at the end of the input replay vector. This way, the next replay information to be used is at the end of the input replay vector and can be popped in constant time compared to linear time relative to the vector size if it was in the front.

The scheduler logic in replay mode is then described in Algorithm 9. Line 3 is responsible for testing if the upcoming evaluation phase has a constrained order by checking the phase identifier of the last element of the input replay vector; Line 4 retrieves this order; Line 5 pops the last entry in the input replay vector; Line 10 performs the constrained evaluation.

Algorithm 9 Replay run scheduler logic outline. *irv* is the input replay vector.

```
1:  procedure evaluateNextPhaseReplay(irv)
2:      ++phaseID
3:      if phaseID == irv.back().phaseID then
4:          seq ← irv.back().orderedworkers
5:          irv.pop_back()
6:      else
7:          seq ← ∅
8:      end if
9:      par ← {pid | pid is ready and pid ∉ seq}
10:     parAndSeqEval(par, seq)
11: end procedure
```

Memory accesses instrumentation can also be completely disabled in replay mode, making the instrumentation function to return immediately. It is assumed in that case that the model instrumentation was correct during the recording run. Thus, all dependencies have been correctly identified. In that case, assuming the same $2 \rightarrow 3 \rightarrow 0$ constrain on the upcoming evaluation phase. Because 1 and 5 do not depend on any worker, they can run at any moment during the replayed evaluation phase. In particular, 1 and 5 can run in parallel to $2 \rightarrow 3 \rightarrow 0$ that will run sequentially. This replay evaluation strategy provides a speedup by superposing the parallel and sequential phase.

Replay is not only used by the user for debugging purpose but also in case of rollback as developed in the next section.

3.3.4 Rollback-Based Conflict Recovery

In case of conflict, the simulation is no longer valid, and it must rollback to the last valid checkpointed state.

Existing SystemC Simulation Checkpoint and Restore Solutions

While rollback has never been considered for optimistic PDES of SystemC models due to its cost and complexity, it has been used for other purposes. For instance, [KLP+09] proposes a Checkpoint/Restore (C/R) framework for SystemC virtual platform that enables resuming a regular SystemC simulation at different points for debugging purpose. Similarly, [Geo09] is an industrial solution that proposes Save and Restore for SystemC virtual platforms. The implementation is not detailed but it allows to save on boot time by saving the end-of-boot state of the platform and loading it in later simulations.

In [MES+10], the state of a SystemC simulation can be saved by wrapping up the member variables of all modules in a special class called `gs_param`. For instance, `int x;` becomes `gs_param<int> x;`. This wrapper is in charge of saving and restoring each member variable state and to restore it when needed. Simulation kernel state save and restore procedures are also available to save the simulation context. TLM.open [Hel09] offers a similar approach at the module level instead of the member level. Because these two approaches are being done exclusively on the user side (e.g., no system calls) and because it focuses exclusively on the data that compose the simulation state, it is very fast and space efficient. However, it requires a good amount of code changes to update SystemC models using `gs_param`, if the model source code is available at all.

In a completely different approach, [JSD+19] proposes to rely on POSIX's `fork()` to recover from timing errors (e.g. causality violations) caused by temporal decoupling in the context of regular sequential SystemC simulation. The `fork()` POSIX system call allows to rapidly duplicate a process using the Copy on Write (CoW) mechanism. That is, instead of duplicating the whole process memory when calling `fork`, the memory is copied in a lazy fashion every time a page is written by either the caller or callee of `fork`. However, `fork()` cannot be used for multithreaded process checkpointing because threads do not survive forking.

A complimentary and non SystemC-specific approach can be found in the *rr-project* [Rr-]. *rr* —for *record and replay*— runs a process in a virtually sequential environment while recording all its inputs (e.g., data returned by system calls). It allows to replay execution of large-scale processes such as a full featured browser but enforces sequential execution where SCale 2.0 aims at the opposite: adding parallelism to an originally sequential and deterministic simulator. These two approaches to record and replay are not to be confused but could be combined to increase SCale 2.0 compatibility with simulations that rely on external outputs.

In SCale 2.0, rollback is used for a different purpose than all the aforementioned

approaches. Indeed, rollback is not supposed to be used by the user but by the simulation kernel to recover from simulation errors. In that sense, it is similar to optimistic PDES but errors in SCale 2.0 are not timing errors but loss of atomicity errors. In practice, the error frequency in SCale 2.0 compared to timing errors in a typical optimistic PDES makes the rollback approach much more realistic.

Process-Level Rollback Performance Analysis

For simulation rollback, we rely on process C/R using CRIU [Cri]. This tool can perform full OS process state checkpoint to drive and restore a process from these generated files. Also, CRIU supports incremental checkpoints: it can checkpoint only the memory that has been modified since the last process checkpoint. This speeds up drastically the checkpointing operation allowing to increase their frequency. Together with OS automatic file caching or the use of a RAM disk, process checkpoint overhead is limited.

Prior to selecting CRIU for our rollback mecanism, (and even prior to envision the possibility of using rollback in SCale 2.0), we have performed synthetic benchmarking to characterize checkpoint and restore speed. Results are shown Figures 3.11 and 3.12. The test application consists in N threads allocating S bytes of memory between each checkpoint, N and S being the variables of the experiment. After each checkpoint, the application exits and is restored to the last checkpoint. Checkpoint and restore times are measured separately and the average of 8 repetitions is reported to cope with performance instability. Checkpoints are stored on a *tmpfs* file system, that is a directory located in RAM as it provides the fastest speed by a great margin compared to regular storage. Such file system is not persistent across reboot, but it is usually not a concern in the context of process checkpointing for rollback.

Overall, both checkpoint and restore costs have two components: a fixed cost associated to the number of threads in the application and a proportional cost associated to the amount of memory used by the application. The cost associated to the number of thread alone can reach 1.5s when using 128 threads. However, in the context of parallel SystemC simulation, such a high number of thread is very unlikely. Typically, up to 36 threads are used in our experiments with SCale 2.0 as we run them on a 36 core host. In that case, the fixed cost associated to the 36 threads is closer to 120ms. The cost associated with the amount of checkpointed/restored memory depends on the storage medium speed. In the case of a ramdisk, 3GB/s is a typical bandwidth. Based on these figures, a simulation using around 1 GB of physical memory can be checkpointed in less than 500ms while incremental checkpoints take about 120 ms for a few tens of megabytes of modified memory between checkpoints.

SCale 2.0 Rollback Infrastructure

In order for checkpoint and restore to be used as a rollback mechanism, additional mechanisms must be implemented. In particular, one must be able to pass information

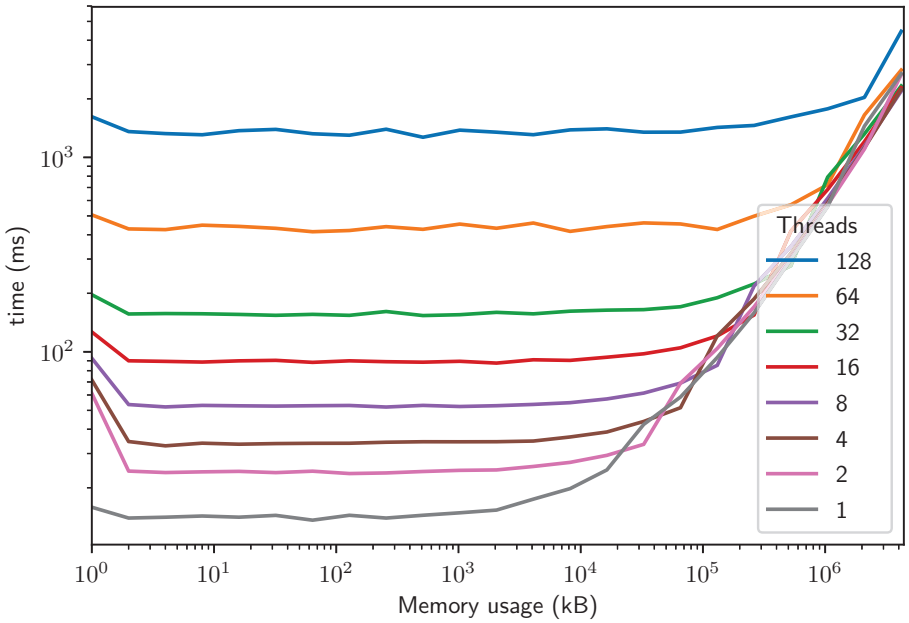

Figure 3.11 – *Checkpoint time to RAM disk using CRIU depending on the number of threads in the application and the total memory used by all threads.*

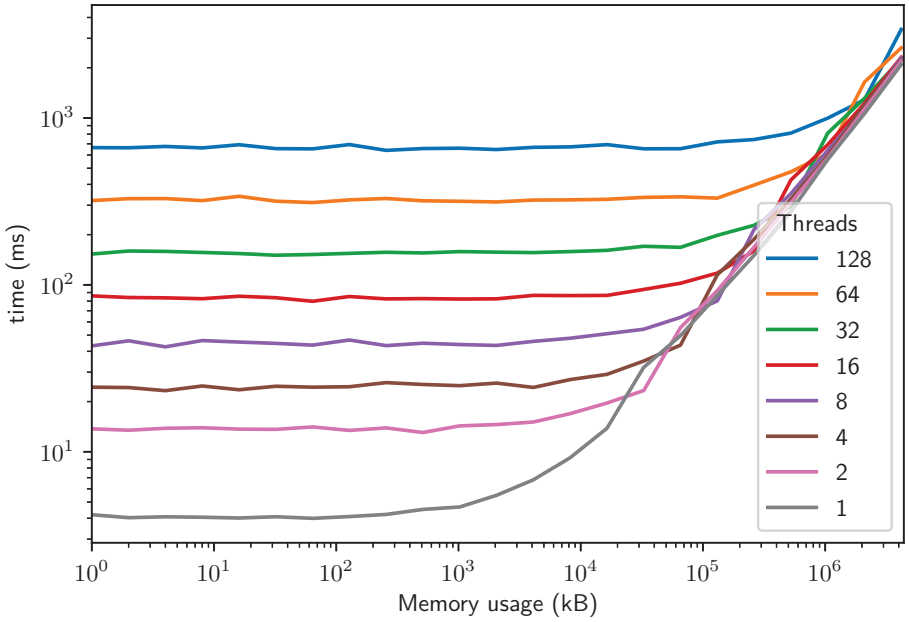

Figure 3.12 – *Restore time from RAM disk using CRIU depending on the number of threads in the application and the total memory used by all threads.*

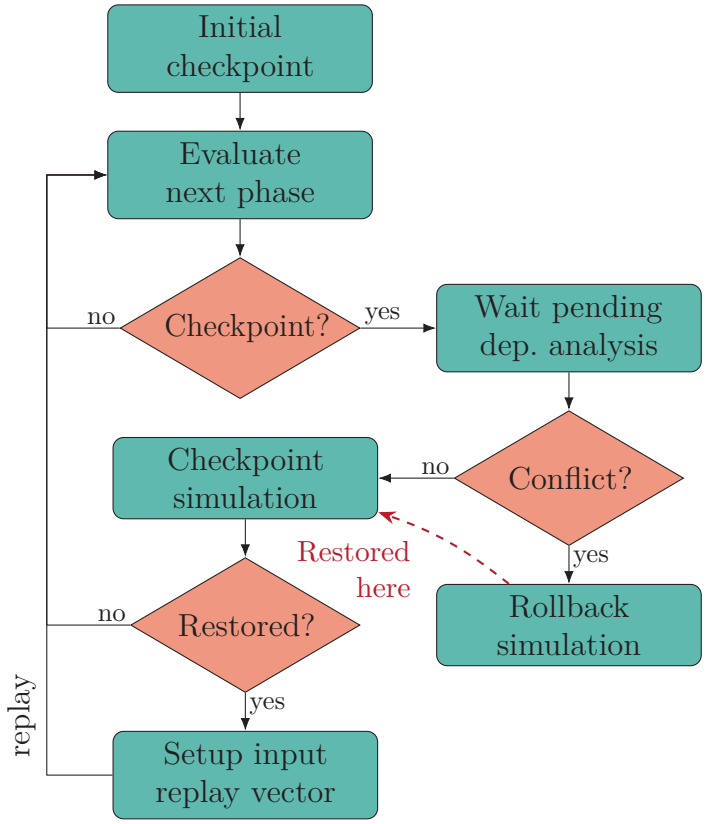

Figure 3.13 – *Rollback-based conflict recovery. Note that when living the "Checkpoint simulation" block, it must be tested if it follows an actual checkpoint or a rollback. Indeed, a rollback restores the simulation inside the checkpoint procedure.*

from the erroneous simulation to the rolled back simulation in order for the later to avoid the error that led to a rollback.

Figure 3.13 illustrates the overall rollback-based conflict recovery logic of SCale 2.0. An initial dump is performed before the first evaluation phase. Then, the simulation runs until a conflict occurs. If so, the simulation is restored to the last valid checkpointed state and run again in replay mode until the conflicting evaluation phase is reached. This phase is sequentially evaluated to prevent the conflict from occurring again. A new snapshot is made right after this phase to be used as the next restore point in case another conflict arises later. Checkpointing once the conflicting evaluation phase is finished guarantees faster simulation progress in case another conflict occurs briefly after. This is especially true as conflicts tend to be grouped together after our experience.

Notice that simulation replay must be used between the restored checkpoint and the conflicting phase. Otherwise, a new conflict could occur before the conflict that caused a rollback in the first place. It would then cause the simulation to rollback to the same checkpoint several times in a raw, preventing progress. It is also mandatory to wait for all pending dependencies analysis to complete before checkpointing to ensure that the

next checkpoint contains a valid state of the simulation.

An important parameter of rollback is the checkpoint frequency. If conflict frequency does not depend on it, checkpoint frequency impacts performance in two ways:

- Each incremental checkpoint takes a few hundred milliseconds.
- The more time since last checkpoint the more there is to simulate a second time in case of rollback.

Figure 3.14 shows the simulation speed depending on the checkpoint interval. The benchmarks used for this experiment are further detailed in Section 4.1.3 page 100. It appears that the impact on global performance does not vary much above a checkpoint interval of about 2 seconds. This is due to the fact that if the base checkpoint frequency is lower than the conflict frequency, checkpoint will be done exactly once after recovering from a conflict, no matter how large the base frequency (flat section on the right of the curves) is. On the opposite, if the base checkpoint frequency is too high, it will dominate the simulation time (decreasing section on the left of the curves).

Still, excessively increasing the checkpoint interval will incur a longer replay phase after each extended conflict-free period, canceling the small benefit of a large checkpoint interval (incremental checkpoints are cheap). These two factors however appear to self-balance resulting in an almost flat curve, which is probably a coincidence. We settled on a 2-second checkpoint interval as a good compromise.

Performing a rollback is useful only if able to memorize the information required to avoid the conflict the next time. In our case, we want to transfer the conflicting phase index and the replay instructions between the last checkpoint and the conflicting phase. The conflicting phase index is stored as an extra replay instruction: if the conflict occurred during evaluation phase P, the entry $P \rightarrow (0, 1, ..., N)$ is added to the replay instructions, effectively causing the entire phase P to be evaluated sequentially.

Figure 3.15 illustrates the principles of the rollback system designed for conflict recovery in SCale 2.0. The rollback infrastructure is packaged as an external library as rollback-based error recovery could be applied to other applications than SystemC simulation.

While the simulation can self-checkpoint by sending a request to the CRIU service, it cannot self-restore. Hence, for a C/R cycle to complete autonomously, we need two processes: the driver and the simulation. The simulation is the actual workload that can encounter errors and require a rollback. The driver is an idle process that only spawns the simulation and wakes up to serve rollback requests from the simulation. The simulation and the driver communicate together with named pipes as the link must survive one of the two processes dying. The CRIU service process (third party software) listens to checkpoint and restore requests on a UNIX-domain socket.

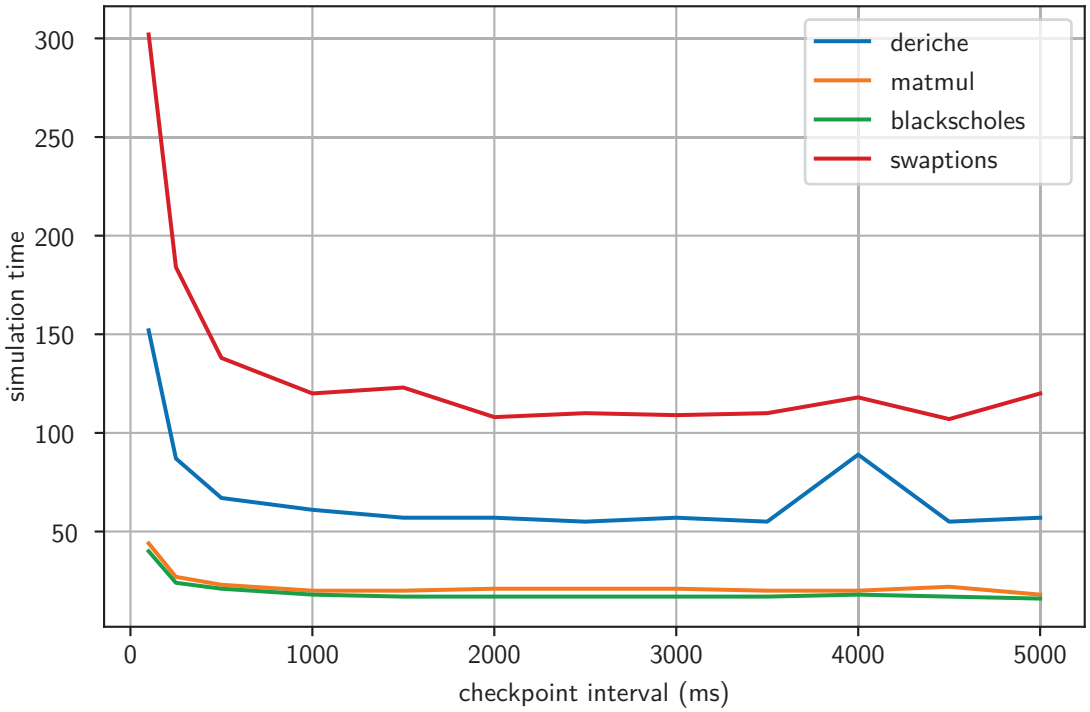

Figure 3.14 – *Influence of checkpoint frequency on overall simulation speed with 32 simulated cores and 32 workers.*

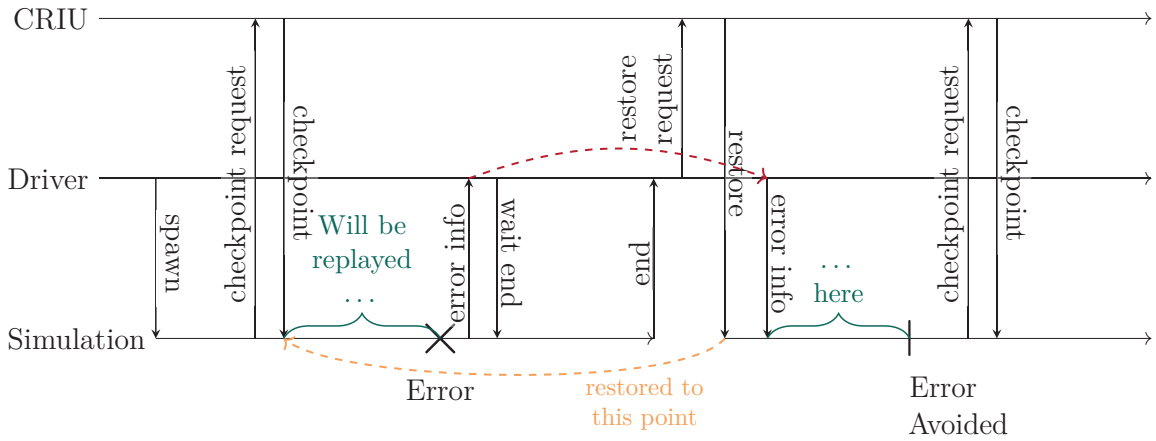

Figure 3.15 – *Protocol of interaction between CRIU, driver and simulation. In SCale 2.0, an error always corresponds to a conflict during an evaluation phase.*

When the simulation needs to be restored, it sends a request to the driver together with some serialized data before aborting. Once the simulation process is done, the driver sends to CRIU a restore request. Finally, once the simulation is restored, it retrieves the serialized data that the driver immediately sends back and starts simulating again. In our case, the serialized data are the replay instructions between the last checkpoint and the conflicting phase.

Technical Limitations of CRIU

However, while being a fast and powerful C/R tool, CRIU has not been designed for error recovery using rollback. It presents several limitations that require some additional engineering on our side to circumvent. The major limitation is that a process can only be checkpointed and restored with the same OS process identifier. However, if such an identifier has been recycled for another process between simulation abort and restore, CRIU will not restore the simulation until the process identifier is freed. In order for rollback to be usable, we had to run simulations in a process identifier (PID) namespace [Man].

A PID namespace is a feature offered by Linux-based OS. It allows to spawn a new process (e.g., a terminal) so that it appears to be the *root* process (PID 1) of the system from inside this process tree only. As a result, listing existing processes from inside a PID namespace only shows the process spawned at the namespace creation and its children.

A subtlety of PID namespaces is that the root process is not *init* as it usually is on a Linux system. Among other things, the init process is responsible for adopting *orphaned* processes, that is processes whose parent died. When a process terminates, it becomes a *zombie* until its parent *waits* on it (it is called *zombie reaping*). When the parent is either a shell or the init process, zombies are instantly reaped as both shell and init expect their subprocesses to finish. But in the context of PID namespaces, orphaned processes — which become children of the root process — never get reaped if the root process chosen to spawn the namespace is not designed to do it.

Zombies are usually not a big issue as they only occupy an entry in the PID table. But in our case, the simulation PID must be freed so that it can be reused by CRIU to restore the simulation. In other words, the simulation must be reaped every time it asks for a restore. If the driver were the parent of the simulation, it could take care of that. However, the simulation is not spawned as a child of the driver but rather as a daemon, that is as a child of PID 1. This is due to the fact that CRIU cannot restore the parent-child relationship of a process, so it always restores processes as children of init. As a result, a special root process called *dumb-init* [Kue] is used at the PID namespace creation. It mainly takes care of zombie simulations reaping for us so that CRIU can restore them afterward.

Additional minor limitations are CRIU's flat refusal to restore processes with handles to files whose size have changed. For instance, if a file contained $35kB$ at checkpoint

time but is $42kB$ at restore time, CRIU refuses to proceed for safety reasons. This could be fixed as a patch on CRIU side to ignore such check (thus overwriting the content written after the last checkpoint) or by truncating all files to their checkpoint size before sending the restore request. The simplest fix is to flush streams to files only right before each checkpoint, when the simulation has been fully checked up to this point and it is guaranteed it will never need to revert to an earlier checkpoint.

Also, CRIU accesses some kernel facilities restricted to `root`, thus requiring starting its service manually or to run the whole simulation as `root`. Lastly, incremental memory dump is not supported for the first checkpoint after a rollback, requiring a full and slower dump. As a result, even if memory can be checkpointed and restored at close to $3GB/s$, freeing unnecessary memory before checkpoints can be beneficial to both restore and non-incremental checkpoint speed. We do so by freeing all memory access records used as part of the dependencies analysis system. The cost of reallocating access records buffers every 2 seconds or so is negligible. This limitation could however also be relatively easily fixed on CRIU side and is currently under discussion on their side.

3.4 Generalization to Any Shared Resources

As explained in the Section 3.1, we have assumed up to this point that the worker interactions only come from shared model memory accesses. However, interactions could happen at many other locations of a SystemC model. For instance, the interrupt management system of an ISS such a QEMU is a complex set of variables read and written from both inside and outside the ISS. All these *shared resources* must be handled carefully.

We distinguish 2 types of shared resources based on the type of interactions they cause:

1. *with side effects*: changing the order of interaction affects the behavior of the simulation processes. For instance, model memory accesses, interrupt raising, or timer component accesses often have consequences on the rest of the simulation.

2. *without side effects*: changing the order of interaction does not affect the behavior of the processes. For instance, incrementing an atomic access counter on a component, allocating system memory, or reading a constant shared variable does not influence the rest of the simulation. However, the value of an atomic counter must never be read by a SystemC process to perform another action or this counter will become a shared variable with side effects.

It is up to the user to detect all potential interactions between SystemC processes, just as when checking interactions between threads in a regular multithreaded program. A number of these interactions cause data races but end up being without external

side effects once protected either using atomic operations or mutexes. The rest of the interactions (i.e., with external side effects) must be protected using our instrumentation mechanism.

To that extent, the user first must define each shared resource *perimeter*. Just as memory bytes are grouped to increase memory access monitoring efficiency, a shared resource such as the interrupt management system of an ISS can either be considered as a set of shared resources (each one of its variables) or as a unique resource. Similarly, a timer module like the one conceptually described in Listing 3 has some internal logic that is shared between several processes (initiators and timer SC_THREAD). Making this timer thread-safe only (e.g., using mutexes) could change the final event delivery order: two interrupts scheduled at the same date could be triggered in a different order depending on which one has been registered first. This might influence the rest of the simulation. Consequently, this makes the timer a resource with external side effects, thus requiring monitoring accesses to its internal data. The question is again whether the timer should be considered as a single resource or as a set of resources. Our experience shows that considering such aggregated resources as a unique resource is often the safest and fastest choice.

At this point, shared resources are identified and delimited, but we have only explained how we can protect a full 64-bit memory map. We generalize this approach by classifying all operations on a resource as reads or writes. The former are operations that do not modify the state of the accessed resource while the later are the rest. Then, we define identifiers for all these resources and associate an FSM to each one of them. The identifiers can be any value that can act as a unique identifier such as contiguous integers or character strings.

In the context of this work, one of a set of contiguous integers is assigned to each non-address resource. An FSM is associated to each one of these identifiers using a pre-allocated vector of FSMs. Finally, the user just needs to insert calls to the provided `generic_instr` function that does the same as the `mem_instr` function but for the other resources designated as *generic resources*. In case it is not clear whether some code, for instance hidden inside an ISS, performs reads or writes, it is a conservative choice to chose to declare the whole operation as a write.

All the required elements for standard-compliant parallel simulation of time-decoupled TLM-LT models have been exposed. Chapter 4 will demonstrate that speedup as high as ×21 can be achieved with the exposed functionalities of SCale 2.0. Yet, Chapter 5 will discuss some extra functionalities brought to answer some Linux-specific issues raised by experiments conducted in Chapter 4.

Listing 3 Pseudo implementation of a SystemC timer. Only the important internals and the interrupt scheduling and raising are shown. Most of SystemC boilerplate relative to the definition of a module and its processes is omitted.

```
1  class timer: public sc_module{
2      std::map<cpu*, sc_time> pending_interrupts;
3      sc_event new_pending_interrupt;
4  public:
5      // This method is typically called by an initiator
6      // through a TLM access
7      void schedule_interrupt(cpu* cpu_ptr, sc_time t){
8          pending_interrupts[cpu_ptr] = t;
9      }
10     // This method can be called by both an initiator
11     // or the time SC_THREAD
12     void cancel_interrupt(cpu* cpu_ptr){
13         // sc_max_time() returns the maximum simulated time value.
14         // It is assumed to never be reached.
15         pending_interrupts[cpu_ptr] = sc_max_time();
16     }
17 private:
18     void timer_scthread(){
19         // sc_thread usually never returns
20         while(true){
21             // get_next_pending_interrupt() is supposed to retrieve
22             // the next item in pending_interrupts with respect to
23             // simulated time
24             auto cpu_and_t = get_next_pending_interrupt();
25             cpu* cpu_ptr = cpu_and_t.first;
26             sc_time t = cpu_and_t.second;
27             // If scheduled time is not reached yet
28             if(t > sc_time_stamp){
29                 sc_time to_wait = t - sc_time_stamp();
30                 // Wait until the next pending interrupt time is reached
31                 // or pending_interrupts is updated
32                 wait(to_wait, new_pending_interrupt);
33             }
34             // If wait returned because pending_interrupts has been
35             // updated, scheduled time might not be reached,
36             // hence this test
37             if(sc_time_stamp() >= t){
38                 cpu_ptr->raise_timer_interrupt();
39                 cancel_interrupt(cpu_ptr);
40             }
41         }
42     }
43 };
```

Chapter 4

Evaluation of the Proposed Simulation Technique

4.1 Experimental Setup and Use Cases 98
4.1.1 The Host Computer 98
4.1.2 Simulated Architecture 99
4.1.3 Simulated Software 100
4.1.4 Metrics and Measurement Protocol 102
4.2 Functional Validation 104
4.2.1 Case Study: the Spinlock-Based Barrier 104
4.2.2 Experimental Functional Validation 108
4.3 Performance on Baremetal and Linux-Based Use Cases 109
4.3.1 Baremetal Performance Evaluation 109
4.3.2 Linux Performance Evaluation 113

The previous chapter described the core of the proposed parallel SystemC simulation technique. This chapter now evaluates the efficiency of this technique on both baremetal and Linux-based simulated applications. The experimental setup is presented together with the simulated platform and the benchmarks. A preliminary reflection is also conducted on the behavior of SCale 2.0 during the simulation of a thread barrier. It will help understanding how SCale 2.0 efficiently avoids conflicts when simulating a classic shared-variable-based synchronization like a barrier. The baremetal benchmarks performance is then presented before the first results on Linux-based benchmarks that are further detailed in the next chapter.

4.1 Experimental Setup and Use Cases

4.1.1 The Host Computer

Experiments have been conducted on a 36-core bi-Xeon Gold 6154 server running Ubuntu server 18.04 with kernel version 4.15.1. CPU cores were downclocked at 3.5GHz maximum boost. The P-state frequency driver recommended for this generation of Intel processors does not allow the user to set an arbitrary lower limit for the frequency. As a result, only the maximum frequency has been caped to prevent thermal throttling and the governor has been set to *performance* mode to raise the processor frequency as aggressively as possible.

Two effects must be considered when looking at the performance results. These two effects are very hard to quantify so they are only listed to give a better context for experimental results interpretation. First, even with the most aggressive governor, cores frequency is not maintained at its maximum. In particular, the less heavy-loaded cores, i.e., those executing the less heavily loaded workers, tend to drop their frequency, resulting in abnormally long evaluation times when the load suddenly raises. This effect should not be present on baremetal benchmarks as they are very homogeneous both over time and among workers. It is likely to have more of an effect in Linux-based benchmarks which are less homogeneous.

Second, in a dual-socket host, two NUMA nodes are exposed. This can negatively impact the performance of multi-threaded applications with a lot of data sharing. In the case of SCale 2.0, data sharing between nodes reflects data sharing in the simulated application behavior for one part: if the simulated application shares a lot of data, SCale 2.0 will too, via the shared resources monitoring infrastructure, too. More communication between NUMA nodes is required upon dependencies analysis to transfer the access records of workers that run on a different node than the dependencies analysis threads. Dependencies analysis frequency heavily depends on the simulated application, though.

The biggest downside of having several NUMA nodes is likely to be the latency that SCale 2.0 is subject to during the kernel phases. Indeed, all workers bring their process queues, event lists and other worker-specific data into their node's memory

hierarchy during the evaluation phase. The kernel thread then needs to bring back all that data into its own node consequently to successive L3 cache-misses which are notoriously time consuming when not hidden by prefetching. The simulation kernel's pseudo-random memory access patterns prevent most automatic hardware prefetching. Manual kernel phase optimization was out of the scope of this thesis, despite having a potentially strong impact on performance. Insufficient kernel phase performance is partially hidden by longer quantum durations as explained in Section 4.3.1.

4.1.2 Simulated Architecture

The reference VP used for the evaluation of our contributions is a RISC-V SMP platform illustrated Figure 4.1. Each core is modeled by an instance of QEMU encapsulated in a SystemC wrapper implemented after [CBM+19]. More details are given in Section 5.3.2. The platform is composed of 1 to 32 simulated cores for baremetal benchmarks assigned to 1 to 32 workers. Three dependencies analysis OS threads were used in all cases, saturating the last remaining cores when using 32 workers (i.e., 1 SystemC kernel thread, 32 workers and 3 dependencies analysis threads). For Linux benchmarks, only the 32-core version of the platform is used with a variable number of workers. Simulated cores are connected through a bus to a RAM, a UART, a real time clock (RTC) and an interrupt controller.

A general mindset adopted during the model implementation was to provide maximum simulation speed while using the Accellera kernel before accelerating it with SCale 2.0. Functional validity with maximum speed was the main goal while timing accuracy was a secondary objective. As a result, when adding SCale 2.0 instrumentation to such a fast SystemC model, any small overhead would immediately result in mediocre speedups.

To that extent, DMI is used to access the main model memory. In our case, SystemC simulation of memory accesses only adds to the internal QEMU logic for memory access emulation: a couple of virtual functions calls, bounds checking to assert that the current access targets the RAM, a 4-case switch on the memory access size, and a copy of the value to its destination before returning to QEMU. This is pretty much the shortest memory access simulation that can be done in SystemC. Any extra logic like, for instance, a `b_transport` transaction that reaches cache models and/or a complex interconnect would severely increase memory access simulation time. In particular, the call to `mem_instr` that is inserted right before the actual access to the model memory is required to have a very small overhead.

Also, relatively big quantums are used as explained in Section 4.3.1 so that most of the simulation time is actual guest code simulation instead of SCale 2.0 kernel fiddling with events and processes. This choice has been made as SCale 2.0 event and process management is mostly inherited from SCale 1.0 with a lot of room left for optimization. In order to focus on the evaluation phase performance, which is the target of this thesis, kernel phase time has been reduced by using a larger quantum of about 10,000 ns, that

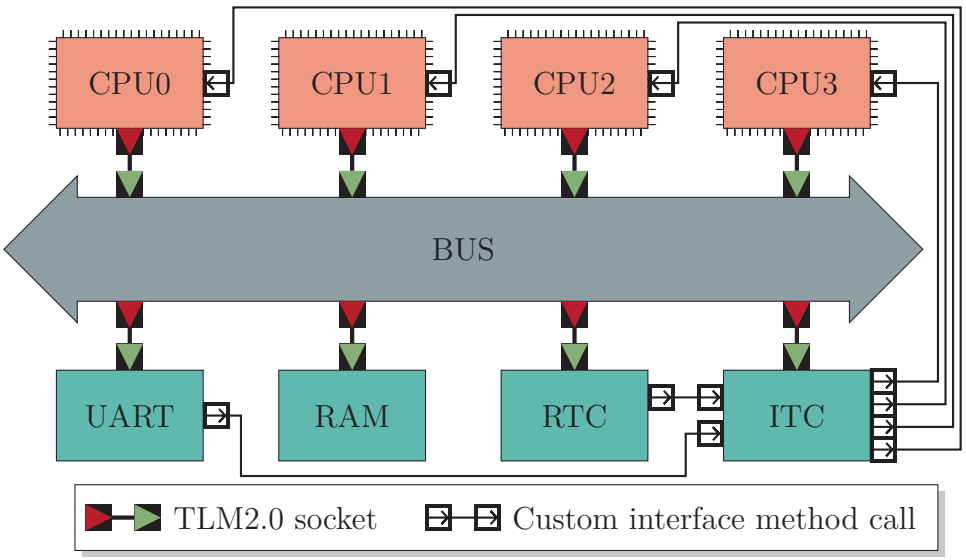

Figure 4.1 – *Architecture of the simulated platform, here with 4 cores*

is 10,000 instructions per simulated CPU in our configuration.

4.1.3 Simulated Software

We have selected five benchmarks to evaluate the performance of the proposed approach. Three of them are implemented both in baremetal, that is to run without the support and complexity of a guest OS, and on Linux:

1. Matmul: 10 iterations of a parallel multiplication of two square matrices of size 512. Each thread computes a horizontal block of the result as illustrated in Figure 3.4. Threads only synchronize between each of the 10 multiplications.

2. Deriche [Der87]: A 10-pass Deriche filtering is applied in place to a 4-megapixel image. This benchmark is composed of horizontal and vertical filtering, making the whole image shared by all threads as illustrated previously in Figure 3.5. Threads synchronize between each pass.

3. MobileNet [HZC+17]: 3 iterations of a 31-layer classification convolutional neural network analyzing a triple channel 160×160 image. The parallelism potential varies depending on the computed layer and much more synchronizations occur than in the first two benchmarks as a barrier is placed between each layer.

All these three applications were ported to Linux using POSIX's threads. Synchronization between simulated threads is achieved in both Linux and baremetal versions using the same spinlock-based barrier. This barrier is used as a case study in Section 4.2.1. No extra system-provided features have been used in the Linux version of the baremetal benchmarks. In other words, except for threads creation, binaries are approximately the

same for baremetal and Linux versions of these three applications. Despite baremetal and Linux binaries being very similar, virtual memory management and file system management will significantly alter the actual behavior of the program.

It must be noted that the Linux variant of Mobilenet have exhibited errors under replay for unknown reasons, yet. This might be due to a subtle bug in SCale 2.0 implementation like a data race or to a badly instrumented shared resource deep inside QEMU. In any case, issues can show up much later after a bug occurs and debugging becomes an extremely difficult task, especially when the issue happens relatively rarely and only with a complex simulated software like the Linux kernel. In fact, debugging the core of SCale 2.0 has mostly consisted in careful reading of the about 3,000 lines of code that compose it, trying to guess which part of it could eventually cause the observed behavior.

The fact that all other benchmarks have shown perfect stability under all conditions while MobileNet under Linux is very unstable raises questions that does not have an answer yet. However, this does not question the theory under SCale 2.0. An efficient debug infrastructure dedicated to complex execution flow produced by parallel execution could help in tracking the implementation/usage error underneath. It must be highlighted that more than 650 simulations of hundreds of thousands of evaluation phases involving up to hundreds of rollbacks and generating up to hundreds of kilobytes of replay instructions have been conducted without a single failure when using the latest version of SCale 2.0.

Two other benchmarks from the PARSEC 3.0 suite [BKS+08] were only used on Linux as they make extensive use of OS-supported functionalities that were not available in our baremetal environment (e.g., dynamic memory allocation or file system accesses):

1. Blackscholes: An Intel Recognition, Mining and Synthesis (RMS) benchmark computing options pricing using the Black-Scholes partial differential equation.

2. Swaptions: Another Intel RMS benchmark computing options pricing but using the Heath-Jarrow-Morton framework.

Despite looking similar from the description standpoint, these two applications have drastically different behaviors when simulated using SCale 2.0. Blackscholes exhibits few conflicts and good parallelism during most of the benchmark duration while Swaptions causes numerous conflicts but mostly during the first part of the benchmark where parallelism is not optimal. The *simlarge* dataset was used in both case to increase the simulation time and reduce variations in measurements.

4.1.4 Metrics and Measurement Protocol

MIPS and time

Two performance metrics are used depending on whether baremetal or Linux-based benchmarks are studied. Baremetal benchmark performances are reported using MIPS as a widely adopted metric. Baremetal applications also exhibit the highest simulation speed in terms of MIPS so they can be used as a reference for SCale 2.0 peak performance when using the QEMU ISS.

However, Linux benchmarks performance cannot be measured in a meaningful way using MIPS due to the Wait For Interrupt (WFI) instruction that happens to never be used in our baremetal benchmarks. When executed, this instruction causes a processor to *idle* until it gets interrupted, which is a mandatory power-saving feature exploited by most if not all modern OS. Natively, WFI is implemented in QEMU as a NOP[1], making idle CPU spin inside the idle loop of Linux instead of truly idling.

While decently fast when using QEMU in standalone mode, that is without it being integrated into a SystemC model, spinning in the idle routine generates an incredible amount of useless memory accesses that severely hamper performances, whatever the SystemC kernel in use is. Indeed, when calling WFI, it is guaranteed that the processor will not be interrupted before it yields as enforced by process atomicity, making polling pointless. As a result, in our platform, the WFI instruction is instrumented on QEMU's side. Instead of only doing a NOP, control is handed over to the SystemC model to handle the WFI as in Algorithm 10. That way, the process of the processor that is executing a WFI can yield instantly for the rest of the evaluation phase (i.e., for the quantum duration). When an interrupt is raised by a peripheral in the SystemC model, the process of the targeted processor returns to QEMU at the beginning of the next evaluation phase. Overall, when a processor executes a WFI instruction and until it gets interrupted, each evaluation phase is simulated in a short constant time.

Algorithm 10 WFI simulation

```
1: procedure waitForInterrupt
2:     wasInterrupted ← true   ▷ wasInterrupted can be modified by other processes
3:     while not wasInterrupted do
4:         remainingTime ← quantumKeeper.getRemainingTime()
5:         wait(remainingTime)
6:     end while
7: end procedure
```

Now comes the question of the MIPS relevance in such a context. Strictly speaking,

[1] A NOP instruction (for No Operation) causes a processor to do *nothing* during one cycle. This is an approximate but satisfying definition in this context. Of course, the program counter is incremented by a NOP, hardware counters might be modified, and superscalar or pipelined processors might still do something else while executing the NOP but this is of little interest in the context of QEMU, which is far from a cycle-accurate processor simulator.

a WFI is a single instruction so until the next interrupt, a single simulated instruction should be accounted for, which artificially lowers the MIPS when idle time increases. Even worse, the longer a processor idles, the lower the *sim_time*/*sim_duration* ratio (lower is better) but the lower the MIPS (higher is better). This demonstrates that MIPS is not a relevant metric in the presence of the WFI instruction.

However, if the quantum duration is, say, 1,000 ns — which corresponds to 1,000 cycles on a 1 GHz processor —, a thousand cycles can be simulated in the time required to loop once inside Algorithm 10, which is basically none. More generally, if the quantum size tends toward infinity, the number of simulated cycles per second tends toward infinity as well, which makes it a bad metric, too.

In the end, we chose to use only wall clock time to compare all results on Linux benchmarks and report speedups as a relative performance metric.

Rollback and Timeline

Timing various actions like, checkpoint, restore or sequential evaluation time during a simulation seems straightforward: it suffices to time each action using regular timing functions provided by the C++ language or by the system and accumulate the resulting durations in counters. But let us focus on the actions that took place since the last checkpoint. In case of rollback, if no care is taken, the durations accounted for these actions would vanish as everything gets reset to the state saved in the checkpointed image, including the time counters.

In particular, restore time would always be zero as from the terminated simulation standpoint, there have been no errors, hence no restores. More generally, everything that took place between a checkpoint and the point where it gets restored is definitely lost if not saved before restore. When profiling a simulation to measure the time taken for various actions, omitting those that took place during these rolled-back periods is not a reasonable approximation.

Hence, a dedicated timeline mechanism has been developed. First, time points returned by `std::chrono::steady_clock::now()`[2] are recorded instead of durations. For instance, when starting a checkpoint, a timestamped *checkpoint_start* event is recorded. When it is done, a timestamped *checkpoint_end* event is recorded. The duration of the checkpoint then corresponds to the difference between these two timepoints. Then, the timeline is split into two sections: the committed and the pending sections. New events are always recorded in the pending section which gets committed right after each checkpoint.

When a rollback is initiated, the committed section of the timeline is anterior to the checkpoint being restored so it will remain after rollback. Though, the pending

[2] `std::chrono::steady_clock` is the only clock in the C++ standard that is guaranteed never to go back in time. On our host system, it is the same as `std::chrono::high_resolution_clock`, which is the most accurate clock with a resolution of 10 ns.

section needs to be serialized to be sent together with the simulation replay data. After rollback is done, the pending timeline is rebuilt using the serialized information and the simulation proceeds. After the next checkpoint — which happens right after the phase that caused a conflict in the first place —, the pending section is committed. In the end, the final timeline contains the timestamps for all events, even those that got rolled back. To determine the time spent on each action, post-processing is applied to a JSON dump of the timeline to compute the various time intervals required.

4.2 Functional Validation

In this section, the functional validity of the approach is studied. The goal is not to perform a formal proof of the technique correctness. The fact that processes can run in parallel as long as they are independent, then get evaluated sequentially and finally dependencies are analyzed to check that they are equivalent to a sequential order is a runtime verification that processes were evaluated atomically.

Whether this principle is correctly *implemented* in SCale 2.0, that is without bugs, is a different question. Using a formal method to verify SCale 2.0 correctness might very well be even more complex than SCale 2.0 itself so this approach was not considered. Instead, we performed an experimental validation, which is to consider as a proof-of-concept more than as a formal proof[3]. This is detailed in Section 4.2.2.

Before diving into the benchmark used for SCale 2.0 functional validation, let us first analyze the simulation flow of a spinlock-based barrier under SCale 2.0 to understanD how process atomicity violation gets avoided efficiently in such cases.

4.2.1 Case Study: the Spinlock-Based Barrier

Atomic Operations Simulation

Before diving into the barrier case study, a discussion about atomic operations simulation must take place. So far, we have considered that all operations on shared resources are either reads or writes. As a reminder, a read is a non-modifying access to a resource, anything else being a write.

What about a hybrid operation such as a fetch-add which reads and then writes to an address? According to the definition of read and write given above, a fetch-add is a modifying access so it is a write from SCale 2.0 perspective.

However, many ISS do not support parallel execution natively as they assume that they are the only instance accessing the simulated system memory. As a result,

[3]As explained in Section 4.1.3, it is proven that the current implementation of SCale 2.0 or of the simulated platform instrumentation presents a small non identified flaw as MobileNet on Linux cannot be replayed properly in certain configurations. It mostly demonstrates that the proof-of-concept implementation of the presented techniques presents a small bug that needs to be fixed but is very unlikely to question the whole approach which works properly in the other tested situations.

they often simulate complex atomic operations such as fetch-add using three distinct operations (and sometimes a lot more): a load, an arithmetic operation, and a store. This breaks the atomicity of the simulated operation. This is the case of QEMU and it is the user's responsibility to restore atomicity in such a situation. At that point, even in classic sequential SystemC, atomic simulation of such an atomic instruction is not guaranteed as a process could yield between the load and the store that delimit the fetch-add operation.

In order to ensure correct atomic instructions simulation, it is the responsibility of the user to prevent a process from yielding during the simulation of an atomic instruction. To that extent, every time QEMU begins to simulate an atomic operation, a flag is set on SystemC side until the atomic operation completes. The process that simulates an atomic instruction is then prevented from yielding as long as this flag is set.

An optimization can then be performed with SCale 2.0 to reduce slightly the risk of conflict related to atomic operations. It consists in instrumenting all accesses to shared resources involved in an atomic operation as writes. It ensures that if the first access that simulate the atomic operations is a read like in the fetch-add case, it will still be considered as a write by SCale 2.0. This provides exclusive access to the targeted address for the first process accessing it. The next processes get unscheduled *before* their first access related to the atomic operation on the shared variable.

On the opposite, if the atomic operation was instrumented as a read followed by a write instead of two writes, several processes might get read access before one of them tries to perform the write. In the case of an atomic read-modify-write operation, no two processes should be allowed to read the initial value of the variable as it violates the atomicity of the operation. Thus, it would lead to a conflict necessitating a rollback. Instrumenting all atomic operations as writes thus reduces the risk of atomicity violation of the atomic operations and thus of the process simulating them altogether.

The Barrier Case

In this section, the spinlock-based barrier case is analyzed. Analyzing the interaction of SCale 2.0 with the spinlock programming pattern gives some interesting insights about how SCale 2.0 behaves. It helps in understanding why most programs that only access shared data protected with mutex-like constructs cannot cause conflicts.

Let us first analyze the barrier code itself. It is given as C++ code in Listing 4. Variables `sense` and `count` are shared amongst all callers of the barrier function as they are static. Each thread T entering the function first saves the current value of `sense`. T then atomically increments `count` and gets the resulting value to compare it against `n`. If the test fails, other threads must be waited-upon and T falls into a loop that waits for the value of `sense` to be toggled. If the test succeeds, then T is the last thread, and it can release the other threads. To that extent, the last thread toggles `sense` after resetting `count` to prepare for the next use of the barrier. T will then pass

Listing 4 C++ implementation of the spinlock-based barrier used in the baremetal benchmarks. It takes the number of participants as arguments, that is the number of threads that must synchronize.

```
1  void barrier(size_t n){
2      static atomic_bool sense{false};
3      static atomic_size_t count{0};
4
5      bool current_sense = sense;
6      if(n == ++count){
7          count = 0;
8          sense = !current_sense;
9      }
10     while(current_sense == sense)
11         ; //LOOP
12 }
```

the loop after a single test and leave the barrier function.

It must be noted that all operations on atomic variables are atomic and sequentially consistent by default. It means that no reordering between operations can happen around the atomic operations. Also, all threads see operations in the same order relatively to atomic operations. In other words, this code exhibits no unexpected behavior and does exactly what is written.[4]

Let us now decompose in Table 4.1 the sequence of load (L) and store (S) operations performed on the shared variables `sense` and `count` by each thread. We assume that the FSMs related to the barrier variables are reset. On step 1, all threads read `sense` before `sense` gets written to on step 4 by the last thread reaching step 2 (last increment of `count`). Thus, `sense` ends up in the READ_SHARED state and all threads are granted read access on step 1 without a single dependency introduced.

On line 2, we assume that the atomic increment of `count` is simulated with a separate load and store. If the load is instrumented as a read, then several processes might be granted access for the load before one of them attempts to do the store. It will obviously result in a conflict as no two processes can read the same value when they are all performing fetch-add operations only. However, if this load is instrumented as a write as recommended previously in the case of atomic operations, then the first worker that executes line 2 (called W_0) sets `count`'s FSM state to OWNED and the following processes are all unscheduled before doing their load. W_0 will then jump to

[4] An optimized version of this algorithm would specify more relaxed memory ordering constraints for each atomic operation. Here, all non-ynchronizing operations can be relaxed, that is without memory ordering constraints. Only the `sense` toggling and the `sense` reading in the loop condition must use respectively the release and acquire semantics.

Step	Code line	Variable	Access
1	5	sense	L
2	6	count	L S
(3)	7	count	S
(4)	8	sense	S
5+	10	sense	L

Table 4.1 – *Sequence of accesses performed on the shared variables of the barrier function:* `sense` *and* `count`*. The line column points to the corresponding line of code in Listing 4. L stands for load and S for store. The* 2^{nd} *step corresponds to the increment and can either be seen as a single atomic operation or as two operations depending on the ISS. The* 3^{rd} *and* 4^{th} *steps are only performed by the thread that unlocks the barrier. The* 5^{th} *step can repeat an unlimited number of times.*

line 5 and loop there until the end of its evaluation phase.

Then follows the sequential phase where all unscheduled workers are resumed at the beginning of line 2. All unscheduled workers except the last to be resumed (which we call W_l) will execute line 2 and then jump to line 5 where they will finish their evaluation phase on the spinlock, one after another. Line 2 introduces a dependency between each worker and the next to execute it but line 5 does not introduce any dependencies (`sense` is still only read).

Up to this point, no worker can depend on the last evaluated worker W_l as W_l has only read `sense`, an address that has only been read so far. Also, whatever W_l does after it is resumed, it cannot make the other workers depend on it as it is the last evaluated worker. So, the last worker W_l will then execute line 2, pass the test and finally execute lines 3, 4 and 5 before leaving the barrier function. Eventually, no conflicts can form until the end of the evaluation phase.

At the beginning of the next evaluation phase, all workers blocked on line 5 will see that `sense` has been toggled so they will be released after a single test. No dependencies will be introduced as `sense` is only read by these workers. The final dependency chain created by the barrier function eventually strictly reflects the order in which workers have incremented `count`.

Finally, as long as memory accesses that are part of an atomic operation are all instrumented as writes, no conflicts can be introduced by the barrier function. Thus, if no data is shared by the code between the barriers (except read-only data), no conflicts can be introduced either by the barriers themselves nor by the rest of the code and the whole benchmark is guaranteed to be conflict-free. This is what we have observed in the baremetal benchmarks as synchronization is exclusively performed with this barrier function.

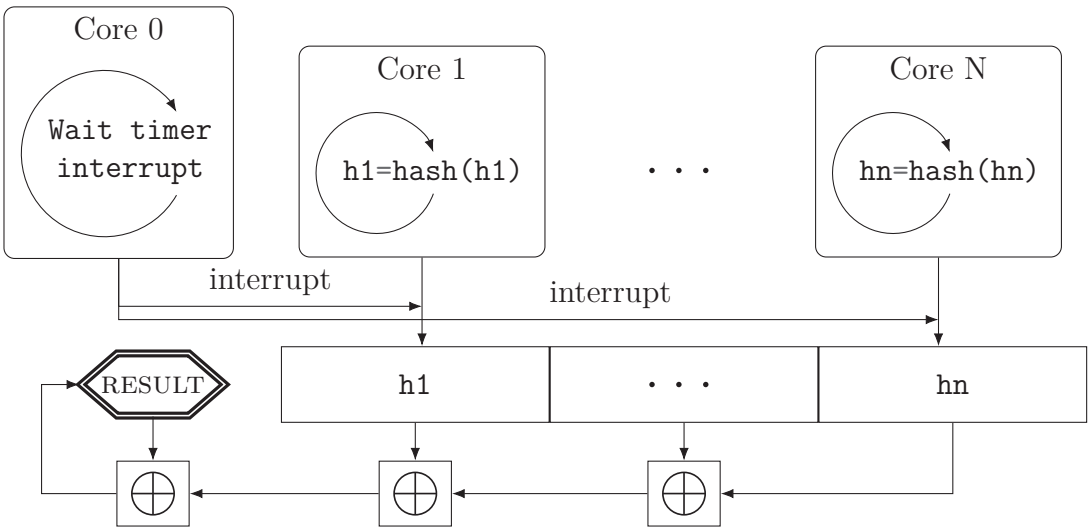

Figure 4.2 – *Principle of the non-deterministic benchmark used to assert SCale 2.0 ability to reproduce highly non-deterministic parallel simulations. Looping arrows symbolize loops while straight arrows are active upon interrupt or value change. All values are combined using a XOR operation (⊕) to yield a single* RESULT *value that is the output of the program.*

4.2.2 Experimental Functional Validation

In addition to performance evaluations, functional validity and robustness of simulation replay have been experimentally asserted using a synthetic baremetal benchmark. The benchmark diagram is presented Figure 4.2. It consists in a master thread (Core 0) that sends software interrupts to a set of slave threads upon reception of a timer interrupt scheduled at regular intervals. Between interrupts, slave threads hash a local variable h_i repeatability in an infinite loop. When these threads receive an interrupt, they all pause and the value of h_i for each thread is collected by the master thread and hashed into a TOTAL variable. This cycle is repeated up to 1024 times varying the timer interrupt interval. The output of the program is the final value of TOTAL which shows pseudo random variations from an execution to the next when **not** using deterministic replay as interrupts are raised and handled at non-deterministic times during each threads' evaluation. To the contrary, the value of TOTAL at the end of the replay runs is always the same as the one at the end of the corresponding recording run.

We also checked on Blackscholes and Swaptions as well as on other complex Linux applications from the PARSEC suite (e.g., ferret, fluidanimate, freqmine) that replay with monitoring enabled exhibits no conflict nor deadlocks caused by atomicity violations for instance. Considering some of these applications can cause hundreds of rollbacks and generate replay traces with tens of thousands of scheduling constraints, we consider this as an additional experimental proof of validity.

4.3 Performance on Baremetal and Linux-Based Use Cases

As the baremetal and Linux-based applications behave differently with SCale 2.0, they are analyzed in separate sections, respectively Sections 4.3.1 and 4.3.2.

4.3.1 Baremetal Performance Evaluation

Baremetal applications present the advantage of being very predictable and homogeneous. They offer the highest simulation speed in most simulators, including SCale 2.0. We use them as semi-synthetic use cases to characterize the impact of our approach on simulation speed (mainly the speedup from parallelization and instrumentation overhead) and as a comparison against SCale 1.0. The baremetal benchmarks presented here never cause conflicts for the reason explained in Section 4.2.1 as they implement a fork-join model based on the same barrier as in the case study. For that reason, rollback impact cannot be observed here. Yet, it is an opportunity to test SCale 2.0 instrumentation and conflict checking without interfering with rollback. SCale 2.0 with all its features enabled is tested in Section 4.3.2.

Figure 4.3 illustrates the impact of quantum size on simulation speed using 32 workers. As expected, increasing the quantum size results in a significant speedup reaching up to 2300 MIPS with Matmul. However, when the quantum gets too large, speed decreases for Deriche and MobileNet and stagnates for Matmul. This is due to the much higher number of synchronizations in a single quantum. Relying on shared variables, each synchronization leads to process sequentializations and FSMs reset. When the quantum increases, the amount of time spent waiting for the barrier in sequential phases because of process unscheduling increases to a point where it is no longer compensated by the speedup in the parallel phase. For the rest of the baremetal evaluations, we use a quantum of 30,000 ns as a performance compromise between the three benchmarks.

To evaluate the influence of memory accesses instrumentation and processes sequentialization, four versions of SCale are compared in Figure 4.4 using 32 workers: 3 variants of SCale 2.0 and SCale 1.0. The fastest variant of SCale 2.0 (1) enables free parallel evaluation. In this variant, instrumentation and process unscheduling are disabled but a few extra protections for atomic memory accesses are added to preserve functional validity. The second fastest variant of SCale 2.0 (2) only disables process unscheduling but instrumentation is preserved. The same atomic memory accesses extra protection as for (1) is used. The last variant (3) is the actual SCale 2.0 with all features enabled. SCale 1.0 is the fourth variant (4).

The overhead of instrumentation and sequentialization compared to fully parallel simulation ((3) Vs. (1)) ranges from 34 to 48%. A part of this speed reduction is due to sequentialization, the rest being caused by shared resources access monitoring. It should be noted that the overhead of instrumentation in Matmul is more than twice as

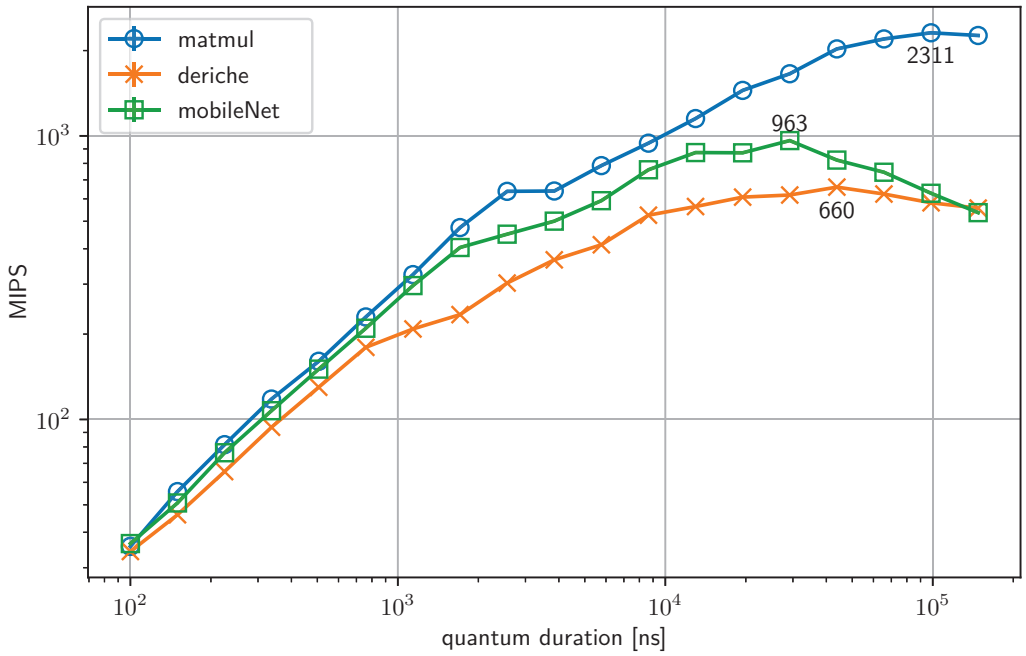

Figure 4.3 – *Simulation speed analysis depending on the simulation quantum size on baremetal benchmarks with 32 simulated cores and 32 workers. The highest point of each curve is annotated.*

big as in the two other benchmarks. This can be explained by the huge base speed of this benchmark. Indeed, Matmul is twice as fast as the other benchmarks.

For its part, sequentialization overhead is hardly compressible as it mostly results from strict co-routine semantics enforcement. Also, the increase in speed compared to [VS16] is significant ranging from ×60 to ×110. It is mostly due to the much faster instrumentation technique together with the asynchronous conflict checking. However, such speed difference implies that SCale 1.0 would lag far behind the reference Accellera kernel in these benchmarks. This is due to the especially demanding configuration of this experiment: memory intensive benchmarks running on 32 cores which saturate SCale 1.0 instrumentation system.

As a reminder, SCale 1.0 memory access instrumentation procedure requires to take 2 to 3 mutexes per memory access. Also, dependencies analysis was performed after every evaluation phase. Such approach brought satisfying result on the model it was tested against when under development. This model was composed of multiple processors with non-coherent caches. As a result, only the traffic leaving the last level cache needed to be instrumented, reducing by about 2 orders of magnitude the number of instrumentations. On the contrary, SCale 1.0 has been tested here without adding the required synchronization that preserve ordering between instrumentation and accesses. Any naive implementation of such synchronization by the user would have further reduced performances by enforcing strong contention on this synchronization

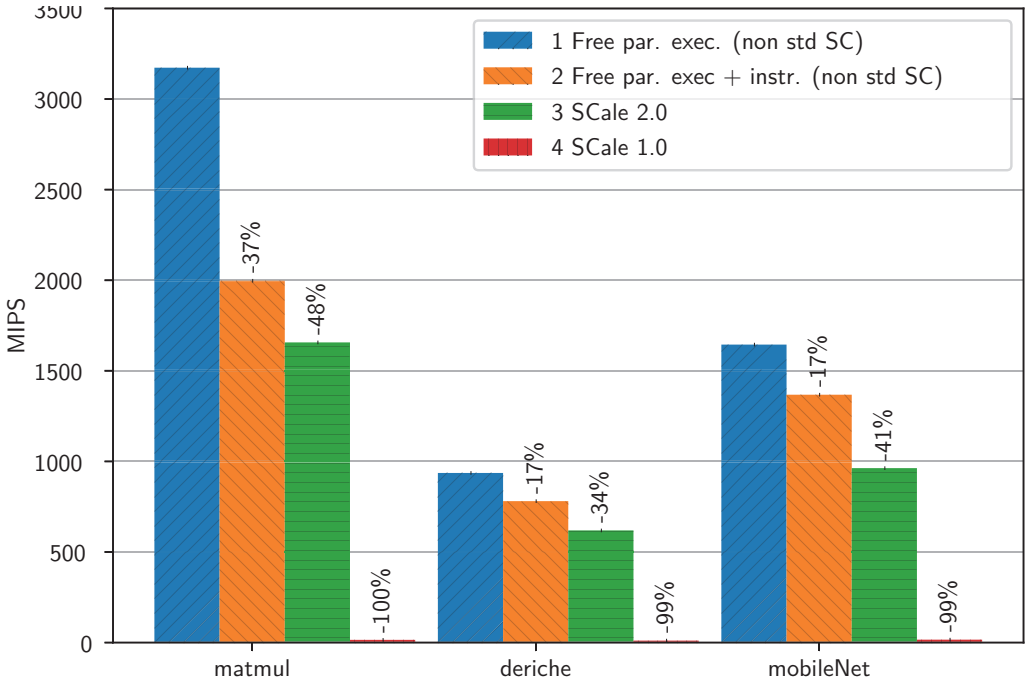

Figure 4.4 – *Impact of instrumentation and processes unscheduling compared to free parallel execution when simulating baremetal applications on 32 cores with 32 workers. Version (1) consists in a parallel simulation without enforcing processes atomicity. It includes the required synchronization for peripheral accesses or atomic instructions simulation for instance. Version (2) shows the overhead of instrumentation without process sequentialization. The same extra synchronization as version (1) is used. Version (3) implements all the contributions of this paper and is standard-compliant. Yet, rollback was not solicitated in these baremetal benchmarks. Version (4) is the same platform model but linked with version 1.0 of SCale from [VS16] without the ensuring correct memory order recording as explained in Section 3.2.3.*

mechanism.

Figure 4.5 illustrates how our simulation kernel scales with the number of workers used to simulate a 32-core platform. It compares SCale 2.0 using N workers against SCale 2.0 using 1 worker. It can be noted that using SCale 2.0 with a single worker is pointless in real world applications but is done here for scaling measurement. If sequential simulation is desired, linking with the Accellera kernel whose SCale 2.0 is a drop-in replacement for is a much better approach. Indeed, in order not to slow down SCale 2.0 in the multi-worker case, the special case with a single worker is not handled differently than the others: instrumentation is still performed, even if no conflict can occur.

Figure 4.5 shows that for homogeneous benchmarks, SCale 2.0 provides significant speedups: almost linear up to 16 workers and reaching between $\times 17$ and $\times 21$ using 32 workers. It must be noted that linear speedup is theoretically not achievable according to Amdahl's law: the kernel phase is sequential and is not negligible starting from 8

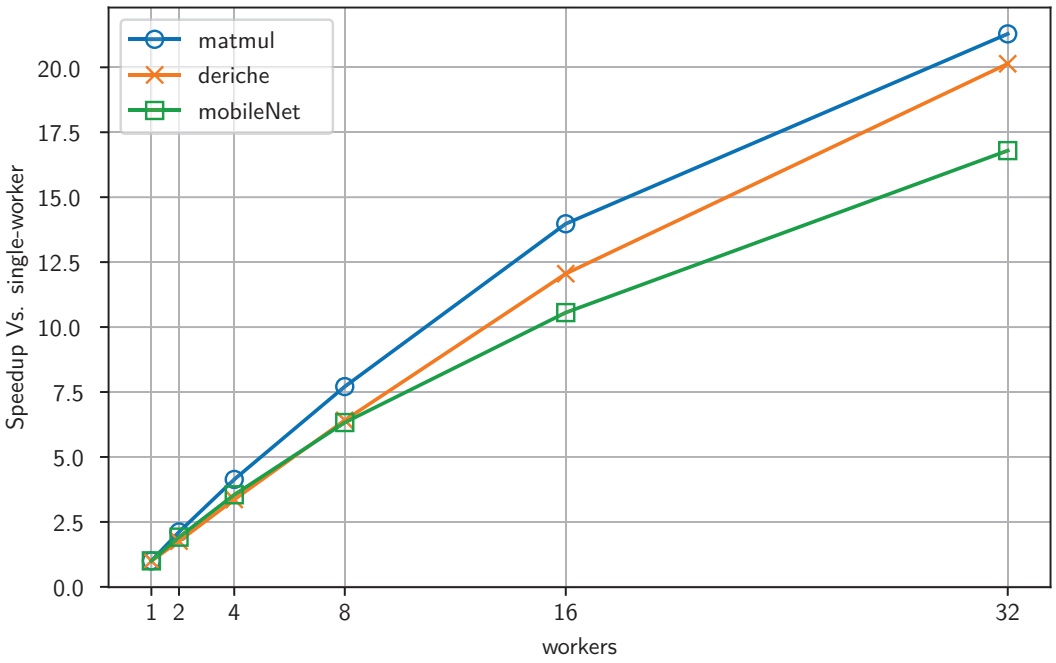

Figure 4.5 – *SCale 2.0 performance scaling with the number of workers of the simulation speed of baremetal benchmark on 32 simulated cores.*

workers. It is demonstrated by the homogeneous worker's host processors idle time observed during the experiments that ranges between 10% and 40% depending on the benchmark, which corresponds to the time they wait for the kernel phase to finish. Kernel phase acceleration is one of the next steps in SCale 2.0 but is not part of this work.

Figure 4.6 shows the impact of simulated platform complexity (number of simulated cores) on speed when always using one worker per simulated core. The goal here is to demonstrate that the simulation speed does not decrease proportionally with the platform complexity like it is the case with a sequential kernel but rather remains stable. As explained above, using the Accellera kernel is faster to simulate a single core platform by about 30% in Deriche and MobileNet and nearly 100% in Matmul due to a simpler scheduler and the absence of instrumentation. Yet the speedup is already greater than ×1 on a dual-core platform simulated in parallel and increases the more cores are to be simulated. Speedup against Accellera kernel reaches up to ×15 on a 32-core simulated platform running Matmul.

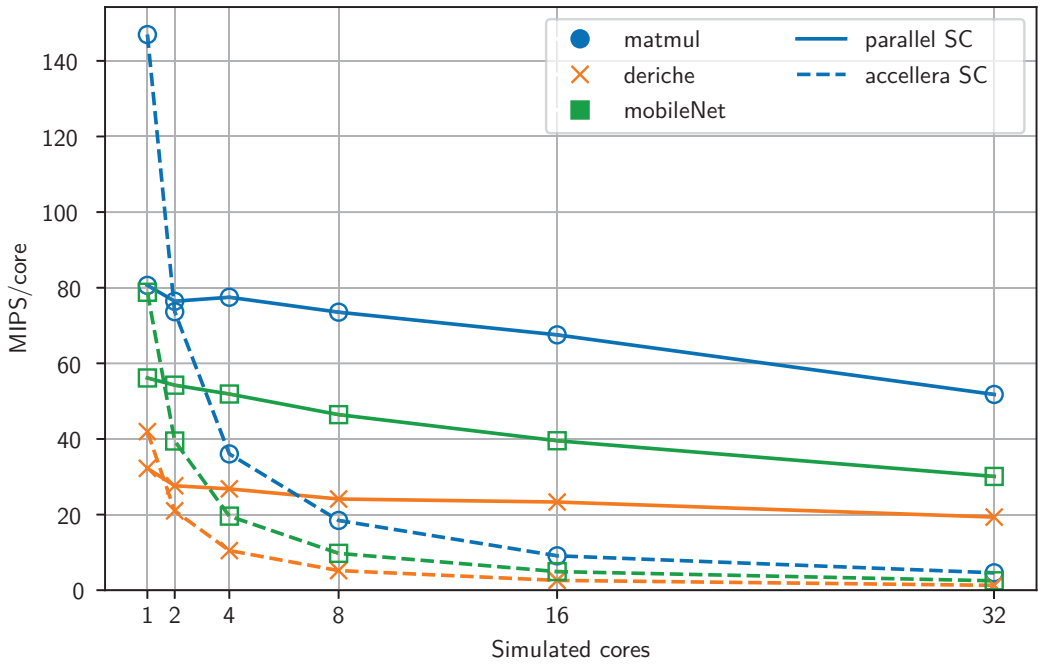

Figure 4.6 – *Baremetal simulation speed per simulated core with parallel (1 worker per simulated core) and sequential simulation (Accellera kernel).*

4.3.2 Linux Performance Evaluation

In this section, SCale 2.0 is directly compared against the reference Accellera kernel. For the reasons exposed in Section 4.1.4, MIPS are no longer a suitable unit of measurement and the raw time spent to complete simulations is used instead.

Figure 4.7 shows SCale 2.0 speedup against the Accellera kernel in both recording and replay simulation run on several benchmarks. The time spent between boot start and end of power off is measured to produce these speedup values. A quick look at the graph reveals deceiving results: speedups cap at $\times 3.2$ during the recording run and is sometimes lower than $\times 1$, going down to $\times 0.5$. During the replay run, results are better with speedup of at least $\times 2$ but never higher that $\times 8.5$.

The bad performance of Linux-based benchmark parallel simulation has several identified causes either internal or external to SCale 2.0. They are investigated in detail in the next chapter but the difference between recording and replay run on Figure 4.7 already indicates that conflicts and associated rollbacks are in cause. Solutions to restore good performances in this context are also presented.

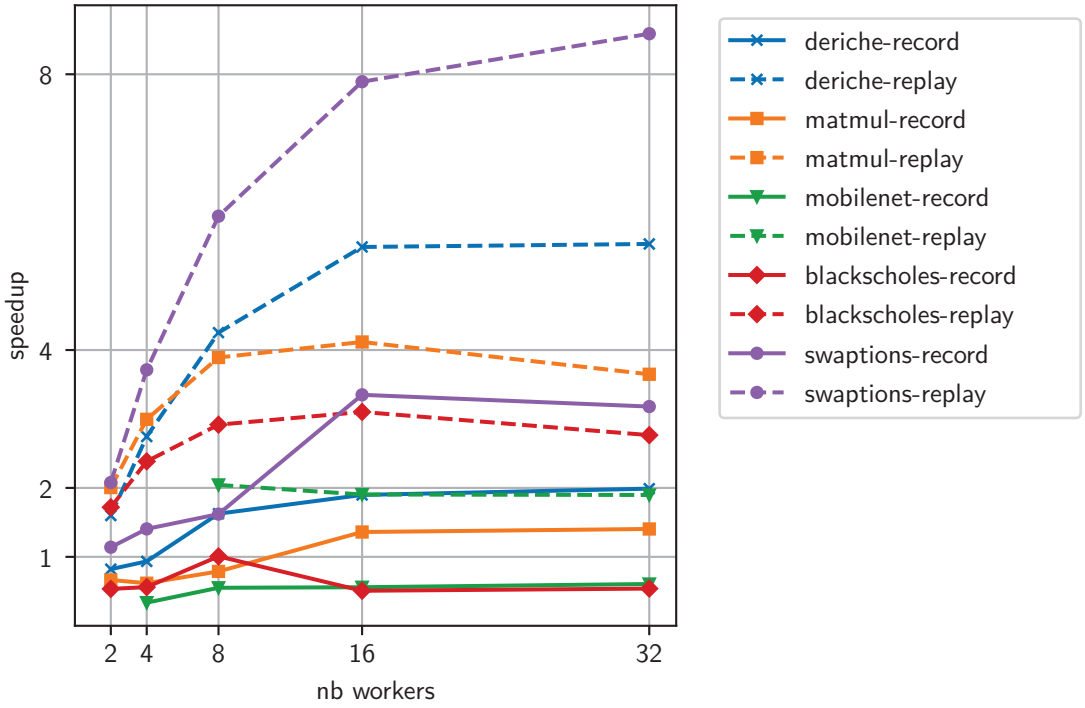

Figure 4.7 – *Speedup of SCale 2.0 compared to the Accellera reference SystemC kernel depending on the number of workers in recording or replay mode on Linux-based simulated application. Results in this graph include the Linux boot and power off. (Due to instabilities using 2 and 4 workers with the MobileNet benchmark, some values are absent from the graph for the MobileNet benchmark.)*

Chapter 5

Full Software Stack Simulation Challenges and Solutions

5.1 Introduction 116
5.2 Investigating the Performance of Linux-Based Benchmark Simulation . 116
5.3 Fast Sequential Mode 119
5.3.1 Region of Interest 119
5.3.2 Variable Accuracy 120
5.3.3 Dynamic Scheduling Policy 122
5.4 CPU-Mode-Based Unscheduling 123
5.4.1 Conflicts Study 123
5.4.2 Executing OS Kernel Code Sequentially 128
5.5 Final SCale 2.0 Performance Evaluation 130

5.1 Introduction

In this chapter, *Linux-based benchmarks simulation* will be referred to as *Linux simulation* for the sake of brevity. We showed that the base principles of SCale 2.0 allowed getting functional correctness on Linux-based benchmarks but were not sufficient to achieve good performance. In this chapter, the cause of these lower performances of Linux simulation will be investigated in the first part of this chapter. First, an overview of the potential causes of slowdown is given to set up the context of the analysis. It is based on a profiling of the simulations to identify accurately the root causes of slowdown that need to be tackled. It will also help set some realistic expectations on the performance level that can be reached in such situation.

Unsurprisingly, the main cause of slowdown is the relatively important number of evaluation phases resulting in a conflict that arise during Linux simulation leading to as many rollbacks. However, this is not the only culprit as Linux simulation exhibits less parallelism potential than baremetal benchmarks simulation because of much more process interactions caused by memory management and file system as well as intrinsically sequential procedures like boot. While the intrinsic lack of parallelism of the simulation cannot be fixed by attempting to add more parallelism, it can be circumvented by adopting alternative acceleration techniques when possible. Linux boot and power off are particularly badly suited to parallel simulation at the TLM level. Thus, SCale 2.0 can fall back to sequential mode in these sections to enable more aggressive simulation techniques only applicable to sequential simulation. It mainly relies on variable-accuracy simulation developed in Section 5.3.2.

Yet, the sections of the simulation that exhibit good levels of parallelism still cause many conflicts. The origin of these conflicts is investigated in Section 5.4.1. It appears to be Linux code itself that makes parallel simulation very hard. Countermeasures are then elaborated. They consist in running sequentially the small portions of Linux code to prevent reliably conflicts coming from them. The resulting performances are finally measured in Section 5.5.

5.2 Investigating the Performance of Linux-Based Benchmark Simulation

Poor performances can result from many factors both on SCale 2.0 side as well as on the model and the simulated software side. For instance, bad parallelism in the *simulated platform* causes unbalanced load between workers. This is not the case here as each worker evaluates the same number of SystemC processes, each simulating symmetrical processor cores. This is verified by the good scaling observed in baremetal benchmarks that would not be possible without a good balance among workers.

Bad parallelism in the *simulated software* can also cause bad load balancing among workers. For instance, simulating an idle core is almost instantaneous in our model

thanks to the WFI instruction simulation technique described in Section 4.1.4. It results in workers simulating more idle cores waiting for those simulating busier cores. This is the case during the Linux boot and power off which are mostly sequential procedures. It also occurs during some parts of each benchmark executed under Linux: setup, result aggregation, etc. are all procedures that are often single threaded. The actual parallel section of the benchmark can thus account for a small part of the whole simulation. For instance, the parallel section of the Blackscholes benchmark accounts for less than 50% of the total simulation duration when simulated on the Accellera kernel, the rest of the simulation being mostly single threaded. According to Amdahl's law, the maximum speedup achievable through parallelization is less than 2 in that case.

In general, when simulating a full software stack that includes numerous sequential sections, the maximum achievable speedup cannot be proportional to the number of workers, but the upper limit is very hard to determine accurately. To eliminate this unknown from the causes of sub-linear speedup, all subsequent measures ignore the boot, the power off and the benchmark loading procedure from the measure to focus on the parallel workload only. These ignored parts of the simulation are also accelerated using variable accuracy and scheduling described in Section 5.3.

In addition, OS provided functionalities such as virtual memory or file system management can introduce massive amounts of synchronizations compared to the benchmark workload itself. Synchronization is mandatory as the role of the OS is to provide threads access to resources they are likely to be competing for (e.g., memory or file system accesses). The resulting numerous SystemC process interdependencies lead to a lot of sequential process evaluations that can be a major cause of sub-linear speedup in some portions of the simulation. Still, sequential process evaluation accounts for less than 5% of the total simulation time in all benchmarks as shown in Figure 5.1, making it a secondary cause of reduced speedup.

It is important to note that the sequential simulation time reported in Figure 5.1 is the time spent in the sequential phases. It does not refer to the time spent simulating sequential parts of the benchmark. For instance, the Linux boot despite being very sequential is not reported as sequential evaluation time except for when sequential phases occur.

Frequent checkpointing cost is also to be considered in the slowdown causes. While baremetal benchmarks require little to no checkpointing as conflicts are rare if at all present, Linux benchmarks require much more of them. As a rough estimate, checkpointing frequency is best at around one checkpoint every 2 seconds as illustrated by Figure 3.14. While incremental checkpointing drastically reduces its cost, each checkpoint causes the simulation to pause (including all its threads). Then, context switches to CRIU for memory dump and then switches back to the simulation. Many small potential slow-down causes add up, like colder caches after checkpoints, possible worker migration on another processor or CPU frequency drop due to the short idle

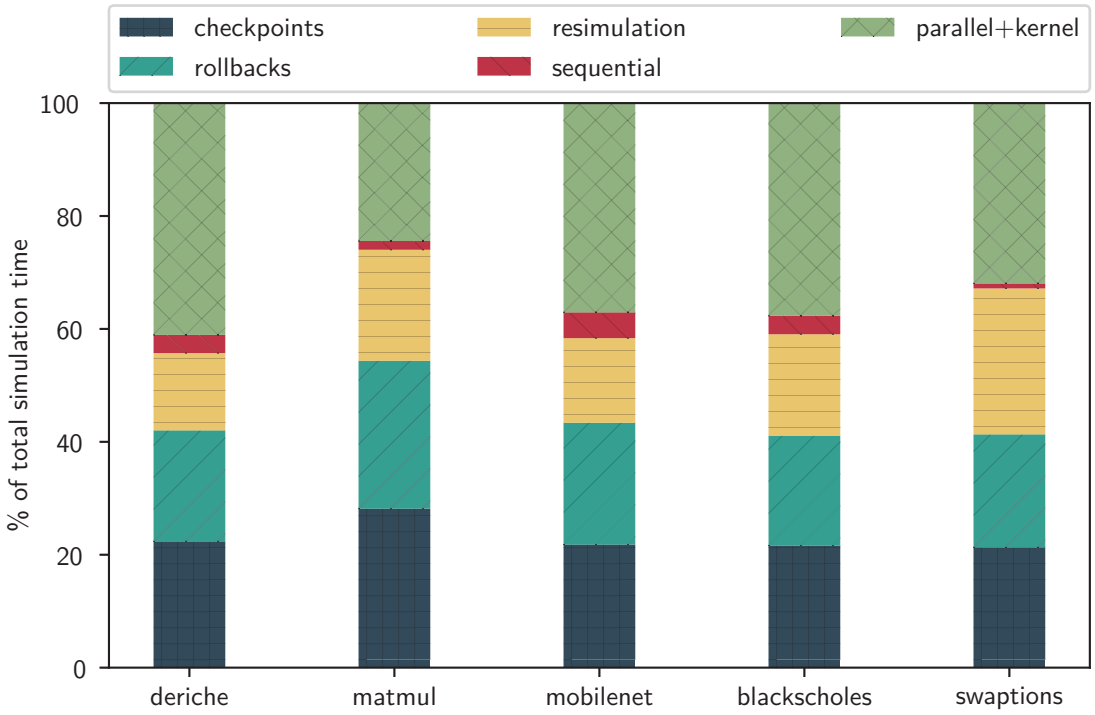

Figure 5.1 – *Analysis of the time spent in parallel and sequential simulation, in checkpointing and rollback procedures as well as the overhead caused by re-simulation after rollback, when using 8 workers (the trend is the same with all numbers of workers). It should be noted that non-parallel simulated code can still be simulated during the parallel phase. Thus, the sequential time only reflects the amount of time spent in sequential evaluation phases, not the nature of the simulated code.*

period. In our case, checkpointing alone accounts for up to 25% of the total simulation time according to Figure 5.1.

However, the cost of the sources of slowdown are dominated by the cost of conflicts as shown in Figure 5.1. Indeed, the following time-consuming steps must be taken to recover from each conflict:

1. Wait for all ongoing conflict checks to complete to know when the first conflict is[1].
2. Terminate the erroneous simulation process.
3. Restore the simulation in the last valid recorded state.

[1]It must be remembered that dependencies analysis is conducted asynchronously. As a result, results from later evaluation phase can be available before those of earlier phases. To give proper replay instructions, the first conflicting phase in phase number order must be identified. A small optimization consists in not waiting for conflict checks of phases that are latter than the earliest known conflicting phase. This has not been implemented as conflict check results collection does not account for a significant enough part of rollback.

4. Redo the simulation up to the last detected conflicting phase (called resimulation).

5. Perform a costly full simulation checkpoint as incremental checkpoints cannot be done right after a restore due to limitations in CRIU.

Rollback and resimulation alone account for close to 40% of the total simulation time. Adding the cost of the non-incremental checkpoint required after each rollback, conflicts account for closer to 50% of the total simulation time. As a result, reducing the number of conflicts to a minimum is essential to preserve overall simulation speed. This issue is tackled in Section 5.4.

5.3 Fast Sequential Mode

As explained in the previous section, some parts of the simulation expose a bad level of parallelism. This can be caused by the simulated software itself. In that case, little can be done on SCale 2.0 side. Instead, an alternative strategy has been adopted to efficiently handle parts of the simulation like Linux boot, power off or benchmark setup. It relies on two components: variable accuracy detailed in Section 5.3.2 and deactivable parallel scheduling described in Section 5.3.3. But first, the notion of Region of Interest (ROI) is defined in Section 5.3.1.

5.3.1 Region of Interest

Some simulated code sections are of little interest to the simulator user who usually wants to analyze the behavior of its platform in the application under development, not the OS boot and power off procedures.

We define the ROI as the part of the code where the user needs more accurate information about the simulated platform behavior such as timing or cache behavior. On the opposite, we assume that the user only needs the platform behavior to be functionally correct outside of the ROI. It generally requires that the instruction set and peripheral accesses are consistently simulated to guarantee progress and valid model state. Yet, timing information or cache hierarchy simulation is of little interests outside of the ROI. The simulation of these elements thus is optional. However, SystemC semantics must be observed both inside and outside of the ROI to ensure overall simulation repeatability. The two next sections develop how this is achieved.

Yet, to adapt the simulator behavior to the section of simulated code, the ROI must be identified. The only solution is for the simulated application to signal to the platform by one way or another the ROI entry and exit points. It is up to the user to find out the most convenient way to achieve that. One solution could be to have a *simulation controller* peripheral with a memory mapped register in which the simulated software can write various codes that correspond to certain events. However, writing into a device requires a (simple) driver to be developed and is not a trivial task.

We have chosen a simpler approach. Instead of using a dedicated peripheral, we have exploited the existing UART peripheral that happens to be very well natively supported by Linux. We have defined a list of strings associated to the ROI start and end: "ROI start" and "ROI end". These strings are then printed at the beginning and end of the ROI respectively using any of the tty printing techniques available (e.g., `printf`, `cout` or `echo`). Then, the UART component has been wrapped to snoop the traffic sent to it. Whenever one of the registered patterns is detected, a callback is called to enable or disable the suitable features described below.

As is, this technique is perfectly suited to dynamic configuration of sequential simulations. However, it is not compatible with SCale 2.0. Indeed, if a string printed on the UART triggers a callback in the middle of the parallel phase to change some arbitrary parameters, it is likely that this will violate some other processes' atomicity. The first solution would then consist in monitoring accesses to these simulation settings that can be accessed by SystemC processes. However, if this setting is read a lot while being written only a few times during the simulation, requiring instrumentation before every read is not optimal. An example of such setting is the one used for variable accuracy simulation and detailed in Section 5.3.2. Instead, we have introduced the concept of *critical process* to better handle such situation.

A critical process simply is a process that is scheduled in isolation before the parallel evaluation phase starts. In the case of the UART driven settings change, delegating callback calls to a critical process prevents process atomicity violation in case these callbacks interact with other processes. Any small process can be defined as critical as an alternative to instrumenting all its accesses to shared resources. In addition, if such process is the only one to write a given resource (e.g., a simulation setting), making it critical also removes the need for instrumenting all the reads performed by other processes on this setting. This is a major advantage in the case of variable accuracy in Section 5.3.2 whose setting value is read before every single simulated memory access. However, because the variable accuracy setting is modified by a critical process only, all regular processes can read it without extra precautions during the parallel phase.

5.3.2 Variable Accuracy

Accurate and Fast mode

To maximize simulation speed outside of the ROI, parallel execution is not always the fastest option if timing accuracy is optional. As explained previously, if the simulated software parallelism is not sufficient, parallel simulation cannot bring significant speedups. However, QEMU on its own can reach very high simulation speeds at the instruction set simulation level thanks to dynamic binary translation. Though, as detailed in [CBM+19], integrating QEMU into a timed SystemC/TLM model requires memory accesses (i.e., load, stores, and fetches) to be instrumented to hand over to the SystemC model when required. This instrumentation, while being a simple pair of function calls followed by a DMI memory access in our case significantly slows down QEMU but still maintains top level performances in the SystemC/TLM models' class.

To take advantage outside of the ROI of the huge raw speed of QEMU while preserving modeling accuracy inside, we have setup *dynamic accuracy* also briefly described in [CBM+19]. It allows to switch during the simulation between what we call *accurate mode* and *fast mode*. The accurate mode enables memory access instrumentation in QEMU so that they are simulated by the SystemC model. This allows for more accurate platform behavior simulation but is about an order of magnitude slower than the fast mode.

The fast mode restricts memory access instrumentation to peripheral accesses that can only be simulated by the SystemC model and instruction counting for rough timing estimation. The memory accesses that cannot be simulated directly by QEMU correspond to the accesses that cannot use the DMI interface provided by TLM 2.0. The other accesses can be executed directly inside QEMU provided that QEMU has access to the DMI pointer of the targeted component.

Optional TLM Memory Access simulation

QEMU executes target code using Dynamic Binary Translation (DBT). When given a target binary to execute, QEMU divides it into Basic Block (BB). QEMU's definition of BB is a bit off of the classic one but it basically consists in a block of consecutive instructions of maximum size with a single branching instruction at the end. Every time the execution flow enters a BB that has not been translated to host binary instructions, the BB is translated and connected to the already translated BB before it gets executed. That way, the behavior of all target instructions is emulated using equivalent fast host instructions and the translation is done only once for each instruction.

Memory access instructions are special in that they have an effect outside of the simulated core, that is outside of QEMU. Natively, QEMU emulates load and store instructions using equivalent host instructions. However, to integrate QEMU to a SystemC model that will simulate memory accesses, all memory accesses in QEMU translated code are instrumented with a function call that optionally delegates the memory access simulation to the SystemC model through a TLM transaction or a DMI access.

The variable accuracy mechanism consists in switching at runtime between using the translated host memory access instruction or the function call to the SystemC model. As a result, QEMU needs to decide for each access whether it can directly execute the host memory access or if it must delegate it to the SystemC model. This decision is to be taken on a per access basis because peripheral accesses cannot be simulated using a host memory access instruction (accessing peripherals control registers usually triggers extra processes) while memory accesses can. To that extent, the user sets the desired behavior on a per-address-range basis. For instance, it can set the RAM to be accessed using host instructions and the UART to only be accessed through SystemC transactions. A flag called `call_systemc` is then attached to each address range set by the user. Every time QEMU is about to perform a memory access, it now first checks

the flag associated to the accessed address and proceeds accordingly.

It must be noted that each component can cancel fast (DMI) accesses (and thus host instruction-based access) whenever it wants using the standard function `invalidate_direct_mem_ptr()`. Thus, the `call_systemc` value for a given address is `false` when and only when both the user (through the variable accuracy setting) and the targeted component (through DMI pointer validity management) allow direct access to the corresponding memory. This way, even if the user sets a peripheral to be accessible with direct host memory accesses, the peripheral will prevent it through DMI pointer invalidation to preserve model functional correctness.

Finally, when the user decides to change the accuracy setting (e.g., to switch from fast to accurate mode on the RAM), the corresponding `call_systemc` flag value is set to `true` on the SystemC model side and QEMU will immediately see the change when it will check it again. Such accuracy setting change can be triggered by any event programmed by the user. In the model presented in this manuscript, the UART is used to react to specific printed strings that correspond to the benchmark ROI bounds. As explained in Section 5.3.1, the accuracy settings are only changed using a critical process in order not to incur process atomicity violations with SCale 2.0.

5.3.3 Dynamic Scheduling Policy

One major drawback of the previously introduced fast memory access simulation mode is that it cannot use SCale 2.0 memory accesses instrumentation, that is `mem_instr` cannot be called before fast memory accesses. This is because that SCale 2.0 Application Programming Interface (API) is not directly reachable from inside QEMU. Also, leaving QEMU generated code always has a very high cost relatively to QEMU speed. Thus, it has been decided that when using the fast memory accesses simulation mode, parallel evaluation would not be allowed as process dependencies analysis cannot be conducted.

To that extent, SCale 2.0 scheduler can now skip the parallel phase to evaluate all workers exclusively during the sequential phase. Whether parallel or sequential scheduling policy is adopted is decided based on a SCale 2.0 flag called `force_seq_eval` which is set by the user. Just like the dynamic accuracy setting, `force_seq_eval` value can be changed in response to strings printed on the UART for instance. In the present work, sequential evaluation is chosen outside of the ROI to preserve process atomicity despite the fast memory access simulation mode. Parallel evaluation is enabled inside the ROI, where the accurate memory accesses simulation mode is enabled.

As a result, Linux boot and power off speed with SCale 2.0 is now much closer to that of the Accellera kernel. The small difference is mainly due to processes being evaluated by workers which all have their own context, and which frequently communicate with the kernel thread. Also, the scheduling logic is more complex in SCale 2.0 than in the Accellera kernel. Though, this difference can be further reduced using of a larger quantum outside of the ROI. Variable quantum size has not been implemented as

efforts have been focused on the parallel evaluation of the ROI.

Overall, Linux boot for a 32-core platform requires under 15 seconds and power off, which is strongly affected by quantum size as it includes a long idle time, takes a comparable amount of time. The rest of this chapter now focuses on the ROI which is the only significantly parallel section of the simulated software.

5.4 CPU-Mode-Based Unscheduling

In this section, the focus is on the benchmark ROI simulation. Only this part of the simulation is done in parallel while the rest relies on fast but less accurate sequential simulation. Also, as a reminder, only the 32-core version of the simulated platform is used with Linux benchmarks to reduce the number of variables. The first part of this section studies the conflicts repartition in all five Linux-based benchmarks. It will give important hints about the main conflict cause in Linux-based benchmarks. The second part presents the solution adopted to avoid these conflicts and drop the conflict frequency to an acceptable level.

5.4.1 Conflicts Study

Experimental Conflict Analysis

We have studied the conflicts temporal distribution in each benchmark ROI using 2, 8 and 32 workers in Figure 5.2 with rollback disabled as it is useless in this context. On these curves, the total number of conflicts that occurred during the ROI is reported as a function of the wall clock time. For instance, on the Deriche benchmark with 8 workers, the number of conflicts rapidly reaches around 75 before stabilizing for a long time. A few more conflicts arise at the end of the ROI.

It is hard to draw universal conclusions based on these benchmarks, yet trends take shape. First, the more workers the more conflicts. It is especially visible in Deriche and Blackscholes while Swaptions shows that 32 workers cause twice as much conflicts as 2 and 8 which are almost identical. MobileNet, for its part, shows around 10 times more conflicts with 8 and 32 workers compared to 2 workers. More workers usually leads to more conflicts because the more workers the higher the probability of an illegal interleavings occuring during their parallel evaluation.

Yet, less workers can cause more conflicts in some situations like with Matmul. Almost 4 times more conflicts occur with 2 workers than with 8 and 32 workers, which is unexpected. The exact cause is yet to be identified but could lie in false positives. For instance, let us assume a 4-core platform simulated on 2 workers W_0 and W_1. Cores C_0 and C_1 are simulated by W_0 and cores C_2 and C_3 are simulated by W_1. Let us now assume that at the end of the evaluation phase, the following dependencies exist: $C_0 \rightarrow C_3$ and $C_2 \rightarrow C_1$. At the process level, no circular dependencies exist, yet at the worker level, the first dependency is $W_0 \rightarrow W_1$ and the second is $W_1 \rightarrow W_0$, which is a

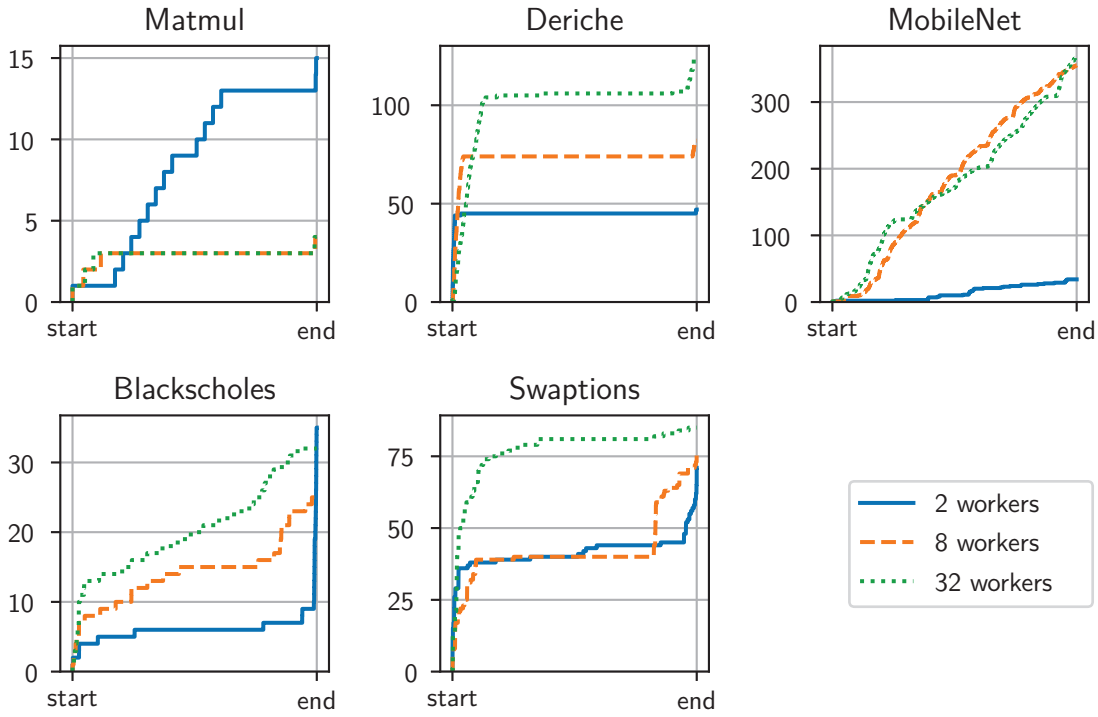

Figure 5.2 – *Temporal repartition of conflicts in ROI when simulating a 32-core platform (cumulative curves).*

conflict. If each process were evaluated on a different worker, the conflict would likely not have existed. However, dependencies are analyzed at the worker level for historical reasons and switching to a process-level mechanism would involve heavy modifications to SCale 2.0 which belong to the future works. Still, less workers can, in particular situations, cause more conflicts than more workers.

The other trend shown by Figure 5.2 is that conflicts are often located at the beginning and end of the ROI. This is especially true for Deriche, Swaptions and Matmul. This holds also for Blackscholes to a lesser extent, yet the curves are steeper at the extremities. MobileNet is the perfect counter example with a very homogeneous conflict repartition during the whole ROI. Let us forget about MobileNet in the reasoning below. Final results will show that the conclusions also apply to it despite it behaving differently at first sight.

Based on the observation that more conflicts occur at the beginning and end of the ROI, let us figure out what is different from the middle of the ROI. The quick answer is the amount of *kernel code* executed at the extremities of the ROI. To explain that, an introduction about binary loading and execution in Linux and most OS is required. This is a big picture that obviously does not describes accurately all the subtleties of a modern OS. A more detailed introduction to ELF files execution can be found in [Dry15].

When a binary gets executed, a new process is spawn. For this new process, a clear virtual address space is created, and the binary is mapped to it[2]. However, the binary content (code, static data, etc.) is not loaded into physical memory in order not to waste memory with instructions and data that might not even get used. Instead, every time a page of memory that does not reside in main memory is hit, a *page fault* occurs and the operating system figures where the expected data lives. If it is in a file like the executed binary, then the relevant part of this file is copied into physical memory and the process resumes from the *page fault* without even knowing that the accessed data was not in memory when the access was initiated.

A good experiment to demonstrate the lazy binary loading phenomenon is to create a C program with a default initialized static array of about a gigabyte[3]. Then, progressively accessing the array at random indices shows the physical memory usage of the process raising gently by a few kilobytes at a time[4].

Let us go back to the benchmarks with conflicts showing up mostly at the beginning and end of the ROI. The three benchmarks derived from the baremetal ones have all their input data and working memory buffer preallocated in the form of static arrays thus located inside the binary. Blackscholes has its input data located in an external file and Swaptions generates it procedurally. Both Blackscholes and Swaptions rely on dynamically allocated buffers for loaded inputs and working buffers. In any case, the first access to any page of memory causes a page fault which results in a complex handling procedure by the OS. Once memory buffers are warmed up, though, page faults become much rarer and OS code is seldom executed, if at all. Regarding the end of the ROI, the only common point between all benchmarks is the thread join operated by the main thread. We did not investigate the thread joining mechanism, but it definitely includes some system-level resource releasing and cross thread synchronization.

From this brief overview of the binary execution mechanism and especially lazy memory management, the biggest suspect in these conflicts is the OS code, like page fault handlers and threading machinery. To validate these assumptions, four versions of the Matmul, Deriche and MobileNet benchmarks have been edited:

[2]In the case of an ELF file [Dry15] for instance, the file does not map directly to memory. Some sections like the text or the data section do, yet the BSS section where all zero-initialized static variables live is only described by its size and the loader deduces the corresponding memory space image.

[3]Runtime initialization would cause the entire array to be mapped to physical memory and defy the purpose of the experiment.

[4]Another interesting experiment consists in comparing zero-initialization of the static array (default behavior if no explicit initialization) versus initialization with non-null data only in the very first cell (e.g., `static char t[HUGE_VALUE] = {1};`). The zero-initialized static array is stored implicitely in the BSS section where only the size of the section in stored, resulting in a tiny compiled binary file whatever the array size. However, the non-zero-initialized static array, even if it contains a single non-zero byte, goes into the data section where it is explicitly initialized, resulting in a binary file slightly bigger than the array size. Yet, executing the binary results in roughly the same tiny memory usage independently from the array size.

Table 5.1 – *Number of conflicts depending on benchmark variant and number of workers*

Benchmark	Variant	Conflicts (nb. workers)		
		2	8	32
Matmul	original	15	4	4
	no join	2	5	3
	warmup	14	5	3
	warmup no join	11	0	0
Deriche	original	50	99	130
	no join	34	107	132
	warmup	2	5	5
	warmup no join	4	0	1
MobileNet	original	91	418	518
	no join	103	411	461
	warmup	16	28	13
	warmup no join	6	5	5

1. *native*: No modifications.
2. *no join*: The end of the ROI is triggered before the final thread join.
3. *warmup*: All data buffers used during the benchmark are accessed once before the ROI starts[5].
4. *warmup no join*: Both 2. and 3.

The number of conflicts in each one of these variations is shown Table 5.1. Setting aside Matmul with 2 workers which behaves abnormally compared to the other benchmarks and configurations, each variant in the list is an improvement over the previous one, with the variants 3 and 4 being significantly better than the others. Indeed, the warmup aims at loading into physical memory all buffers outside of the ROI before they get used inside. It results in the actual code executed during the ROI being much closer to the baremetal code where all data is statically stored into memory even before simulation starts. It must be noted that the warmup is an efficient countermeasure even in the case of the MobileNet benchmark that did not exhibit the same conflict repartition as the other benchmarks in Figure 5.2.

Conjecture on the Conflict Cause

While it has been demonstrated in Section 4.2.1 that a spinlock-based barrier cannot cause conflicts alone and that it can be extended to most spinlock-based synchronization patterns, other programming techniques enable data-race free multi-threaded execution.

[5]This version is not achievable on the more complex Blackscholes and Swaptions benchmarks, hence their exclusion of this experiment.

Listing 5 `foo()` and `bar()` both define two integer variables initialized to zero which they both increment concurrently twice. `foo()` and `bar()` use respectively the lock-based function `incr_lock()` and the lock-free function `incr_lockfree()`.

```
#include <atomic>
#include <thread>
// Lock-based atomic increment of two variables (each increment is atomic)
void incr_lock(int& x, int& y){
    // A boolean flag that can be tested an set atomically
    static std::atomic_flag flag = ATOMIC_FLAG_INIT;
    while(!flag.test_and_set())
        ; // Wait
    ++x;
    // Release one waiting thread
    flag.clear();
    while(!flag.test_and_set())
        ;
    ++y;
    flag.clear();
}
void foo(){
    int x{0}, y{0};
    // Increment both variables in a different order in two threads
    std::thread(incr_lock, std::ref(x), std::ref(y));
    std::thread(incr_lock, std::ref(y), std::ref(x));
}
////////////////////////////////////////////////////////
// Lock-free atomic increment
void incr_lockfree(std::atomic_int& x, std::atomic_int& y){
    x.fetch_add(1, std::memory_order_relaxed);
    y.fetch_add(1, std::memory_order_relaxed);
}
void bar(){
    std::atomic_int x{0}, y{0};
    std::thread(incr_lockfree, std::ref(x), std::ref(y));
    std::thread(incr_lockfree, std::ref(y), std::ref(x));
}
```

Lock-free programming that is based on atomic memory operations is one of them and probably the most famous one. Such programming technique encourages disordered accesses to shared variable as opposed to lock-based patterns that makes threads queue at the entry of critical sections and enter one at a time. These two approaches can be compared on Listing 5.

In the case of the `incr_lock` lock-based function under SCale 2.0, the atomic flag causes all but the first process entering the function to get unscheduled on the atomic

flag before accessing any other shared variable. As a result, dependencies will form in the order in which processes get to successfully set the atomic flag, avoiding a circular dependency from forming. On the contrary, in the case of the `incr_lockfree` function, the first thread might increment `x` while the second is incrementing `y`. When they will try to increment the second variable, they will both get unscheduled by SCale 2.0 but the circular dependency will form anyway during the sequential phase. Note that before the second increment of both thread, there is no reason to think that a conflict might happen from SCale 2.0 perspective: two independent variables `x` and `y` have been accessed by two different threads, which is not a reason for preventing parallel evaluation of any of them. As a matter of fact, the Linux kernel makes heavy use of lock-free code like the Read-Copy-Update (RCU) pattern [MBW12].

One major advantage of lock-free programming against regular lock-based programming lies in performance. Lock manipulations are expensive operations which sometimes are not necessary. The typical scenario where locks are notoriously inefficient occurs in the presence of "often read, rarely written" data. This is precisely the first use case of the RCU where it noticeably increases performance and scalability. While it is still a conjecture, this kind of programming techniques are likely to be responsible of the numerous observed conflicts. Confirming this conjecture is part of the future works.

The conclusions of this study is that the kernel code is responsible for the vast majority of conflicts. The next section proposes a simple solution to that problem that will further demonstrate these conclusions.

5.4.2 Executing OS Kernel Code Sequentially

To reduce the number of conflicts, we have chosen to run all the OS kernel code during the sequential phase. Doing so requires two features:

1. Detecting kernel code execution.
2. Unscheduling the process executing kernel code during the sequential phase.

Even if it does not hold true for every single line of code that compose an OS kernel, kernel code is mostly composed of what is called *privileged code*. At least virtual memory management and file system operations are handled by privileged code. Most ISA's including RISC-V define several privilege levels [WLP14] that are necessary to implementing safety and security features in OS. In RISC-V, for instance, 4 levels of privilege are defined, from the lowest to the highest: user, supervisor, hypervisor, and machine. Each level of privilege increases the set of actions that the CPU can do. The level of privilege of a CPU is called its *mode*.

By default, a program runs in the *user* privilege level as it offers the least freedom and, therefore, the best amount of protection against programming mistakes or malicious

software. When a user level program needs to perform an action that requires a higher level of privilege, it asks the OS to do it instead, either explicitly through a *system call* or implicitly by triggering an *exception*[6]. Either possibility leads to the privilege level of the processor to be raised to execute specific handlers that can serve the request after careful permission control. The most famous example of unauthorized action showing protection mechanisms at work is probably the "segmentation fault". The page fault handler decides that the program is trying to perform an illegal memory access and sends a SIGSEGV signal to the offending process that results in its immediate death unless the signal is handled.

As a result, it is assumed that kernel code is executed with a non-user level of privilege (usually supervisor) and that only user code can run during the parallel phase. In QEMU for RISC-V, the privilege level is represented by a simple `enum` which is only modified in the `set_privilege()` function. A single callback in `set_privilege()` has been added to QEMU to catch CPU mode changes and notify the SystemC model of those changes.

Finally, a `force_sequential(bool)` function has been added to SCale 2.0. This function allows a worker to self unschedule and be executed sequentially when calling `force_sequential(true)`. When this worker calls `force_sequential(false)`, it will run in the sequential phase during the next evaluation phase. The function `force_sequential(bool)` uses the exact same unscheduling mechanisms as the `mem_instr` function. Thus, to run all privileged code (e.g., most system code) sequentially,

```
force_sequential(new_mode != user_mode)
```

is called upon every CPU mode change. An example of evaluation phase that includes privileged code is represented in Figure 5.3.

Finally, if the user code makes use of lockless programming for instance, SCale 2.0 will struggle to prevent conflicts from happening

[6]Kernel loadable modules (i.e., drivers) also give the ability to run privileged code outside of the kernel to support new hardware. Communication with modules uses special files located in the `/dev` directory on all major Linux distributions.

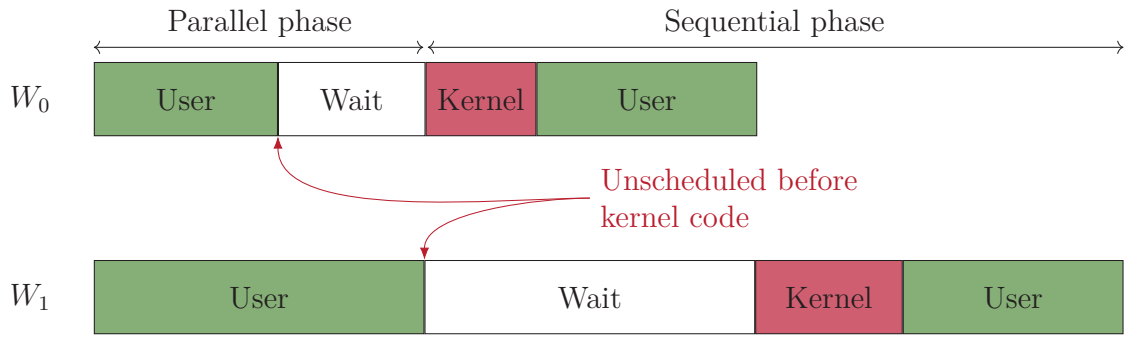

Figure 5.3 – *Two workers that get unscheduled at the moment they start executing kernel (privileged) code. They resume during the sequential phase. Conflicts caused by the kernel code are all avoided that way. It must be noted that the relative size of the user and kernel sections on the figure is not representative of the five tested benchmarks: the user code accounts for the vast majority of the benchmark code in practice.*

5.5 Final SCale 2.0 Performance Evaluation

In this last section, the overall performance gains offered by SCale 2.0 against the Accellera kernel are exposed. The comparison focuses on the ROI as it is where speedups are to be expected. The rest of the code being mostly sequential and of little interest in the user perspective (boot, power off and benchmark setup), it is fast forwarded thanks to the variable accuracy and dynamic scheduling features. The speed of both SCale and Accellera is comparable in these sections when using a 10,000 ns quantum.

The number of conflicts when executing the system code sequentially as well as the speedup in the ROI compared to the Accellera kernel are shown on Table 5.2 and Figure 5.4. It can be observed that the number of conflicts is drastically reduced to between 0 and 3 for the whole simulation (this number varies slightly from one recording run to the other). It leads to a speedup greater than 1 in all benchmarks using 2 to 32 workers. Overall, depending on the benchmark, the recording run speedup ranges from ×9 to ×13 and the replay run speedup ranges from ×11.5 to ×24.

Specifically, Swaptions, Blackscholes and Matmul present very similar speedup profiles during the recording run with almost ×6 using 8 workers but then diminishing

Table 5.2 – *Number of conflicts during the ROI using* `force_sequential()` *with benchmarks Matmul, Deriche, Blackscholes and Swaptions*[7]*, enabling 2, 4, 8, 16, and 32 workers*

Nb. workers	2	4	8	16	32
Deriche	2	1	0	0	0
Matmul	1	1	1	1	1
Blackscholes	2	5	0	0	3
Swaptions	0	0	2	2	3

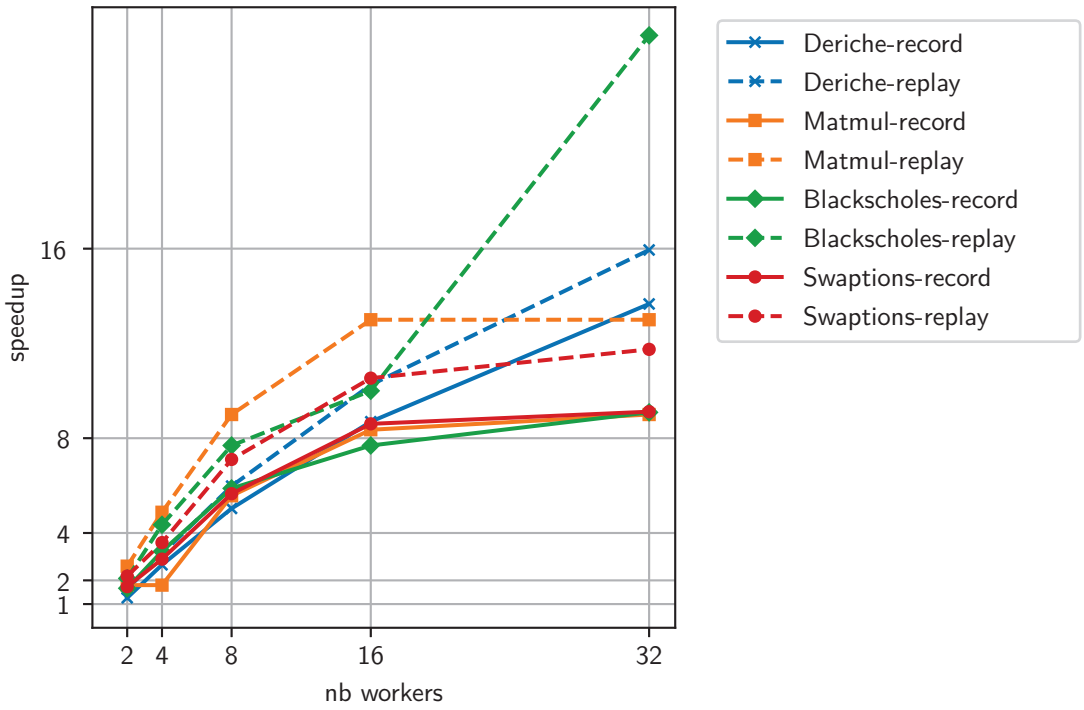

Figure 5.4 – *Speedup Vs. OSCI kernel during the ROI using* `force_sequential()` *with benchmarks Matmul, Deriche, Blackscholes and Swaptions, enabling 2, 4, 8, 16, and 32 workers*

returns appear. Going from 8 to 16 workers raises the speedup close to ×8 and going from 16 to 32 workers only to ×9 on these three benchmarks. Several factors can explain that, including longer checkpointing times with more workers, increased number of conflicts, worse socket locality (18 cores per socket) or non-ideal benchmark parallelism. To explain these results, Figure 5.5 presents profiling results of the recording runs reported in Figure 5.4.

Focusing on the recording run, several results can be explained by Figure 5.5. First, the amount of time spent in parallel evaluation and kernel phases has greatly improved compared to Figure 5.1, proving the great efficiency of sequential OS kernel code simulation. Yet questions remain like, for instance, the bad speedup of Matmul when using 4 workers is the result of a surprisingly high amount of sequential evaluation. The cause of this high sequential simulation time in this configuration remains unknown, though. On the opposite, the good scaling results on Deriche are the consequence of the absence of conflicts using 8 workers and more.

For Matmul (excluding the 4-worker configuration), Blackscholes and Swaptions, it appears that the total time spent on sequential evaluation, checkpoints, rollback and resimulation does not vary much. However, the time spent in the parallel phase and in the kernel does not shrink proportionally to the number of workers, especially between 16 and 32 workers. The cause of the bad speedup between these two configurations is to

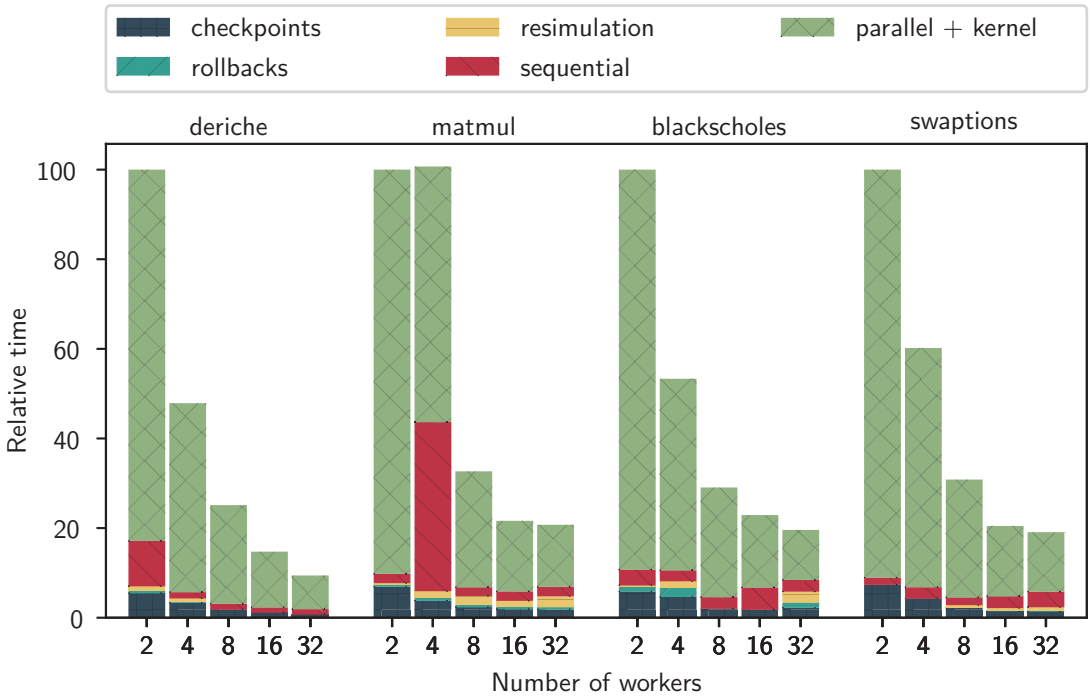

Figure 5.5 – *Profiling of the recording run of the ROI simulation when executing OS kernel code during the sequential phase. All times are displayed using the 2 workers configuration as the 100 base time.*

be found behind this constatation. However, a more detailed profiling would be required to separate the parallel evaluation time from the kernel time. This would require a different tool than the timeline used to generate these figures, though. The outcome would then either exhibit long kernel phases reducing the amount of parallelism or long parallel phases revealing inefficient parallelism. The first is a known issue that will be tackled in future works while the second would require even more investigations to find the cause of this inefficient parallelism.

Though, when including the replay runs in the analysis, two profiles show up: Blackscholes and Deriche that both benefit from going from 16 to 32 workers and Swaptions and Matmul that do not. The first profile corresponds precisely to the benchmarks that demonstrate little reduction in the parallel+kernel time going from 16 to 32 workers in the recording run while the seconds show a clear reduction. Because the replay run involves no shared resource accesses monitoring, no worker unscheduling and no dependencies analysis, these mechanisms that could slow down either the evaluation or the kernel phase are exonerated. As a result, a bad scaling in performance during replay can only be a consequence of insufficient parallelism between processes. If caused by too much replay constraints causing frequent sequential evaluations, then the amount of sequential evaluation during the recording run would be significant, which is not the case. The only option that remains is insufficient parallelism in the simulated software. Still, this is counterintuitive especially in the case of Matmul that

is embarrassingly parallel so this conclusion would deserve a confirmation.

In any case, the proposed technique augmented with sequential OS kernel code simulation (and variable accuracy for less parallel simulated code sections) has shown very good results on demanding scenarios like manycore platform simulation running real-world applications under Linux. The very high level of abstraction of the simulated model and the great native speeds it exhibits make efficient parallelization even more challenging. Achieving such speedups thus is a very satisfying result and the few observations made at the end of this section sets up interesting case studies to further improve SCale 2.0 and better understand very high speed parallel SystemC simulation.

Conclusion

Summary

Simulation speed has increased by several orders of magnitude thanks to the transition from RTL to TLM and the introduction of DMI and temporal decoupling. However, besides the advent of multi-threaded host architectures, raw simulation speed has been stagnating for close to a decade as none of the proposed parallelization solutions was compatible with the aforementioned acceleration techniques.

In this thesis, a new parallel SystemC kernel named SCale 2.0 that tackles this very challenge has been developed. It enables standard-compliant parallel simulation of any SystemC model and its main differentiating features focus on LT-TLM models, the fastest class of models, where it brings an additional order of magnitude in simulation speed. Specifically, temporal decoupling that induces long running SystemC processes and the DMI protocol are supported, including for full stack Linux-based software simulation.

SCale 2.0 shares its fundamental architecture with SCale 1.0. In particular, parallel simulation is achieved by distributing SystemC processes among workers, each worker running its processes sequentially in a different OS thread. Using SCale 2.0 only requires from the user to add a few lines of instrumentation to existing SystemC models to register all accesses to shared resources of the model. Based on this information, SCale 2.0 can unschedule workers to avoid violating processes atomicity. Because of the sheer number parallel accesses to possibly shared resources performed by the workers, SCale 1.0 monitoring mechanism was overwhelmed, leading to poor performances. Thus, the monitoring strategy of SCale 2.0 has been completely reworked compared to SCale 1.0 and heavily optimized to propose the smallest overhead possible while scaling up to high numbers of workers to take advantage of the high core count of modern hosts.

A fundamental new property of SCale 2.0 is the zero dependencies guarantee that ensures that no two workers can interact during the parallel phase. This property enables several vital optimizations like relaxed ordering requirements for access recording and instantaneous dependencies analysis after most evaluation phases. The zero dependencies guarantee is enforced by a shared resources access policy based on an FSM instance attached to every single shared resource of the model like addresses of the

simulated memory. Heavily optimized implementation, storage and reset mechanisms for the FSM have also been introduced, with each one of these components being essential to the final achieved performances.

However, evaluation conflicts, that is process atomicity violations can still occur. SCale 2.0 thus includes a process-level rollback mechanism based on CRIU to recover from such errors. Relying on process-level rollback is not the fastest available option but it enables rollback for arbitrary models without any modification from the user.

This setup enables standard-compliant parallel SystemC simulation of arbitrary models but suffers from too many rollbacks in more complex applications and especially full-software stack simulation including an OS like Linux. It has been observed that Linux kernel code was responsible for the vast majority of conflicts, so it has been decided to prevent parallel simulation of privileged code. By simply monitoring the privilege level of the simulated CPU to evaluate them sequentially when needed, the number of conflicts has been reduced by 2 orders of magnitude in Linux-based benchmarks, restoring good speedups in all tested applications.

Finally, some simulated applications present little inherent parallelism, which translates into unbalanced SC_THREAD complexity leading to poor speedups in SCale 2.0. In case the user is not interested in simulating accurately some parts of the software like the boot procedure, SCale 2.0 provides the possibility to switch to sequential evaluation in conjunction with variable simulation accuracy to fast forward through these parts.

Experimental Results

SCale 2.0 has been evaluated on a 36-core dual socket host. On baremetal benchmarks, speedups reach ×15 against the Accellera kernel with 32 simulated CPU on 32 workers and greater than ×1 speedups starting from 2 simulated CPU in all benchmarks. Equally important is the good scaling exhibited when raising the number of workers for a given model with speedups between ×17 and ×21 with 32 workers compared to using a single worker.

In details, shared resources accesses monitoring overhead has been measured using 32 workers on baremetal applications at 17% on Deriche and MobileNet and 37% percent on Matmul. This is a reasonable overhead when considering the instrumentation-free parallel simulation speed reaching 3,200 MIPS on Matmul, 950 MIPS on Deriche, and 1,700 MIPS on MobileNet, constituting upper bounds to the maximum achievable speed. When including worker unscheduling, the overhead ranges between 34% and 48% for simulation speeds ranging from 800 to 2,000 MIPS. Perhaps the most interesting result to put SCale 2.0 performance in perspective is the comparison with SCale 1.0. While SCale 1.0 was a strong performer when experimented on a previous relatively slow model with much less memory traffic, SCale 2.0 runs 60 to 110 times faster on our demanding SMP RISC-V model based on QEMU and DMI.

Finally, on Linux-based benchmarks, SCale 2.0 has first shown very mitigated results with speedup using 32 workers caped at $\times 3$ and going as low as $\times 0.5$ during the recording run. During the replay run, speedups were better but there was still room for improvement: from $\times 2$ to $\times 8.5$ using 32 workers. This was due to the boot and power off being poorly parallelizable and to the Linux kernel code causing many conflicts in general. These two issues have been mitigated by focusing parallelization efforts on the parallel parts of the benchmarks — the only that can exhibit a good speedup — and simulating Linux kernel code sequentially. As a result, speedups between $\times 9$ and $\times 13$ were achieved during the recording run and between $\times 12$ and $\times 24$ during the replay run, an unprecedented result in the state of the art especially when simulating Linux-based software.

Future Works

Now that it has been identified that Linux kernel code is responsible for many conflicts, it would be interesting to investigate exactly which parts of this code are causing these conflicts to mitigate the cause in a more refined way. Two approaches have been considered: memory accesses pattern analysis and Linux source code analysis. We have experimented the former method that revealed complex conflicting access patterns that require more advanced data processing to deliver exploitable results. In particular, many conflicts involve more than two workers (often all workers) and more than one cycle (often nested and interleaved cycles) in the dependency graph. Also, tens of shared resources can be involved in conflicts, making the identification of specific memory access patterns very hard. This is even harder as the observed conflicting addresses are hardware addresses that cannot be easily associated to software addresses and thus to simulated program variables.

In addition, the influence of quantum size on conflicts needs to be investigated in further details. While a shorter quantum seems to reduce the risk of conflict in each evaluation phase, it also increases the number of evaluation phases. First-hand experience has shown no evident correlation between the quantum size and the number of conflicts so more work is needed.

In any case, SCale 2.0 kernel logic related to notification phase and specifically processes scheduling are not up to the speed of recent ISS's like QEMU anymore. In the context of parallel SystemC simulation, a fast kernel phase is mandatory not to spoil the speedup gained by evaluation phase parallelization. Also, monopolizing an extra core for the kernel is not necessary and wastes resources, especially on smaller workstations. Seizing the opportunity of a SCale 2.0 kernel refactoring, switching to a per-process-based dependencies analysis and scheduling granularity instead of the current per-worker one could be profitable.

Finally, SCale 2.0 provides repeatable simulation on closed models, that is on models that do not communicate with the outer world through, e.g., user I/O or networking. However, enabling such features would instantly cancel repeatability. Coupling SCale

2.0 with a tool like rr [Rr-] for system calls recording would generalize parallel simulation reproducibility to any class of model.

Acronyms

API Application Programming Interface. 122

BB Basic Block. 121

C/R Checkpoint/Restore. 87, 88, 91, 93

CAS Compare And Swap. 72, 73, 76, 78

CEA Commissariat à l'Énergie Atomique et aux Énergies Alternatives. 13

CoW Copy on Write. 87

CPU Central Processing Unit. 13, 25, 33, 36, 44, 47, 51, 52, 54, 69, 98, 100, 102, 117, 128, 129, 136

DBT Dynamic Binary Translation. 121

DES Discrete Event Simulation. 29, 31, 43, 54

DES Discrete Event Simulator. 40

DMI Direct Memory Interface. 15, 18, 34, 36, 37, 40, 43, 44, 55, 99, 120, 121, 122, 135, 136

EDA Electronic Design Automation. 14

ESL Electronic System Level. 13

FSM Finite State Machine. 10, 11, 16, 63, 69, 71, 72, 73, 74, 75, 76, 77, 78, 79, 95, 106, 109, 135, 136

GPU Graphical Processing Unit. 13, 26, 40, 50

HDL Hardware Description Language. 14, 15, 18, 27, 29

HLS High Level Synthesis. 54

HPC High Performance Computer. 51

IMC Interface Method Call. 49

IoT Internet of Things. 13

IP Intellectual Property. 13, 14, 19, 26, 27, 40, 54

ISA Instruction Set Architecture. 25, 128

ISS Instruction Set Simulator. 15, 44, 65, 94, 95, 102, 104, 107, 137

LECA Design Automation & Architecture Laboratory. 13

LIP Laboratoire de l'Informatique du Parallélisme. 13

MIPS Million Simulated Instructions Per Second. 35, 37, 44, 55, 102, 103, 109, 113, 136

NoC Network on a Chip. 26, 27, 34

NUMA Non-Uniform Memory Access. 26

OS Operating System. 19, 25, 26, 47, 60, 61, 84, 88, 93, 99, 100, 101, 102, 117, 119, 124, 125, 128, 129, 131, 132, 133, 135, 136

PDES Parallel Discrete Event Simulation. 37, 43, 45, 50, 51, 52, 53, 87, 88

PFE Projet de Fin d'Études. 13

PID process identifier. 81

RAW Read After Write. 58, 59, 74

RCU Read-Copy-Update. 128

RMS Recognition, Mining and Synthesis. 101

ROI Region of Interest. 119, 120, 121, 122, 123, 124, 125, 126, 130, 131, 132

RTL Register Transfer Level. 15, 18, 27, 31, 41, 42, 43, 45, 47, 49, 50, 52, 54, 55, 135

SIMD Single Instruction Multiple Data. 69

SMP Symmetric Multiprocessing. 54, 55, 99, 136

SMT Simultaneous MultiThreading. 25, 26

SoC System on a Chip. 13, 14, 15, 26, 27, 49, 52

TLM Transaction-Level Modeling. 14, 15, 18, 27, 31, 33, 34, 36, 37, 40, 42, 43, 44, 45, 46, 48, 49, 50, 52, 54, 55, 95, 116, 121, 135

UMA Uniform Memory Access. 52

VP Virtual Prototyping. 14, 27, 35

VP Virtual Prototype. 14, 15, 27, 33, 99

WAR Write After Read. 58, 59, 74

WAW Write After Write. 58, 59

WFI Wait For Interrupt. 102, 103, 117

WID worker identifier. 72

Bibliography

[ARM20] ARM. *Arm Architecture Reference Manual*. 2020. URL: https://documentation-service.arm.com/static/5f106e060daa596235e81ea4?token=.

[Ayn09] J. Aynsley. "OSCI TLM-2.0 language reference manual". In: *Open SystemC Initiative (OSCI), Tech. Rep* (2009).

[Bec17] Denis Becker. *Parallel SystemC/TLM Simulation of Hardware Components described for High-Level Synthesis*. Tech. rep. Dec. 2017. URL: https://hal.archives-ouvertes.fr/tel-01709813v3.

[Bel05] Fabrice Bellard. "QEMU, a Fast and Portable Dynamic Translator". In: *USENIX Annual Technical Conference, FREENIX ...* (2005).

[BKS+08] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. "The PARSEC benchmark suite: Characterization and architectural implications". In: *Parallel Architectures and Compilation Techniques - Conference Proceedings, PACT*. 2008. ISBN: 9781605582825. DOI: 10.1145/1454115.1454128.

[BMC16] Denis Becker, Matthieu Moy, and Jerome Cornet. "Challenges for the parallelization of loosely timed SystemC programs". In: *Proceedings - IEEE International Symposium on Rapid System Prototyping, RSP*. Vol. 2016-Febru. IEEE, Oct. 2016, pp. 54–60. ISBN: 9781467382762. DOI: 10.1109/RSP.2015.7416547. URL: http://ieeexplore.ieee.org/document/7416547/.

[BSV+20] Gabriel Busnot, Tanguy Sassolas, Nicolas Ventroux, and Matthieu Moy. "Standard-Compliant Parallel SystemC Simulation of Loosely-Timed Transaction Level Models". In: *Proceedings of the Asia and South Pacific Design Automation Conference, ASP-DAC*. Vol. 2020-Janua. Institute of Electrical and Electronics Engineers (IEEE), Mar. 2020, pp. 363–368. ISBN: 9781728141237. DOI: 10.1109/ASP-DAC47756.2020.9045568.

[C] C++. *ISO - ISO/IEC 14882:2003 - Programming languages — C++*. URL: https://www.iso.org/standard/38110.html (visited on 05/01/2020).

[CAD20] Zhongqi Cheng, Emad Arasteh, and Rainer Dömer. "Event Delivery using Prediction for Faster Parallel SystemC Simulation". In: Institute of Electrical and Electronics Engineers (IEEE), Mar. 2020, pp. 357–362. DOI: 10.1109/asp-dac47756.2020.9045492.

[CBM+19] Amir Charif, Gabriel Busnot, Rania Mameesh, Tanguy Sassolas, and Nicolas Ventroux. "Fast virtual prototyping for embedded computing systems design and exploration". In: *ACM International Conference Proceeding Series*. Vol. Part F1483. New York, New York, USA: ACM Press, 2019, pp. 1–8. ISBN: 9781450362603. DOI: 10.1145/3300189.3300192. URL: http://dl.acm.org/citation.cfm?doid=3300189.3300192.

[CCZ06] Bastien Chopard, Philippe Combes, and J. Zory. "A conservative approach to SystemC parallelization". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 3994 LNCS. Springer Verlag, 2006, pp. 653–660. ISBN: 3540343857. DOI: 10.1007/11758549_89.

[CHD12] Weiwei Chen, Xu Han, and Rainer Dömer. "Out-of-order parallel simulation for ESL design". In: *Proceedings -Design, Automation and Test in Europe, DATE*. IEEE, Mar. 2012, pp. 141–146. ISBN: 9783981080186. DOI: 10.1109/date.2012.6176447. URL: http://ieeexplore.ieee.org/document/6176447/.

[Cri] Criu. *Criu*. URL: https://criu.org/Main_Page.

[Den05] Peter J. Denning. *The locality principle*. July 2005. DOI: 10.1145/1070838.1070856. URL: http://portal.acm.org/citation.cfm?doid=1070838.1070856.

[Der87] Rachid Deriche. "Using Canny's criteria to derive a recursively implemented optimal edge detector". In: *International Journal of Computer Vision* 1.2 (1987), pp. 167–187. ISSN: 09205691. DOI: 10.1007/BF00123164.

[Döm16] Rainer Dömer. "Seven Obstacles in the Way of Standard-Compliant Parallel SystemC Simulation". In: *IEEE Embedded Systems Letters* 8.4 (Dec. 2016), pp. 81–84. ISSN: 19430663. DOI: 10.1109/LES.2016.2617284. URL: http://ieeexplore.ieee.org/document/7589063/.

[Dry15] David Drysdale. "How programs get run : ELF binaries". In: 3 (2015). URL: https://lwn.net/Articles/631631/.

[Fuj90] Richard M. Fujimoto. "Parallel discrete event simulation". In: *Communications of the ACM* 33.10 (Oct. 1990), pp. 30–53. ISSN: 15577317. DOI: 10.1145/84537.84545. URL: http://portal.acm.org/citation.cfm?doid=84537.84545.

[Geo09] Georgef. *How to Save OS Boot Time In Your SystemC Virtual Platform With Save and Restore - System Design and Verification - Cadence Blogs - Cadence Community*. 2009. URL: https://community.cadence.com/cadence_blogs_8/b/sd/posts/how-to-save-os-boot-time-in-your-systemc-virtual-platform-with-save-and-restore (visited on 08/29/2020).

[Hel09] Claude Helmstetter. *TLM.open: a SystemC/TLM Front-end for the CADP Verification Toolbox*. Tech. rep. Oct. 2009. URL: https://hal.archives-ouvertes.fr/hal-00429070.

[HLH+09] Ziyu Hao, Qian Lei, Li Hongliang, Xie Xianghui, and Zhang Kun. "A parallel SystemC environment: ArchSC". In: *Proceedings of the International Conference on Parallel and Distributed Systems - ICPADS*. 2009, pp. 617–623. ISBN: 9780769539003. DOI: 10.1109/ICPADS.2009.28.

[HLR10] Tim Harris, James Larus, and Ravi Rajwar. "Transactional memory, 2nd edition". In: *Synthesis Lectures on Computer Architecture*. Vol. 11. Morgan & Claypool Publishers, July 2010, pp. 1–263. ISBN: 9781608452354. DOI: 10.2200/S00272ED1V01Y201006CAC011.

[HLX+09] Yong Qin Huang, Hong Liang Li, Xiang Hui Xie, Lei Qian, Zi Yu Hao, Feng Guo, and Kun Zhang. "ArchSim: A System-Level Parallel Simulation Platform for the Architecture Design of High Performance Computer". In: *Journal of Computer Science and Technology* 24.5 (Sept. 2009), pp. 901–912. ISSN: 10009000. DOI: 10.1007/s11390-009-9281-9.

[HZC+17] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications". In: (Apr. 2017). arXiv: 1704.04861. URL: http://arxiv.org/abs/1704.04861.

[IEE12] IEEE. *IEEE Standard for Standard SystemC ® Language Reference Manual IEEE Computer Society*. Vol. 2011. January. 2012, p. 638. ISBN: 9780738168012. DOI: 10.1109/IEEESTD.2012.6134619.

[Int19] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*. 2019. URL: https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf.

[JSD+19] Matthias Jung, Frank Schnicke, Markus Damm, Thomas Kuhn, and Norbert Wehn. "Speculative Temporal Decoupling Using fork()". In: *Proceedings of the 2019 Design, Automation and Test in Europe Conference and Exhibition, DATE 2019*. IEEE, Mar. 2019, pp. 1721–1726. ISBN: 9783981926323. DOI: 10.23919/DATE.2019.8714823. URL: https://ieeexplore.ieee.org/document/8714823/.

[Kah74] Gilles Kahn. *The semantics of a simple language for parallel programming*. 1974. DOI: 10.1007/BF00288686.

[KLP+09] Stefan Kraemer, Rainer Leupers, Dietmar Petras, and Thomas Philipp. "A checkpoint/restore framework for systemC-based virtual platforms". In: *2009 International Symposium on System-on-Chip - Proceedings, SoC 2009*. IEEE, Oct. 2009, pp. 161–167. ISBN: 9781424444670. DOI: 10.1109/SOCC.2009.5335656. URL: http://ieeexplore.ieee.org/document/5335656/.

[Kue] Chris Kuehl. *Yelp/dumb-init: A minimal init system for Linux containers*. URL: https://github.com/Yelp/dumb-init (visited on 05/28/2020).

[LSD16] Guantao Liu, Tim Schmidt, and Rainer Dömer. “A segment-aware multi-core scheduler for system C PDES”. In: *2016 IEEE International High Level Design Validation and Test Workshop, HLDVT 2016*. IEEE, Oct. 2016, pp. 100–107. ISBN: 9781509042708. DOI: 10.1109/HLDVT.2016.7748262. URL: http://ieeexplore.ieee.org/document/7748262/.

[Man] Manpages. *pid_namespaces(7) - Linux manual page*. URL: http://man7.org/linux/man-pages/man7/pid_namespaces.7.html (visited on 05/25/2020).

[MBW12] Paul E McKenney, Silas Boyd-Wickizer, and Jonathan Walpole. “RCU usage in the Linux kernel: one decade later”. In: *Tech Report* (2012). URL: https://pdos.csail.mit.edu/6.828/2018/readings/rcu-decade-later.pdf.

[MES+10] Màrius Monton, Jakob Engblom, Christian Schröder, Jordi Carrabina, and Mark Burton. “Checkpoint and restore for systemC models”. In: *Lecture Notes in Electrical Engineering*. 2010. ISBN: 9789048193035. DOI: 10.1007/978-90-481-9304-2_3.

[MMG+10] Aline Mello, Isaac Maia, Alain Greiner, and François Pêcheux. “Parallel simulation of systemC TLM 2.0 compliant MPSoC on SMP workstations”. In: *Proceedings -Design, Automation and Test in Europe, DATE* (Mar. 2010), pp. 606–609. ISSN: 15301591. DOI: 10.1109/date.2010.5457136.

[Moy13] Matthieu Moy. “Parallel programming with SystemC for loosely timed models: A non-intrusive approach”. In: *Proceedings -Design, Automation and Test in Europe, DATE* (2013), pp. 9–14. ISSN: 15301591. DOI: 10.7873/date.2013.017. URL: http://dl.acm.org/citation.cfm?id=2485288.2485294.

[Nan93] Richard E. Nance. “A history of discrete event simulation programming languages”. In: *ACM SIGPLAN Notices* 28.3 (Jan. 1993), pp. 149–175. ISSN: 15581160. DOI: 10.1145/155360.155368. URL: http://portal.acm.org/citation.cfm?doid=155360.155368.

[Rr-] Rr-project. *rr: lightweight recording & deterministic debugging*. URL: https://rr-project.org/ (visited on 06/30/2020).

[SCD18] Tim Schmidt, Zhongqi Cheng, and Rainer Dömer. “Port call path sensitive conflict analysis for instance-Aware parallel SystemC simulation”. In: *Proceedings of the 2018 Design, Automation and Test in Europe Conference and Exhibition, DATE 2018*. Vol. 2018-Janua. IEEE, Mar. 2018, pp. 349–354. ISBN: 9783981926316. DOI: 10.23919/DATE.2018.8342034. URL: http://ieeexplore.ieee.org/document/8342034/.

[ScR] ScRefImpl. *SystemC reference implementation*. URL: https://www.accellera.org/downloads/standards/systemc (visited on 07/03/2019).

[SLD17] Tim Schmidt, Guantao Liu, and Rainer Dömer. “Hybrid analysis of SystemC models for fast and accurate parallel simulation”. In: *Proceedings of the Asia and South Pacific Design Automation Conference, ASP-DAC*. IEEE, Jan. 2017, pp. 226–231. ISBN: 9781509015580. DOI: 10.1109/ASPDAC.2017.7858324. URL: http://ieeexplore.ieee.org/document/7858324/.

[SLP+10] Christoph Schumacher, Rainer Leupers, Dietmar Petras, and Andreas Hoffmann. “parSC: Synchronous parallel SystemC simulation on multi-core host architectures”. In: *Embedded Systems Week 2010 - Proceedings of the 8th IEEE/ACM/IFIP International Conference on Compilers, Architecture and Synthesis for Embedded Systems, CODES+ISSS'2010* (2010), pp. 241–246. DOI: 10.1145/1878961.1879005.

[SWL+14] Christoph Schumacher, Jan Henrik Weinstock, Rainer Leupers, Gerd Ascheid, Laura Tosoratto, Alessandro Lonardo, Dietmar Petras, and Andreas Hoffmann. “LegaSCi: Legacy systemc model integration into parallel simulators”. In: *ACM Transactions on Embedded Computing Systems* 13.5s (Oct. 2014), pp. 1–24. ISSN: 15583465. DOI: 10.1145/2678018. URL: https://dl.acm.org/doi/10.1145/2678018.

[VCB+12] Sara Vinco, Debapriya Chatterjee, Valeria Bertacco, and Franco Fummi. “SAGA: SystemC acceleration on GPU architectures”. In: *Proceedings - Design Automation Conference* (2012), pp. 115–120. ISSN: 0738100X. DOI: 10.1145/2228360.2228382. URL: http://dl.acm.org/citation.cfm?doid=2228360.2228382.

[Ver91] Verilog. *Verilog Reference Guide Verilog Reference Guide Foundation Express with Verilog HDL Description Styles Structural Descriptions Expressions Functional Descriptions Register and Three-State Inference Foundation Express Directives Writing Circuit Description*. Tech. rep. 1991.

[VHD97] VHDL. *VHDL Reference Manual*. Tech. rep. 1997. URL: www.synario.com.

[VPG06] Emmanuel Viaud, François Pêcheux, and Alain Greiner. “An efficient TLM/T modeling and simulation environment based on conservative parallel discrete event principles”. In: *Proceedings -Design, Automation and Test in Europe, DATE*. Vol. 1. 2006. ISBN: 3981080114. DOI: 10.1109/date.2006.244003.

[VPS+14] N. Ventroux, J. Peeters, T. Sassolas, and James C. Hoe. “Highly-parallel special-purpose multicore architecture for SystemC/TLM simulations”. In: *Proceedings - International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation, SAMOS 2014*. 2014, pp. 250–257. ISBN: 9781479937707. DOI: 10.1109/SAMOS.2014.6893218.

[VS16] Nicolas Ventroux and Tanguy Sassolas. “A new parallel SystemC kernel leveraging manycore architectures”. In: *Proceedings of the 2016 Design, Automation and Test in Europe Conference and Exhibition, DATE 2016* (2016), pp. 487–492. DOI: 10.3850/9783981537079_0325. URL: http://ieeexplore.ieee.org/abstract/document/7459359/.

[VSV+16] Janne Virtanen, Panu Sjövall, Marko Viitanen, Timo D. Hämäläinen, and Jarno Vanne. “Distributed systemc simulation on manycore servers”. In: *NORCAS 2016 - 2nd IEEE NORCAS Conference*. IEEE, Nov. 2016, pp. 1–6. ISBN: 9781509010950. DOI: 10.1109/NORCHIP.2016.7792920. URL: http://ieeexplore.ieee.org/document/7792920/.

[WLA+16] Jan Henrik Weinstock, Rainer Leupers, Gerd Ascheid, Dietmar Petras, and Andreas Hoffmann. “SystemC-link: Parallel SystemC simulation using time-decoupled segments”. In: *Proceedings of the 2016 Design, Automation and Test in Europe Conference and Exhibition, DATE 2016*. 2016, pp. 493–498. ISBN: 9783981537062. DOI: 10.3850/9783981537079_0114.

[WLP14] Andrew Waterman, Yunsup Lee, and David Patterson. “The RISC-V Instruction Set Manual”. In: I (2014).

[WML+16] Jan Henrik Weinstock, Luis Gabriel Murillo, Rainer Leupers, and Gerd Ascheid. “Parallel SystemC Simulation for ESL Design”. In: *ACM Transactions on Embedded Computing Systems* 16.1 (Oct. 2016), pp. 1–25. ISSN: 15399087. DOI: 10.1145/2987374. URL: http://dl.acm.org/citation.cfm?doid=3008024.2987374.

[Zak17] Yannick Zakowski. “Verification of a Concurrent Garbage Collector”. PhD thesis. Dec. 2017. URL: https://hal.inria.fr/tel-01680213v2.