HAL
open science

New MP-SoC profiling tools based on data mining techniques

Sofiane Lagraa

To cite this version:

Sofiane Lagraa. New MP-SoC profiling tools based on data mining techniques. Artificial Intelligence [cs.AI]. Université de Grenoble, 2014. English. NNT : 2014GRENM026 . tel-01548913

HAL Id: tel-01548913

https://theses.hal.science/tel-01548913v1

Submitted on 28 Jun 2017

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

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

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministériel 7 août 2006

ISBN: 978-2-11-129190-4

Présentée par

Sofiane LAGRAA

Thèse dirigée par **Frédéric PÉTROT**[1]
et codirigée par **Alexandre TERMIER**[2]

préparée au sein des **Laboratoires** [1]**TIMA et** [2]**LIG**
et de l'École Doctorale **Mathématiques, Sciences et Technologies de l'Information, Informatique (MSTII)**

Nouveaux outils de profilage de MPSoC basés sur des techniques de fouille de données

« New MPSoC profiling tools based on data mining techniques »

Thèse soutenue publiquement le **13 Juin 2014**
devant le jury composé de:

M. Albert Cohen
Directeur de recherche INRIA, École Normale Supérieure, Examinateur
M. Bernard Goossens
Professeur, Université de Perpignan, Rapporteur
M. Pascal Poncelet
Professeur, Université Montpellier 2, Rapporteur
M. Miguel Santana
Directeur du centre IDTEC à STMicroelectronics, STMicroelectronics - Grenoble, Examinateur
Mme. Peggy Cellier
Maître de conférences, INSA Rennes , Examinatrice
M. Frédéric Pétrot
Professeur, Institut Polytechnique de Grenoble, Directeur de thèse
M. Alexandre Termier
Maître de conférences (HDR), Université Joseph Fourier, Co-Directeur de thèse

Abstract

Miniaturization of electronic components has led to the introduction of complex electronic systems which are integrated onto a single chip with multiprocessors, so-called Multi-Processor System-on-Chip (MPSoC). The majority of recent embedded systems are based on massively parallel MPSoC architectures, hence the necessity of developing embedded parallel applications. Embedded parallel application design becomes more challenging: It becomes a parallel programming for non-trivial heterogeneous multiprocessors with diverse communication architectures and design constraints such as hardware cost, power, and timeliness.

A challenge faced by many developers is the profiling of embedded parallel applications so that they can scale over more and more cores. This is especially critical for embedded systems powered by MPSoC, where ever demanding applications have to run smoothly on numerous cores, each with modest power budget. Moreover, application performance does not necessarily improve as more cores are added. Application performance can be limited due to multiple bottlenecks including contention for shared resources such as caches and memory. It becomes time consuming for a developer to pinpoint in the source code the bottlenecks decreasing the performance.

To overcome these issues, in this thesis, we propose a fully three automatic methods which detect the instructions of the code which lead to a lack of performance due to contention and scalability of processors on a chip. The methods are based on data mining techniques exploiting gigabytes of low level execution traces produced by MPSoC platforms. Our profiling approaches allow to quantify and pinpoint, automatically the bottlenecks in source code in order to aid the developers to optimize its embedded parallel application. We performed several experiments on several parallel application benchmarks. Our experiments show the accuracy of the proposed techniques, by quantifying and pinpointing the hotspot in the source code.

Key Words Multi-Processor System-on-Chip (MPSoC), Parallel Embedded Software, Profiling, Data Mining, Execution Traces, Contention, Scalability.

Résumé

La miniaturisation des composants électroniques a conduit à l'introduction de systèmes électroniques complexes multiprocesseurs intégrés sur une seule puce, les *Multi-Processor System-on-Chip* (MPSoC). La majorité des systèmes embarqués à venir est basée sur des architectures avec un grand nombre de processeurs, d'où la nécessité de développer des applications parallèles embarquées. La conception et le développement d'une application parallèle embarquée est de plus en plus difficile, notamment pour les architectures multiprocesseurs hétérogènes ayant différents types de contraintes de communication et de conception, tels que le coût du matériel, la puissance et la rapidité.

Un défi à relever par les développeurs est le profilage des applications parallèles afin qu'elles puissent passer à l'échelle. Cela est particulièrement important pour les systèmes embarqués de type MPSoC, où les applications doivent fonctionner correctement sur de nombreux cœurs. En outre, la performance d'une application ne s'améliore pas forcément lorsque l'application tourne sur un nombre de cœurs encore plus grand. La performance d'une application peut être limitée en raison de multiples goulots d'étranglement, notamment la contention sur des ressources partagées telles que les caches et la mémoire. Il devient difficile et long pour un développeur de faire un profilage de l'application parallèle et d'identifier les goulots d'étranglement dans le code source qui diminuent la performance de l'application.

Pour surmonter ces problèmes, dans cette thèse, nous proposons trois méthodes automatiques qui détectent les instructions du code source conduisant à une diminution de performance due à la contention et à la croissance du nombre de processeurs sur une même puce. Les méthodes sont basées sur des techniques de fouille de données exploitant des gigaoctets de traces d'exécution de bas niveau produites par des simulateurs de plateformes MPSoC. Nos approches de profilage permettent de quantifier et de localiser automatiquement les goulots d'étranglement dans le code source afin d'aider les développeurs à optimiser leurs applications parallèles embarquées. Nous avons effectué plusieurs expériences sur plusieurs applications parallèles embarquées. Elles montrent la précision des techniques proposées, en quantifiant et localisant avec précision les lignes de code dans le code source qui induisent des ralentissements.

Mots Clès Système MultiProcesseur sur Puce (MPSoC), Logiciel Parallèle Embarqué,

Acknowledgments

Il m'est agréable à travers ces quelques lignes d'exprimer toute ma gratitude et mes remerciements envers les personnes qui m'ont aidé et m'ont soutenu tout au long de ma thèse.

Mes sincères remerciements vont à Mr Frédéric Pétrot et Mr Alexandre Termier, qui m'ont proposé ce sujet de thèse à travers lequel j'ai pu apprécier deux domaines de recherche : le data mining et les systèmes embarqués. Je les remercie pour leur confiance et leur soutien tout au long de ce projet.

J'adresse mes remerciements à Mr Albert Cohen qui m'a fait l'honneur de présider le jury de ma thèse. Je souhaite aussi remercier Mr Goossens et Mr Poncelet pour avoir accepté de rapporter mon travail de thèse. Mme Cellier et Mr Sontana pour avoir accepté d'examiner mon travail.

Je remercie mes collègues de l'équipe SLS et l'équipe HADAS respectivement au laboratoire TIMA et LIG pour leurs conseils, leur gentillesse et leur bonne humeur.

En fin, ce travail n'aurait pu avoir lieu sans le précieux soutien de ma famille.

List of Figures

1.1 Predictive evolution of number of cores in a chip and high performance computing (source [ITR07]) . . . 2
1.2 MPPA®-256 block diagram (source [KAL]) . . . 3
1.3 Overview of dissertation . . . 6

2.1 Example of MPSoC platform (Source: Texas Instruments) . . . 8
2.2 Concurrent memory accesses latency versus Time across CPUs . . . 12
2.3 Concurrent memory access by CPUs in a given time window . . . 13
2.4 Speed-up as a function of the number of processors for matrix multiplication, ocean (SPLASH-2) and MJPEG multi-threaded applications. . . . 14
2.5 Overview of the simulation of embedded software on MPSoC architecture 17
2.6 Trace file size according to the number of CPUs in each MPSoC platform 19

3.1 Classification of Embedded Software Profiler . . . 28
3.2 Reasons for large traces . . . 30
3.3 Map of Data Mining domains . . . 32
3.4 A behavior graph dataset and Frequent graphs [LYY+05] . . . 34

4.1 Profiling Process Overview . . . 42
4.2 Two groups or clusters of data points . . . 45

5.1 Example of 4 x 4 mesh NoC . . . 51
5.2 The windowed frequent events trace . . . 54
5.3 Contention Pattern discovery methodology from execution traces in MPSoC . . . 55
5.4 Overview of Windowed frequent events trace computation . . . 56
5.5 Example of frequent patterns . . . 56
5.6 Example of hotspot detection from patterns . . . 57
5.7 Contention Pattern discovery methodology from execution traces in MPSoC . . . 59
5.8 Boxplot . . . 60
5.9 The windowed events trace . . . 61

6.1 Global approach for scalability hotspot in MPSoC platforms 70
6.2 Hot cluster evolution 73

7.1 MJPEG Application described with communicating tasks 80
7.2 Description of the used architecture 81
7.3 Simulated platform (1) 83
7.4 Access rate of the nodes to the pages P_3090 and P_3088 running Mandelbrot application 84
7.5 Results Representation 84
7.6 Period between the successive *stores* of the address `0x1000f914` 86
7.7 New period between the successive *stores* of the address 0x1000f914 . . . 86
7.8 Memory access frequency 88
7.9 Run time of *memcpy, idct, memset* in parallel application 88
7.10 Scalability hotspot in assembly code for the matrix multiplication application. 90
7.11 Visualizing the evolution of hot clusters in each multi-threaded matrix multiplication application according to platform instances 92
7.12 Growth rate evolution over platform instances running five multi-threaded applications 92
7.13 InitA function of LU application 93
7.14 Improvement of InitA function of LU application 93

A.1 An example of disassembly of executable MJPEG code 101

B.1 InitA function of LU application 104

List of Tables

2.1 Raw trace format 18

3.1 Access memory addresses of CPUs 33
3.2 Frequent Access memory addresses of CPUs 34
3.3 A dataset in the context of system trace analysis [CBT+12] 35
3.4 Works on program analysis using on Traces 39

4.1 Example data 46
4.2 Frequent itemsets 46

5.1 Raw trace format for NoC 51
5.2 Frequent Contention patterns 58
5.3 Frequent Patterns 63
5.4 Comparison of contention analysis methodologies in MPSoC 65

6.1 Scalability hotspots 75
6.2 Comparison of Scalability Bottlenecks Detection Methodologies 76

7.1 Characteristics of applications 80
7.2 Summary of the simulation characteristics 82
7.3 Frequent patterns 84
7.4 Frequent patterns 85
7.5 Contention windows 87
7.6 Frequent Patterns 88
7.7 Scalability hotspots 91
7.8 Coverage of clusters in each multi-threaded application across platform instances 95

A.1 New Raw trace format 102

B.1 Frequent Contention Patterns in FFT Application 103
B.2 Frequent Contention Patterns in Mandelbrot Application 104
B.3 Data variable called by floating point functions in platform with 4 CPUs 104

B.4 Data variable called by floating point functions in platform with 8 CPUs 105
B.5 Frequent Contention Patterns in RADIX Application 105
B.6 Frequent Contention Patterns in LU Application 105

Contents

Abstract iii

Résumé v

Acknowledgments vii

List of Figures ix

List of Tables xi

1 Introduction 1

2 Problem Definition 7

2.1 Context . . . 7

2.1.1 MPSoC . . . 7

2.1.2 Terminology . . . 9

2.1.2.1 MPSoC platform . . . 9

2.1.2.2 Multi-threaded programs . . . 9

2.1.2.3 Parallel Embedded Software . . . 9

2.2 Difficulties of Writing Parallel Programs for MPSoC . . . 9

2.3 MPSoC Profiling Problems . . . 10

2.3.1 Contention problems . . . 11

2.3.2 Scalability bottlenecks . . . 13

2.3.3 Profiling tools and Profiling tools based on Simulation . . . 15

2.4 Execution traces . . . 17

2.5 Execution trace analysis . . . 18

2.6 Conclusion . . . 19

3 Background and Related Work 21

3.1 Profiling Tools . . . 21

3.1.1 Software Based Profiling . . . 22

3.1.2 Hardware Based Profiling . . . 24

3.1.3 FPGA Based Profiling 25
3.2 Related Work in Contention and Scalability Bottlenecks Discovery 25
3.2.1 Contention Discovery 25
3.2.2 Scalability Bottlenecks Discovery 26
3.3 Positioning Relative To Existing Profiling Tools 27
3.4 Traces 28
3.4.1 Execution Traces Representation 29
3.4.2 Dealing with the Large Size of Traces 29
3.5 Multi-Threaded Programs Analysis based on Traces 31
3.6 Data Mining 31
3.6.1 Frequent Pattern Mining 32
3.7 Traces Analysis Using Data Mining 35
3.7.1 High level analysis 35
3.7.1.1 Software Analysis 35
3.7.2 Low level analysis 37
3.7.2.1 Hardware Analysis 37
3.7.2.2 Software Analysis 37
3.8 Summary 38
3.9 Conclusion 40

4 New MPSoC Profiling Tools based on Data Mining 41
4.1 Profiling Process Overview 41
4.1.1 MPSoC Simulation 42
4.1.2 Trace Collection 42
4.1.3 Traces Preprocessing 43
4.1.3.1 Low-Level and High-Level Traces 43
4.1.3.2 The Windowed Events Trace 43
4.1.3.3 Feature of Traces 43
4.1.4 Data Mining Tools 43
4.1.4.1 Clustering 44
4.1.4.2 Frequent Itemset/Pattern Mining 45
4.1.5 Knowledge Discovery 46
4.2 Summary 46

5 Contention Pattern Discovery in MPSoC 49
5.1 Introduction 49
5.2 Preliminaries and Problem Formulation 50
5.2.1 NoC 50
5.2.2 Trace Definitions 51

5.2.3 Problem Statement 52
5.2.4 Objective 53
5.3 Contention Pattern Discovery Methodology in MPSoC I 53
5.3.1 Patterns definitions 54
5.3.2 Pattern discovery method 54
5.3.2.1 Windowed frequent events trace computation 55
5.3.2.2 Patterns computation 56
5.3.3 Hotspot detection from patterns 56
5.3.4 Preliminary Results 58
5.4 Approach limitations 58
5.5 Contention Pattern Discovery Methodology in MPSoC II 58
5.5.1 Pattern discovery method 59
5.5.2 Long latencies determinations 59
5.5.3 Slicing the execution traces into contention windows 60
5.5.4 Mining the frequent contention patterns 61
5.5.5 Preliminary Results 63
5.6 Comparison of Methodologies 64
5.7 Conclusion 66

6 Scalability Bottlenecks Discovery in MPSoC 67
6.1 Introduction 68
6.2 Preliminaries and Problem formulation 68
6.2.1 Definitions 68
6.2.2 Problem Statement 69
6.3 Scalability bottlenecks discovery method 70
6.3.1 Overview of the method 70
6.3.2 Trace collection 71
6.3.3 Feature extraction 71
6.3.4 Feature-based clustering 72
6.3.5 Growth rate of hot cluster 72
6.3.6 Frequent scalability bottlenecks mining 74
6.4 Preliminary Results 75
6.5 Comparison of Scalability Bottlenecks Detection Methodologies 75
6.6 Conclusion 76

7 Experimentations and Results 77
7.1 Parallel embedded software 78
7.1.1 Ocean 78
7.1.2 FFT 78

7.1.3 LU 78
7.1.4 RADIX 79
7.1.5 Mandelbrot 79
7.1.6 MJPEG 79
7.1.7 Matrix Multiplication 79
7.2 Simulation environment and Hardware architecture 81
7.2.1 Simulator 81
7.2.2 Operating System 81
7.2.3 Hardware Architecture 81
7.3 Experiments Set I: Contention discovery 82
7.3.1 Experiments I.1 82
7.3.2 Experiment I.2 85
7.3.3 Results analysis 85
7.3.4 Experiments II 86
7.3.5 Results Set II 86
7.3.6 Discussion 89
7.4 Experiments Set II: Scalability bottlenecks discovery 90
7.4.1 Results analysis 90
7.4.2 Discussion 95
7.5 Conclusion 96

8 Conclusions and Future Work **97**
8.1 Conclusions 97
8.2 Future Work 99

A Trace preprocessing **101**

B Contention Patterns **103**

Glossary **107**

List of Publications **109**

References **111**

Chapter 1: Introduction

Un poète doit laisser des traces de son passage, non des preuves. Seules les traces font rêver.
René Char

Today, embedded systems are found in cell phones, digital cameras, portable video games, personal digital assistants, and many other devices/ gadgets. Behind these different devices, it is the advancement of technologies, especially the rapidly progressing semiconductor technology, that makes the development and production of such devices possible, and make these devices smaller, while improving their performances. To further improve performances while reducing costs and energy consumptions, Systems-On-a-Chip (SoC) were introduced. They combine on a single chip general computation units, memory, and I/O components. Recently, the need for more computing power as well as increased graphics powers as leads to SoC with multiple computation cores as well as GPU cores, called Multi-Processor System-On-Chip (MPSoC).

Industrial companies have already developed many MPSoC platforms for multimedia and wireless communication applications. For example, TI's OMAP (Open Multimedia Application Platform) [Ins] product line targets the mobile phone and personal multimedia device market. Other companies have also developed their own MPSoC platforms in the past few years, e.g., the STMicroelectronics's STiH416 [STM], the Samsung Exynos 5 Quad [Sam] and KALRAY's MPPA (Multi-Purpose Processor Array) platform [KAL].

According to the ITRS prediction [ITR07], it will be possible in ten years to integrate more than 500 processors in a single chip (figure 1.1). Indeed, as a real example, the figure 1.2 shows the KALRAY's MPSoC architecture. This chip contains 256 processors organized in an array of 16 clusters connected by a high-speed Network-on-Chip (NoC). Each cluster contains 16 processing cores and 2MB of memory shared among the cores.

Hardware/software interaction is therefore very complex and often not analysable at design time because of the dynamicity of the current applications and architectures. MPSoC include complex memory hierarchies, components and more processors, which in turn makes it difficult to develop / profile and optimize a parallel application on a MPSoC.

Embedded Parallel Software

Writing parallel programs in MPSoC (embedded parallel software) is more difficult than writing parallel programs or sequential programs on a classical computer. In classical

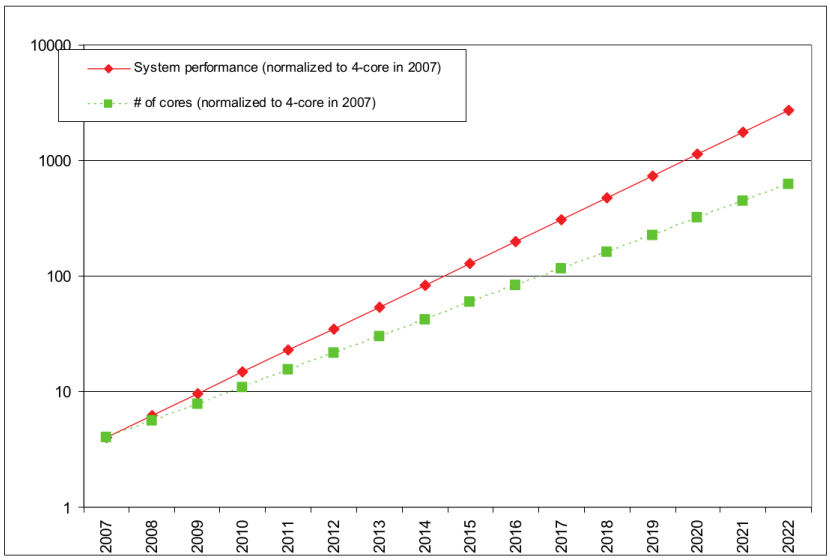


Figure 1.1: Predictive evolution of number of cores in a chip and high performance computing (source [ITR07])

sequential and parallel programming, the programmer must design an algorithm and then express it to the computer in some manner that is correct, clear, and efficient to execute. Additional to the problems of sequential programming, the parallelism adds difficulties in several categories:

- The programmers need to understand the implications of parallelism and how to transform common tasks into parallel programs.
- Parallel problems such as deadlocks, race condition, finding and expressing concurrency, managing data distributions, managing interprocessor communication, balancing the computational load, inefficiency of parallel software in utilizing the hardware resources, scalability of processes.
- Increased difficulty to debug / profile N parallel processes.
- Increased difficulty to verify correctness of program.
- The researchers and practitioners lack of experience with parallel systems.

Parallel programming in MPSoC involves these same issues. Furthermore, for writing efficient parallel programs, the programmer must know a good deal about the MPSoC hardware complexity and characteristics of hardware resources limitation such as limitations of processor-to-memory bandwidth, and small size of cache and memory compared to cache and memory in a classical machine. MPSoCs lack dedicated performance analysis methodologies of parallel applications. The hardware complexity and characteristics of hardware resources are accompanied by dramatic constraints in energy consumption. This is especially critical for embedded systems powered by MPSoC, where ever demanding applications have to run smoothly on numerous cores, each with

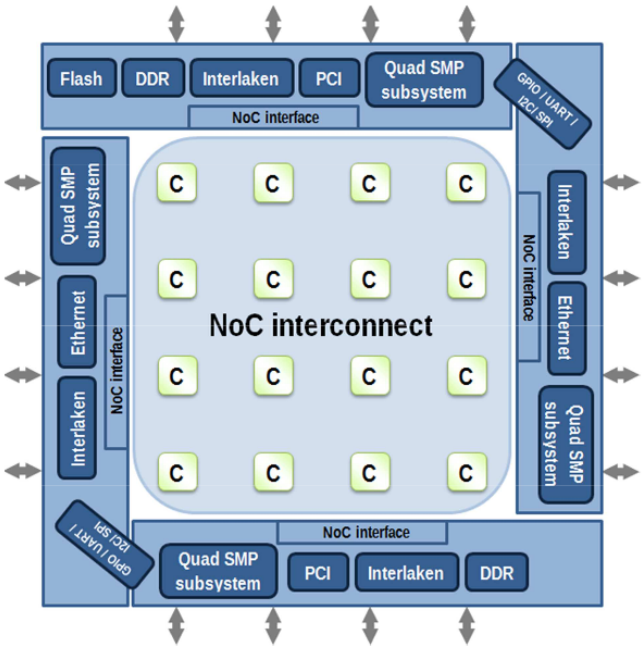


Figure 1.2: MPPA®-256 block diagram (source [KAL])

modest power budget. Other issues can appear such as the cache miss and contention which occur due to specific characteristics of the chip architecture.

Embedded parallel software performance is an essential and fundamental key in modern development of MPSoC. With increased complexity and requirements, applications are expected to perform more computation in less time with lower power consumption. In order to enhance the performance of embedded parallel application on MPSoC, it is important to focus on optimization issues. Optimization is the process of transforming a piece of code to make it more efficient (either in terms of time or space) without changing its output or side-effects. The only difference visible to the code's user should be that it runs faster and/or consumes less memory. Optimizing parallel programs is difficult [MMW96], specifically for embedded parallel program running on MPSoC platforms. In order to optimize code, a tool for locating inefficient resources usage is necessary. Such tool is called a *profiler* or *profiling* tool.

Profiling tools are computer aided software development tools that collect and measure performance information of a software that is running on a target hardware platform. The performance information they can provide are the amount of time needed for each software function to execute in its entirety, the amount of cache read/write misses, and other notable performance metrics. Profiling is deemed of pivotal importance for embedded systems design.

However, profiling techniques in MPSoC domain face the following major challenges:

- the existing profiling tools are not adapted and targeted to embedded system in particular MPSoC software programs on MPSoC due to their intrusivity i.e. the programmer must inject profiling code in source code sections to profile. The intrusivity changes the application behavior.

- the increasing complexity of applications and systems require profiling tools to be fast and quickly available, so that they can be employed as early as possible in the evaluation of embedded software.
- the accuracy of profiling information is imperative to capture the characteristics of today's highly diverse embedded applications and systems.
- the challenge to develop a powerful application suitable on parallel platform.

In order to raise up these challenges, execution traces is a commonly used technique for debugging and performance analysis of a program in a machine or embedded system. Concretely, execution traces implies generation and storage of relevant events during run-time, for later off-line or post-mortem analysis.

Execution Traces Based Profiling

Execution traces have been used in program behavior comprehension to facilitate understanding of interactions between threads of an embedded software system. Further in this thesis, execution traces have been used to profile, analyze program interactions, and discover hotspots / bottlenecks in order to aid developer for program optimization.

However, another problem is the large amount of traces generated from the execution of a parallel embedded software on massively parallel MPSoC platform. The execution traces can vary from tens up to a hundreds gigabytes. Identifying the relevant information in this mass of data is a challenge. The large amount of traces complicates the process of applying existing analysis techniques such as visualization or statistical techniques. However, due to the complex nature of execution traces, most existing works recognise the fact that there is a need for more advanced trace analysis techniques. Therefore, we propose to use data mining techniques.

General Problem and Thesis objectives

Given the following context and problems:

- Evolution of the number of processors on a chip and particular hardware resources limitation,
- Difficulties in writing embedded parallel software taking the characteristics of hardware architecture into account,
- Difficulties of optimization of such programs,
- Difficulty to quantify and pinpoint the hotspots in the source code,
- Difficulty to analyse huge amounts of execution traces.

The objectives of this thesis is to develop and evaluate profiling techniques to aid developers to take decision for optimization of the parallel programs running on MPSoC in a fully automated way.

The profiling tools are based on execution traces and must have the following properties: ① capabilities to analyse and understand the content of complex and large

execution traces; ② an ability to extract meaningful and useful information from execution traces, ③ identifying recurrent behaviors that indicate performance issues while running several tasks of applications concurrently on a MPSoC platform, and ④ an ability to pinpoint the hotspots in the parallel program source code when:

- Multiple accesses to the same resource occurring at the same time cause the decline of performance, this phenomena is called *contention*. The contention is one of the principal cause of low performance that can stem from either hardware resource limitations or from inefficiency of software applications in utilizing the hardware resources.
- Bottlenecks prevent the performance of scaling embedded parallel application linearly, when the number of processor increases. Application scalability refers to the improved performance of running applications on a scaled-up MPSoC platform.

In order to answer these problems:

- we opt for using data mining techniques for analysis of huge amount of execution traces.
- we propose three techniques in order to find the cause of contention and scalability bottlenecks using data mining on execution traces.

Our experimentations demonstrate the efficiency of our approaches by quantifying and pinpointing hotspots in embedded parallel applications.

Thesis Organization

The remainder of the thesis is organized as follows:

- **Chapter 2** presents in details the problems addressed in this thesis.
- **Chapter 3** reviews a large state of art of profiling tools, the works using data mining techniques on execution traces and the works on contention and scalability bottlenecks. Also a positioning of our works is given.

Our three contributions are in frontier of two domains: MPSoC software engineering and data mining. These contributions have in common the use of data mining techniques and algorithms on execution traces for profiling embedded parallel applications:

- **Chapter 4** presents an overview of new profiling tools based on the use of data mining techniques on execution traces.
- **Chapter 5** presents our first two contributions to identify and pinpoint frequent contentions during the concurrent memory accesses and interaction between processors.
- **Chapter 6** presents our third contribution, a parallel scalability bottlenecks discovery method in MPSoC platforms using data mining on execution traces.
- **Chapter 7** presents a set of experiments and results to validate our approaches.
- **Chapter 8** concludes this thesis and presents future directions works of this thesis.

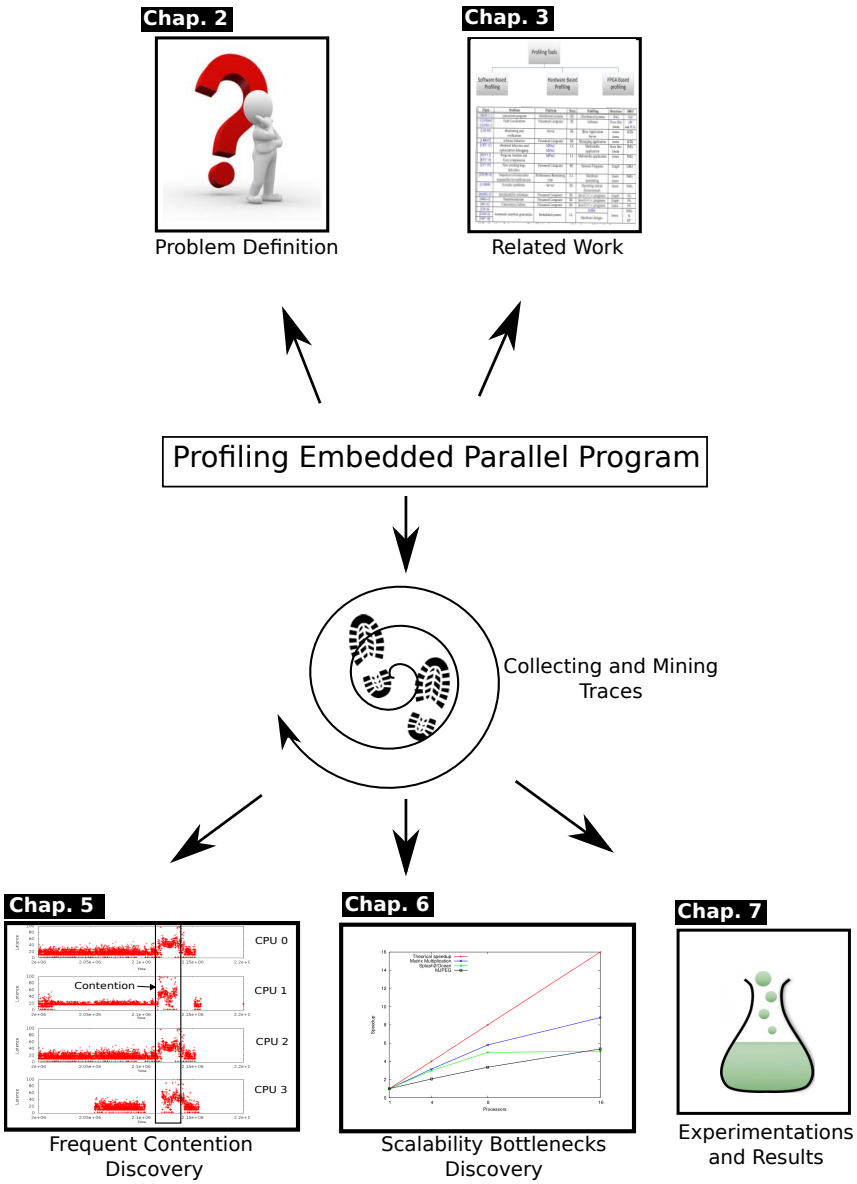


Figure 1.3: Overview of dissertation

Chapter 2: Problem Definition

Contents

2.1 Context 7
2.1.1 MPSoC 7
2.1.2 Terminology 9
2.2 Difficulties of Writing Parallel Programs for MPSoC 9
2.3 MPSoC Profiling Problems 10
2.3.1 Contention problems 11
2.3.2 Scalability bottlenecks 13
2.3.3 Profiling tools and Profiling tools based on Simulation 15
2.4 Execution traces 17
2.5 Execution trace analysis 18
2.6 Conclusion 19

This chapter deals with performance bottlenecks and profiling issues for software running on Multi-Processor System on Chip MPSoC. First, we present the context of our work, followed by a detailed description of the main MPSoC profiling problems. We then describe the difficulties in tracking the parts of source code decreasing the performance of the software running on MPSoC. Finally, we summarize the problems we intend to solve.

2.1 Context

2.1.1 MPSoC

A System-on-Chip (SoC) is an integrated circuit that implements most or all of the functions of a complete electronic system. The system may contain memory, processor, specialized logic, busses, and other digital functions. System-on-Chips are usually targeted for embedded applications and widely used in cell phones, telecommunications, networking, multimedia, and many other applications.

A MPSoC is a SoC that contains multiple processors, Constituting an evolution in computer architecture, that is justified by the requirements of recent embedded systems: real-time, low-power, and more demanding multitasking applications. The MPSoC performance is determined by its hardware and software it runs. The hardware components include the capacity of the node processors (e.g. CPU speed, cache size,

etc.), the interconnect network that connects the processors, the memories, and more. The software is basically the embedded application that is running which is the key contributor to embedded system performance and power consumption.

To give an example of a MPSoC, Fig. 2.1 shows the the upcoming next generation OMAP™4 mobile applications platform [Ins], OMAP44x, which is designed specifically for mobile multimedia telecommunication devices. It mainly consists of two ARM Cortex™- A9 MPCore™RISC processors, a Digital Signal Processor DSP based programmable video accelerator, a PowerVR™2D/3D graphics accelerator, an image signal processing processor, a number of peripherals for video, I/O peripherals, and the interconnection between the functional components.

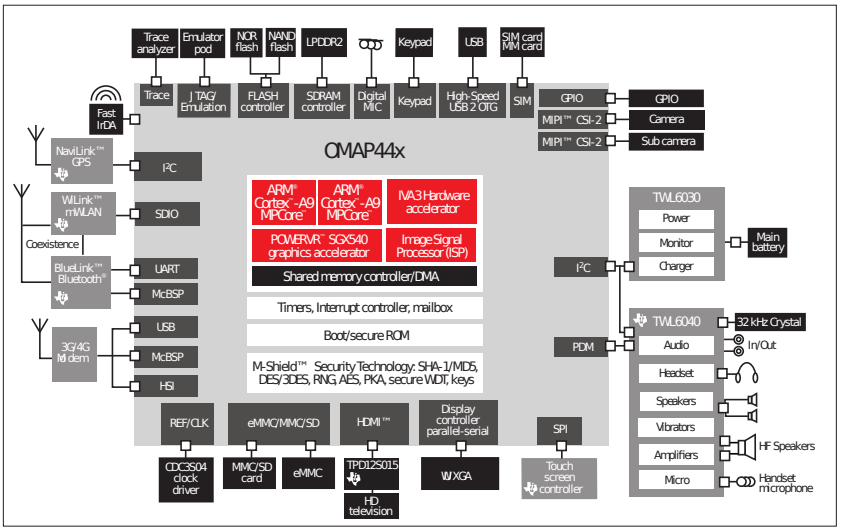


Figure 2.1: Example of MPSoC platform (Source: Texas Instruments)

Multi-core MPSoC architectures allow the efficient execution of parallel programs. This serves well in modern programming languages and operating systems as they are designed to support multi-threaded application development with multiple concurrent tasks. However, writing parallel programs is difficult and the resultant concurrent tasks may not run as efficiently as expected on the MPSoC. Therefore, it is vital to be able to evaluated and study the performance of embedded application on multi-core architectures in order to improve their performance. Performance measurement is based on collecting information about the execution characteristics of a program.

Performance measurement tools, also called *profiling tools*, help the developer to profile its application in terms of execution time, memory usage, cache misses and other important performance metrics. The results of the profiling tools help developers in optimizing their applications.

Currently, the problem of how to do software analysis and efficient profiling is one of the biggest challenges for MPSoC design [Mar06]. This thesis aims at advancing the state of the art towards this goal.

2.1.2 Terminology

2.1.2.1 MPSoC platform

A platform refers to the whole hardware necessary to run the embedded software. A platform can be real (FPGA-based prototype, final SoC) or virtual, allowing to model and simulate the hardware with the software running thereon.

2.1.2.2 Multi-threaded programs

A multi-threaded program contains two or more threads that can run concurrently. Each thread defines a separate path of execution.

2.1.2.3 Parallel Embedded Software

Parallel embedded software runs on MPSoC platform. It is a multi-threaded program operating on multiple CPUs. In this thesis, we use the term embedded software, program or application for sake of brevity.

2.2 Difficulties of Writing Parallel Programs for MPSoC

The development of parallel programs on multi-core architectures is needed for several applications such as video decoding, 3D games,...etc. However, developing parallel programs is not an easy task and is more difficult than sequential programs [MH89, NM92]. They consist of concurrent processes / threads, each executing different tasks with communication and coordination between them. Parallel programs are complex dynamic systems and interactive ones. They include complex interactions among the processes, and between the processes and the platform components. The difficulties of writing parallel programs for MPSoC are summarized as follows:

- **Finding the parallelism**: The first problem to which the developer is confronted to identify the parallel code sections in the application.
- **Debugging and Profiling issues**: For a sequential application, the developer has only one application to debug / profile, but for a parallel application running on multiple cores, the developer faces several threads. Analysing the complex interactions, the concurrent processes, and the relationships between program processes is a challenging task.
- **Characteristics of MPSoC architecture**: Embedded software development is challenging because of the hardware complexity of MPSoC. It requires parallel programming for homogeneous or heterogeneous multiprocessors. It also must take into account diverse communication architectures and design constraints, such as hardware cost, power, and timeliness. Therefore the developer must understand the various complete characteristics of MPSoC hardware.
- **Time to market**: Manufacturers have to integrate new hardware technologies, to develop new software, and to provide new functions in a very short time.

In addition, the developer must cope with the different challenges such as scheduling tasks at the right granularity onto processors, associating data with tasks, resource sharing, threads synchronization, etc.. These issues can severely impact runtime performance and can be very hard to fix [Fos95]. The problem of efficient programming of complex multi-threaded applications for MPSoC is not new, however it remains one of the biggest hurdles in the embedded system community.

In order to help the developer write an optimized parallel program, in this thesis, we profile multi-threaded applications running on MPSoC platforms, in order to quantify application performance and pinpoint the code sections that are amenable to optimization.

2.3 MPSoC Profiling Problems

A profiler is a program analysis tool that collects data on a program in order to track the performance of the running software. It measures, for example, the space (memory) or time complexity of a program, the usage of particular instructions, or frequency and duration of function calls in order to track performance. The most common use of profiling information is to detect bottlenecks and aid program optimization.

A *bottleneck* sometimes known as *hotspot* is a software resource (sections of code) that is a key limiting factor in improving system performance.

There are many reasons that can prevent the scaling of a parallel application and can degrade the performance of applications. The major bottlenecks for multi-threaded workloads on multi-core architectures are [EEKS06, HCC+11, EDBE12]:

- **Resource sharing**: Multi-core architectures typically have many shared resources such as: interconnection network, memory, caches, etc. Resource sharing improves the utilization of a hardware component and can improve overall system performance. However, resource sharing may also have a negative impact on the performance. For example, if several threads need to access the same device, they will compete for its access and which may cause contention, that may propagate to the network or lead to very high latencies.

- **Cache coherency**: Cache coherency ensures that cache is consistent with respect to shared data. Cache coherency introduces extra traffic on the bus or interconnection network, and causes additional misses when local cache lines that are invalidated through upgrades by other cores, are re-referenced later. For example, if threads attempt to update the same cache line (false sharing) or the same data (true sharing), a cache invalidation will be occurred. Cache invalidation is an expensive operation because it causes memory operations to stall and wastes memory bandwidth. Moreover, cache invalidations can cause cache misses later when other cores access the same cache line or the same data again. If cache invalidations and cache misses occur frequently, they create a contention and the performance of the application may suffer severely.

- **Synchronization**: The two most commonly used synchronization primitives are locks and barriers. Locks are typically used to define critical sections to guarantee atomicity when modifying shared data. A barrier, on the other hand, imposes

ordering and denotes a point in the execution beyond which a thread is only allowed to go after all other threads have reached that point. Also, locks and barriers make the application execute sequentially which causes loss of performance for the parallel application.

- **Load imbalance**: Load imbalance means that one or a more threads need (substantially) more time to execute than the other threads. This puts a limit on the achievable speedup, as the execution time of a multi-threaded application is determined by the slowest thread.

- **Parallelization overhead**: Is the amount of time required to coordinate parallel tasks. Parallel overhead includes factors such as: task start-up time, synchronizations, data communications, software overhead imposed by parallel languages, and task termination time.

The growth of these factors results in significant traffic increase in the MPSoC platform. Consequently the execution time of the parallel program is also increased.

Different bottlenecks can limit performance at different times. In particular, contention for different code segments can be very dynamic. In this thesis, we focus on *contention problems*, and *scalability issues* linked to contention.

Two of the major problems of parallel applications in MPSoC are : *contention problems* and *scalability bottlenecks* that result from contention.

2.3.1 Contention problems

The hardware components in an MPSoC, such as memory, input / output and processing elements, are usually referred to as resources. If certain resources are shared among the programs such as memory, there will be a potential resource *contention*. Contentions occur when a memory unit or one or more links of the MPSoC platform are accessed simultaneously by more processors than they were designed to satisfy. This leads to delays in response time, and increases the memory access latency, and reduces the bandwidth from the processor to memory [MTQ07].

Contention depends on bottlenecks, the number of read/write accesses, amount of private and shared data present in an application, and cache capacity. Understanding and discovering when, where, and how contentions occur in shared memory resources that impact application performance is a challenging task.

Figure 2.2 illustrates the notion of high latency contention in a small period of the execution of a parallel Motion-JPEG (MJPEG) video decoding application. It shows an example of concurrent memory accesses by 4 CPUs over time. In each CPU, we see the memory access latency in y-axis is according to the time (cycles) in x-axis. We can observe in the region highlighted by a rectangle that *a*) memory access latencies are much higher than the other regions of the curve, and *b*) the latencies of CPUs are correlated i.e whenever the memory access latency increases in a CPU, it grows across all CPUs. : Thereby, there is contention in this period.

The principal causes of contention are:

- Different synchronization sections and resource sharing.

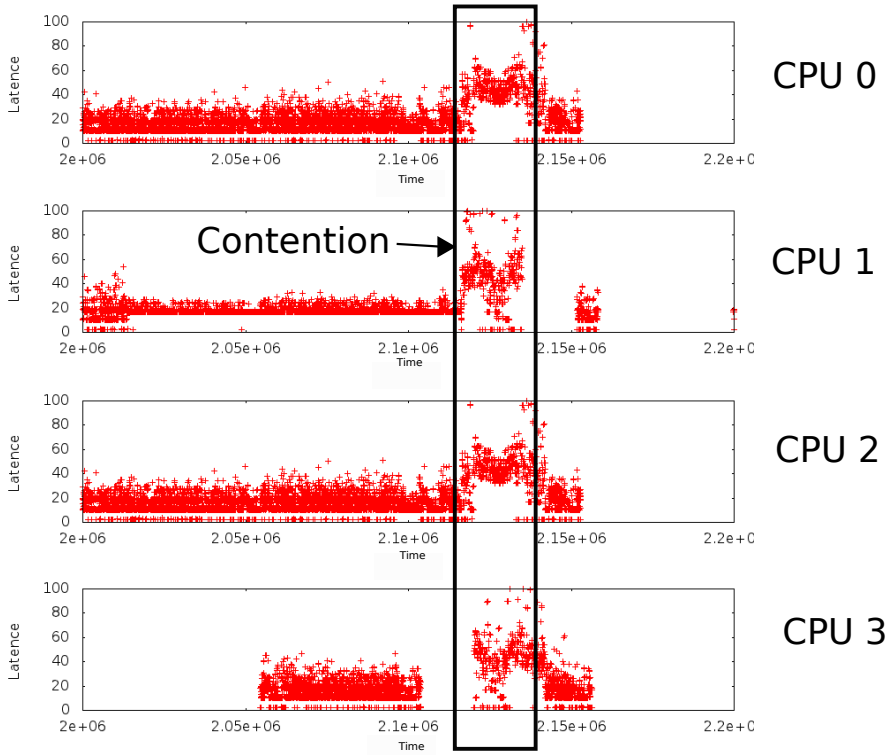


Figure 2.2: Concurrent memory accesses latency versus Time across CPUs

- **Concurrent accesses** to memory segments and/or IPs. The concurrent accesses can be read/write memory accesses. The more frequent concurrent accesses or synchronization, the higher the latency.

The concurrency is only about running different pieces of code at the same time during a time window. **Time window** represents time period in which a set of concurrent accesses occur. The interest of time window is to have a slice of time periods in order to help identify resources that are highly shared. The concurrent accesses is defined as a set of multiple events that occur within a given time period. We are interested in concurrent accesses because in a given time period, the contention may be caused by one or more processors accessing the same resources not exactly at the same time.

If we make a zoom of Figure 2.2 in terms of memory addresses, we have Figure. 2.3. In the given time period Δ, we can find one or more concurrent accesses by the CPUs to the addresses belonging to the same memory page. The different concurrent accesses may occur multiple times by different CPUs in the same memory page and can create contention (highlighted in Figure. 2.3). Furthermore, the presence of multiple concurrent threads or programs are some of the challenges that make MPSoC programming so difficult. Although many efforts have been done, concurrent programming remains hard [LC10, KKJ+08].

The phenomenon of memory contention is well known to practitioners and can occur one or more times during program execution. Thus, frequent contentions decrease drastically the parallel program performance. Also, the performance suffers when too many processors attempt to access the same memory component / page simultaneously.

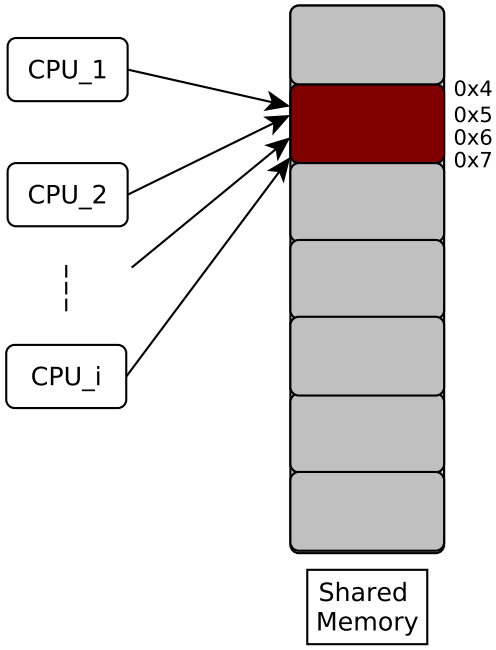


Figure 2.3: Concurrent memory access by CPUs in a given time window

The performance of parallel algorithms is heavily influenced by contention. Nevertheless, even though contention is one of the principal considerations affecting the performance of algorithms on multiprocessors, to the best of our knowledge, there are no tools for analyzing and pinpointing the contention in source code in MPSoC development environment.

Therefore a contention detection tool is needed to help programmers identify whether their multi-threaded programs suffer from contention or not, and pinpoint the contention in source code if it exists.

2.3.2 Scalability bottlenecks

It is expected that the number of cores will increase in the coming years given the continuous transistor density improvements predicted by Moore's law. For example, Intel's Many Integrated Core (MIC) architecture with more than 50 cores on a chip, and the 288 cores of the KALRAY's MPPA (Multi-Purpose Processor Array) platform. A major challenge with increasing core counts is the ability to analyze and optimize performance of parallel programs for multicore architectures. Developers need performance analysis tools and methodologies to identify the performance scaling bottlenecks and understand the behavior of the multi-threaded programs running on MPSoC platforms having 2, 4, 6,...any number of CPUs on a chip.

Intuitively, a parallel program is *scalable* if it runs n times faster on n cores than on 1 core. In this case, it is said that there is a *linear speedup*. Formally, the speedup is described with the equation 2.1.

$$SpeedUp = \frac{T(1)}{T(n)} \tag{2.1}$$

Where $T(n)$ is the time it takes to execute the program when using n processors.

Speedup is often used to understand scaling behavior of an application.

In practice such scaling cannot be obtained by all programs, and the well known Amdahl's law [Amd67] states that the maximum speed up of utilizing n processors in a program is equal to:

$$\frac{1}{(1-P)+\frac{P}{n}} \tag{2.2}$$

where P is the portion of the program that can be made parallel, and thus $1-P$ is the portion of the program that runs sequentially.

Figure 2.4 shows the actual speedup on a given hardware platform according to the number of processors running multi-threaded programs on MPSoC platforms. Each of them has different and growing number of processors $\{1,4,8,16\}$. The speedup achieved is not *linear*. For example, it only achieves a speedup of 2.16 with four cores for the parallel MJPEG application. Thus, the multi-threaded program does not fully utilize the increasing number of processors.

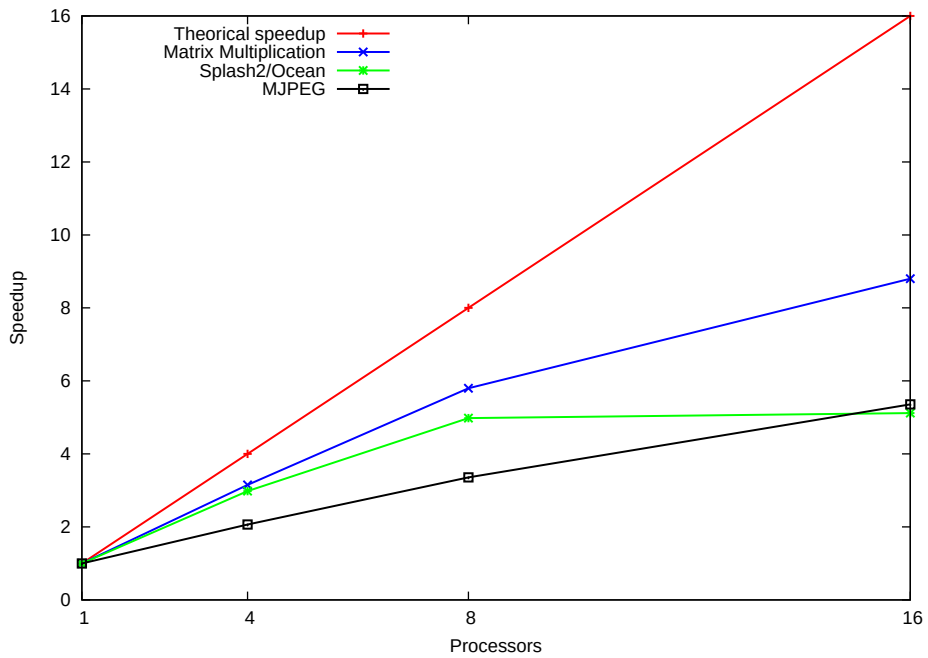


Figure 2.4: Speed-up as a function of the number of processors for matrix multiplication, ocean (SPLASH-2) and MJPEG multi-threaded applications.

Although a speedup curve gives a high-level view on application scaling behavior, it

does not provide the developer information on the inefficient parts of the source code and any insight with respect to :

- Why an application does not scale ?
- What are the critical regions or parts of source code that create the scalability bottlenecks problems ?
- Are the bottlenecks frequent or not in each platform ?

The bottlenecks (described in 2.3) serialize execution, hence wasting valuable execution cycles, and limiting scalability of applications. However, it can be difficult to identify which sections of code are likely to reduce multi-threaded application performances. It is tedious for an application developer to find the correct reason for a lack of scalability without any tools.

2.3.3 Profiling tools and Profiling tools based on Simulation

Using profiling tools, developers can identify sections of code that, if optimized, would yield a better speed-up. Such sections of code are referred to as hotspots or bottlenecks. The benefits of using such tools are to optimize the application by decreasing execution time, increasing efficiency of resource utilization, or a combination of the two. Using the results from the profiling tool, the developers can optimize parts of the program and then run the tool again. This iterative refinement method allows the developer to eliminate parts of the program that dominate the execution time until satisfactory results are obtained.

The following approaches of profiling tools for computers or servers already exist in the literature [MAT09]:

- **Instrumentation based profiling**: Instrumentation consists of injecting extra code into the application's source code before or during compilation. Instrumentation code introduced into a program can change its behavior which in turn, can lead to collecting, i.e the profiled information, which does not represent the original un-instrumented program. Example: The profilers *gprof* [FS08] and *Intel®VTune™* [Rei05] instrument the target program with additional instructions to collect the required information.
- **Hardware Based Profiling** Hardware Counter Based Profiling (HCBP) tools [TK08a] utilize on-chip hardware counters that are available on advanced processors. These hardware counters are dedicated to monitoring specific events that occur during runtime execution of an application. An example of such (HCBP) tool is Performance Advanced Programming Interface (PAPI) [BDG+00].

These categories of profiling tools are not suitable for parallel programs because they change its implementation and behavior during execution. The problems with these categories are :

- They do not provide the results of complex interactions in parallel programs on multiprocessor architecture such as MPSoC platforms.

- The intrusivity (adding instructions in code source to profile it) is not suitable for hardware and software architecture of MPSoC platform because it changes the execution behavior of the application.

In order to tackle these issues, we use simulation methods with non-intrusive trace collection.

- **Simulation**: Today simulation techniques are widely used to help developing and designing SoCs and software that they run. Figure 2.5 shows the general principle of the simulation. The simulation takes into consideration both embedded parallel software and hardware architecture. In the context of this work, the hardware is modeled using SystemC [1]. Simulation is used to obtain the state of the whole simulated system. It allows also to test the software without the real SoC. This last point is crucial because, when designing a SoC, the time-to-market pressure is very hard. The possibility to test software parts before having the chip finished is a key factor for accelerating the release of a SoC. It allows too to verify that the SoC is well suited to execute the required application. Several simulation techniques exists, each one having its advantages and drawbacks. The main parameters of a simulation technique is its speed and precision. A trade-off between this two aspects has to be chosen depending on the needs. Our need in term of precision is at *Cycle-Accurate / Bit-Accurate* CABA level of simulation, at which the system is described in detail with respect to time. The interest of the use of CABA is to get precise performance analysis. Simulations can have varying times to complete depending on the complexity of the software code. It may take several hours to run an entire simulation which may only cover a few seconds of real-time.

When hardware/software simulation is used, we must distinguish the host processor, which is the processor of the machine on which the hardware/software simulation is run, from the target processor, which is the processor embedded into the simulated platform.

Architecturally, the host processor is usually different from the target processor. A target processor may be simulated in a virtual platform, running on the host processor like any software. In this work, we focus only on the target processor in a virtual platform. In the following, we will refer to virtual platform as platform and target processor as processor for sake of brevity.

During the simulation, the non-intrusive trace system generates execution traces of processors. The trace system has a global view to everything that's going on during the platform simulation. A trace system consists in tracing hardware events that are produced by instrumented models of the platform components. The produced trace contains the instructions executed by the different processors of the MPSoC. The related memory accesses are also traced up to the memory by every component relaying them. These memory accesses are used to recover the inter-processors instructions dependencies. Thus, a very low-level trace terms of address of assembler instruction is generated by the cycle accurate simulator. These execution traces are used for software analysis.

[1] SystemC is a language that allows designers to develop both the hardware and software components of their system together. This is all possible to do at a high level of abstraction. Strictly speaking, SystemC is not a language, but rather a library for C++, containing structures for modeling hardware components and their interactions.

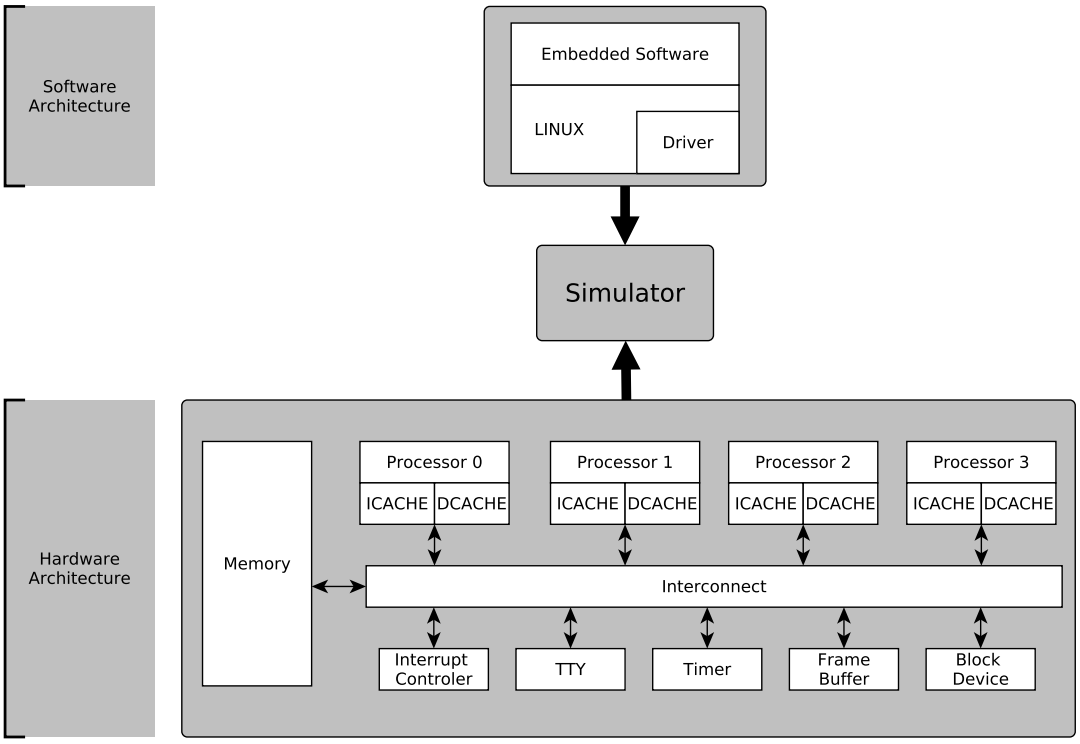


Figure 2.5: Overview of the simulation of embedded software on MPSoC architecture

2.4 Execution traces

Execution traces (or traces) are a description of events that occur during the execution of a software application. They can be generated from simulated or real platforms such as CoreSight trace macrocells tracer [Cor13] for ARM processors or STMicroelectronics's Application Trace Logger (ATL) [STM13] for STiH415 and STiH416 processors [2].

Execution traces are a collection of fine grained information of the execution thanks to the simulation of hardware/software platform. Traces are at a low level of granularity i.e. gathered at the register/signal level during simulation. Table 2.1 shows an example of traces that represent memory accesses (event) performed by CPU. Each event corresponds to a *trace event*. A *trace event* consists of, the global date at which the event occurred in cycles since the power-up of the system, the CPU that initiated the transaction, the program counter of the instruction that produced the access, the instruction type (a fetch, load/store, load-link/store-conditional pairs), and the memory access latency by the CPU.

The interests of (the low level of) traces are:

- Traces are a very powerful tool that can be used to locate the complex interactions between processors and inconspicuous code defects.
- They are necessary for debugging applications and used for monitoring and profi-

[2] STiH415 and STiH416 are dual-core ARM Cortex-A9 CPU, designed for use in Set-top-boxes.

Table 2.1: Raw trace format

CPU ID	Cycle Number	Program Counter	Instruction Type	Data Address	Access Latency
1	212305	0x10009d60	fetch	0x10009d60	28
2	212310	0x10009d60	load	0x10001a40	40
1	212333	0x10009d60	load	0x10001a40	52

ling the performance of applications.

- They provide accurate timestamps of processors events, thanks to the CABA level simulation of parallel program execution.

The problems associated with traces are:

- **Volumetry of traces**: The traces can be very large from tens of megabytes to hundred of gigabytes depending on several functions such as the number of processors in the platform, simulation time, number of event parameters to be traced and the data input size of the application.

  Figure 2.6 shows the trace file size of MJPEG video decoding application against the number of cores in the platform. The traces correspond to the decoding required at 10 color images with a resolution of 255 x 144 pixels. We see that the trace file size grows significantly when the number of cores in a platform grows.

- **Trace analysis**: Traces analysis is the process of applying techniques such as statistical techniques to describe and illustrate, condense and recap, and evaluate traces. Due to the volume of traces, it can be difficult to perform trace analysis.

2.5 Execution trace analysis

Analysing the concurrent processors behaviors and their interactions within a program is a complex task due to the interleaving of events among processors. The large number of events occurring in each processor in a given time period or in all periods, leads to combinatorial explosion during analysis. Thus the analysis from huge amount of trace may take a long time.

Trace analysis is a set of techniques that synthesizes, transforms, processes traces, and/or factual elements to answer questions with the goal of discovering useful information. It is difficult to analyse traces with existing simple tools and visualize the large amount of execution traces in different steps of execution.

Capturing, extracting, and discovering information from execution traces from multiple cores requires significant tools support to automatically extract and present useful information to developer. For developing a trace analysis tool, there is a need to overcome the following hurdles :

- Handling the huge amount of execution traces generated from hardware/software simulation.

- Execution trace format: each tracing tool has its own trace format.

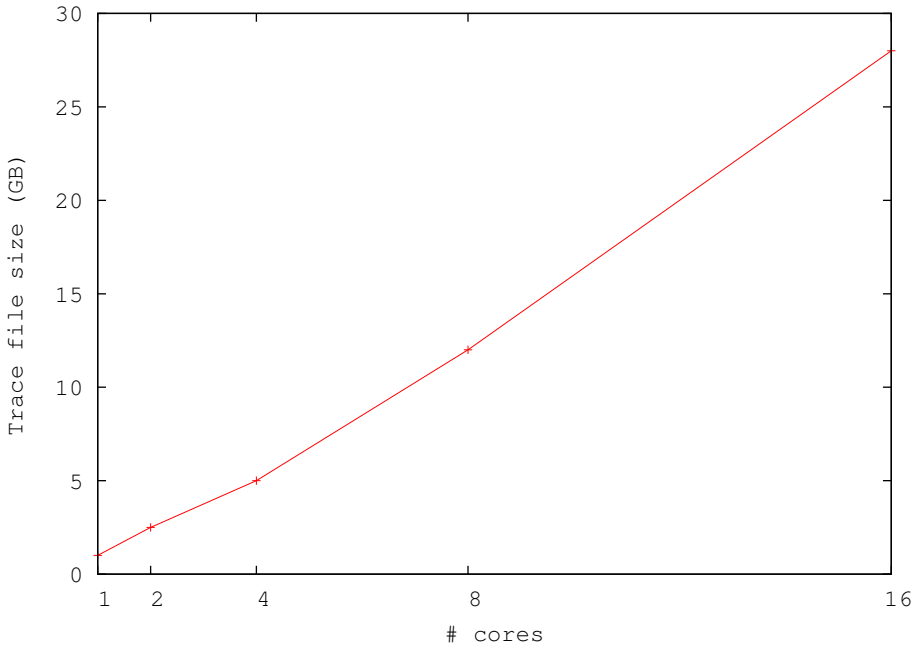


Figure 2.6: Trace file size according to the number of CPUs in each MPSoC platform

- Which techniques can discover and extract automatically meaningful knowledge of the bottlenecks from traces without developer intervention ?
  - How to pinpoint and quantify the contention and scalability bottlenecks from traces ?
  - How to discover and extract automatically recurrent hotspots / bottlenecks in parallel platform ?
  - What is the frequent concurrent accesses leading to contention ? How to discover and extract them ?
  - How to design the profiling tools for contention and scalability bottlenecks ?

2.6 Conclusion

In this chapter, we defined our focus as addressing the delicate problem of memory contention and scalability issues, targeting explicitly MPSoCs and/or multi-core processors. This raises the following problems:

- Finding both hotspots of contention points across multiple cores and presenting explicitly what happens frequently at these points. The frequency of concurrent accesses indicates that it is not a rare or difficult to predict situation, rather a misuse of the resources that come from the application design.

- Discovering automatically the frequent concurrent interactions where contention occurs and the scalability bottlenecks impacting the parallel program.
- Isolating, pinpointing and quantifying the bottlenecks that arise due to contention or scalability of processors in parallel applications running on MPSoC platform.

In MPSoC domain, profiling a multi-threaded application from a huge amount of execution traces is not an easy task given the complexity of the hardware and software architecture. A detailed analysis of the internal task structure of an application is required to determine bottlenecks that should be fixed to improve overall performance. It should help the developer by pointing and suggesting the part of source code to improve. Also, analysing parallel performance and identifying scaling bottlenecks is key in optimizing software and/or hardware design. To the best of our knowledge, if performance evaluation tools exists, no automatic performance analysis tool exists for MPSoC software yet. Such as make performance analysis, detect the most important interactions such as hotspots between platform components, and identify scalability bottlenecks.

In the rest of this manuscript, we describe the existing profiling tools and propose solutions for each problem we raised. These solutions are disjoint but complementary for profiling the embedded application in MPSoC.

Chapter 3: Background and Related Work

Contents

3.1 Profiling Tools 21
3.1.1 Software Based Profiling 22
3.1.2 Hardware Based Profiling 24
3.1.3 FPGA Based Profiling 25
3.2 Related Work in Contention and Scalability Bottlenecks Discovery 25
3.2.1 Contention Discovery 25
3.2.2 Scalability Bottlenecks Discovery 26
3.3 Positioning Relative To Existing Profiling Tools 27
3.4 Traces 28
3.4.1 Execution Traces Representation 29
3.4.2 Dealing with the Large Size of Traces 29
3.5 Multi-Threaded Programs Analysis based on Traces 31
3.6 Data Mining 31
3.6.1 Frequent Pattern Mining 32
3.7 Traces Analysis Using Data Mining 35
3.7.1 High level analysis 35
3.7.2 Low level analysis 37
3.8 Summary 38
3.9 Conclusion 40

In this chapter the related works of profiling tools, techniques for traces analysis programs, and especially trace analysis using data mining techniques are presented. This state of the art is not limited to embedded systems but also encompasses domains that require trace analysis. We also define the positioning of our approach w.r.t existing works.

3.1 Profiling Tools

In software engineering, *profiling* is a commonly used technique for the investigation of software behavior. This helps in identifying the code where the application spends most of its time, i.e. the hotspot functions in order to optimize it. Profile information is

collected during the execution of the program, hence it is a form of dynamic software analysis method.

A lot of research work have been done in computer science in profiling. We identify three main categories for profiling techniques: software based profiling, hardware based profiling, and FPGA based profiling [TK08b, PR12]. (See Figure 3.1)

3.1.1 Software Based Profiling

It is the most common technique for measuring the performance of the application software. Globally, we can distinguish between four different software based profiling methods: insertion of instrumentation code, source-level performance estimation, emulation based profiling, and simulation. Each of these methods have advantages and disadvantages in terms of speed and accuracy of the profiling.

Instrumentation code Insertion: Instrumentation consists of injecting extra code into the application's source code before or during compilation. The extra code is software counters instantiated by the profiling tool on the host machine. It is responsible for recording the execution time and the number of calls for different functions. The recording is performed by sampling the Program Counter PC of the target processor at regular interval during program execution. The best example of such profiler is GNU's *gprof*. Instrumentation code can be done at source level, assembly level or binary level [GKM82], such that profiling information is collected during execution. Unlike *gprof*, PIN dynamic instrumentation provides a simple, and flexible API for transparently inserting new code at runtime into an application. The new code is used to observe the behavior of the program, and can be used to write profilers, memory leak detectors,... etc.

The advantages of instrumentation code insertion is the ease of injecting profiling code into the application's source code and is faster than the existing approaches described in the next sections. The disadvantages are:

- Some compiler optimizations (e.g., function inlining) might be suppressed due to the introduction of profiling code [GHC+09].
- The timing information observed from the environment is also affected by the execution of profiling code.
- The accuracy is poor due to the software overhead introduced by the instrumentation code.
- Code injection can change the behavior of an application when collecting the profiling information.

Moreover, this approach only presents profiles for native execution and can not provide the convenience of profiling an application for a target architecture on a host machine. In order to tackle this problem different profiling approaches for MPSoC were proposed:

Source-level performance estimation (SLPE) techniques address instrumentation code-based profiling issues especially in embedded systems. The approach proposed in [LLSV99, LBH+00] obtains the timing information by decompiling the application binaries to C code. The generated C code (instead of the original source code) can

be compiled natively and executed on the host computers. This approach resembles closely a compile time binary translation that is enhanced with timing information. SLPE [LLSV99, LBH+00, KFK+05, HAG08] uses machine-independent optimizations provided by the host compilers for the sake of accuracy. An improvement of SLPE tools is the **intermediate profiling technologies** or **emulation based profiling**.

Emulation based profiling is an improvement of previous approaches in terms of accuracy by enabling the generation of detailed application information and performance estimates. In these techniques the application source code is first lowered into compiler intermediate representation (IR) and then translated into virtual assembly (VA) by a backend. The resulting VA is then compiled and executed for profiling and performance estimation [HAG08, GHC+09, EWL13].

The SLPE and emulation based profiling techniques target Application Specific Instruction-set Processors (ASIPs). Both SLPE and emulation based profiling techniques, have prohibitively limited accuracy, especially for VLIW architectures [GHC+09].

Instruction Set Simulators (ISS) based profilers: ISS are common for all design types and are widely used tools for studying new architectures and developing software closely related to hardware such as operating systems and embedded system applications. Simulations take place in virtual environments that simulate the behavior of processors as the software code is running in a virtual environment. The advantages of using an ISS for profiling is as follows:

- The designer is able to view the entire data flow inside processors during the simulation.
- The simulation is done on any host machine with the help of ISS model of the target architecture.
- The simulation is more accurate than intrusive approaches.
- ISS based profiling does not require any modification at any level of the software code of the application to be profiled.

The disadvantage of ISS based profilers is low speed: the intrusive approach is considerably faster than the existing software based profiling approaches because the executable is running in the real environment.

A simulator virtualizes the targeted processor hardware, its drawback is: it takes from minutes to hours to run a simulation which only covers a few seconds of real-time.

In embedded systems, the most straightforward and widespread approach is *ISS based profiling*. *ISS based profiling* such as SimpleScalar simulator is used for computer architecture simulation [BA97b]. SimpleScalar measures the performance of several parts of a superscalar processor and its memory hierarchy. It estimates the amount of time (or other measurement) that the simulated processor will need to execute the program. In order to distinguish between the existing ISS-based profiling approaches w.r.t our approach, we call that *profiling performed during simulation based on estimation*. Each estimation approach can be evaluated on the basis of speed, accuracy and abstraction level. Abstraction level defines the hardware design level details either high (Transaction Level Modeling TLM) or low (Cycle Accurate Bit Accurate CABA). Abstraction level is important because during early estimation, detailed models of the

processing elements may not be available. Also, software estimation techniques rely on instruction set abstraction. In addition the speed and accuracy are natural concerns and based on the abstraction level, if TLM is used then the simulation is much faster but also far less accurate than CABA.

Examples of ISS based profiling tools are described in [KKW+06, BKL+00, BFSS01]. Each of these tools are based on different performance estimation techniques.

3.1.2 Hardware Based Profiling

The most straightforward example is hardware performance counters [ABD+97] that widely exist in modern processors. Hardware Counter Based Profiling (HCBP) tools [TK08a] uilize on-chip hardware counters that are available on advanced processors such as *Sun Ultrasparc* [Mic06], *Intel Pentium Processors* [Cor06] and *Advanced Micro Device* (AMD) Processors [AMD02]. These hardware counters are dedicated to monitoring specific events that occur during runtime execution of an application. The types of events which can be monitored are: memory accesses, cache misses, pipeline stalls, types of instructions executed among others. HCBP tools do not require the use of instrumentation code since these counters are designed to collect performance of the software program. Example of HCBP tools include:

- Intel's VTune [Rei05] provides an interface for accessing and utilizing the hardware counters to profile application code executing on Pentium based processors.
- The Performance Advanced Programming Interface (PAPI) [BDG+00] provides users with a high level interface to access the profiling counters and can support many different processors [Spr02].

Using hardware counters for profiling software is beneficial in the following points:

- It does not introduce any instrumentation code.
- They do not add any performance overhead since the data collection of these counters occurs transparently by the hardware during runtime execution of the software.

However, there are some limitations when using HCBP tools.

- Some HCBP tools may require the user to reconfigure and reprogram the counters to detect different events.
- There is a limited number of hardware counters available. The programmer must run the application many times to obtain data for different monitoring events [Spr02].
- The hardware counters cannot be used in early chip design phases, when the prototypes are not available yet [GHC+09]

Furthermore, both software and hardware based profiling approaches can benefit from *sampling profiling technique* [MSR+07, Rei05] that reduces the runtime overhead. Sampling profiling technique [MLG05] generates an interruption at a regular interval or writes a task that samples the content of a program counter or other important registers

of the processor to statistically determine execution behavior later on. Handling of interrupts affects the gathered data since the interrupt service routines (ISR) used add to the number of events.

3.1.3 FPGA Based Profiling

Some embedded systems utilize FPGAs as an implementation platform due to their versatility. Such FPGAs are comprised of hardware customized logic, peripherals along with soft-core processors running on the same chip. The profiling tool is implemented on the FPGA and utilizes the soft-core processors for collecting the profiling information in a nonintrusive manner. Examples of such tools are:

- *Snoop* [SC04b] which is an on-chip function level profiler that was implemented on the Xilinx Virtex-II 2000 FPGA board.
- Frequent Loop Analysis Tool (FLAT) is a tool that detects functions in software that heavily uses loops [GRV05].
- WOoDSTOCK [SC04a] (Watches Over Data STreaming On Computing element linKs), is a profiling tool that monitors the dataflow between computing processor elements.

The advantages of FPGA based profiling tools are

- They provide improved results compared to the profiling tools described earlier.
- They do not use the sampling profiling technique

However, the disadvantages of FPGA based profiling are

- Not all applications can make use of FPGA (not viable for high volume high performance).
- Can be very difficult to program the profiling tool if the knowledge of the user is limited.

A recent alternative to hardware/software based profiling is Trace/Debug interface. It is designed on-chip and can be utilized to monitor the execution of software applications in real time. For example, ARM introduces two components to the CoreSight architecture, the System Trace Macrocell (STM) and Trace Memory Controller (TMC) [Cor13].

3.2 Related Work in Contention and Scalability Bottlenecks Discovery

3.2.1 Contention Discovery

In [TdRC+10, AZXC11, RL10], the authors propose an adaptive routing algorithm for detecting contention patterns. However, their method is limited to detecting only contention in NoC routers. It is based on composite contention metrics and does not consider execution traces.

In [QLD09], the authors investigate a technique to derive communication delay bounds and energy consumption in NoCs. These techniques adopt analytic models, specific methods like machine learning-based regression are also considered in [JKLS10].

In [JKLS10], the authors propose non-parametric statistical regression models such as Multivariate Adaptive Regression Splines MARS [Fri91] to overcome the limitations in ORION2.0 [KLPS09]. ORION [WZPM02] and ORION2.0 [KLPS09] are architectural models that use micro architecture and technology parameters for the router component blocks.

Bottleneck Identification and Scheduling (BIS) technique [JSMP12] tries to identify and solve the most critical bottlenecks at runtime in parallel applications, and migrates dynamically threads that include these bottlenecks to a big core in a heterogeneous multicore. Their method identifies bottlenecks using synchronization primitives (e.g., barriers) provided by parallel languages. The developer annotates the source code with bottleneck identification instructions in order to profile the bottlenecks characteristics such as process identifier, thread waiting cycles, and others. The authors of [JSMP12] showed that different bottlenecks can limit performance at different times. In particular, contention for different critical sections is dynamic.

3.2.2 Scalability Bottlenecks Discovery

Profiling critical jobs in scalable platforms and identifying bottlenecks inside the application is hard [HR07] and few related works raise this issue. Scal-Tool [SLT99] is a tool that isolates and quantifies scalability bottlenecks in parallel applications running on Distributed Shared Memory DSM multiprocessors machines. Scal-Tool is based on an empirical model that uses Cycles Per Instruction (CPI) equations, and uses as inputs the measurements from hardware event counters in the processor. It isolates and quantitatively estimates the cycle count impact of different scalability bottlenecks such as insufficient caching space, load imbalance, and synchronization. From a number of measurement (varying number of processors and varying the size of the dataset on a single processor) the parameters in the CPI equations can be estimated. Globally, to identify the scalability bottlenecks, Scal-Tool is based on estimation of CPI.

A cycle stack (also known as CPI stack) breaks down a single-thread program's execution time into a number of cycle components, each represent the number of cycles the program is stalls due to various miss events, such as cache and TLB misses; and branch miss predictions. Cycle stacks are widely used by software developers and computer architects to gain high-level insight into the behaviors of applications [EEKS06]. In [HCC+11], the authors proposed using cycle stacks to analyze multi-threaded programs and understand performance bottlenecks for multi-core environments as there could be other factors that were not seen in a single core environment. They simulate the analyzed program and capture cycle stacks for each individual thread. In order to analyze the profiled program's scaling behavior, they take similar approach like ours in which, they perform several simulations with increasing number of cores. In their methodology, the authors employ Principal Component Analysis (PCA), which is a statistical data analysis technique that extracts important trends from data. Clustering is performed to obtain dendrograms that represent the (dis)similarity across workloads.

Speedup stacks [EDBE12], is an analysis of the causes of an application lake of perfect scalability. It is a representation that quantifies the impact of individual components

of scalability bottlenecks such as synchronization and interference in shared hardware resources, and attributes in the gap between achieved and ideal speedup to the different possible performance delimiters. Their method is based on hardware performance counter architecture for obtaining speed up stacks. They used the proposed counters to measure a set of performance metrics while the analyzed program is running on a simulated multi-core implementing their permanence counters architecture. Based on these metrics, they estimate the execution rate of the analyzed scalability bottlenecks. However, speedup stacks do not identify the thread that caused the limitation. They also do not suggest how to overcome the scalability limitations they identify. Pinpointing critical threads is important for optimization.

In [CLZ+11], the authors conduct a study of parallel algorithms benchmark on real machines and try to find out implications of program behavior on the real commodity hardware. In order to fully understand the performance characteristics of the multi-threaded application, the authors conduct study experiments to the impact of different factors of multi-threaded applications on multicore server. The factors are the size of the input data set, the number of threads, and parallelism mode (data parallelism or task parallelism). Their study led to the following result: the shared cache and memory bandwidth are indeed the performance bottleneck, that limits the scalability of parallel programs.

In [GKS12], the authors performed a scalability analysis of parallel applications on a 64-threaded Intel Nehalem-EX server. The authors measured how the hardware performed according to performance counters. They used the measurements they acquired to demonstrate that application performance can be limited due to contention for shared resources. While additional threads are active, the contention caused by shared resources increases, and the application may experience stagnation or slowdown in performance.

In [JSMP12], the authors identify and accelerate the critical bottlenecks due to critical sections, barriers, and pipeline stages. The identification of bottlenecks is done in hardware based on information provided by the instrumentation of software. The programmer instruments the instructions in source code for each potential bottleneck.

3.3 Positioning Relative To Existing Profiling Tools

Existing profiling techniques are useful, but do not fullfill our constraints which are:

- Non intrusiveness and availability before fabrication
- Parallelism of embedded applications running on MPSoC
- Behaviors of concurrent processors and their interactions within a parallel program.
- Analysis of hotspots due to contention and scalability bottlenecks.

In order to profile the embedded application without altering its execution behavior, we propose a new profiling tool in the branch of ISS based profiling tools. The new profiling tool is based on post mortem analysis of execution traces. It generated execution traces of software running on MPSoC. Once traces are obtained (without affecting

program execution), trace analysis can be performed. In addition, the profiling aims at identifying the interactions between processors and the causes of bottlenecks that result in contention and scalability problems which decrease the parallel application performance.

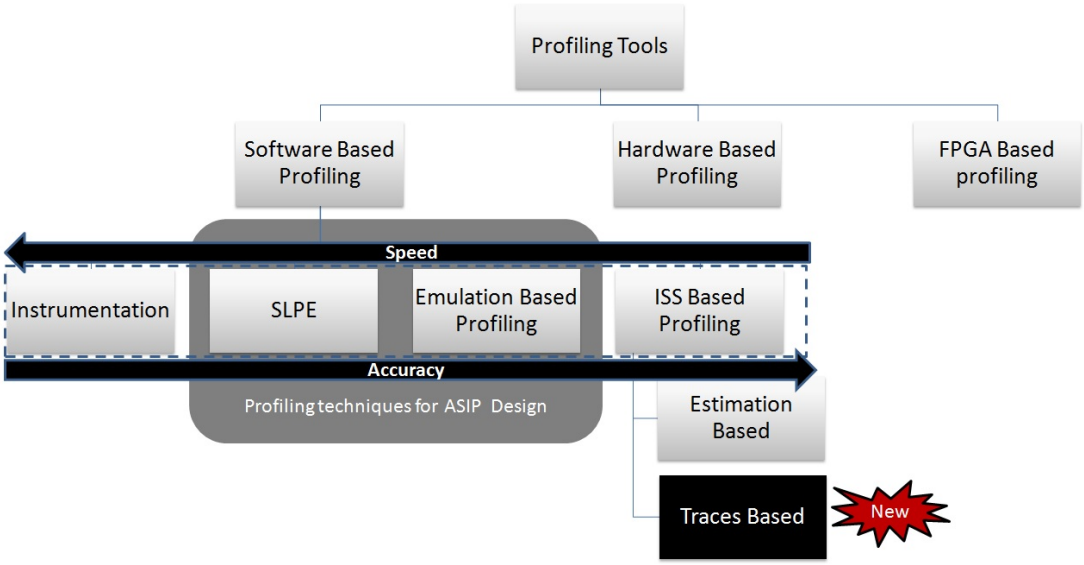


Figure 3.1: Classification of Embedded Software Profiler

The advantages of using traces for software profiling in MPSoC platform are:

- Non-intrusive profiling tool.
- Do not require estimation mechanisms for profiling.
- No sampling based profiling is performed for collecting software metrics or hardware counters.
- Achieve the highest level of accuracy using hardware CABA model.

The disadvantages of using traces for software profiling are:

- Simulation can last longer due to the different parameters such as number of processors to be simulated, simulation time, level of granularity, number of event parameters, and data input size. All these parameters are described in details in the following section 3.4.2.
- A huge amount of traces can be generated from simulation, hence can lead to long post-mortem analysis.

3.4 Traces

In this section, we discuss the characteristics of traces in terms of representation and volumetry.

3.4.1 Execution Traces Representation

Modelling and representing execution traces is an important aspect for analysing and profiling long program executions. It is useful to develop a compact representation for execution traces which capture both sequence events and memory access information. This compact trace should be generated *on-the-fly* during program execution.

Globally in literature, there exists two representations for description of events trace: items tables and graphs. The item can be an event, message, address,...etc. which is saved into a table or a database, and is the most used representation for detecting of systemic problems recurring during execution [LXM08], monitoring and verification of program [LKL08], analyzing the behavior of software system [LMK07], discovering sequences of instruction responsible for inefficiencies [ZXHW10], verify the design using traces [CW10].

In [GMCP13], the authors present the structure of program execution traces in a real machine in order to understand the structure of the program execution and the instruction level parallelism. For their study they use Pin to instrument and generate traces. Pin is a profiling tool that utilizes binary instrumentation techniques for Linux applications. The traces represent the pipeline of the assembler instructions after each cycle.

Another possibility is to represent execution traces into a graph [SMWG11, KKRL11, LYY+05].

3.4.2 Dealing with the Large Size of Traces

The size of a trace file may easily exceed the per-user or per-file disk quota of the operating-system, commonly limited to 2 gigabytes on 32-bit machines. For example, the trace file size for decoding 10 color images using MJPEG application with 16 cores is greater than 25 gigabytes.

The reasons accounting for such size can be divided into five categories :

- **Number of processors/threads**: Whenever the number of processors/threads increases in a platform then the number of raw traces increases too. Also, the amount of communication and synchronization events usually grows with the number of processors/threads.

- **Simulation time**: The simulation time can take few minutes to several hours depending to the number of processors to be simulated in CABA model. It is obvious that restricting the simulation time to smaller interval can substantially decrease the amount of traces.

- **Levels of Granularity**: The granularity means the level of detail captured through tracing. The different levels of abstraction of a software system depend on the trace tool used. We can distinguish between two contents of an execution trace at the following levels of granularity:

  - **High level (HL)**: at this level, the execution traces are described in terms of functions, blocks call levels, messages, diagrams, operations, programs names.

- **Low level (LL)** : at this level, the execution traces consist of fine grained information. They are generated in terms of machine instruction-level or signals and saved one by one instruction at a time. In the trace file, the instructions are described by their addresses and so are the data they access. The tracing tool at instruction-level allows the fine grained analysis which we are interested in.

- **Number of event parameters**: A trace line is composed of set of attributes recorded as part of an event, typically a time stamp. In addition, there may be one or more type-specific parameters. The trace file size can be as a consequence related to the number of attributes.

- **Data input size**: this factor is related to the input size of the algorithm. A typical example is the size of the video for MJPEG application, the size of video can extend both the execution time and the number of events to be traced.

Graphically, Figure 3.2 summarizes causes of large traces.

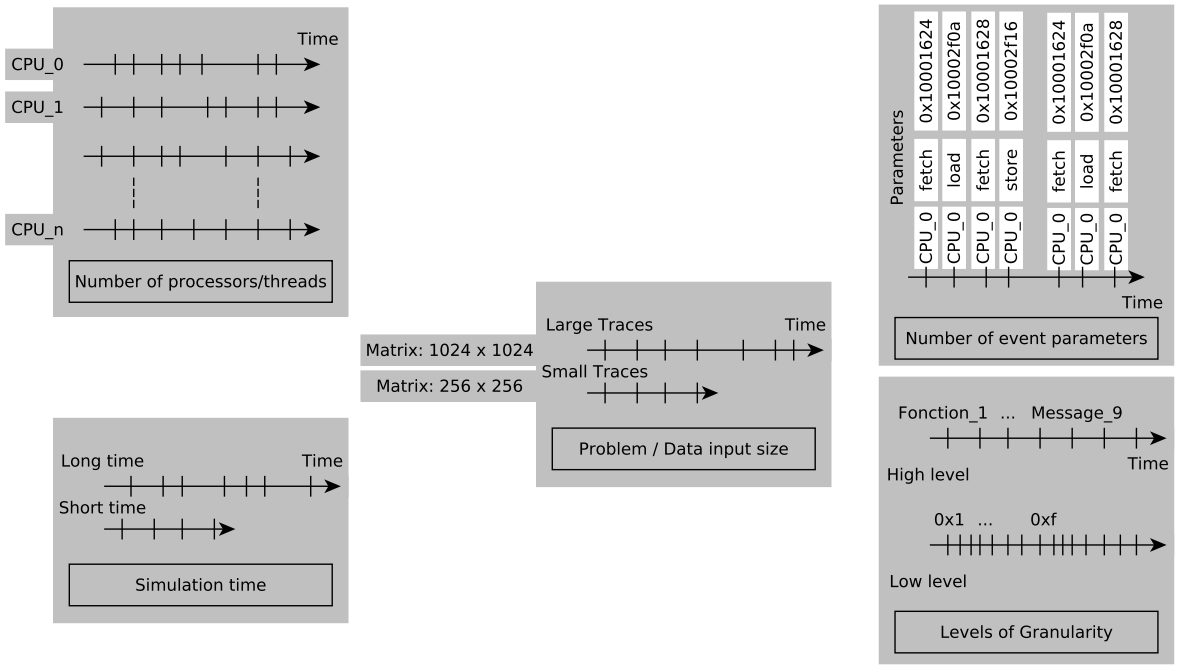


Figure 3.2: Reasons for large traces

Building robust and efficient tools to collect, analyse, and extract knowledge from a large amounts of execution trace is a very challenging task. The trace analysis itself can be very time consuming, especially when the number of processors of the target platform grows and simulation is done on many cycles.

3.5 Multi-Threaded Programs Analysis based on Traces

Predictive analysis is the most used technique for multi-threaded programs analysis based on execution traces. Predictive analysis is a broad term describing a variety of statistical and analytical techniques used to develop models that predict future events or behaviors.

Predictive analysis technique is used for :

- detecting violations of safety properties from apparently successful executions of multi-threaded programs [SRA05]
- detecting concurrency errors during runtime by monitoring a concrete execution trace of a concurrent program [WKGG09, WCGY09, WLGG10].
- detecting serializability violations and nondeterminism in multi-threaded programs [SMWG11, SMG12]

In [SMWG11], [SMG12], the authors addressed the problem of detecting serializability violations in a concurrent program [SMWG11] and nondeterminism in multi-threaded programs [SMG12]. Two threads are considered to be deterministic if when in the same initial state and applying the same sequence of operations, they reach the same final state. In their works, the authors proposed a graph-based predictive analysis method to derive a predictive model from a given traces. This model is based on read-write and synchronization events in the observed trace.

In [WG12], the authors developed a tool for predicting concurrency failures in the generalized execution traces of x86 executables of Linux application. They use PIN [LCM+05] as profiling tool to instrument both the executables and all the dynamically linked libraries upon which the applications depend. The additional code injected during this instrumentation process is used to monitor and control the synchronization operations such as lock/unlock, wait/notify, thread create/join, as well as the shared memory accesses. Then they use a logical constraint based predictive analysis as [WKGG09, WCGY09, WLGG10] to detect runtime failures by generalizing the recorded execution trace. Predictive analysis aims at detecting concurrency errors during runtime by monitoring a concrete execution trace of a concurrent program.

For our problems described in the chapter 2, we need powerful analysis tools and techniques for trace analysis by extracting knowledge of hotspots and bottlenecks from large amount of traces in order to profile embedded software on MPSoC. Such analysis tools are from the **Data Mining** domain.

3.6 Data Mining

Data mining is the exploration and analysis of large quantities of data in order to **discover** valid, **novel**, potentially **useful**, and ultimately understandable **patterns** in data [FPsS96]. There are lots of concurrent and interleaving events among processors leading to an exponential number of event combinations in parallel programs. Data mining is well adapted to tackle this aspect and offers techniques and tools for processing, aggregating, extracting, analyzing, and mining combinatorial processor events in execution traces.

In the last few years, data mining tools and knowledge discovery methods have been used in embedded system area on simulation traces in order to extract automatically assertions [CW10, LSAV11, LLV12, VSP+10], discover recurring runtime execution patterns in the Linux kernel [LXM08], analyse hardware sample data [ZXHW10], improve the performance of software [CH08], verify design specification [LFS10], analyse of multimedia application [KFI+12] and debug embedded multimedia application [CBT+12]. More details are given in the next section.

Data mining techniques are well adapted to traces analysis because they offer the following strong points:

- data mining offers a large variety of tools, techniques and combinatorial algorithms for mining specific data, data pre-processing / post-processing, interestingness metrics and complexity considerations.
- data mining algorithms provide the opportunity to extract useful patterns from enormous amount of data.
- data mining algorithms can provide exact and accurate results of analysis.
- data mining techniques highlight relationships between events within traces of events.

Data Mining contains different tools for classification, clustering, anomaly detection and pattern mining (See Figure 3.3). In this state of the art, we focus on frequent pattern mining algorithms used recently in traces analysis.

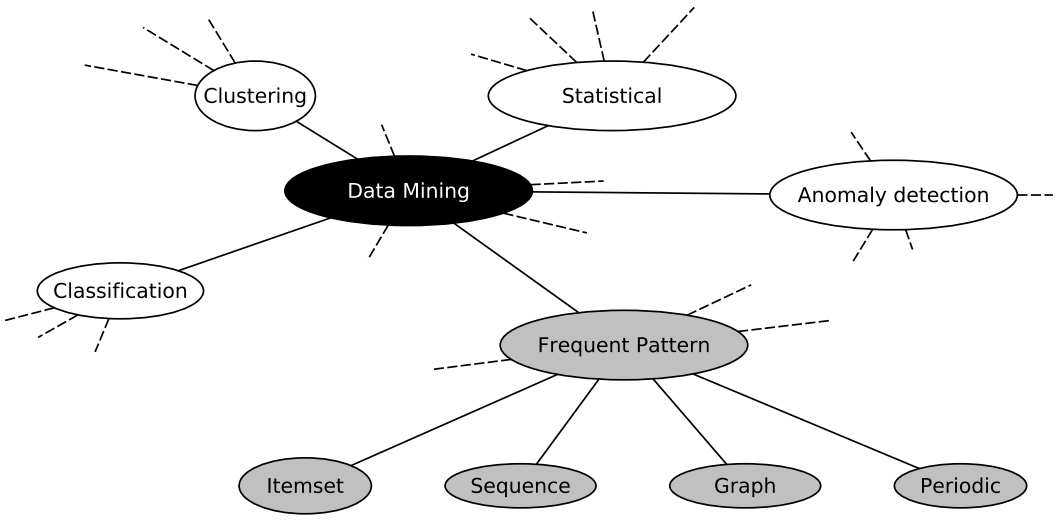


Figure 3.3: Map of Data Mining domains

3.6.1 Frequent Pattern Mining

Frequent pattern mining has been an active field in data mining research for over a decade.

Frequent pattern mining consists of discovering patterns that appear *frequently*, *i.e.* more than a given number of times, in the data. These patterns can be sets of elements, sequences [Zak98], trees [Zak05] or graphs [YH02] depending on the nature of the data and of the analysis to perform.

Each kind of pattern has a specific mining algorithm, for example if the data consist of items, subsequences, substructures, then we have to apply an appropriate algorithm for mining itemsets, sequences, graphs, respectively:

Given a set of items $I = \{I_1, ..., I_n\}$. The data is represented as a list of *transactions* $T = \{T_1, ..., T_n\}$, where each transaction T_i consists of a unique identifier $i \in [1, n]$ and is a set of elements of I: $T_i \subseteq I$. An itemset is also a subset of I.

- **Itemset mining algorithms** mine the database of items looking for repetitive patterns having a frequency greater than a given frequency threshold or called *minimum support* (min_sup). The patterns are known as frequent itemset. More detail is given in next chapter.

  Example 3.6.1. *Given access memory addresses as itemset performed by CPUs in Table 3.1 and frequency threshold = 2.*

CPU_ID	Memory Addresses
1	{0x0100000a0 0x000000a4 0x000000a8 0x000000ac}
2	{0x0100000a0 0x000000a8 0x000000ac 0x100000b0}
3	{0x0100000a0 0x000000a8 0x000000ac 0x100000b0}

Table 3.1: Access memory addresses of CPUs

  Then, the frequent pattern discovered by itemset mining algorithm are in Table 3.2. The frequent patterns or the frequent memory addresses having the frequency = 3 means that 3 CPUs access to such patterns, otherwise, by 2 CPUs.

- **Sequence mining algorithms** mine the sequence database looking for frequent sequences patterns lending themselves to discovering frequent itemsets and the order in which they appear.

  Example 3.6.2. *Given the frequency threshold = 2 and the table 3.1. The frequent sequence patterns are:*

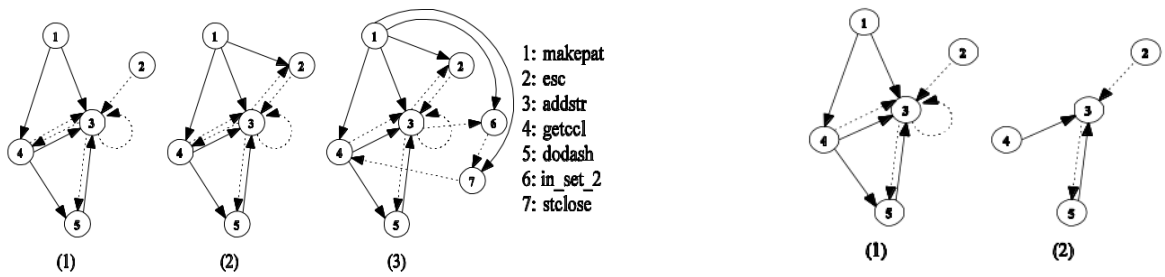
  - {0x0100000a0 0x000000a8 0x000000ac 0x100000b0} *with the frequency equal to 2. It means 2/3 of CPUs perform the same access order to addresses.*
  - {0x000000a8 0x000000ac} *with the frequency equal to 3. It means 3/3 of CPUs perform the same access order to addresses* {0x000000a8 0x000000ac}.
  - {0x000000ac 0x100000b0} *with the frequency equal to 2.*

- **Graph mining algorithms** allow the identification of frequent subgraphs within graph data sets.

  Example 3.6.3. *Figure 3.4a shows behavior graph segments derived from three different runs of a program. Graph mining algorithm discovers the two frequent subgraphs shown in Figure 3.4b.*

Frequent Pattern	Frequency
{0x0100000a0}	3
{0x000000a8}	3
{0x000000ac}	3
{0x0100000a0 0x000000a8}	3
{0x0100000a0 0x000000ac}	3
{0x0100000a8 0x000000ac}	3
{0x0100000a0 0x000000a8 0x000000ac}	3
{0x0100000b0}	2
{0x0100000a0 0x0100000b0}	2
{0x0100000a0 0x000000a8 0x0100000b0}	2
{0x0100000a0 0x000000a8 0x000000ac 0x0100000b0}	2
{0x0100000a0 0x000000ac 0x0100000b0}	2
{0x000000a8 0x0100000b0}	2
{0x000000a8 0x000000ac 0x0100000b0}	2
{0x000000ac 0x0100000b0}	2

Table 3.2: Frequent Access memory addresses of CPUs



(a) A behavior graph dataset

(b) Frequent graphs

Figure 3.4: A behavior graph dataset and Frequent graphs [LYY+05]

- **Periodic mining algorithms** targets frequent temporal regularity by mining time series and focus on characterizing cyclic behaviors in dataset, i.e a periodic pattern can be defined as the repeating activities at certain locations with regular time intervals.

  Example 3.6.4. *Given Table 3.3 presents a program behavior dataset of execution intervals. Each interval is represented by a transaction t_k in a table. Periodic mining algorithm discovers the following periodic patterns: {getFrame}, {displayFrame}, {getFrame displayFrame} at a period $p = 2$ on transactions from t_1 to t_8, therefore they form a cycle of length $l = 2$,*

The pattern mining algorithms are used for knowledge discovery and enhancing the application area of a dataset. The goal of such algorithms is to find inherent regularities in data. For example, discovering frequent sequences of instructions responsible for inefficiencies [ZXHW10] or discovering frequent process that creates operating system issues during execution [LXM08].

t_k	**Itemset**
t_1	getFrame displayFrame
t_2	int16 swint16
t_3	getFrame displayFrame
t_4	write16 cpu_clock
t_5	read16
t_6	getFrame displayFrame printk
t_7	sum_up sem_down
t_8	getFrame displayFrame

Table 3.3: A dataset in the context of system trace analysis [CBT+12]

3.7 Traces Analysis Using Data Mining

We classify the related works of traces analysis using data mining techniques into two groups of types of analysis: high level analysis and low level analysis.

3.7.1 High level analysis

3.7.1.1 Software Analysis

Software analysis using data mining techniques is used for the following problems:

Bug detection: In [LYY+05], the authors treat program executions as software behavior graphs and develop a method to integrate graph mining algorithm and classification for detecting suspicious regions of non-crashing bugs such as logical errors. This study discover structural patterns in call graphs, which are characteristic for failing executions.

Fault localization: In [CDFR08, CDFR11], the authors focus on fault localization on execution traces. When the result of a program execution is not the same as the expected one, that execution is called a *failure*. Fault localization is a software engineering task that tries to find an explanation to the failures by examining information from the executions. To each execution is associated an execution trace that contains information about the execution: the executed lines and the verdict of the execution (*Pass* when the result of the execution is the same as the expected one, otherwise *Fail*). The authors proposed a combination of association rules and Formal Concept Analysis (FCA) to assist in fault localization. **Association rules** is a data mining technique for discovering interesting relations between items in large databases. The pattern is often presented as a collection of if-then rules, called *association rules*. The form of an association rule is $I \rightarrow J$, where I is a set of items and J is an item. The implication of this association rule is that if all of the items in I appear in some transaction, then J is "likely" to appear in that transaction as well. **Formal concept analysis** is formulated based on the notion of a formal context, which is a binary relation between a set of objects and a set of properties or attributes [GW97, Wil09].

A data structure is proposed to organize program elements in a multi-dimensional space called the failure lattice. The failure lattice is a partial ordering of the elements of the traces that are most likely to lead to failures. Lattice structure gives a reasoning basis to find similar execution traces. Actually, all differences between the sets of executed lines of passed and failed executions are represented in the trace lattice [CDFR08,

CDFR11].

In [CFDC11], the author propose an application of Logical Concept Analysis [1] [FR04] to build a generic framework to explore fault localization patterns extracted in [CDFR08]. The kinds of patterns taken into consideration are association rules and sequential patterns from execution traces. The framework is based on a data structure which organizes the set of patterns in a partial order over in the set of patterns. The partial order of pattern organisation allows to compact and structure the patterns and navigate through them.

Information extraction from logs: In [SBL+09a, SBL+09b], the authors extract information from log files. The log files are generated by digital systems such as integrated circuit design tools. They contain essential information on the conditions of production and the final products in order to evaluate the quality of the design. The level of granularity of log files is high and describes the events by a nature language such as English. Therefore, for their analysis, they use Natural Language Processing (NLP) and Information Extraction (IE) techniques. The size of log files is not much and is about hundred of kilobytes.

In [SBL+09a], the authors propose an extension of their approach EXTERLOG (EXtraction of TERminology from LOGs), presented in [SBL+09b], that is developed to extract the terminology from these log files. They study how to adapt the existing terminology extraction methods to the particular and heterogeneous features of log files generated from different tools. They also present a filtering method of extracted terms based on a ranking score in order to emphasize the precision of extracted relevant terms.

Concurrent program behavior analysis: In [KKRL11], the authors present MSGMiner, a framework for building message sequence graphs from execution traces in the concurrent domain using dependency graphs. Their goal is to understand concurrent program behavior in order to depict important "phases" or "interaction snippets" involving several concurrently executing processe. Their work supports concurrent and distributed systems. However, MSGMiner requires each function to be clearly identified in advance and the sequence of function messages to be partially ordered. MSGMiner converts each trace in the trace set into a dependency graph whose vertices contain the events in the trace. The dependency graph captures the partial order of events across processes. The chronological order of events within each process is maintained by a minimal set of directed edges. There is also a set of edges from send events to their respective receive events. The edges in this dependency graph are similar to the "happened before" relationship defined by Lamport [Lam78]. The MSGMiner framework breaks down the dependency graphs obtained from the set of traces into sequences of smaller dependency graphs. From these sequences it mines for a message sequence graph using a variant of the sk-string algorithm [RPN97].

Monitoring and verification: In [LKL08] presented approaches to mine preceding events that often occur before certain series of events through rules stating that "whenever a series of events occurs, previously another series of events must have happened". Specifications of this format are commonly found in practice and useful for program testing and verification.

Software behavior: In [LMK07], the authors mine Live Sequence Chart (LSC) from program execution traces. LSC can be viewed as a formal form of a sequence diagram

[1] **Logical Concept Analysis** is Formal Concept Analysis where logical formulas replace sets of attribute [FR01]

of UML2 described in [DH01]. In [LMK07], temporal rules are mined from LSC with the same principal of rules as [LKL08] stating that "whenever a pre-chart is satisfied, eventually another main-chart is satisfied".

3.7.2 Low level analysis

3.7.2.1 Hardware Analysis

Hardware analysis consists in analysing hardware components or architectures. For hardware analysis, data mining techniques have been proposed for the following problems:

Automatic assertion generation: In [CW10], the authors propose an approach that can automatically extract relationship among several signals from simulation traces. Functional assertions describe the functional relationship through rules among several signals from traces in an abstract level that is different from design implementation, and such information can provide useful hints to synthesis tools because it may contain boolean relations that do not exist in the implementation itself [CCCK11, LSAV11, CW10]. Typically, functional assertions are written by designers or verification engineers. Assertions are used for validating hardware designs at different stages through its life-cycle like pre-silicon formal verification, dynamic validation, runtime monitoring and emulation [BCZ07, BM05], as well as post-silicon debug and in-field diagnosis [BCZ07, BZ08]. Alternatively, tools that try to generate assertions using data mining algorithms have been developed recently [LSAV11, VSP+10, CW10].

In [VSP+10], the authors present *GoldMine* a methodology for generating assertions automatically. Their method involves a combination of data mining and static analysis of the Register Transfer Level (RTL) design. GoldMine mines the simulation traces of a behavioral Register Transfer Level (RTL) design using a decision tree [2] based learning algorithm to produce candidate assertions. These candidate assertions are passed to a static analysis using formal verification engine.

3.7.2.2 Software Analysis

Software analysis using data mining techniques is used for the following problems:

Systemic problems: In [LXM08], the authors developed a framework for mining kernel trace data aiming at the detection of recurring runtime execution patterns and isolating processes responsible for systemic problems, such as inter-process communication patterns. The work finds the set of all temporally proximal events that occurred frequently in a trace. This helped identify the processes that are heavy consumers of system resources but still remain invisible to traditional tools such as `top`.

Monitoring sequences of instruction: In [ZXHW10], the authors developed an efficient sequence mining algorithm for hardware sample data (also called hardware profile data) that can discover short sequences of instruction and their frequency responsible

[2] Decision Tree (or classification tree) is represented as a tree-like of decisions and their possible consequences. A decision tree is composed of internal nodes and terminal leaves. Each decision node implements a splitting function with discrete outcomes labeling the branches. This hierarchical decision process that divides the input data space into local regions continues recursively until it reaches a leaf [BA97a].

for inefficiencies. Their algorithm is less robust to noise in the trace, coming for example from different schedulings.

Another kind of data mining algorithm which is periodic pattern mining algorithm is used on execution traces in [CBT+12].

Abnormal behaviors and optimization debugging: In [CBT+12], the authors propose a pattern mining approach for automatically discovering all periodic behavior occurring in a multimedia application execution traces in order to identify abnormal behaviors and optimize the debugging phase.

Program analysis and Trace compression: In [KFI+12, KFT+13], the authors use a sequential pattern mining algorithm in order to reduce the size of execution traces by automatically discovering a set of blocks that maximally covers the traces. The blocks simplify the exploration of large traces by allowing programmers to see an abstraction instead of low level events.

3.8 Summary

Table 3.4 summarizes the existing non exhaustive list of trace mining techniques discussed in the previous section. It also shows which tool is prescribed in the literature against each problematics, techniques, level of trace granularity and trace type. This way we can assess which part of the trace mining techniques is covered by existing tools and which part is still left open. For example, no existing proposal focuses on high level analysis of architectural hardware using data mining.

From the table, we observe that data mining algorithms have been recently used for different problems including sequential and parallel algorithms. We can also infer that there are few works on the trace analysis of embedded applications on MPSoC. The only two works [CBT+12, KFT+13], we could find are carried out in parallel with this thesis within LIG laboratory in a collaboration with STMicroelectronics.

The originality of our approach compared to the related works is as follow:

- It explicitly targets MPSoCs and/or multi-core processors and addresses the delicate problem of memory contention and scalability bottlenecks discovery.
- It aims at discovering, quantifying and pinpointing automatically the bottlenecks from trace with a low level of granularity.
- It provides exact and accuracy results for answering contention and scalability problem in MPSoC.

Table 3.4: Works on program analysis using on Traces

Paper	Problem	Platform	Trace	Profiling	Structure	DMT
[KKRL11]	concurrent program	Distributed systems	HL	Distributed systems	DAG	SAS
[CDFR08] [CDFR11]	Fault Localization	Personnel Computer	HL	Software	Trace line (item)	AR and FCA
[LKL08]	Monitoring and verification	Server	HL	JBoss Application Server	items items	MTR
[LMK07]	software behavior	Personnel Computer	HL	Messaging application	items	MTR
[CBT+12]	abnormal behaviors and optimization debugging	MPSoC MPSoC	LL	Multimedia application	Trace line (item)	PMA
[KFI+12] [KFT+13]	Program Analysis and Trace compression	MPSoC	LL	Multimedia application	items	SMA
[LYY+05]	Non-crashing bugs detection	Personnel Computer	HL	Siemens Programs	Graph	GMA
[ZXHW10]	Sequences of instruction responsible for inefficiencies	Performance Monitoring Unit	LL	Hardware monitoring	items items	SMA
[LXM08]	Systemic problems	Server	HL	Operating system (Linux kernel)	items	IMA
[SMWG11]	Serializability violations	Personnel Computer	HL	Java/C/C++ programs	Graph	PA
[SMG12]	Nondeterminism	Personnel Computer	HL	Java/C/C++ programs	Graph	PA
[WG12]	Concurrency failure	Personnel Computer	HL	Java/C/C++ programs	items	PA
[CW10]	Automatic assertion generation	Embedded system	LL	AMBA	items	SMA & DT
[LSAV11]				Hardware designs		
[VSP+10]						

DMT: Data Mining Tool
SAS: String algorithm for Sequence
AR: Association rules
MTR: Mining Temporal Rules
PA: Predictive Analysis
HL: High Level
LL: Low Level
IMA: Itemset Mining Algorithm
FCA: Formal Concept Analysis
SMA: Sequence Mining Algorithm
PMA: Periodic Mining Algorithm
GMA: Graph Mining Algorithm
DT: Decision tree

3.9 Conclusion

Profiling is a common software analysis technique which is widely used today. It has been the subject of lots of papers, and several tools are available today on any computers.

In this chapter, we describe and give a classification of profiling approaches, trace analysis of programs using data mining and a positioning of our work. Even though existing profiling tools are very useful, they cannot cope well with parallelism easily, so the use of traces has been proposed to perform more complex analysis.

In the MPSoC domain, the use of data mining to analyze execution traces is relatively recent, and we believe data mining algorithms are powerful tools for mining, discovering and extracting knowledge, system behavior or relations between events from a huge amount of traces.

As can be seen from our literature analysis, no work has been done to profile parallel programs running on MPSoC in a fully automated way.

Chapter 4: New MPSoC Profiling Tools based on Data Mining

Investigation, c'est une espèce de quête où l'esprit suit à la piste les traces d'une cause ou d'un effet, présent ou passé

Denis DIDEROT

Contents

4.1 Profiling Process Overview **41**
4.1.1 MPSoC Simulation 42
4.1.2 Trace Collection 42
4.1.3 Traces Preprocessing 43
4.1.4 Data Mining Tools 43
4.1.5 Knowledge Discovery 46
4.2 Summary **46**

This chapter presents a global overview of new MPSoC profiling tools based on using data mining techniques on simulation traces. The first profiling tool identifies frequent contentions during the concurrent memory accesses and pinpoints the hots-pots in source code. The second profiling tool discovers and quantifies the scalability bottlenecks over MPSoC platforms and localizes the bottlenecks in source code. These profiling tools have in common the knowledge discovery process using data mining algorithms and techniques on execution traces.

4.1 Profiling Process Overview

Our profiling tools for contention and scalability bottlenecks discovery follow a process described in Figure 4.1. The process contains the following steps: ① a non-intrusive trace collection from a MPSoC simulator. ② Trace preprocessing in order to have a high level of traces in terms of functions in addition to low level traces, and therefore facilitates the mining, comprehension and analysis. ③ Contention and scalability bottlenecks discovery using data mining algorithms such as frequent itemsets mining algorithm, clustering and statistical tool. Data mining algorithms extract knowledge to show the

user in order to improve software or hardware properties. In the following, all these steps are presented in details.

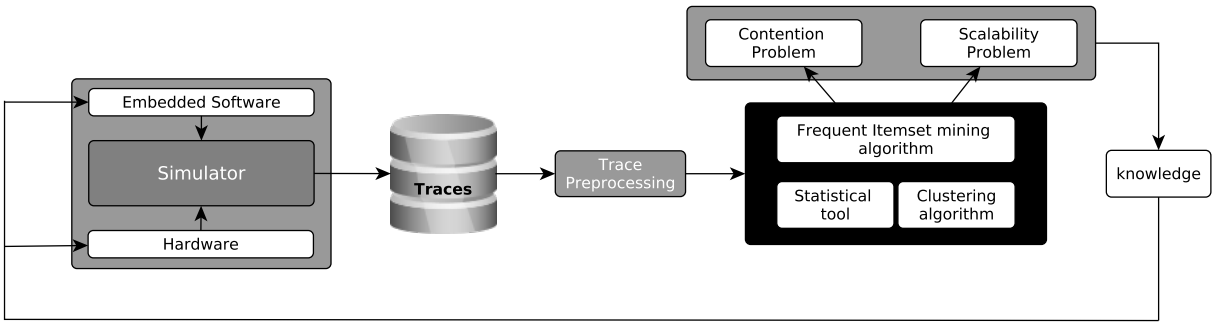


Figure 4.1: Profiling Process Overview

4.1.1 MPSoC Simulation

A MPSoC simulator that we use is SystemC Accurate System Simulator (SystemCASS). SystemCASS[1] is a SystemC cycle-accurate simulator for SoCs 15x times faster than the SystemC simulation kernel [BPG04]. Its goal is to provide cycle accurate based simulation of systems built upon hardware and software components, in order to evaluate performances (hardware/software partitioning, system validation). The hardware platforms are described using SoCLib library. SoCLib [Soc] is an open platform for virtual prototyping of MPSoC. It is a SystemC library of component models. It supports several models of processors (MIPS, ARM, PowerPC, Sparc, MicroBlaze, etc.), of buses, of memories, and several operating systems. SoCLib comes with debugging features like a GNU debugger. SoCLib supports two abstraction level of simulation models: Transaction Level Modeling (TLM) and Cycle Accurate Bit Accurate (CABA). In TLM level, a set of abstraction levels simplifying the description of intermodule communication is defined and the communication between hardware modules are modeled as transactions.

The CABA level models are accurate to the bit and cycle.TLM permits faster simulation but less accurate estimates than CABA. We use CABA abstraction level, in our simulations for cycle-accurate and bit-accurate simulation in order to get a precise performance analysis, timestamp precision of events, and efficient handling of time series of events such as slicing the traces into time windows, or computing the runtime.

4.1.2 Trace Collection

For trace collection, we use a non-intrusive trace system developed by Hedde et al. [HP11]. This trace system consists in tracing hardware events that are produced by models of multiprocessor platform components. They consist of fine grained information about the execution. These traces, viewed as a set of events, are traced and collected with tracing tools system for MPSoC analysis. The component models are instrumented in a

[1] https://www-asim.lip6.fr/trac/systemcass

non-intrusive way so that their behavior in simulation is not modified. Using this trace results allow to run precise analysis of the software that is executed on the platform. Globally, a trace file contains both system calls and the embedded application behavior performed by CPUs and represented by the following information: the timestamps, the CPU identifier that initiate the memory access and the information related to this access. More details are given in the next section.

The traces results contain records of events of each CPU (see 2.4), which occurred during the execution. Traces allow to have precise fine-grain analysis of the software running on the MPSoC platform.

The traces are saved into binary files and can be very large from tens of megabytes to hundred of gigabytes.

4.1.3 Traces Preprocessing

The collected traces are low-level traces. Three kinds of preprocessing are performed and used in different contributions for manipulating the low-level traces:

4.1.3.1 Low-Level and High-Level Traces

The raw traces, as output by the simulator, contain for each trace event information that are useful for our analysis: CPU identifier, timestamp, program counter, instruction type, data address, and memory access latency. But not sufficient for profiling and interpreting results. Furthermore, the embedded application source code is developed in high-level programming language (`C/C++`), hence the necessity of a preprocessing step in order to have both low-level and high-level traces in terms of functions. The transformation process is outlined in Appendix A.

4.1.3.2 The Windowed Events Trace

In order to analyse and discover hotspots during a time period, slicing a trace file into time windows is necessary. The time window gives a snapshot of what happens during a given time period, the set of concurrent accesses of each processor. In addition, time window facilitates the visualization of the concurrent processors. More details are given in the next chapter.

4.1.3.3 Feature of Traces

Due to the huge amount of execution traces, reducing them is a challenge, Thus, this processing may include the transformation and features extraction of the original traces into simplified ones, while keeping the interesting characteristics of traces. More details are given in 6.3.3.

4.1.4 Data Mining Tools

For quantifying and pinpointing contention and scalability bottlenecks, We are interested in :

- Finding groups of memory accesses with similar behavior in a given MPSoC platform.

- Identifying groups of memory accesses according to their access frequency and the access time.
- Comparing multiple and scalable MPSoC platforms for discovering frequent groups of bottlenecks over such platforms.
- Discovering co-occurrence of memory accesses repeating themselves often and occurring contentions.

However, we need automatic tools that allow us to extract knowledges from execution traces. Thus, there are two data mining techniques particularly relevant to this thesis, namely *clustering* and *frequent itemset/pattern mining*.

Clustering is an unsupervised learning process where data are divided into groups. Similar data are grouped into the same group and different data are separated into different groups.

Frequent pattern mining (FPM) is a process in which patterns appearing frequently in a dataset are extracted. Frequent pattern mining often serves as an intermediate step for knowledge discovery, improved data understanding and more powerful data analysis. For example, it can be used as a feature extraction step or classification. For improved data understanding, patterns can be used for annotation or contextual analysis [HK06].

4.1.4.1 Clustering

Clustering is one technique in data mining for finding and organizing data instances into similarity groups, called *clusters* such that the data instances in the same cluster are similar to each other and data instances in different clusters are very different from each other.

There are many clustering algorithms. A good summary and categorization is available in [HK06]. Clustering algorithms can be categorized broadly into hierarchical or partitional. In the hierarchical approaches, the clusters are build step by step by merging or dividing previously formed clusters. In the partitional approaches, all the clusters are build at once, further refinement are then performed by shifting data samples from one cluster to another in the successive clustering steps. One of the most well-known and classic clustering algorithm is *k-means* [HW79], which belong to the partitional family of clustering algorithms.

The *k-means* algorithm is the best known partitional clustering algorithm. It is also widely used among all clustering algorithms due to its simplicity and efficiency. The k-means algorithm divides the dataset into k groups where k is a number of clusters specified by the user. The distance of each data items from the mean the group it belongs (hence the name k-means). Several distance metrics have been proposed. Some of them includes: Manhattan distance, Euclidean distance, and many more. In Manhattan distance the distance between two data points in x-y space is defined as $|x_1 - x_2| + |y_1 - y_2|$. In Euclidean distance the distance between two data points in x-y space is defined as $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$. Algorithm 1 presents the pseudo-code of the K-means algorithm. First, the algorithm forms an initial set of k groups randomly. The mean of each group is then computed. Next, each data point is assigned to the group where the distance (Euclidean distance) between the data point and the group's mean is minimized. The procedure is repeated until a fix point is reached, namely no more data items move into a different group/cluster.

Algorithm 1 Pseudo-code of the K-Means algorithm

Input: D: a data set containing n objects, K: the number of clusters
Output: Set of K clusters

```
arbitrary choose K objects from D as the initial cluster centers;
repeat
    (re)assign each object to the clusters which has the closest mean, based on distance;

    update the cluster means, that is, calculate the mean value of the objects for each
    cluster;
until no change
return WT
```

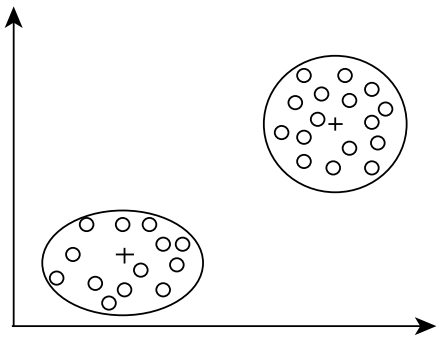


Figure 4.2: Two groups or clusters of data points

Each cluster has a cluster center, which is also called the cluster **centroid**. The centroid, usually used to represent the cluster, is simply the mean of all the data points in the cluster, which gives the name to the algorithm, i.e., since there are k clusters. Fig. 4.2 gives the k-means clustering algorithm with $k = 2$.

4.1.4.2 Frequent Itemset/Pattern Mining

In this thesis, we use frequent *itemset* mining [AS94]. In this setting, we consider a set of items $I = \{I_1, ..., I_n\}$. The data is represented as a list of *transactions* $T = \{T_1, ..., T_n\}$, where each transaction T_i consists of a unique identifier $i \in [1, n]$ and is a set of elements of I: $T_i \subseteq I$. An itemset is also a subset of I.

The support $support(P)$ of an itemset P is defined as the proportion of transactions in the data set which contain the itemset. Formally,

$$support(P) = \frac{no.\ of\ transactions\ which\ contain\ the\ itemset\ P}{total\ no.\ of\ transactions} \tag{4.1}$$

Given a minimum support threshold min_p, an itemset P is *frequent* if $support(P) \geq min_p$.

FPM algorithms search for patterns in a combinatorial search space, which is generally very large. But, the **anti-monotone** property allows fast pruning: which states, "If

Table 4.1: Example data

	Transaction
T_1	$\{A, B, C\}$
T_2	$\{A, C\}$
T_3	$\{A, B, C\}$

Table 4.2: Frequent itemsets

Itemset	Frequency	Itemset	Frequency
$\{A, B, C\}$	2	$\{B, C\}$	2
$\{A, B\}$	2	$\{A, C\}$	3
$\{A\}$	3	$\{C\}$	3
$\{B\}$	2		

a pattern is frequent, so are all it's sub-patterns; if a pattern is infrequent, so are all its super-patterns." Efficient data structure and algorithmic techniques on top of this basic principle enable FPM algorithms to work efficiently on database of millions events.

Usually, effective search space pruning strategy needs to be employed to render FPM algorithms feasible to handle datasets. Some well known algorithms are Apriori [AS94], Eclat [Zak00] and FP-Growth [HPY00].

Lets consider a simple example. We have set of items $I = \{A, B, C\}$, the data is represented in Table 4.1.

The frequent itemsets with their support are given in Table 4.2. This table shows that the frequent itemsets contain many redundant information. In practice, we are only interested in *closed* frequent itemsets, which are the maximal itemsets by set inclusion for a given support value. A pattern is called *closed*, if it has no super-pattern with the same support. Closed frequent pattern can also be mined within the FPM process. They can be an order of magnitude less numerous than frequent itemsets while retaining the same amount of information. The most efficient closed frequent itemsets according to the FIMI contest [FIM04] is LCM [UKA04a]. In our example the closed frequent itemsets are $\{A, B, C\}$ with support 2 and $\{A, C\}$ with support 3.

4.1.5 Knowledge Discovery

Knowledge discovery is the process of discovering "valuable" information from patterns extracted from large traces using data mining algorithms. According to the definition of the knowledge discovery in databases by Fayyad et al [FPsS96]. The information should be novel, non-trivial, previously unknown and potentially useful for developers in order to improve software or hardware proprieties.

4.2 Summary

This chapter introduced an overview of new techniques that we use in our profiling method for MPSoC platforms, which involves trace collection, pre-processing, and data mining steps. Through platforms simulation and adequate tracing tool, the trace tool

generates a trace file containing the behavior of CPUs simulated in platform such as concurrent memory access information.

Data mining algorithms are powerful algorithms, they find valuable patterns and regularities in large volumes of dataset and group similar behaviors, which we will allow to implement methods for contention and scalability bottlenecks detection. In the next chapter, we detail these methods.

Chapter 5: Contention Pattern Discovery in MPSoC

Contents

5.1 Introduction **49**
5.2 Preliminaries and Problem Formulation **50**
5.2.1 NoC 50
5.2.2 Trace Definitions 51
5.2.3 Problem Statement 52
5.2.4 Objective 53
5.3 Contention Pattern Discovery Methodology in MPSoC I **53**
5.3.1 Patterns definitions 54
5.3.2 Pattern discovery method 54
5.3.3 Hotspot detection from patterns 56
5.3.4 Preliminary Results 58
5.4 Approach limitations **58**
5.5 Contention Pattern Discovery Methodology in MPSoC II **58**
5.5.1 Pattern discovery method 59
5.5.2 Long latencies determinations 59
5.5.3 Slicing the execution traces into contention windows 60
5.5.4 Mining the frequent contention patterns 61
5.5.5 Preliminary Results 63
5.6 Comparison of Methodologies **64**
5.7 Conclusion **66**

This chapter presents the first contributions of this thesis: a framework for automatic contention discovery in MPSoC using data mining on simulation traces. The contributions have been the subject of articles in [LTP12, LTP13].

5.1 Introduction

Hardware/software interaction is complex and often not analysable at design time because of the dynamicity of the current applications and architectures. Given this

context, there is a dramatic need for tools that will ease this integration and optimization process.

Since the traffic between two components processor-memory is proportional to the number of processors, for MPSoC platform containing a large number of processors, the resulting interconnection network can become complex and expensive. Even if the interconnection network meets the processor-memory bandwidth requirement, performance degradation can result if several processors share a common link in the interconnection network. This type of contention is referred to as *communication contention*.Another contention is referred to as *memory contention* which is related to the fact that a memory module can handle only one memory request at a time. When several processors request the same memory module, the requests are serialized. Thus, one or more links of the MPSoC platform are accessed simultaneously by more processors than they have been designed to satisfy, leading to delays in response time and increasing the memory access latency from the processor to the memory [MTQ07]. When several processors repeatedly access the same memory location, it gives rise to *hotspot contention*. This is because it creates hotspot in either the memory module or the interconnection network that get overloaded with the requests which create a bottleneck for system performance.

For this reasons, our goal is to detect automatically such inefficiency i.e. concurrent accesses to memory segments leading to hotspots: high latencies and low throughputs. Also, discovering and extracting automatically the frequent and repeated accesses that cause the contention and pinpointing them in source code is a challenge.

The rest of the chapter is organized as follow. Section 5.2 formally presents definitions, problem statement and objectives. We detail our approach and give preliminary results in section 5.3. We enumerate their limitations in section 5.4, and we propose an improved approach with preliminary results in section 5.5. We compare our methodologies with the existing ones, in section 5.6. Section 5.7 concludes this chapter.

5.2 Preliminaries and Problem Formulation

This section gives necessary definitions and problem statement and objectives of the approach.

5.2.1 NoC

Network-on-Chip (NoC) is an efficient on-chip communication architecture for SoC architectures based on network infrastructure, with similarities and differences relative to the classical networks. It enables integration of a large number of computational nodes and storage blocks on a single chip. Each node consists of CPU and their caches, local memory and a router. Lots of topologies have been proposed for NoCs so far, such as 2D Mesh [DYL02], Torus [DT01], Star [AK89], Octagon [KND02], FATTREE [ACG+03]. Among these topologies, mesh topology has gained more consideration by designers due to its simplicity and their grid-type shapes and regular structure which are the most appropriate for the two dimensional layout on a chip/ For example, Figure 5.1 shows a 4 x 4 mesh NoC. Each core communicates with other cores by sending and receiving messages through a network interface controller (NIC) that connects the core to a router (hence the network).

A XY routing algorithm is implemented as routing algorithm of NoC. XY routing is a dimension order routing which routes packets first in x or horizontal direction to the receiver's column and then in y or vertical direction to the receiver. XY routing suits well networks using mesh topology. Addresses of the routers are their xy-coordinates. XY routing never leads to deadlock [DNAKN05].

5.2.2 Trace Definitions

Let TS be a set of trace symbols containing instruction addresses, data addresses and memory accesses types that have been performed by CPU cpu_id at time t with a latency $latency$. A *trace event* $e = (t, s)$ consists of a timestamp $t \in [0, t_{max}]$ and a set of symbols $s \subseteq TS$ representing a memory access that has been performed by an initiator. A raw

Table 5.1: Raw trace format for NoC

Cycle Number	CPU ID(1)	Access type	Page Number	Program Counter	Hop Count	Node ID (2)	Access Latency
212305	1	INST	785408	0xbfc00740	3	10	48

trace, see Table 5.1, consists of, in order of occurrence, the global date at which the event occurred in cycles since the power-up of the system, which CPU initiated the transaction (CPU ID(1)), the transaction type (which can be instruction fetch, data read and data write), the 4Kb page number in which the access takes place, the program counter of the instruction that produced the access, the hop count to destination (CPU ID(2)) which represents the Manhattan distance between the CPU ID(1) that initiate the operation and the memory receiving the requested address, and finally the latency between the start of the transaction and the reception of its acknowledgement. For the rest of this chapter, we use the term "event" to represent both actual CPU event and the corresponding trace event.

The transaction of Table 5.1 is based the topology of the 4x4 mesh network-on-chip NoC of Fig. 5.1.

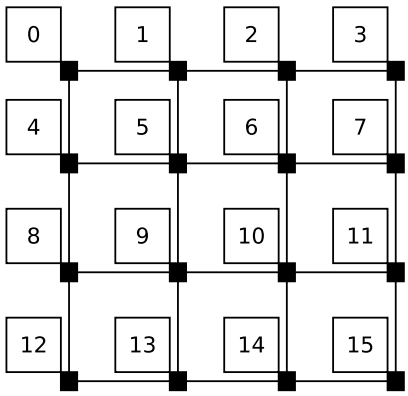


Figure 5.1: Example of 4 x 4 mesh NoC

Let $C = \{1, ..., k\}$ be a set of CPUs. A *CPU trace* T^c, with $c \in C$ a CPU identifier, is a

list of trace events $T^c = \{e_1, ..., e_n\}$ ordered by their timestamps. It represents all the trace events emitted by CPU c during the execution.

The *execution trace* $ET = \{T^1, ..., T^k\}$ is the set of CPU traces for all the CPUs of the MPSoC.

For a CPU trace T^c, we denote by $T^c_{[a,b]}$ the list of trace events of T^c that happen between time a and time b. An execution trace represents a very fine grained information, especially for traces coming from cycle-accurate simulators. In order to coarsen the grain of analysis and discover interesting correlations, the trace is split into *time windows*. The interest of time window allows to: regroup the co-occurring events belonging to the same time period that may be the cause of contention.

Let δ be a *window duration*. Then there are $N_w = \lceil \frac{t_{max}}{\delta} \rceil$ time windows, whose starting and ending times are defined as:

$$\forall i \in [0, N_w - 1] \;\; w_i = (bw_i, ew_i) = (i * \delta, \min((i+1) * \delta, t_{\max}))$$

We consider that inside a time window, all the events appear simultaneously. Hence, the *windowed CPU trace* of a CPU $c \in C$ having CPU trace T^c is $WT^c = \{wt^c_0, ..., wt^c_{N_W - 1}\}$ where for all $i \in [0, N_W - 1]$ $wt^c_i = T^c_{[w_i, w_{i+1}[}$.

Due to the exhaustive nature of our traces, the accesses leading to contention are captured in the trace and called *contention patterns*.

Definition 1 (Contention pattern)**.** *A contention pattern is a set of co-occurring events i.e. whose timestamps differ by at most δ cycles, that are each on a different CPUs which access a similar resource (a memory address), and where at least one of the events exhibits an unusually long latency, indicating contention on the resource.*

Definition 2 (Frequent Contention pattern)**.** *A contention pattern is* frequent *if it occurs in the trace more than ε times.*

5.2.3 Problem Statement

The problem that is addressed in this chapter is to automatically discover recurrent hotspots due to contention (i.e. bad usage of the MPSoC) in an execution. We also want to have hints on the reasons of this contention, in order to be able to improve either task/data placement or directly the code of the application.

In MPSoC, contention develops when multiple processors access the same memory module concurrently, it causes reduced bandwidth and increased memory access time (latency). Profiling contention problem, we can ask the following questions:

- What are the accesses leading to the contention ?
- What happens during a contention ?
- How to detect and identify co-occurrence of events ?
- How to analyse them among large traces ?

5.2.4 Objective

We then rely on advanced data mining techniques which allow to automatically identify and report access patterns whose latency deviate significantly from the average behavior of the traces. This method allows to help developers to extract automatically the parts of the traces exhibiting contention and mining them in order to discover both frequent interactions between several processors and the patterns that create this contention.

The data mining techniques on huge amount of simulation traces allow to analyse and discover frequent patterns of co-occurrence of events that create contention are well adapted to this task because:

- by definition, they can identify patterns repeating themselves often,
- they identify the co-occurrence of some events, which is usually what happens during a contention scenario: for example, CPU1 and CPU2 are both contending for resource A at the same time.

In order to resolve the problem statement and achieve our objectives, we have proposed two approaches according to analysis granularity:

- Either we want to know precisely, ① how many processors create contention in a given time window ? by specifying the minimum number of processors as input of the approach that create contention. ② Are patterns found in ① frequent in multiple time window ? (Approach 5.3)

  The advantage of this approach is: the developer can specify the minimum number of processors that create contention. The approach discovers with accuracy the number of processors greater than the minimum number of processors and their identifiers participating at the contention in a given time window. The disadvantage of this approach is: it takes a lot of event combinations and requires a large computing power.

- Either we partially abandon the user input parameter of the previous approach (i.e. minimum number of processors) but in this approach we keep only time windows with potential contention. To make this choice, the discovered time window is based on latency treatment. And after, we mine the time window. (Approach 5.5)

  The advantage of this approach is: the latency is a posteriori indicator of contention, and the approach is faster than the first one. The disadvantages of this approach: we have not any information about the number of CPUs involved in the contention. At best, if the contention is stable, then we can know the CPUs involved.

5.3 Contention Pattern Discovery Methodology in MPSoC I

We describe in this section our method to detect hotspots in the trace of a multicore execution by discovering frequent itemsets in this trace. We do this in two phases: first, we identify the most recurrent patterns, and second we search within the resulting set the ones that have the highest execution times in the considered window. First we give some definitions.

5.3.1 Patterns definitions

In order to detect contention, our principle is twofold: we want to identify groups of events that occur simultaneously in a significant number of CPUs and in a significant number of time windows.

We first define a way to aggregate all the windowed CPU traces in a single trace that keep only trace events appearing simultaneously in a significant number of CPUs.

Definition 3. *Let w_i $i \in [0, N_w - 1]$ be a time window and $min_sup_c \in [1, k]$ be a threshold on the number of CPUs. A trace event e is* frequent *if there exits at least d CPUs $> min_sup_c$ such as $e \in wt_i^d$.*

The frequent events window *fw_i corresponding to window w_i contains all the frequent trace events for window w_i.*

The windowed frequent events trace *$FWT = \{fw_0, ..., fw_{N_w-1}\}$ contains all the frequent events windows.*

Figure 5.2 shows an example of a frequent events window. It shows the events occurred in each CPUs during the window 1. We assume that $min_sup_c = 2$. The events A and B occur in 3 CPUs, whereas C,M and N appear only in 1 CPU, so only A and B are frequent events.

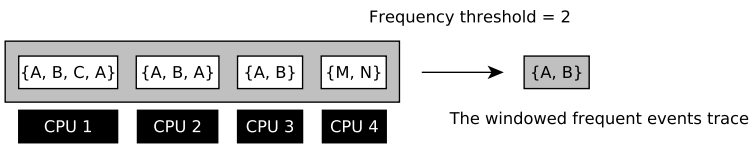


Figure 5.2: The windowed frequent events trace

The next step is to discover groups of frequent trace events that appear in at least a given number of frequent event windows.

Definition 4. *Let $min_sup_w \in [0, N_w - 1]$ be a frequency threshold on the number of windows. A* frequent pattern *is a set of frequent trace events that occur simultaneously at least min_sup_w windows.*

The Figure 5.5 shows 3 time windows, with their frequent trace events on the right. We consider a frequency threshold $min_sup_w = 2$. The frequent event A is the only one that appears in 2 windows, so $\{A\}$ is the frequent pattern discovered.

5.3.2 Pattern discovery method

Given an execution trace ET and three thresholds min_sup_c, min_sup_w and δ time window size as user input, our goal is to discover automatically the frequent contention patterns. If the min_sup_w threshold is set sufficiently high, the frequency of a contention pattern indicates that it is not a rare and difficult to predict situation, but a misuse of the resources which comes from the application design, and that should be fixed to improve overall performances.

Our discovery method is based on three steps: preprocessing the raw traces, computing the windowed frequent events trace, eventually compute the frequent patterns

and hotspot detection. Fig. 5.3 shows the global methodology of the contention pattern discovery process from execution traces in MPSoC.

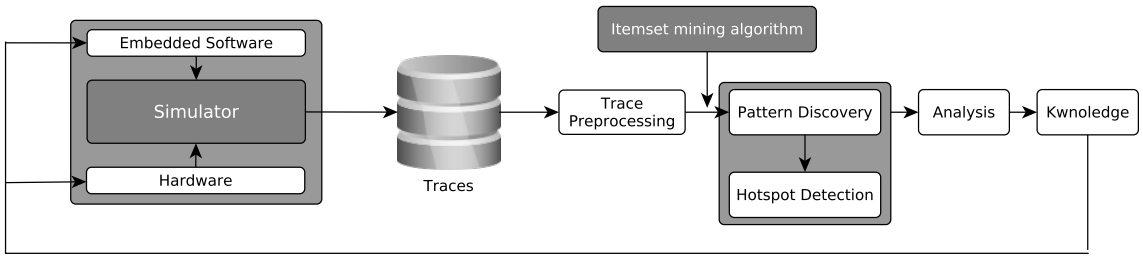


Figure 5.3: Contention Pattern discovery methodology from execution traces in MPSoC

The execution traces are generated from MPSoC simulator using a tracing tool system. The simulator has the hardware description platform and multithreaded application as inputs. Now, we show the rest of the steps.

Discretization of Latency

Discretization is one of the most important, and often required, preprocessing methods in data mining. The task of discretization of numerical value is well known to statisticians.

The goal of discretization is to reduce the number of values, by grouping them into a number n of intervals (bins or slots). Also, the reduction is necessary for further data mining tasks because frequent itemset mining algorithms are not adapted to mine numerical values.

Thus we discretized numeric attributes into larger bins in order to regroup events exhibiting similar values.

As an example, the memory access latencies are discretized by bins of 20 up from 0 to 140, (*i.e.* all latencies between 0 and 20 are represented by the bin lat_0_20). The choice to take bins of 20 is that the memory access average latency is approximately 20 cycles, in our experimental setup.

From Traces To Transactions

Using the window duration parameter δ, we split the trace of each CPU into windows, merge the trace events in each window and produce the windowed CPU traces as described in 5.3.1.

5.3.2.1 Windowed frequent events trace computation

This computation is straightforward. For each window w_i and each event $e_j \in \{e_1, ..., e_n\}$, we compute the number of CPUs in which the event e_j occurs and we keep only the frequent events, *i.e.* the frequency of the event is greater than the frequency threshold min_sup_c, for the next step. These events are inserted into the window fw_i of the windowed frequent events trace FWT. An overview is shown in Figure 5.4.

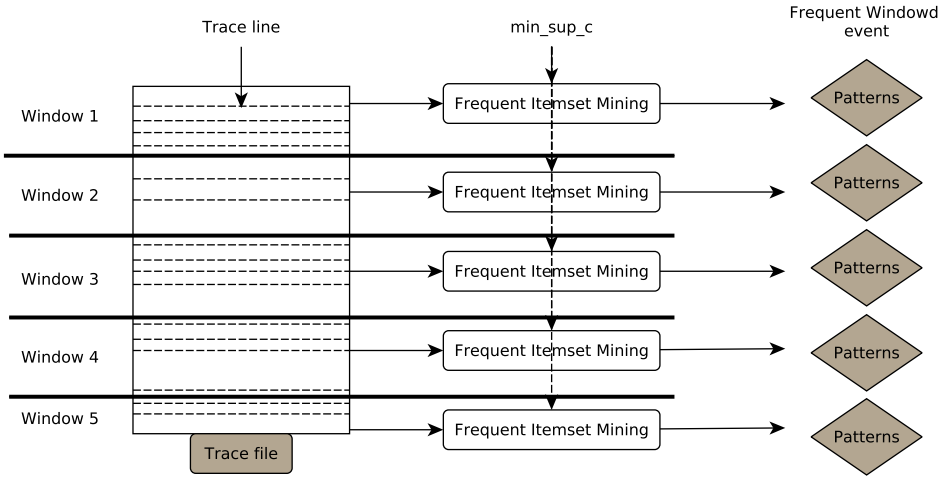


Figure 5.4: Overview of Windowed frequent events trace computation

5.3.2.2 Patterns computation

From the windowed frequent events trace defined before, we build a matrix where the lines are the windows and the columns represents the frequent trace events. This matrix is fed to the most efficient closed frequent itemset mining algorithm, LCM[UKA04b], with the min_sup_w support threshold, which outputs the frequent patterns as defined before.

Figure 5.5 shows that the frequent pattern $\{A\}$ with the intermediate matrix used to apply the mining algorithm.

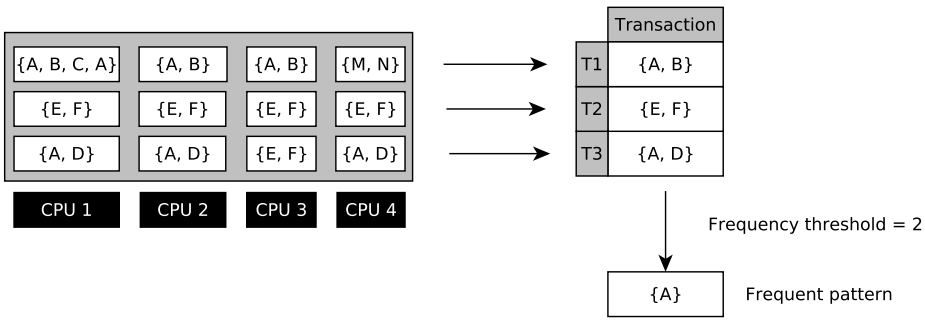


Figure 5.5: Example of frequent patterns

5.3.3 Hotspot detection from patterns

The frequent patterns by themselves can give good hints for potential contention periods and their reasons. However, they can be very numerous and be of unequal interest (for example, presence of obvious patterns).

We thus need a way to score the patterns in order to determine the patterns responsible for the most contention and present them in priority. For this score, we observe that

the problem of contention in a MPSoC can be assimilated to the problem of scheduling tasks in a single processor.

Liu and Layland [LL73] were the first to study the problem of scheduling tasks on single processor. They did show that an optimum scheduler possesses an upper bound to processor utilisation which may be as low as ≈ 70 percent for large task sets.

For detecting the frequent contention, we thus use the frequent patterns found in the last method and we use the following test based on Liu and Layland work:

$$\forall \omega \in W, \forall p \in P, U_{i\omega}^{p} = \sum_{i=1}^{n} \frac{D_{i\omega}^{p}}{\delta} \leq n(\sqrt[n]{2} - 1) \tag{5.1}$$

$$\forall p \in T^{c}, D_{i\omega}^{p} = \sum d_{p} \tag{5.2}$$

where W is the set of windows, and n is the number of CPU. When this number of CPU tends towards infinity the expression (5.1) will tend towards:

$$\lim_{n \to \infty} n(\sqrt[n]{2} - 1) = \ln 2 = 0.6931... \tag{5.3}$$

$D_{i\omega}^{p}$ is the duration time of frequent pattern P into each window ω. It is defined as follows: Let a frequent pattern P discovered from the last method. For computing the duration D of the pattern P into each window, we proceed as follows:

- Firstly, we project the pattern P on the trace events T^{c} by the matching between P and T^{c}.
- Secondly, we extract the memory access time of the pattern P from its latency.
- Thirdly, we sum the latency of the events for each window of the CPUs (Eq 5.2).

So $U_{i\omega}^{p}$ is applied for each patterns P to determine a memory access rate by the CPUs in the window ω. Thus, if this measure is greater than threshold ($ln\ 2$), then we say this pattern create a contention in the window ω .

Fig. 5.6 shows an example of the hotspot detection from a pattern found in Fig. 5.5. The frequent pattern A is projected in the trace events and its duration extracted at each time window for all CPUs. For each window, we compute U such as the window duration $\delta = 200$. We see that there is a contention in the third window.

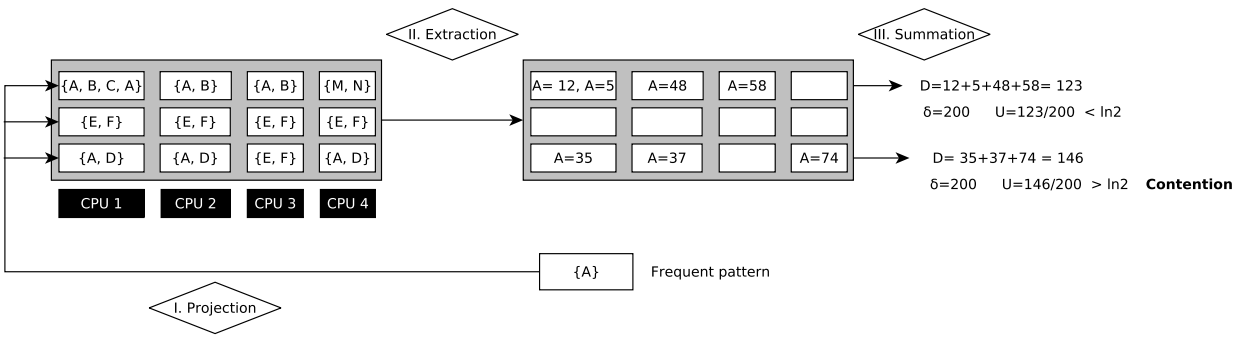


Figure 5.6: Example of hotspot detection from patterns

5.3.4 Preliminary Results

We experiment this approach in a 4x4 mesh NoC hardware platform. The nodes run multithreaded Mandelbrot fractal application. More details on the simulation environment are given in chapter 7.

We discovered interesting patterns about the nodes running the Mandelbrot application 5.2. The most accessed instructions by the concurrent CPUs: CPU_8 to CPU_15 are located in the following memory pages: P_3091, P_3090, P_3088. The pages P_3091 and P_3090 contain the code of the floating point helper functions (as our processors do not support floating point operations directly, the compiler has automatically instantiated *ad-hoc* functions to perform the operations) and the page P_3088 contains the code of the Mandelbrot application. Among these frequent concurrent accesses, we found that the frequent contention pattern is the accesses to pages containing floating point operations after each access.

Table 5.2: Frequent Contention patterns

Application	Frequent Pattern	Support
Mandelbrot	CPU[8,15] INST P_3090 fpadd_parts id_10	96%
	CPU[8,15] INST P_3090 __muldf3 id_10	62%
	CPU[8,15] INST P_3088 mandelbrot id_10	95%
	CPU[8,15] INST P_3091 __unpack_d id_10	87%
	CPU[8,15] INST P_3090 __pack_d id_10	80%

More details, discussions and experiments about this approach in the chapter 7.

5.4 Approach limitations

The first version of contention pattern discovery methodology is intended for NoC architecture using specific and adequate tracing tool system.

The approach described in this chapter admits the following limits:

- Two key problems associated with discretization of latency values are how to choose the number of intervals (bins), and how to decide on their width.
- Frequent itemset mining algorithm is applied twice for: identifying patterns that occur simultaneously in a significant number of CPUs and in a significant number of time windows. This process can take twice the execution time of Frequent itemset mining algorithm.

Thus, the extended version of contention pattern discovery methodology use another tracing tool for SMP architecture and a totally data mining driven method for hotspot detection and frequent contention pattern discovery. These limits are improved in the next sections.

5.5 Contention Pattern Discovery Methodology in MPSoC II

Discovering contention patterns is a multi-step process. As we detect contention through instruction latency value, the first step is to determine what is an *unusually* high latency.

This information can of course be given as input, however we show a simple method to determine it semi-automatically. Then the execution trace must be filtered to keep only events that are around high latency events, and this filtered execution trace requires further preprocessing in order to be fed to a pattern mining algorithm that will discover the contention patterns.

5.5.1 Pattern discovery method

Given an execution trace ET and two thresholds ε and ω, our goal is to discover automatically the frequent contention patterns. If the ε threshold is set sufficiently high, the frequency of a contention pattern indicates that it is not a rare and difficult to predict situation, but a misuse of the resources that come from the application design, and that should be fixed to improve overall performances.

Our discovery method is based on three steps: preprocessing the raw traces, computing the windowed frequent events trace, eventually compute the frequent patterns and hotspot detection. Fig. 5.7 shows the global methodology of the contention pattern discovery process from execution traces in MPSoC.

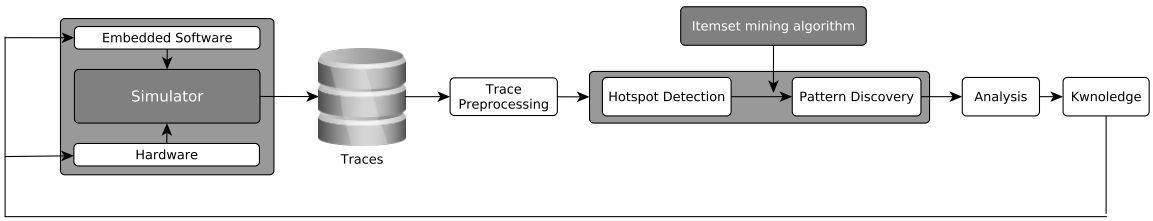


Figure 5.7: Contention Pattern discovery methodology from execution traces in MPSoC

5.5.2 Long latencies determinations

The latencies will the analysed through simple statistical techniques. Let L denote the list of all non-trivial (i.e. not cache hits) latencies found in the execution trace, whatever the CPU: $L = \{e.latency \mid e \in ET\}$. Without domain knowledge, the basic assumption that we make is that most memory accesses are done without contention. The median of the latency values in L is thus supposed to be representative of the normal access latency. In order to allow for some variations in the latency value, we only consider as unusual latency values that are in the *upper quartile*, i.e. the highest 25% of the latency values. $Q_3(L)$ denotes the lowest latency of the upper quartile. This is a standard statistical way to identify high values in a dataset [Pot06]. It can be represented graphically by a boxplot, as in the Fig 5.8. In this figure the median is at the middle, 50% of values are lower (left of median) and 50% of values are higher (right of median). Q_3 represents the limit of the upper quartile, with 75% of values below and 25% of values above.

Hence for latencies, the set L_H of high latency values contains all latencies above $Q_3(L)$, i.e right of Q_3 in the figure, $L_H = \{l \mid l \in L \wedge l \geq Q_3(L)\}$, in Fig 5.8. We see that the set L_H contains all latencies above $Q_3 = 12$ cycles.

Note that in the case where most memory accesses are done under contention, the median will be higher than in a case without contention. Our set L_H will contain only

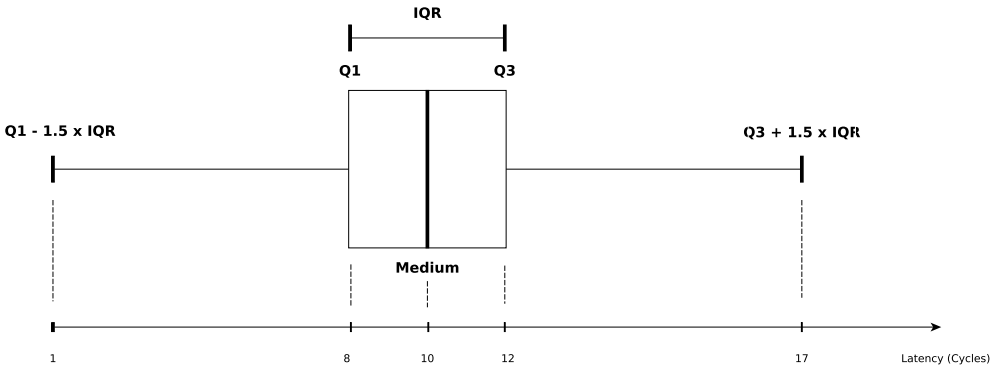


Figure 5.8: Boxplot

the extremely high latency values, i.e. memory accesses for which contention have the most adverse effects. Focusing on these accesses is thus anyway a priority, so our method can also be applied to these cases.

5.5.3 Slicing the execution traces into contention windows

By having identified high latencies, the execution trace can be filtered to focus on events having these latencies and their immediate surroundings. The output of this filtering step is a sequence of *contention windows*. A contention window is a slice of an execution trace having a duration ω, and which contains one or more high latency events. It thus give us the context of occurrence of high latency events. This context is important as if ω is not too long, we can be sure that the contention patterns are located inside.

For constructing the windowed events trace, our solution is presented in Algorithm 2. It receives as input a contention window duration ω, the execution trace ET, and a set of high latency events HL defined by: $HL = \{e \mid e \in ET \wedge e.latency \in L_H\}$. HL is sorted on increasing timestamp in order of events, as well as is ET. The algorithm outputs the set WT of contention windows. It's principle is as follows: for each high latency event $e_H \in HL$ increasing according to the timestamp order (line 3), all the events from the execution trace ET that surround it (at most $-\omega/2$ cycles before or $\omega/2$ cycles after) are inserted into the current window (lines 4-5). If one of these events is a high latency event, it is removed from HL (lines 6-8) to avoid making a near-duplicate window in the next iteration of the for all (line 2). A single time window can thus have several high latency events, which is expected as high latencies come from several CPUs competing for a single resource and thus slowing down each other.

Our algorithm authorizes some overlap between windows, but it will be limited to at most $\omega/2$ cycles for a couple of overlapping windows.

Fig.5.9 shows an example of the windowed execution trace on 4 CPUs. The window 1 is constructed according to the first high latency event encountered in any of the CPUs. Here it is `C` on CPU_0. The window 1 contains the events of all CPUs occurring from $(-\frac{\omega}{2})$ before `C` to $(+\frac{\omega}{2})$ after `C`. Here a second high latency event occurs in this window (event `A` on CPU_1). The window 2 is constructed according to the high latency event encountered in any of the CPUs after exiting window 1. Here for example lets consider it is `D` on CPU_1. The window 2 contains the events of all CPUs occurring from $(-\frac{\omega}{2})$

Algorithm 2 Windowed events trace

Input: duration ω, execution trace ET, high latency events HL
Output: Windowed trace WT

```
1:  n ← 0
2:  for all e_H ∈ HL do
3:      WT[n] ← ∅
4:      while ET[i].ts ≥ e_H.ts − ω/2 AND ET[i].ts ≤ e_H.ts + ω/2 do
5:          WT[n] ← WT[n] ∪ {ET[i]}
6:          if ET[i] ∈ HL then
7:              HL ← HL \ {ET[i]} {Avoid some overlapping windows}
8:          end if
9:          i ← i + 1
10:     end while
11:     n ← n + 1
12: end for
13: return WT
```

before D of CPU_1 to $(+\frac{\omega}{2})$ after D of CPU_1. We also see that window 2 overlaps partially window 1.

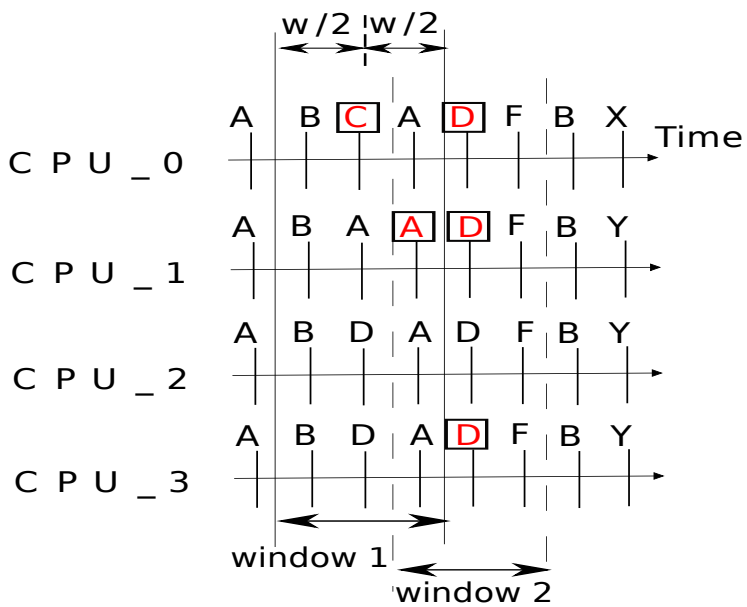


Figure 5.9: The windowed events trace

5.5.4 Mining the frequent contention patterns

There exists many different algorithms for mining patterns in data. In our case, the most important information is the frequent co-occurrence of set of instructions, memory address and memory access types represented by the repetition of the set of trace

symbols over several CPUs. Our assumption is that due to the way contention windows were selected, any set of events appearing frequently in these windows is suspicious and have a high chance to be involved, directly or indirectly, in the contention that drives up the latencies. The data mining technique used to discover such sets of frequently occurring events is called *frequent itemset mining algorithm* [AS94, UKA04b]. In this technique, the first input is a multiset of *transactions* $D = \{t_1,..,t_p\}$ defined over an alphabet of *items* $\Sigma = \{i_1,..,i_q\}$, where $\forall t_i \in D \;\; t_i \subseteq \Sigma$. The second input is a *minimum support threshold* $\varepsilon \in [0,p]$. Frequent itemset mining algorithms then extract all the *frequent itemsets*, i.e. all the *itemsets* $is \subseteq \Sigma$ that appear in more than ε transactions of D. More formally, is must satisfy $support(is) \geq \varepsilon$, where $support(is) = |\{t_i \mid t_i \in D \wedge is \subseteq t_i\}|$.

In order to exploit this technique, we transform the set of windows WT into a set of transactions D. This is presented in Algorithm 2. Each window $w \in WT$ becomes a transaction (lines 2-5), i.e. a set of items. We thus loose the sequencing of events inside a window, for the data mining algorithm all the events of a single window are considered simultaneous. This is not a problem, as inside a window we are interested in the co-occurrence of different events and not in their precise sequencing at the cycle scale.

Algorithm 3 Windowed events transactions

Input: Windowed trace WT
Output: Transactions dataset D
1: **for all** $w \in WT$ **do**
2: $D[i] \leftarrow \emptyset$
3: **for all** $e \in w$ **do**
4: $D[i] \leftarrow D[i] \cup \{e.cpuid_e, e\}$
5: **end for**
6: **end for**
7: **return** D

Pattern mining algorithms are complex algorithms that explore a large combinatorial space i.e the algorithms have exponential time complexity according to the number of items in order to compute the exacts solutions. In order to do so efficiently, they exploit several properties of sets over the alphabet Σ, and of the frequency definition. Changing any of these properties prevents from using the most efficient algorithms. Thus, in order to mine complex data while keeping good scale up properties, a delicate problem is to find an alphabet Σ that allows to find informative patterns while fitting with the pattern mining framework defined above. In our case, the alphabet Σ of transaction items should at least contains all the possible trace symbols TS. But consider the case where two CPUs CPU_1 and CPU_2 make the same memory access, represented by $a \in TS$, in a single contention window. The associated transaction is a set and not a multiset, so it will only be the singleton $\{a\}$. This will not allow to discover any contention: a single CPU issuing access a would have given the same transaction. Our solution is to prefix each trace event with the CPU that issued it: here this will give $\{CPU_1_a, CPU_2_a\}$ such that all CPUs of the platform here CPU_1, CPU_2 are in transaction, and it gets possible to find contention patterns. Now consider the case where we have two transactions $t_1 = \{CPU_1_a, CPU_2_b\}$ and $t_2 = \{CPU_1_b, CPU_2_a\}$. There is no itemset common to both t_1 and t_2, as they contain completely different items: the algorithm cannot see

their similarity and does not extract any frequent pattern. Our solution is to keep also the original event with its CPU prefix (line 4). This gives: $t_1 = \{CPU_1_a, CPU_2_b, a, b\}$ and $t_2 = \{CPU_1_b, CPU_2_a, b, a\}$, with a common pattern being to have simultaneously the events a and b. We thus have $\Sigma = TS \cup (\{CPU_1, ..., CPU_k\} \times TS)$. Once we have the transactions, we can use a state of the art frequent itemset mining algorithm. We use LCM [UKA04b], the most efficient one according to the FIMI contest [FIM04]. The resulting frequent itemsets are the contention patterns that we are looking for.

5.5.5 Preliminary Results

In this section, we show the the preliminary results of this approach, taking into consideration the following characteristics:

The hardware platform is a shared memory multiprocessor that contains n MIPS32 processors such as $n = \{1, 4, 8\}$, interfaced with one data cache and one instruction cache. It also contains one memory and others peripherals components: a timer, an interrupt controller, a frame buffer, a block device, a tty. More details of simulation environment in the chapter 7.

We perform two simulations on different platforms: platform 1 and platform 2 which contain 4 and 8 processors, respectively. The software that runs on these platforms is a parallel Motion-JPEG decoder on top of an operating system for embedded system that includes a Pthread library. These platforms should exhibit different contention levels, and thus help us validate our approach of contention pattern discovery.

In order to discover contention patterns, we apply our approach to the contention windows converted to transactions with a minimum support threshold of $\varepsilon = 65\%$: we are interested in interactions between functions, memory locations and CPUs that occur in more than 65% of contention windows, *i.e.* very frequently parts of the traces exhibiting potential contention. We focus on platform with 4 and 8 processors, which exhibit high levels of contention. The most interesting contention patterns discovered for these simulation platforms are presented in Table 5.3.

Table 5.3: Frequent Patterns

Platform	Frequent Pattern	Support
4 CPUs	CPU[0,3] [0x10009ee4, 0x10009f78] idct [0x10016b50, 0x10016f2c] memcpy lat_10_20 lat_20_30	72 %
8 CPUs	CPU[0,7] [0x10009b10, 0x1000a224] idct [0x10016ab0, 0x10016e8c] memcpy lat_10_20 lat_20_30	88 %

The pattern of the platform using 4 CPUs shows a concurrent memory access pattern that creates a contention implying all 4 CPUs and occurring in 72 % of contention windows. This pattern shows a frequent interaction between the functions *idct* and *memcpy*, and more specifically between the loops of *idct* located in address interval [0x10009ee4, 0x10009f78] and the loops of *memcpy* located in address interval [0x10016b50, 0x10016f2c]. The pattern also shows that the usual latencies around these interactions are between 10 and 30 cycles (lat_10_20, lat_20_30). Having in mind that

the high latency threshold is $Q_3 = 12$ for the 4 CPUs trace, this corresponds well to contention latencies.

The pattern for the 8 CPUs platforms is the same as on the 4 CPUs platform, with different addresses due to a different executable. However these addresses correspond to the same assembler instructions than previously: this enforces the importance of the *idct*/*memcpy* interaction. In the 8 CPUs platform the pattern has an even higher support of 88 %, whereas there are more contention windows in this case: this pattern is clearly the main responsible for most of the contention and thus the lack of scalability when the number of cores increases. This pattern thus helps the application developer to know that the *idct* function, which performs the inverse discrete cosine transformation, has negative interactions with *memcpy*, a function for copying data from one address to another. It even pinpoints the specific assembler instructions of both functions that are the most impacted: the developer, which is more likely to work on *idct* than *memcpy*, will know immediately which loop of *idct* he/she has to work on.

More details, discussions and experiments about this approach in the chapter 7.

5.6 Comparison of Methodologies

Table 5.4 shows a comparison of existing methodologies in literature of contention detection. The methodologies are described in details in related work chapter 3.2.1. The comparison is performed in terms of the kind of architecture used for contention detection, use of traces in the contention detection methodology, yes or no if the methodology is integrated in routing algorithm, the specification of contention detection tool.

The originality of our approach compared to the related works is as follow:

- Profiling and analysing concurrent application behavior in MPSoC platform.
- Identifying and extracting the implicit patterns from large execution traces.
- Extracting, Quantifying and pinpointing the hotspots.
- Pinpointing contention in the source code.

Paper	Application Domain	Architecture	Traces Analysis	Routing Algorithm	Tool	
					Hotspot Detection	Co-occurence
Our approaches: Ver 1.0 [LTP12] Ver 2.0 [LTP13]	MPSoC	All	Yes	No	Metrics Statistically: Boxplot	Frequent itemset mining
[TdRC+10, AZXC11, RL10] [QLD09] [JKLS10]		NoC	No	Yes	Metrics Formal analysis Statistical regression: MARS	X
[JSMP12]	Heterog-eneous CMP	SCMP ACMP	No	No	Source code instrumentation	X

Table 5.4: Comparison of contention analysis methodologies in MPSoC

5.7 Conclusion

The automatic identification of application contention is an important issue for the optimization of application deployment in integrated multiprocessor platforms. Using the trace generation capabilities of nowadays well accepted virtual platforms, we have introduced a frequent itemsets mining algorithm that allows to automatically identify the frequent patterns that occur concurrently.

Our contention patten discovery framework provides the developer, the necessary and sufficient knowledge to improve the concurrent application. The framework offers the following originalities, first, it targets explicitly MPSoCs and/or multicore processors and addresses the delicate problem of memory contention. Second, it relies on a completely automatic data mining approach that both finds contention points and presents explicitly what happens frequently at these points. Third, we provide a framework adapted to instruction-level traces.

Chapter 6: Scalability Bottlenecks Discovery in MPSoC

Bottlenecks occur in surprising places, so don't try to second guess and put in a speed hack until you have proven that's where the bottleneck is.

Rob Pike

Contents

6.1 Introduction **68**

6.2 Preliminaries and Problem formulation **68**

6.2.1 Definitions 68

6.2.2 Problem Statement 69

6.3 Scalability bottlenecks discovery method **70**

6.3.1 Overview of the method 70

6.3.2 Trace collection 71

6.3.3 Feature extraction 71

6.3.4 Feature-based clustering 72

6.3.5 Growth rate of hot cluster 72

6.3.6 Frequent scalability bottlenecks mining 74

6.4 Preliminary Results **75**

6.5 Comparison of Scalability Bottlenecks Detection Methodologies . . **75**

6.6 Conclusion **76**

This chapter presents the second contribution: scalability bottlenecks discovery in MPSoC platforms using data mining on simulation traces. Understanding scalability of programs is a difficult problem; indeed, it is one of the fundamental problems in parallel computing. The lack of scalability of parallel applications are numerous, and it can be time consuming for a developer to pinpoint the correct one. In this chapter, we propose a fully automatic method which detects the instructions of the code which lead to a lack of scalability. The method is based on data mining techniques exploiting low level execution traces produced by MPSoC simulators.

6.1 Introduction

In this chapter, we focus on one critical point of parallel applications, their *scalability*. Intuitively, a parallel program is *scalable* if it runs n times faster on n cores than on 1 core. In this case, it is said that there is a *linear speedup*. In practice such scaling cannot be obtained by all programs, and the well known Amdahl's law gives a more precise bound on the maximal speedup that can be reached by a given application. There are many reasons that can prevent the scaling of a parallel application: the program can spend too much time doing synchronization, it can suffer from congestion on memory accesses or accesses to other external resources, or there can be load unbalance, or cache trashing, etc. It is tedious for an application developer to find the correct reason for a lack of scalability among all those.

Thus, we propose a fully automatic method that discovers the main reasons for lack of scalability of an application, and reports the exact code lines involved. The developer can thus directly concentrate on understanding and solving the problem found, gaining a lot of time in the profiling process. Our method is based on the analysis by data mining techniques of low-level execution traces produced by running the application on a MPSoC simulator. Using such simulators is already part of the workflow of MPSoC application development. Indeed, due to the fast evolution rate of these chips, applications often start to be developed before the chip physically exists. Because of the complex execution of these applications on MPSoC, collecting traces and analyzing them *a posteriori* has emerged as the best way to understand the complex interactions between the components of the MPSoC. Our method thus integrates in the existing workflow of MPSoC application development, bringing further benefits for profiling scalability.

The rest of the chapter is organized as follow. Section 6.2 formally presents definitions, problem statement and objectives. We detail our approach in the section 6.3. In the section 6.4, we give preliminary results. We compare our methodologies with the exciting one, in the section 6.5 and in the section 6.6 we conclude this chapter.

6.2 Preliminaries and Problem formulation

The execution traces used in this chapter are the same as the ones described in 2.1.

6.2.1 Definitions

We give definitions and notations used throughout this chapter.

Definition 5 (% Time spent)**.** *Given an execution trace ET and an address $@_i$, $\%_time_spent(@_i, ET)$ is the percentage of the total execution time of the program spent in this adress. Let $ET(@_i) = \{e \mid e \in ET \wedge e.s \supseteq \{@_i\}\}$ be the events of ET that are accesses to $@_i$, we have:*

$$\%_time_spent(@_i, ET) = \frac{\sum_{e \in ET(@_i)} e.latency}{\sum_{e \in ET} e.latency} \times 100$$

Definition 6 (% accesses)**.** *Given an execution trace ET and an address $@_i$, $\%_accesses(@_i, ET)$*

is the percentage of the total number of accesses that were done to $@_i$.

$$\%_accesses = \frac{|ET(@_i)|}{|ET|} \times 100$$

From these metrics, it is possible to evaluate how detrimental to performance an access is likely to be:

Definition 7 (Hot predicate, hot access)**.** *Given an execution trace* ET *and an address* $@_i$*, a predicate* $isHot(@_i, ET)$ *is called* hot predicate *if it answers* $true$ *when both* $\%_time_spent(@_i, ET)$ *and* $\%_accesses(@_i, ET)$ *are significantly higher for* $@_i$ *than for the other addresses, and* $false$ *otherwise.*
An $@_i$ *for which* $isHot(@_i, ET) = true$ *is called a* hot access.

There are many ways to define a hot predicate. Usually, this is done by statistical methods based on characteristics of the distribution of values for $\%_time_spent$ and $\%_accesses$: for example by taking only the upper quartile of both distribution. The disadvantage of such methods is that they require a parameter that determines from which point an access starts to be considered as hot. In this chapter, one of our contributions, is to propose a parameterless way to express the predicate $isHot$.

The definition of hot predicate exhibits the two main characteristics of problematic regions of the code in a parallel trace: first, the time spent and number of accesses are unusually high for a set of accesses, and second this problem occurs several times in the execution, further degrading performance. However, such hot predicate, even if detrimental for performances, may have no impact at all on the parallel scalability of the application considered. We thus propose a definition of hot predicates having an impact on scalability: the *scalability hotspots*.

Definition 8 (Hotspot)**.** *A* hotspot *HA is a set of hot accesses appearing consecutively in the execution trace ET.*

Definition 9 (Scalability hotspot)**.** *Let* $P_1, .., P_k$ *be* k *homogeneous MPSoC platforms only differing in their number of cores, with for all* $i < j \in [1..k]$ P_i *has less cores than* P_j*. Let* $ET_1, ..., ET_k$ *be execution traces of an application, where* ET_i *has been produced on platform* P_i *using all its cores. Let* min_p *be a user given threshold, with* $min_p \in [1..k]$.
A set of accesses HS *is a* scalability hotspot *if:*

- *For the accesses in* HS*, the metrics* $\%_time_spent$ *and* $\%_accesses$ *increase with the number of cores of the platforms where* HS *is a hotspot*
- HS *is a hotspot in at least* min_p *execution traces of* $ET_1, .., ET_k$

These definitions are necessary for identifying code regions executed frequently, allowing a developer to focus on optimizing those regions to build an optimized software or hardware implementations.

6.2.2 Problem Statement

Given a set of execution traces $ET_1, ..., ET_k$ produced by platforms $P_1, .., P_k$ as defined above and user threshold min_p, our goal is to discover the scalability hotspots of the traces.

Such scalability hotspots are the parts of the code that are most likely to impact parallel scalability, and should be investigated in priority by the application developers. Thus, our objective is to quantify and pinpoint the bottlenecks in multi-threaded application.

In order to discover automatically scalability hotspots, our approach exploits data mining techniques. The following section introduces these techniques.

6.3 Scalability bottlenecks discovery method

6.3.1 Overview of the method

The five major scalability bottlenecks of multi-threaded workloads on multi-core hardware are: resource sharing, cache coherency, synchronization, load imbalance, and parallelization overhead [EDBE12]. For discovering such scalability bottlenecks, our proposed approach uses data mining techniques in order to analyze automatically large quantities of execution traces and discover such bottlenecks of software.
This approach takes as input a set of traces resulting from the simulated execution of the same program, with the same parameters, on a simulated MPSoC with a varying number of cores. We call each of these MPSoC instances a *platform*. Our approach outputs addresses that are the more impacted when running on more cores, forming what we call scalability hotspots (Def 9). These addresses can either be addresses of instructions in the code, pinpointing parts of the code that are responsible of the scalability issues, or addresses of data, indicating where the memory of the MPSoC is not used efficiently. Fig. 6.1 shows the outline of our approach, which consists of these principal steps: feature extraction, feature-based clustering, mining and analysis of scalability hotspots. We now present each of theses steps in detail.

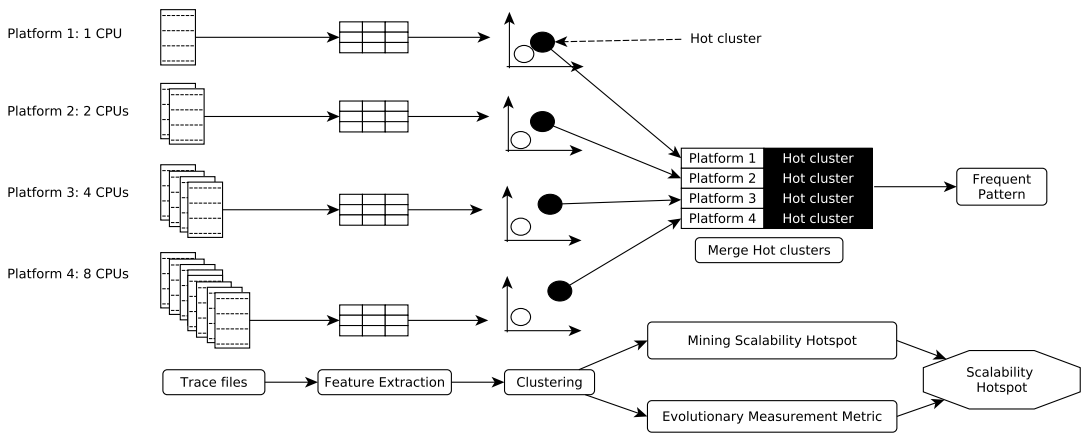


Figure 6.1: Global approach for scalability hotspot in MPSoC platforms

6.3.2 Trace collection

The trace files are obtained by running the same multi-threaded program on different instances of the multi-processor platform, each instance differing from the other by its number of processors (either by instantiating more processors, or simply by having only a subset of all processors actually active). From each platform instance, the execution traces are saved into trace files. These trace files are used for discovering scalability hotspots across the platform instances. In the next step of our approach, each trace file corresponding to a MPSoC platform is applied independently of the other trace files the two following steps: feature extraction and feature-based clustering.

6.3.3 Feature extraction

Trace processing may include transformation of the original traces into simplified ones, along with reduction of dimensionality by extraction of only the most informative features from a huge amount of execution traces. Extracting the features may improve the recognition process and make easier the extraction of the critical zones through the consideration of only the most important traces representation.
For a trace of a given platform, each trace line gives information on an access to an address. For each such address, we compute the following statistics, called *features*: $\%_time_spent$ and $\%_accesses$ according to definitions 5 and 6.
The features are represented by the following address-feature vector of the platform $j \in [1,p]$ where the platforms are numbered in $[1,p]$ and platform j has by convention 2^j cores and has corresponding raw trace ET_j:

$$X_{p_j} = [X_1, X_2, ..., X_{n_j}] \tag{6.1}$$

where $\forall i \in [1,n_j]$ $X_i = v(@_i, x, y, z)$, with n_j the number of different addresses in the trace ET_j, $x = \%_time_spent(@_i, ET_j)$ and $y = \%_accesses(@_i, ET_j)$.

In the traces, we have only the address of the executed instruction and information about the source code (*i.e.* instruction or line number in the source code). In order to have a higher level of granularity in the traces and facilitate the interpretation of the results, we use the symbol table of the executable to determine using well-known techniques [Bal69] the function to which this instruction belongs. In the feature vector, $z = e_i.function$ is the function name.
These features give first performance metrics at the level of granularity of the address, similar to those provided by `gprof` [FS08] at the level of granularity of functions.

Transforming each trace ET_j as a feature vector X_{p_j} first allows an important compression of the volume of data mining algorithms will have to process. This speeds-up further analysis. We also show in the following sections that the features of vector X_{p_j} are sufficient for discovering scalability hotspots.

In the next step, the features extracted are used for automatically discovering the hot accesses.

6.3.4 Feature-based clustering

This step allows to automatically group accesses using a *clustering* algorithm. Clustering is a data mining technique for organizing data elements into similarity groups, called *clusters* such that the data elements in the same cluster are similar to each other and data element in different clusters are very different from each other. A classical clustering algorithm is *k-means* [HW79]. The k-means algorithm is the best known partitional clustering algorithm. It is also widely used among all clustering algorithms due to its simplicity and efficiency. Given a set of data points and the required number of k clusters (k is specified by the user), this algorithm iteratively partitions the data into k clusters based on a distance function such as Euclidean distance. Each cluster has a cluster center called the **centroid**. The centroid, usually used to represent the cluster, is simply the mean of all the data points in the cluster, which gives the name to the algorithm, since there are k clusters. The clustering is the basis of our hot predicate definition. The clusters are obtained by applying *k-means* clustering algorithm on X_{p_j} with the number of clusters k as an input parameter. The result of *k-means* algorithm is the cluster feature vector (6.2) of the platform j which is the extension of the address-feature vector (6.1) with the cluster identifier assigned to each address performed by processors.

$$X_{p_j} clusterVector = [X'_1, X'_2, ..., X'_{n_j}] \tag{6.2}$$

Where $X'_i = v(@_i, x, y, z, C_{@_i})$, and $C_{@_i} \in [1, k]$ is the identifier of the cluster for address $@_i$ such as k is the maximum number of clusters.

In this work, we set the number of clusters $k = 2$ as we are interested to distinguish two types of accesses: *hot accesses* within a *hot cluster* and other accesses in the second cluster called *normal cluster*. Formally, the hot cluster is based on its centroid that satisfy the following definition:

Definition 10 (Hot Cluster)**.** *Let two centroids $c_{p_0}(x_{p_0}, y_{p_0})$ and $c_{p_1}(x_{p_1}, y_{p_1})$ of the platform having p processors such as $c_{p_0} \in C_0$ and $c_{p_1} \in C_1$ where C_0 and C_1 are two clusters and x is the percentage of the time spent %_time_spent and y is the percentage of accesses %_accesses. C_1 is a hot cluster if $c_{p_1} > c_{p_0}$ which true if $(x_{p_1} - x_{p_0}) + (y_{p_1} - y_{p_0}) > 0$, and normal cluster otherwise.*

By definition, we consider that the hot cluster will always have the label C_1. The hot predicate $isHot$ for an address $@_i$ simply consists in testing if $@_i$ is in the hot cluster or not. The set of hot accesses for platform P_j is thus $Hot_j = \{@_i \mid @_i \in [1, n_j] \wedge isHot(@_i, ET_j) = true\}$. Hot cluster gives, for each platform, the set of hot accesses of that cluster. Now when considering all platforms together, it becomes interesting to check if there are set of hot accesses that are found in several hot clusters, indicating that they are problematic for several platforms. Furthermore, for these sets of hot accesses found in several platforms, if their performance statistics decrease with the number of cores, it is a high indication that these hot accesses are scalability hotspots. We present in the next two sections these two last steps for discovering scalability hotspots.

6.3.5 Growth rate of hot cluster

The hot clusters are computed independently for each platform. The next step of the analysis, presented in this section, is to determine if there is a correlation between the

increase of number of cores in platforms and the statistics determining the hot clusters. This way we can determine if the hot clusters can help to determine a scalability problem.

Definition 11 (Performance loss)**.** *Given the hot clusters extracted from each platform, we say that an application loses its performance if the centroid of the hot cluster evolves across platform instances,* i.e. *both %_time_spent and %_accesses of the centroid grow with the number of cores.*

Such a performance loss is illustrated in Fig. 6.2. Discovering the impact of the

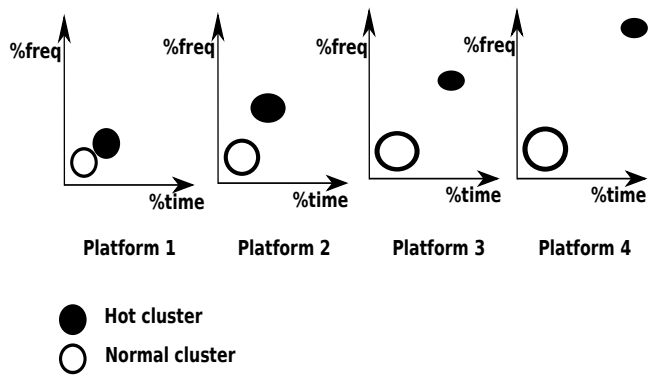


Figure 6.2: Hot cluster evolution

loss of performance of scalability hotspots is not an easy task. When the number of processors grows in the platform instances, the distance between the centroids of the two clusters grows too. Thus, we define a metric based on the euclidean distance between the centroids of the clusters. It measures the evolution of the distance in a multi-core platform relative to the distance in the one core platform. This principle is inspired from the speed-up metric. This metric is necessary to evaluate the impact of scalability hotspots on application performance. Therefore, this metric is called the *growth rate* metric.

Definition 12 (Growth rate)**.** *The Growth rate (Gr_p) refers to how much the distance between the two centroids $c_{p_1}(x_{p_1}, y_{p_1})$ and $c_{p_2}(x_{p_2}, y_{p_2})$ of the platform having p processors grows relative to the corresponding distance between the two centroids $c_1(x_1, y_1)$ and $c_2(x_2, y_2)$ of the platform having* 1 *processor.*

$$Gr_p = \frac{\sqrt{(x_{p_1} - x_{p_2})^2 + (y_{p_1} - y_{p_2})^2}}{\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}} \tag{6.3}$$

Where p is the is the number of processors in a platform. Gr_p is a value, typically between 1 and 100, estimating how many times the hot clusters grows over platforms, compared to how much the hot cluster containing the scalability bottlenecks patterns decrease the performance of the platform, *i.e.* how much effort is wasted in communication, synchronization or waiting state. Thus, the centroid of the cluster provides a scale for measuring the cluster evolution over different platforms.

The interest of computing the growth rate is not only to measure the impact of the scalability bottlenecks but also to aid the developer to make a decision if a given

program needs to be optimized. If the growth rate for two platforms with increasing number of cores is close to zero, it is likely that the program has no parallel scalability problem, or if one exists, it is not possible to detect it with our metrics. Otherwise, large positive values of growth rate indicate that the instructions contained in the hot clusters are likely to cause parallel scalability bottlenecks. The next section will focus on pinpointing the instructions of the hot clusters that the developer should investigate in priority.

6.3.6 Frequent scalability bottlenecks mining

It is vital to understand bottlenecks in platforms and their impact over the scalability for optimizing application performance and design future hardware. From the hot accesses, the developer wants to discover the frequent hot accesses common in each platform in order to focus on the parts of the code to improve. For this, we describe the frequent scalability hotspot mining method which discovers the set of hot accesses on multi-threaded application across multi-core platforms instances. A delicate problem is to find the frequent patterns that decrease the performance when the number of processors increases. We thus need to discover the frequent scalability hotspots, and do so using *frequent itemset mining algorithm* existing in *data mining* for mining instructions memory addresses that belong to the hot clusters through all platform instances. The extracted patterns are the most likely to be responsible of scalability issues. In the frequent itemset mining algorithm, the first input is a multiset of *transactions* $D = \{t_1, .., t_p\}$ defined over *items* in our case, the items are the hot accesses of all platforms, i.e. $\Sigma = \cup_{i \in [1,p]} Hot_j$, where $\forall t_i \in D \quad t_i \subseteq \Sigma$. To do this, we transform the set of hot accesses into a set of transactions D by merging all hot clusters in a same transaction table. Each hot cluster becomes a transaction, *i.e.* a set of hot accesses. The second input is a *minimum support threshold* $min_p \in [1, p]$ where p is the number of platform instances. Frequent itemset mining algorithms then extract all the *frequent hot accesses*, *i.e.* all the *hot accesses is* $\subseteq \Sigma$ that appear in more than min_p transactions of D. Once we have the transactions, we can use a state of the art frequent itemset mining algorithm. We use LCM [UKA04a], the most efficient one according to the FIMI contest [FIM04], to which we provide the minimum support threshold min_p and the transactions of hot accesses.

Example: Let a number of platform instances $p = 3$, a minimum support threshold $min_p = 2$ and set of hot accesses and the functions contained in their hot cluster respectively. We assume the following transactions:
Platform 1 itemset is `{0x01,0x02,0x03,0x9,0x10}` $\in$ Hot_1, where `{0x01,0x02,0x03}` $\in$ function f_1 and `{0x9,0x10}` $\in$ function f_2.
Platform 2 itemset is `{0x11,0x12,0x13,0x19,0x20}` $\in Hot_2$, where `{0x11,0x12,0x13}` $\in$ function f_3 and `{0x19,0x20}` $\in$ function f_4.
Platform 3 itemset is `{0x31,0x32,0x33,0x19,0x20}` $\in$ Hot_3, where `{0x31,0x32,0x33}` $\in$ function f_5 and `{0x19,0x20}` $\in$ function f_4.
The frequent pattern is thus `{0x19,0x20}` $\in$ function f_4, occurring in platforms 2 and 3.

The discovered frequent hot accesses with their frequencies pinpoint the scalability bottlenecks in the source code, and should be investigated by the developer.

6.4 Preliminary Results

In this section, we show the the preliminary results of this approach, taking into consideration the following characteristics:

The hardware platform is a shared memory multiprocessor that contains n MIPS32 processors such as $n = \{1, 4, 8, 16\}$, interfaced with one data cache and one instruction cache. It also contains one memory and others peripherals components: a timer, an interrupt controller, a frame buffer, a block device, a tty. More details of simulation environment in the chapter 7.

We perform two simulations on different platforms: platform 1, platform 2 and platform 3 which contain 1, 4, 8 and 16 processors, respectively. The software that runs on these platforms is a Matrix multiplication algorithm that includes a Pthread library. These platforms should exhibit different contention levels, and thus help us validate our approach of scalability bottlenecks pattern discovery.

Table 6.1: Scalability hotspots

Software	**Scalability hotspot pattern**	**% Occurrence**
Matrix Multiplication	[1000825c:10008268] cpu_mp_wait	75 %

We see that the frequent scalability hotspot contains synchronisation addresses in the *cpu_mp_wait* function, decreasing the performance by its evolution in (3/4) 75 % platform instance i.e. all parallel platforms.

More details, discussions and experiments about this approach in the chapter 7.

6.5 Comparison of Scalability Bottlenecks Detection Methodologies

Table. 6.2 shows a comparison of existing methodologies in literature of scalability bottlenecks detection. The methodologies are described in details in related work chapter 3.2.2. The comparison is performed in terms of the platform on which the methodologies is performed, the tools used for scalability bottlenecks discovery and what results the tool shows the user. We can see that our methodology ([LTP14]) is:

- The only one that gives a precision of results to user by quantifying and pinpointing the bottlenecks using data mining on executions traces among the existing methods.
- The first approach on MPSoC.
- Not using estimation mechanisms or instrumenting source code for bottlenecks extraction.

Paper	Platform	Tools	Data	Results
Scal-Tool [SLT99]	DSM	Based on estimation	X	quantifying + pinpointing
Cycle stacks [HCC+11]	SMP	Statistical tool (PCA) clustering	cycle stack	quantifying
[CLZ+11]	CMP	Statistical tool	Performance counters	Statistical
[GKS12]	CMP	Statistical tool	Performance counters	Statistical
[JSMP12]	ACMP	Instrumentation of source code	X	Monitoring of Bottlenecks
Speedup stacks [EDBE12]	CMP	Based on estimation	X	quantifying
Our approach [LTP14]	MPSoC	Frequent itemsets and Clustering	Traces	quantifying + pinpointing

Table 6.2: Comparison of Scalability Bottlenecks Detection Methodologies

6.6 Conclusion

In this chapter, we have proposed a new approach for discovering the scalability hotspots that reduce the parallel performance in MPSoC platforms running embedded software. The proposed approach uses data mining techniques on simulation traces, thus offering several advantages. It can profile the parallel embedded software in one (intra platform) or multiple platforms (inter platforms), and it is applicable on embedded systems as well as on parallel machines as long as detailed information on the memory accesses are available. It can be performed on homogeneous architectures with different types of processors in order to select the most appropriate processors type running a given multi-threaded program. The approach only requires one parameter, and then is completely automatic. Furthermore, the pinpointing and quantifying the bottlenecks is already a nice service to the programmer. It can thus be very helpful in reducing the amount of work a programmer has to do.

Chapter 7: Experimentations and Results

No amount of experimentation can ever prove me right; a single experiment can prove me wrong.

Albert Einstein

Contents

7.1 Parallel embedded software **78**
- 7.1.1 Ocean 78
- 7.1.2 FFT 78
- 7.1.3 LU 78
- 7.1.4 RADIX 79
- 7.1.5 Mandelbrot 79
- 7.1.6 MJPEG 79
- 7.1.7 Matrix Multiplication 79

7.2 Simulation environment and Hardware architecture **81**
- 7.2.1 Simulator 81
- 7.2.2 Operating System 81
- 7.2.3 Hardware Architecture 81

7.3 Experiments Set I: Contention discovery **82**
- 7.3.1 Experiments I.1 82
- 7.3.2 Experiment I.2 85
- 7.3.3 Results analysis 85
- 7.3.4 Experiments II 86
- 7.3.5 Results Set II 86
- 7.3.6 Discussion 89

7.4 Experiments Set II: Scalability bottlenecks discovery **90**
- 7.4.1 Results analysis 90
- 7.4.2 Discussion 95

7.5 Conclusion **96**

After having described our approaches to automatically discover contention (Chapter 5) and scalability issues (Chapter 6), we now evaluate how effective theses approaches can be to help profiling real word applications on large simulated MPSoC platforms. We first describe the context of the experiments, with the applications considered and the hardware platforms simulated. We then show the patterns extracted by our approaches, and how they can help to improve application performances.

7.1 Parallel embedded software

In order to evaluate our approaches, we selected seven different parallel applications that are well known real world benchmarks. Among them applications from Stanford ParalleL Applications for SHared memory SPLASH-2 [CME+95] parallel benchmarks suite. [CME+95] is a set of parallel applications for use in the design and evaluation of distributed and shared-memory multiprocessing machines. The SPLASH-2 applications are: Ocean, Fast Fourier Transform (FFT) LU and RADIX. Another applications such as MJPEG video decoding application, Mandelbrot fractal application and Matrix Multiplication algorithm.

7.1.1 Ocean

Ocean is a program that simulates large-scale ocean movements. The ocean surface is represented with a grid mesh. Ocean program partitions the grids that represent ocean into square-like subgrids rather than groups of columns to improve the communication to computation ratio. The grids are conceptually represented as 4-D arrays, with all subgrids allocated contiguously and locally in the CPU that own them. Ocean program uses a red-black Gauss-Seidel multigrid equation solver [Bra77]. See [WSH93] for more details. Application domain can be video games.

7.1.2 FFT

The FFT benchmark performs a complex 1-D FFT transform. It uses the radix $\sqrt{n}$ six-step FFT algorithm, which is optimized to minimize interprocessor communication [Bai90]. The data set consists of the n complex data points to be transformed, and another n complex data points referred to as the roots of unity. Both sets of data are organized as $\sqrt{n} \times \sqrt{n}$ matrices partitioned so that every processor is assigned a contiguous set of rows which are allocated in its cache. The FFT algorithm achieves its computational efficiency through a divide and conquer strategy. Application domains of FFT can be image analysis, image filtering, image reconstruction, image compression , and sound processing.

7.1.3 LU

The LU benchmark factors a dense matrix into the product of an upper and lower triangular matrix. The matrix is divided into blocks, which are assigned to processors using 2-D scatter decomposition to reduce communication. Each block is allocated locally on the processor that owns it. The dense $n \times n$ matrix is divided into an $N \times N$

array of $B \times B$ blocks ($n = N * B$). The application domains where such decomposition solution is useful, includes text mining [GGM04].

7.1.4 RADIX

RADIX is a parallel version of radix sort [ZB91], it comprises three phases: build local histograms, build a global histogram, and permute keys. Execution of the algorithm proceeds iteratively, analyzing one digit of an integer per iteration to populate the histograms and then permuting the keys accordingly, starting with the least significant digit.

More details about Splash benchmark suite are given in [CME+95]. This set of applications are designed to evaluate the shared and distributed memory parallel machines. Splash applications are parallel compute-intensive applications and are always a reference in domain. In the following, we describe another benchmarks:

7.1.5 Mandelbrot

The Mandelbrot set is a fractal set defined in the complex plane by the following equation: $z_n = z_{n-1}^2 + z_0$ The Mandelbrot set is a classic example of task parallelism, where the computation is viewed as a set of tasks operating on independent data streams. In this case, generating a pixel of the Mandelbrot image is the computation, and the pixel position is the data stream. The applications domains where such Mandelbrot solution are useful, which include geology, biology, finance, image compression.

7.1.6 MJPEG

The Motion-JPEG video format is composed of succession of JPEG still pictures. It is used by several digital cameras and camcorders to store video clips of a relatively small size. With Motion-JPEG, each frame of video is captured separately and compressed using the JPEG algorithm.

Parallel Motion-JPEG video decoding application is described by a set of communicating tasks through a queue. It is composed of three stages: **fetch**, **compute** and **dispatch** as shown in Figure 7.1. The first stage **fetch** reads a data stream from a file accessed through hardware module TG. It decompress the data and send them in the next stage. The second stage **compute** performs image decoding and is parallelized in several tasks. For example in the figure there are 3 decoder `C0,C1,C2`.

The frames are displayed using a frame buffer. The Frame buffer is a video output device which displays images from a memory buffer containing a complete frame of data.

The interest of this application is that all tasks have a specific function and are working on a data stream. Also the memory accesses by processors vary depending on the task being performed over time, this is typical of the work of a real MPSoC.

7.1.7 Matrix Multiplication

The definition of the matrix multiplication operation is very simple, all of which simplifies its understanding. Given a matrix $A^{(m \times r)}$ of m rows and r columns, and a matrix

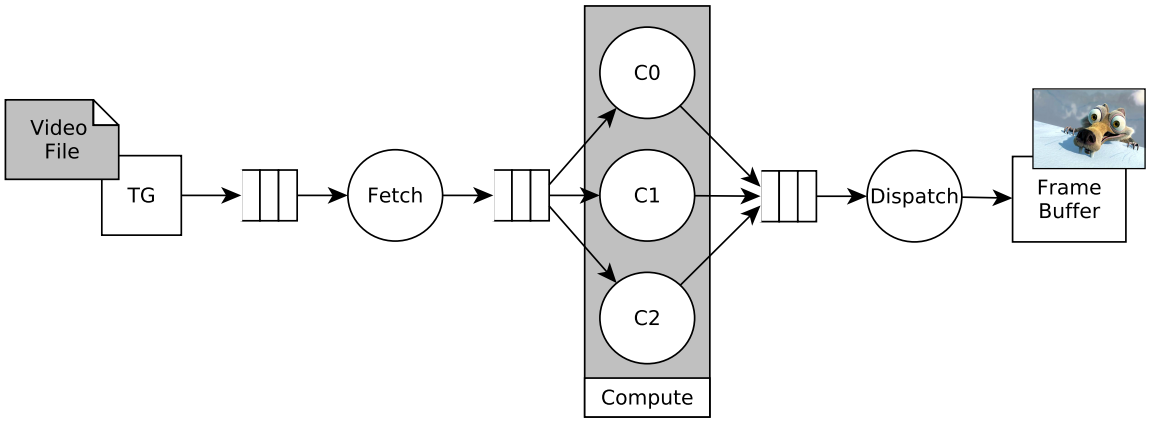


Figure 7.1: MJPEG Application described with communicating tasks

$B^{(r \times n)}$ of r rows and n columns, matrix C resulting from the multiplication operation of A and B matrices, $C = A \times B$. In the parallel Matrix Multiplication, the matrices are divided into sub-blocks which are mapped to the processors in order to perform the multiplication operation. It exists several applications domains of this application such as image processing / recognition.

Table 7.1 shows the problem domain of each application, language, library used to implement the application and the particular methods or algorithms that are dominant in it. Four applications (Ocean, FFT and LU, matrix multiplication) are scientific computations and one application (MJPEG) for the video decoding. We believe that the current set of applications provides a reasonably wide variety of applications.

Table 7.1: Characteristics of applications

Application	Problem Domain	Parallelism Mode	Language	Library	Problem Size
Ocean	Multi-grid ocean simulation	Data parallelism	C	PThreads	130 × 130 grid size
FFT	Signal Processing	Data parallelism			256 data points
LU	Blocked dense LU decomposition	Data parallelism			256 × 256 matrix 8 × 8 blocks
RADIX	Sorting algorithms	Data parallelism			2048 data to sort
Mandelbrot	fractal shape	Task parallelism			256 × 256 pixels
MJPEG	Video decoding	Task parallelism			50 frames, 255 × 144 pixels
Matrix Multiplication	Parallel Matrix Multiplication	Data parallelism			1024 × 1024 matrix

All benchmarks were compiled using GCC version with the -O3 optimization level.

7.2 Simulation environment and Hardware architecture

7.2.1 Simulator

The MPSoC simulator used for experimentations is SystemCASS described in 4.1.1.

7.2.2 Operating System

We use DNA-OS [GP09] as operating system for our experiments. DNA-OS is a kernel-mode lightweight operating system for Multiprocessor System on a Chip. It is build on top of a thin HAL to ease porting on new platforms and processor architecture. It targets the following architectures: ARM, MIPS, Micro Blaze, SparcV8, NiOS. It adapts to all types of architectures: SMP (Symmetric MultiProcessing) DS (Distributed Scheduling). DNA-OS allows to write parallel program using the following libraries: POSIX Threads (PThreads) and *newlibc* with multiprocessor support.

7.2.3 Hardware Architecture

Our simulation platforms were built using the SoCLib components library [Soc]. Most components are described with an accuracy level of the clock cycle and the data bit CABA. Figure 7.2 shows and describes the architecture used of experiments. The hardware platform is a shared memory multiprocessor that contains n MIPS32 processors, interfaced with one data cache and one instruction cache. It also contains one memory and the peripherals components required to perform standard I/O operations : a timer, an interrupt controller, a frame buffer, a block device, a terminal sink (TTY). These components are interconnected with a generic network-on-chip.

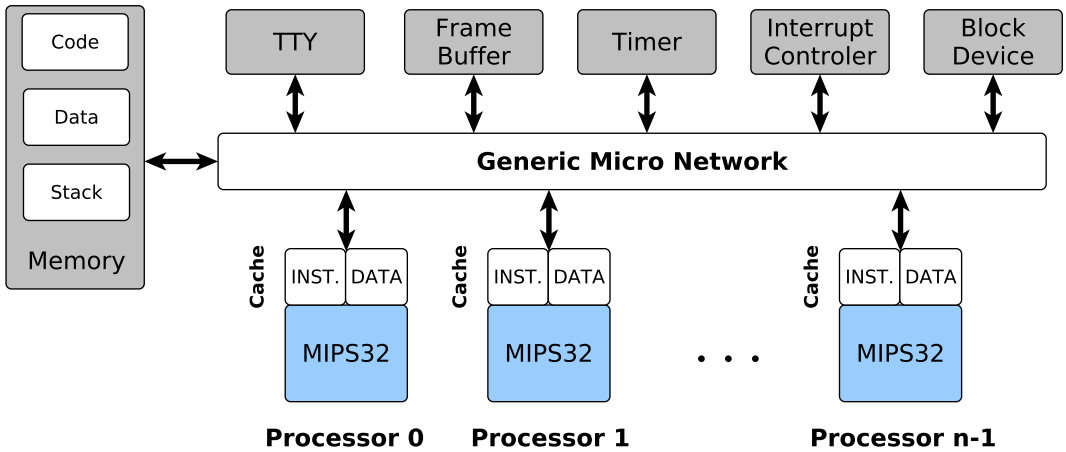


Figure 7.2: Description of the used architecture

The table 7.2 summarizes the characteristics of platforms.

Table 7.2: Summary of the simulation characteristics

Applications	Ocean, FFT, LU, RADIX, MJPEG, Mandelbrot, Matrix Multiplication
Simulator	SystemCASS
Operating System	DNA-OS
Architecture	SMP
Processor model	MIPS32
Number of CPUs	$n = \{1, 2, 4, 8, 16\}$
Number of memory banks	3 (code, data, stack)
Memory size	32MB
Data cache size	4KB
Instruction cache size	4KB
Size of a cache line	32 bytes

7.3 Experiments Set I: Contention discovery

In this section, we describe the experiments and results of two approaches to automatically discover contention presented in (Chapter 5). The first experiments 7.3.1 present results of the first approach and the second experiments 7.3.4 present the results of the second approach.

7.3.1 Experiments I.1

For the first experiments, we take a particular case of the hardware architecture described in Figure 7.2 which is 4x4 mesh NoC with wormhole routing protocol and a XY routing algorithm as shown in Fig. 7.3. Every node contains: MIPS processor with one data cache (with write-through invalidate memory coherency strategy) and one instruction cache, one shared memory or local memory and others peripherals components: a timer, a frame buffer and a tty. For our experiment, three application instances are mapped to this platform: two Motion-JPEG decoders and one Mandelbrot fractal computation.

The Fig. 7.3 shows the mapping of these applications on the MPSoC. Four nodes are allocated for the Motion-JPEG applications and eight nodes are allocated for Mandelbrot. The data memory is located on the node 5 and the code of the applications and operating system is located on the node 10.

Trace preprocessing

The raw traces, as output by the simulator, contain for each traces event informations that are not useful for our analysis, so we dispose of it.

There are no function names but only the `PC` address of the executed instruction: using the symbols table of the executable, we determine using well known techniques [Bal69] (See Appendix A) the function to which an instruction belongs.

After the pre-processing step of the traces files, we cut the trace files into windows with a period of 2500 clock cycles and we run the frequent pattern mining step with $min_sup_c = 2$ CPUs and min_sup_w corresponding to 60 % of windows. Some frequent patterns found are presented in Table 7.3. Here, we have found the combination of itemsets with different CPU_ID: from CPU_8 to CPU_15 and having the same access in

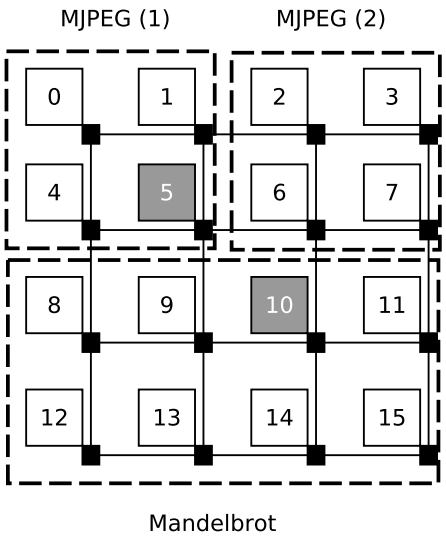


Figure 7.3: Simulated platform (1)

the page P_3090, function fpadd_part and node id_10. For example, the first discovered pattern shown in Table 7.3 is: {{ CPU_8 INST P_3090 fpadd_part id_10 }, { CPU_9 INST P_3090 fpadd_part id_10 }, ..., { CPU_15 INST P_3090 fpadd_part id_10 }}, for ease of writing: CPU[8,15] INST P_3090 fpadd_parts id_10.

For the nodes that run the Mandelbrot application, the most accessed instructions are located in the following pages of node 10: P_3091, P_3090, P_3088.

The pages P_3091 and P_3090 contain the code of the floating point helper functions (as our processors do not support floating point operations directly, the compiler has automatically instantiated *ad-hoc* functions to perform the operations) and the page P_3088 contains the code of the Mandelbrot application. Now, for refining and understanding the hotspots shown by the frequent patterns, we extract the access times to the memory page for each frequent pattern and we apply the equation (1). The Figures 7.4a and 7.4b show respectively the access rate of the nodes to the pages P_3090 and resp. P_3088 during the first 200 windows.

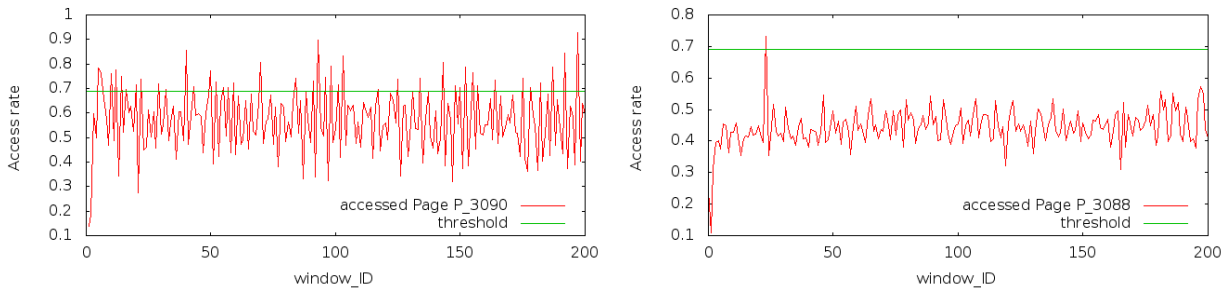
We see that in the page P_3090 some patterns exceed the $\ln 2$ threshold which we interpret as a contention of memory access, and the same result holds for the page P_3091. Unlike the previous two memory pages, the page P_3088 contains very little memory contention. A quick analysis of the patterns allows to exhibit a *cache trashing phenomenon*. In Mandelbrot both floating additions and multiplications must be performed, but the small instruction cache of a core cannot handle both these helper functions. Thus these functions evict each other in turn from the instruction cache, introducing contention on the text memory containing the code of the functions. The latency increases and the performance decreases.

The Fig.7.5 shows a representation of the pattern related to floating point operations realized as functions that create contention detected by our experiments for different architectures.

The software functions "**__unpack_d, __pack_d, _fpadd_parts**" and "**__muldf3**" are

	Frequent Pattern	**Support**
Mandelbrot	CPU[8,15] INST P_3090 fpadd_parts id_10	96%
	CPU[8,15] INST P_3090 __muldf3 id_10	62%
	CPU[8,15] INST P_3088 mandelbrot id_10	95%
	CPU[8,15] INST P_3091 __unpack_d id_10	87%
	CPU[8,15] INST P_3090 __pack_d id_10	80%
MJPEG	CPU_0, CPU_2 INST P_3085 fetch_process id_10	67%
	CPU_0, CPU_2 INST P_3073 fdaccess_read id_10	67%

Table 7.3: Frequent patterns



(a) Access rate of the nodes to the page P_3090

(b) Access rate of the nodes to the page P_3088

Figure 7.4: Access rate of the nodes to the pages P_3090 and P_3088 running Mandelbrot application

all functions associated with the manipulation of floating-point numbers and contained in **fp-bit** and **libgcc** libraries of GCC complier, respectively. If we add up the time taken for all of these functions to execute, we discover that a massive 22,27% of each processor's time in each platform is due to the floating-point calculations performed by these function calls. The conversion processes are done using the **__pack_d** and **__unpack_d** functions that compress or expand data into the target register format. **_fpadd_parts** function performs the addition operation of floating-point. **__muldf3** function performs the multiplication operation of floating-point.

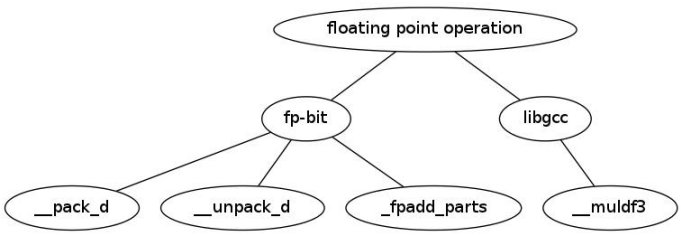


Figure 7.5: Results Representation

Now, we see that floating-point operations consume a lot of time and create a contention. In the next experiment, we integrate floating point coprocessor in MIPS32 processors.

7.3.2 Experiment I.2

In order to tackle the contention pattern which is the floating points problem discovered in the first experiments, the second hardware platform showed in the figure 7.2 is a shared memory multiprocessor (SMP) that contains four MIPS32 processors that include a **floating-point unit**. The software that runs on this platform is the Contiguous Ocean application (that simulate large scale ocean movements) of the SPLASH-2 parallel benchmark suite. We chose this benchmark because it is characterised by the number of floating point operations used[CME+95]. The following results show that the floating points patterns does not appear again and the problem is resolved but another pattens are discovered.

7.3.3 Results analysis

To determine the frequent memory contention during the execution of the application, we use the same parameters as the first experiment such as the decomposition into windows and the minimum threshold in the pattern discovery method. We found that the CPUs 1, 2 and 3 have the same behavior during each time window. Thanks to the integration of a floating point unit in MIPS32 processors, we did not found a frequent pattern that is symptomatic of instruction cache trashing like in the first experiment.

Table 7.4: Frequent patterns

Frequent Pattern	Execution time
CPU[0,3] INST slave1 0x1000f9f4 CPU[0,3] INST slave1 0x1000f914 CPU[0,3] INST slave1 0x1000fab4 CPU[0,3] INST slave1 0x1000f704 CPU[0,3] INST slave1 0x1000f70c CPU[0,3] INST slave1 0x1000fca0	38,39 %

In the Splash benchmarks, the function **slave1** is the top level function replicated on the pool of processors. It does the initialization on its data set and calls the **slave2** function that does the actual computation. Our data mining process did not find any abnormal latency behavior in the computation themselves, but strangely enough, it did in the initialization function. Indeed, we noticed that recurrent patterns with large latencies accounted for almost 40% of the execution time of this function, and were indeed due to array initializations (see Table 7.4). These latencies are due to successive *stores* of constants in loops, which is unexpected as the cache contains a write buffer that should not stall the processor. Figure 7.6 shows successive, frequent and periodic *stores* whose program counter address is `0x1000f914`. We see that the *stores* operations to this address are periodic with regular periods between 23 and 80 cycles, and the average latency of a *store* is 26.01 cycles. This phenomenon is the same for each CPUs and for each patterns discovered, and is due to the fact that all processors are doing the initialization at the same time. In fact, a synchronization barrier is necessary to make sure that the shared data structure is allocated before working on it, so all processor try to access the memory concurrently when the barrier is released, leading to contention.

Improvement: By doubling the size of the write buffer (from 8 to 16 slots, which has

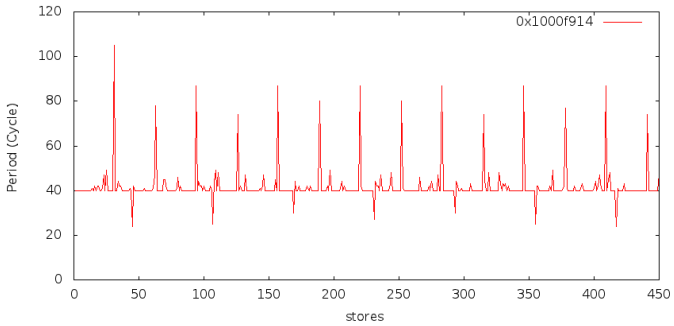


Figure 7.6: Period between the successive *stores* of the address `0x1000f914`

clearly a hardware cost), the patterns were not anymore considered as abnormal by the tool. We see in figure 7.7 that the period between successive *stores* has doubled (because the execution between two iterations is twice as fast) compared with the previous configuration, and the average latency decreases by 32% to 17,75 cycles.

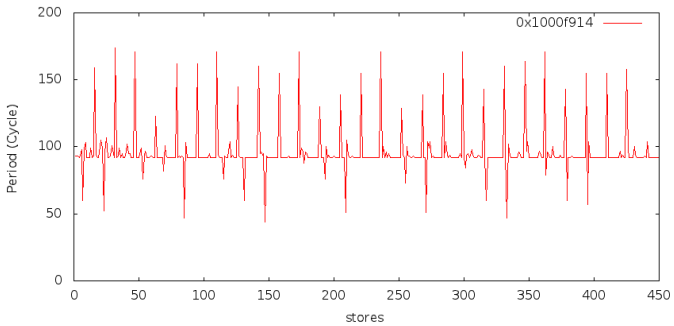


Figure 7.7: New period between the successive *stores* of the address 0x1000f914

This is experiments of how our profiling tools can help to make hardware level decision when the chip is still being prototyped in a simulator.

7.3.4 Experiments II

The second experiments 7.3.4 present the results of the second approach of contention discovery on parallel Motion-JPEG application.

7.3.5 Results Set II

The first criterion taken into consideration when the performances of the parallel systems are analysed is the speed up used to express how many times a parallel program works faster than a sequential one. For this, we performed experiments in order to evaluate the speed up. The speed up of the video decoding application results are 3.3 and 4.5 corresponding for 4, and 8 cores in a platform, respectively. It is just acceptable for 4 cores and bad for 8 cores, hence the video decoding application considered does not scale well with the number of cores. We verified that this bad scalability is not due to a lack of work or a load unbalance issue. Having eliminated these reasons for

lack of parallel scalability, the remaining reason is likely to be contention that slows down memory accesses and thus prevents the application to reach the desired speed up. It is thus justified to apply our approach in order to automatically detect the parts of the trace where contention occurs, and to understand through contention patterns the reasons for this contention. First, we explain what preprocessing was necessary on trace in order to apply our approach.

To be able to use the pattern mining algorithm, we discretized continuous numeric attributes into bins of numeric intervals in order to regroup events exhibiting similar values. As an example, the memory access latencies are discretized by bins of 10 from 0 to 250 *i.e.* all latencies between 0 and 10 are represented by the bin lat_0_10. These information will help the pattern mining algorithm to discover meaniningful patterns.

The first step of our approach compute the high latency values for each trace (trace for 1, 4 and 8 CPUs). The high latency thresholds in different platforms using 1, 4 and 8 CPUs are 9.65, 12 and 15 respectively. However, as the number of CPUs increases the number of the high latency thresholds also increases.

The second step of our approach exploits these thresholds in order to compute the windows identifying the parts of the traces exhibiting contention. We set the window duration to $\omega = 200$ cycles.

Over all the traces, the number of contention windows found and the percentage of the trace they cover is summarized in Table 7.5.

Table 7.5: Contention windows

Platform	Nb of windows	Coverage of trace
1 CPU	86 843	1.20%
4 CPUs	925 951	16.97%
8 CPUs	1 686 785	36.79%

As expected, there are few windows with high latencies for the platform with 1 CPU. However, as the number of CPUs increases the number of these windows also increases, and it covers a significant fraction of the execution time. This is in line with the speed up results, and confirms that contention is a problem for our experiments with 4 and 8 CPUs.

In order to better understand the reasons for contention, we plot in Figure 7.8 the frequency of apparition of instructions in the contention windows for 4 CPUs platform. More precisely, x-axis is the program counter identifying instructions, and y-axis is the frequency of apparition of each instruction over all contention windows. As instructions can be related to functions, this figure indicates which functions are the most responsible for contention. Three highly frequent groups arise that we identified with the corresponding function names: there are *memset*, *idct* and *memcpy* functions. The figure shows that contention is mostly due to *idct* and *memcpy*. However, this figure does not indicate the interactions between *idct* and *memcpy* in contention windows.

Figure 7.9 shows the percentage of CPU time spent in the functions *idct*, *memcpy* and *memset*. It can be noticed that *idct* is more often executed on *CPU_2* and *CPU_3* than on *CPU_0* and *CPU_1*. This is the opposite for *memcpy*. However all CPUs spent at least 10% of their time in each function, so there is no pattern associating a CPU with a function. Depending on the architecture, this information could be of interest to the

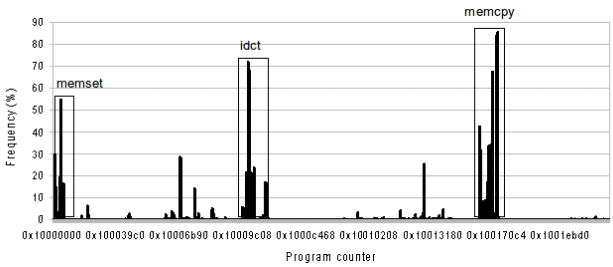


Figure 7.8: Memory access frequency

application developer. For example in cases each CPU has its own memory bank, he/she may want to bind some threads to given CPUs in order to keep good spatial locality.

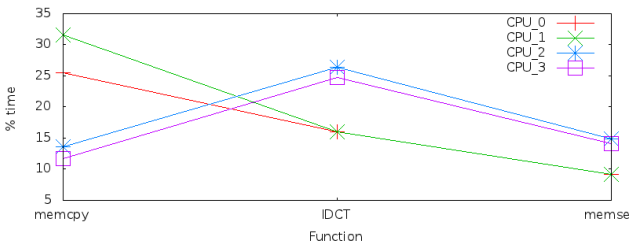


Figure 7.9: Run time of *memcpy*, *idct*, *memset* in parallel application

In order to understand such interactions, we apply the pattern mining algorithm to the contention windows converted to transactions with a minimum support threshold of $\varepsilon = 65\%$: we are interested in interactions between functions, memory locations and CPUs that occur in more than 65% of contention windows, *i.e.* very frequently parts of the traces exhibiting potential contention. We focus on platform with 4 and 8 processors, which exhibit high levels of contention. The most interesting contention patterns discovered for these simulation platforms are presented in Table 7.6.

Table 7.6: Frequent Patterns

Platform	Frequent Pattern	Support
4 CPUs	CPU[0,3] [0x10009ee4, 0x10009f78] idct [0x10016b50, 0x10016f2c] memcpy lat_10_20 lat_20_30	72 %
8 CPUs	CPU[0,7] [0x10009b10, 0x1000a224] idct [0x10016ab0, 0x10016e8c] memcpy lat_10_20 lat_20_30	88 %

The pattern of the platform using 4 CPUs shows a concurrent memory access pattern that creates a contention implying all 4 CPUs and occurring in 72 % of contention windows. This pattern shows a frequent interaction between the functions *idct* and *memcpy*, and more specifically between the loops of *idct* located in address interval [0x10009ee4, 0x10009f78] and the loops of *memcpy* located in address interval [0x10016b50, 0x10016f2c]. The pattern also shows that the usual latencies around these

interactions are between 10 and 30 cycles (lat_10_20, lat_20_30). Having in mind that the high latency threshold is $Q_3 = 12$ for the 4 CPUs trace, this corresponds well to contention latencies.

The pattern for the 8 CPUs platforms is the same as on the 4 CPUs platform, with different addresses due to a different executable. However these addresses correspond to the same assembler instructions than previously: this enforces the importance of the *idct*/*memcpy* interaction. In the 8 CPUs platform the pattern has an even higher support of 88 %, whereas there are more contention windows in this case: this pattern is clearly the main responsible for most of the contention and thus the lack of scalability when the number of cores increase. This pattern thus helps the application developer to know that the *idct* function, which performs the inverse discrete cosine transformation, has negative interactions with *memcpy*, a function for copying data from one address to another. It even pinpoints the specific assembler instructions of both functions that are the most impacted: the developer, which is more likely to work on *idct* than *memcpy*, will know immediately which loop of *idct* he/she has to work on.

More experiments about the rest of applications: FFT, RADIX, LU, Mandelbrot are given in Appendix B.

7.3.6 Discussion

Our approach is more accurate than the existing current works [TdRC+10, AZXC11, RL10] because it identifies the frequent concurrent memory access patterns, extracts from the execution traces the patterns that create a contention and generates a compact and readable output that can be analyzed by the software developers. Our tool helps the developers to highlight concurrent memory access patterns and the impact on the parallel scalability. In the experimentations, we saw, firstly, the high frequency interactions between *idct* and *memcpy* functions in a parallel platform. These interactions lead to contention in different platforms. However, it's difficult to find such interactions with the existing profiling tools [FS08, Rei05]. Secondly, *memcpy* is having a major impact on the parallel scalability of a video decoding application. Thus, the developer can optimize the program with the following possible solutions:

- Using software (or hardware) based prefetching caches or non-blocking caches techniques. These effective techniques allow to decrease memory access latency [CB92].
- In [WDV06], a dedicated hardware accelerator was proposed that works in conjunction with caches found next to modern-day microprocessors, to speed up the commonly utilized *memcpy* operation.
- The *memcpy* function can be put between a lock/unlock pair to serialise the memory access and ensures that when one CPU is executing *memcpy*, no contention will be created. As such, serialization can be detrimental to performance, it can be activated only when *idct* is executing the loops identified in the previous frequent pattern and deactivated the rest of the time.
- Refactor the communication scheme of the whole application.

7.4 Experiments Set II: Scalability bottlenecks discovery

In this section, we describe the experiments and results of the approach to automatically discover scalability bottlenecks presented in (Chapter 6).

7.4.1 Results analysis

After having performed different speed up measures of applications as shown in Chapter 2. In this section, we show the different bottlenecks patterns found in all experimented applications. The Tables 7.7, 7.8 and Fig. 7.12 show the scalability hotspots, the growth rate evolution and the coverage of clusters in each multi-threaded application across platform instances, respectively. They give detailed information about the scalability bottlenecks in different critical zones of the multi-threaded applications. The scalability hotspots in multi-threaded applications that undergo evolution across scalable platforms are discovered and the hot accesses are distinguished from other addresses. The patterns are highlighted by the address range and their function name. Table 7.7 shows the frequent hot patterns discovered having a minimum support threshold $min_p = 25\%$. The frequent patterns discovered from hot clusters in each platform P_i running the same multi-threaded applications highlight the common critical zones.

Matrix Multiplication Algorithm

In the first application which is matrix multiplication application, we see that the scalability hotspot contains synchronisation addresses in the *cpu_mp_wait* function, decreasing the performance by its evolution in each platform instance as shown in Fig. 7.11 and Fig. 7.12. The pattern represents a sequence of instructions in a loop as shown in Fig. 7.10. *cpu_mp_wait* procedure spins until the value of the variable `sync` returns 0. This variable must exist and must be initialized to 1 beforehand.

Figure 7.10: Scalability hotspot in assembly code for the matrix multiplication application.

```
10008250:    <cpu_mp_wait>
                        ...
1000825c:    sync
10008260:    lw v0,-17120(v1)
10008264:    bnez v0,10008260 <cpu_mp_wait+0x10>
10008268:    nop
```

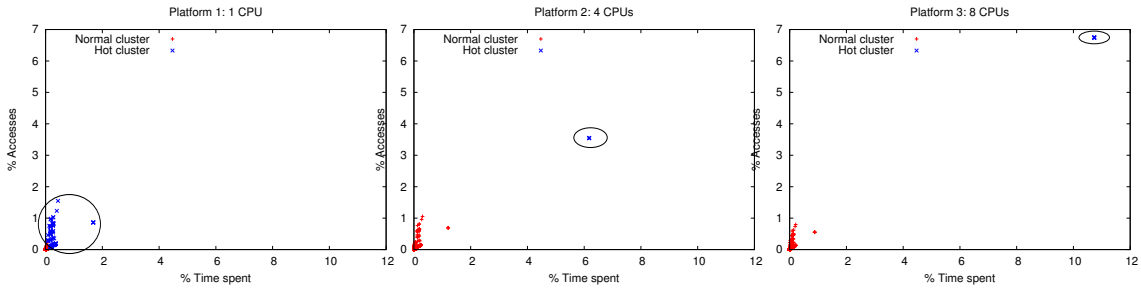
The particular loop means that the processors are in a 'wait' state waiting to be 'notified' of work to be done. Therefore, it induces load imbalance of the tasks in cores, with a high impact on scalability as shown by the growth rate in Fig. 7.12. In order to see in detail the impact of the evolution of the hot cluster in a platform, Fig. 7.11 shows the evolution of hot accesses in the hot clusters in multi-threading matrix multiplication according the platforms, each point plotted represent the coordinates (x, y) of an address where x is the percentage of the time spent and y is the percentage of access.

Improvement of Matrix Multiplication Algorithm

The found synchronisation pattern shows that there was a synchronization barrier `pthread_barrier_wait()` in addition to a `pthread_join()` in the source code

Table 7.7: Scalability hotspots

Software	Scalability hotspot pattern	% Occurrence
Matrix Multiplication	[1000825c:10008268] cpu_mp_wait	75 %
Ocean	[100235b0:100235dc] lock_acquire	75 %
Motion-JPEG	[100023b4:100023d0] soclib_fb_write	75 %
	[100171a4:100171d4] __malloc_lock [100171e0:10017200] __malloc_unlock	50 %
FFT	10044650 (data address)	75 %
	[1000b0a0:1000b0a8] cpu_mp_wait	
RADIX	[1000b940:1000b948] cpu_mp_wait 10027190 : cpu_mp_synchro [100147b8:10014830] lock_acquire	75 %
LU	[10013c60:10013c80] __muldf3 [100147b8:10014830] __unpack_d 100071ec lrand48 [0x10015f9c:0x10015fdc] InitA	75 %
	[10013c60:10013c80] __muldf3 [100147b8:10014830] __unpack_d	100 %
Mandelbrot	[10011c60:10011c94] __fpadd_parts [100238c0:100239d0] __ieee754_sqrt [10008c40:10008c48] cpu_mp_wait [10012cb0:10012d64] __unpack_d [10012a50:10012b58] __pack_d [10012158:1001218c] __muldf3	75 %
	[100238c0:100239d0] __ieee754_sqrt [10012cb0:10012d64] __unpack_d __pack_d [10012158:1001218c]	100 %



(a) Platform 1 : 1 CPU (b) Platform 2 : 4 CPUs (c) Platform 3 : 8 CPUs

Figure 7.11: Visualizing the evolution of hot clusters in each multi-threaded matrix multiplication application according to platform instances

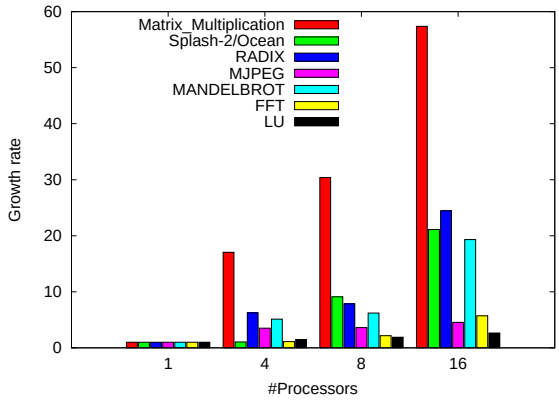


Figure 7.12: Growth rate evolution over platform instances running five multi-threaded applications

and which was not in the right place. A barrier for a group of threads in the source code means any thread must stop at this point and cannot proceed until all other threads reach this barrier. This pattern increases the growth rate in each platform instance.

By eliminating this problem, it allows us to gain in performance 3.79%, 3.91% and 4.32% of runtime in the platform with 4, 8 and 16 CPUs, respectively.

LU

In LU application, we discovered two patterns:

1. The frequent pattern in all platforms is the software functions *__unpack_d* and *__muldf3* are all functions associated with the manipulation of floating-point numbers and contained in *fp-bit* and *libgcc* libraries of GCC complier, respectively. *__unpack_d* functions that compress or expand data into the target register format. *__muldf3* function performs the multiplication operation of floating-point.

2. In addition to the discovered contention pattern in Figure 7.13, the second pattern

Figure 7.13: InitA function of LU application

```
void InitA(double *rhs)
{
...
for (j=0; j<n; j++) {
for (i=0; i<n; i++) {
...
a[ii][jj] = ((double) lrand48())/MAXRAND;//①// executed by all threads
}}
for (j=0; j<n; j++){
for (i=0; i<n; i++){
...
rhs[i] += a[ii][jj];//②// executed by all threads
}}}
```

is the multiple simultaneous accesses at two instructions ①, ② of *InitA* function in LU application. The first instruction ① performs multiple initialization of a matrix using *lrand48*[1] by all threads, and the second instruction ② performs the same operation on the same array index, thus, different results on different executions are obtained. This pattern is considered as a bug by developers. In order to tackle these two patterns and improve this application, we have two solutions: protecting these instructions with a locks, or partitioning a matrix into sub-matrix affected to each thread. We use the second solution because each thread performs a local operations in a given sub-matrix. An improvement in given in Figure 7.14.

Performing this minor improvement, the performance improvement of the new LU application is 3.03%, 3.41%, 4.88% in the platform with 4, 8 and 16 CPUs, respectively.

Figure 7.14: Improvement of InitA function of LU application

```
void InitA(int *nb, double *rhs)
{
...
int thread_id = ( int ) nb;
/*
Determine the portion to be computed by the thread_id.
*/
int startrow = thread_id * n / P;  // P: number of processor.
int endrow = (thread_id+1) * n / P ; // n: The size of the matrix.
for (j=startrow; j<endrow; j++) {
for (i=0; i<n; i++) {
...
a[ii][jj] = ((double) lrand48())/MAXRAND;
}}
for (j=startrow; j<endrow; j++){
for (i=0; i<n; i++){
...
rhs[i] += a[ii][jj];
}}}
```

FFT The same pattern as Matrix Multiplication application is found in **FFT** application. In FFT application, another frequent pattern is the shared data address

[1] *lrand48* is a function in C *stdlib* library. The *lrand48* function uses 48-bit arithmetic to generate non-negative long integers uniformly distributed pseudo-random values.

`0x10044650` protected by load-link and store-conditional (LL/SC) which are a pair of instructions used in multi-threading to achieve synchronization, this pair cause contention during simultaneous requests on address. In order to improve this application, the user can improve by replicating the execution of the section code on processors [LCZ01].

Ocean

In Splash's Ocean, like the first experiment the growth rate increases with the number of cores of the platform. The discovered scalability hotspot contains the address range grouped into *lock_acquire* function. The *lock_acquire* function acquires access to a specific lock being represented by a given lock set. If the lock is already controlled by another thread then the calling thread will spin. It means that the high number of accesses and their high percentage of execution time are grouped in synchronization operations into the number of barriers (locks) encountered by the processors when they access to the critical section. Among the suggestions for improvement is to pin the threads to cores.

Motion-JPEG

In the third application, which is parallel Motion-JPEG, we see that the growth rate is almost the same in all platforms and does not increase significantly like the first two experiments. Some addresses in the hot cluster are present in each platform: the frequent pattern is a loop of *soclib_fb_write* function responsible for displaying the decoded image. The evolution of this loop is stable in both 4, 8 and 16 CPUs platforms.

Other results are detected by the approach, the frequent accesses to the data addresses are called by set of instruction belonging to *memcpy*, and specially to the *__malloc_lock* and *__malloc_unlock* functions that copies the values from one memory block to another, and protect/release that memory blocks from corruption during simultaneous allocations, respectively. In order to improve Motion-JPEG application, the developer can spread its data over several physical memory banks, for example.

RADIX

In RADIX application, the `cpu_mp_wait` and `lock_acquire` functions are frequent bottlenecks. The growth rate of such functions (Fig. 7.12) is quite stable in platforms with 4 and 8 CPUs but it grows in the platform with 16 CPUs.

Mandelbrot

In addition to `cpu_mp_wait` function " `__fpadd_parts`, `__ieee754_sqrt`, `cpu_mp_wait`, `__unpack_d`, `__pack_d`, `__muldf3` " functions associated with the manipulation of floating-point numbers are discovered as bottlenecks in **Mandelbrot** application. The growth rate is greater in a platform with 16 CPUs than the other platforms.

Regarding the impact of discovered scalability hotspots in applications, there are more important and significant growth rate increases across platform instances in matrix multiplication and Ocean applications which is not the case for Motion-JPEG and LU and a little evolution of patterns in FFT (Fig. 7.12).

Table 7.8 shows the number of hot accesses and normal accesses in the hot cluster and the normal cluster, respectively. We see that the number of hot accesses is constant when the number of cores increases in a platform for each application. It shows the

highlighted and localized number of hot accesses in a platform in order to aid the user if the size of hot cluster, (*i.e.* the number of hot accesses) increases across the platforms.

Table 7.8: Coverage of clusters in each multi-threaded application across platform instances

Software	# CPU	# Address	
		Hot Cluster	Cluster
Matrix Multiplication	1	180	38294
	4	4	38470
	8	4	38470
	16	4	38470
Ocean	1	311	82310
	4	8	82613
	8	8	82613
	16	6	82615
Motion-JPEG	1	741	1223628
	4	85	1224284
	8	85	1224284
	16	60	1224309
FFT	1	227	75812
	4	12	76872
	8	5	78747
	16	5	78747
LU	1	151	95280
	4	59	97541
	8	37	97541
	16	37	97541
RADIX	1	197	38030
	4	79	42579
	8	63	45934
	16	63	45934
Mandelbrot	1	79	93930
	4	254	97792
	8	293	96740
	16	153	100180

These results obtained using these five applications confirm the interest of our approach, and its ability to provide both visual clues on the sources of scalability issues as well as precise code location that the developer should examine.

7.4.2 Discussion

The scalability bottleneck patterns discovered can help the user to improve his/her multi-threaded applications. From the hotspots discovered, there are several ways/method to improve the performance of the multi-threaded applications in multi-core platforms. The improvements can be made either at the software or hardware level. If hardware architecture design improvements are possible, they consist of fine tuning the multi-

core architecture in terms of general purpose (cache size and policies, ...) or custom hardware [BSsL+02], interconnects [GG00], interfaces [YYS+04]. It is very important to tune the code so that it exploits as well as the target hardware architecture. Software improvements can be a different level of granularity, either requiring the user to produce other coarse grain parallel version of its code, or at low level by changing synchronization primitives, data placement, and so on. Techniques like *Post-pass optimizers* or *object code optimizer* are capable of applying an extra optimization pass on assembly or object code. It tries to replace identifiable sections of the code like the pattern found in the Motion-JPEG application with replacement code that is more algorithmically efficient to a target architecture.
The critical section synchronization discovered in matrix multiplication and Ocean applications impact the scalability of the overall platform. As platform sizes get larger, the number of processes potentially requesting a lock increases. Several solutions have been proposed in the literature to reduce the idle wait process tile. In [BSPN03] the authors describe an implementation of an optimized operation which combines a global fence operation and a barrier synchronization operation, but usage of simpler atomic operations and lock-free or wait-free algorithms might also be applicable.

7.5 Conclusion

In this chapter, we experimented our approaches on seven parallel embedded applications. The approaches discover hotspots and bottlenecks caused by contention or scalability of platforms. They quantify and pinpoint the hotspots and bottlenecks in source code of application. We discovered that the hotspots and bottlenecks can be *synchronisation* functions, GCC compiler library functions associated with the manipulation of floating-point numbers, acquiring a *lock* in a application's function or functions in `libc` library of GCC compiler such as `malloc_lock()` and `malloc_unlock()` for locking the memory pool. These bottlenecks identifies also the load imbalance. We demonstrate that the developer gains upto 5% of runtime on super optimized applications. However, the few limitation of scalability bottlenecks discovery approach is: the same variable may have different memory addresses in different platform instance or execution trace files. So, the frequent hot access to this variable using frequent itemset mining can not capture all the addresses of the same variable in different platform causing the scalability issues. In this case, the need of a very efficient trace preprocessing or powerful frequent itemset mining algorithm, which is not necessarily trivial from addresses.

Chapter 8: Conclusions and Future Work

This thesis presents new profiling tools for parallel embedded applications in Multi-Processor System-on-Chip MPSoC. The profiling tools are based on data mining techniques applied on simulation traces. Our contributions are threefold. First, we proposed a technique for contention discovery in parallel embedded program. Second, we proposed an approach for scalability bottleneck discovery in MPSoC. Third, we proposed an effective way to trace analysis. Numerous experimental results have been provided to demonstrate the capacity of these approaches to pinpoint and quantify bottlenecks.

8.1 Conclusions

We presented the key problems of MPSoC profiling tools in the chapter 2, where a set of questions were asked that we intend to answer in this thesis. We repeat the same set of questions here and provide answers to them in the following text.

1. Contention problems.

   - What are the accesses leading to the contention ?
   - What happens during a contention ?
   - How to detect and identify co-occurrence of events ?
   - How to scale contention detection to large traces ?

   The automatic identification of parallel application contentions is a major issue for the optimization of application deployed in MPSoC, as it is one of the keys to enable good scalability. Using the trace generation capabilities of nowadays well accepted virtual platforms, we have presented an automatic approach based on data mining that when given only two thresholds, can automatically discover, extract, quantify contention patterns and pinpoint such patterns in source code. We have shown by experiments on different applications that the extracted patterns are interesting and can provide useful and meaningful information that will help the developer to understand the embedded software behavior, and the reasons of the contention in order to ease the optimization process (chapter 5).

   Then, we focused on scalability problems of parallel embedded programs which may be the one of the consequences of contention.

2. Scalability problems.

   - Why an application does not scale ?
   - What are the critical regions or parts of source code that create the scalability bottlenecks problems ?
   - Are the bottlenecks frequent or not in each platform ?

   System scalability is limited by the number of paths between the memory and the processors, which can lead to poor performance due to access contention. In the chapter 6, we have proposed a new approach for discovering the scalability bottlenecks patterns such as resource sharing, synchronization that reduce the parallel performance in MPSoC platforms running parallel embedded software. The proposed approach can profile the parallel embedded software in one (intra platform) or multiple platforms (inter platforms), and it is applicable on embedded systems as well as on parallel machines as long as detailed information on the memory accesses are available. It can be performed on multiple architectures with different type of processors in order to select the most appropriate processors type running a given multi-threaded program.

   For our profiling tools based on execution traces for discovering contention and scalability bottlenecks, we have answered the following questions:

3. Which techniques can discover and extract automatically meaningful knowledge of the bottlenecks from traces without developer intervention ?

   - How to pinpoint and quantify the contention and scalability bottlenecks from traces ?
   - How to discover and extract automatically recurrent hotspots / bottlenecks in parallel platform ?
   - What is the frequent concurrent accesses leading to contention ? How to discover and extract them ?
   - How to design the profiling tools for contention and scalability bottlenecks ?

   The approaches proposed in this thesis are based on data mining techniques on huge amount of execution traces. These techniques offer several advantages and help in the discovery process of profiling embedded applications, extract the unknown co-occurrence of different events of hotspots contention and scalability bottlenecks of a given application from execution traces. For knowledge discovery process, we use a frequent itemset mining algorithm for discovering co-occurrence of memory accesses repeating themselves often and occurring contentions. Also, this algorithm is used for discovering the frequent bottlenecks in scalable MPSoC platforms. Another algorithm used for knowledge discovery process is a clustering algorithm for grouping, discovering, identifying the bottlenecks in a given platform. These approaches are very helpful in reducing the amount of work a programmer has to do for localization of performance decline.

To the best of our knowledge, this is the first work reporting the use of data mining on MPSoC traces to identify hotspots and bottlenecks patterns that create contentions or scalability problems in multi-threaded applications.

Our experimental results demonstrate that our approach based on data mining techniques discovers unknown specific problems of a given application and specific instructions decreasing the performance. The quantified and pinpointed contention or scalability bottlenecks in source code help the developers to understand better the contention and scalability problems of their parallel embedded applications and facilitate the optimization of applications within a complex MPSoC platform.

8.2 Future Work

This thesis opens several perspectives for further research in the context of profiling tools for MPSoC using and data mining.

We classified the perspectives in three categories: short term, middle term and long term. Each one propose either an extension of proposed solution in this thesis or additional profiling tools.

Short term perspectives

The next generation of profiling tools for MPSoC based on data Mining using traces must be oriented toward the "big traces" aspect. These profiling tools must be based on either online trace analysis or postmortem analysis (like this thesis) with parallel / distributed algorithms.

- **Parallel / Distributed Algorithm**: The size of trace files increases drastically with the number of processors and the runtime of simulated application can quickly reach hundreds of giga bytes or tera bytes. Therefore, trace analysis becomes more and more difficult with existing data mining techniques. In this case, compression algorithms, trace management techniques and a parallel / distributed algorithm for trace analysis are required.

  Another characteristics of profiling tool can be proposed such as:

- **Loop profiling tool**: 90% of the execution time of programs is spent in loops [PRW10]. Loop profiling is a process which gives information about the execution time of loops. Thus, loop profiling tool is a necessary step and can be used in the process of speeding up software applications in MPSoC. In addition, the major results found in experiments are loops. We started working on this tool by developing a sequence mining algorithm on execution traces in order to extract and discover the loops executed by CPUs, and their characteristics such as the time spent and the number of calls, automatically.

- **Refinement Patterns**: In order to distinguish more finely several types of performance problems in scalable applications, our future plan consists on varying the number of clusters k of the approach described in the chapter 6.

Middle term perspectives

- **Call Graph Analysis and Mining**: A call graph is a directed graph that represents calling relationships between subroutines in a program. Specifically, each node represents a procedure and each edge (f_1, f_2) indicates that procedure f_1 calls procedure f_2. Each CPU has own call graph, thus in MPSoC, we can have as many graphs as processors. A new algorithm for mining call graphs can be proposed for extracting the following proprieties: frequent sub-graphs of procedure calls, and the profile of sub-graphs in terms of number of calls and time spent in sub-graphs. These proprieties allow to have a program behavior, dysfunctional of sub-graphs or possible anomalies and bugs.

- **Automatic Time Window**: The use of time window duration as input of algorithms for contention discovery in chapter 5 is adapted to dichotomic search. Discovering time window duration, automatically, is a challenge not only for our approaches but also for a general case of data mining algorithms.

Long term perspectives

- **Online Profiling tool** online profiling tool based on data mining traces allow to provide in real time profiling results and avoid to save giga or tera bytes of traces. However, this new profiling tool needs to handle the trace stream incoming from trace generator. So, new data mining algorithms over traces stream are needed. For example: frequent itemset mining algorithm over trace stream or clustering algorithm over trace stream. Also, the online profiling tool must be integrated into the MPSoC platform and executed in parallel.

  In addition to online profiling solution, we believe on an intelligent profiling tool which takes decisions instead of developer:

- **Self-adaptive algorithm**: Given a platform having 16 CPUs, if contention occurs frequently on a specific pattern while threads are active, the contention for shared resources can increase, and the application may experience stagnation or slowdown in performance. In this case, the parallel algorithm must take a decision to deactivate one or more CPUs or killing threads causing the contention. Identifying threads and/or data accesses that need to be asked upon can be done using data mining.

Appendix A: Trace preprocessing

This appendix briefly describes the trace preprocessing that consist to extract the function name from a given instruction address.

There are no function names in execution traces, but only the `PC` address of the executed instruction: using the symbols table of the executable *objdump*, we determine the function to which an instruction belongs and insert the function name in the raw traces.

objdump is used for disassembly a program. It displays and saves information about one object file. Basically, the objdump file contains information about the memory sections and their layout, the addresses of functions, and the assembly code. Figure A.1 shows part of the assembly code section of an objdump file for the MIPS32 processor built-in MJPEG program.

```
                                    • • •
10008250     <cpu_mp_wait>:

10008250:    3c031003     lui      v1,0x1003
10008254:    24020001     li       v0,1
10008258:    ac62bd20     sw       v0,-17120(v1)
1000825c:    0000000f     sync
10008260:    8c62bd20     lw       v0,-17120(v1)
10008264:    1440fffe     bnez     v0,10008260 <cpu_mp_wait+0x10>
10008268:    00000000     nop
1000826c:    03e00008     jr       ra
10008270:    00000000     nop
                                    • • •
Location     Memory Contents     Disassembly Results
```

Figure A.1: An example of disassembly of executable MJPEG code

The information in objdump file is mostly useful to programmers who are working on profiling software in low level, optimization and the compilation tools. Using the objdump file, we can extract the function name of an address in order to have both low and high level of granularity of traces. Table A.1 shows the new raw trace format with the function name inserted in last column. Note that this does not represent the call graph, but the currently active function.

Table A.1: New Raw trace format

CPU ID	**Cycle Number**	**Program Counter**	**Instruction Type**	**Data Address**	**Access Latency**	**Function Name**
1	212305	0x10008260	fetch	0x10008260	28	cpu_mp_wait

Appendix B: Contention Patterns

This appendix briefly describes the contention patterns discovered in different embedded applications.

FFT

Figure B.1 shows the frequent contention patterns in FFT benchmark in a platform with 4 and 8 CPUs. We see that the frequent contention patterns are dominated by an lengthy initialization phase of FFT function (*InitU2*), __kernel_cos and __kernel_sin manipulating trigonometric functions, and cpu_mp_wait function. These patterns are both in a platform with 4 and 8 CPUs with different threshold.

Table B.1: Frequent Contention Patterns in FFT Application

Platform	Frequent Pattern	Support
4 CPUs	CPU[0,3] [`0x10006314:0x10006388`] InitU2	40 %
	CPU[0,3] [`0x10026328:0x10026380`] __kernel_cos [`0x100274ac:0x100274f8`] __kernel_sin	32 %
	CPU[0,3] [`0x1000b0a08:0x1000b0a8`] cpu_mp_wait	88%
8 CPUs	CPU[0,7] [`0x10006314:0x10006388`] InitU2	45 %
	CPU[0,7] [`0x10026328:0x10026380`] __kernel_cos [`0x100274ac:0x100274f8`] __kernel_sin	28 %
	CPU[0,7] [`0x1000b0a08:0x1000b0a8`] cpu_mp_wait	92%

Mandelbrot

The frequent contention patterns in Mandelbrot application in a platform with 4 or 8 CPUs (Table B.2) are a set of data variable manipulating by floating points functions *__fpadd_parts __unpack_d __ieee754_sqrt __pack_d __muldf3*. These data variable are shown in Table B.3 and Table B.4 for patterns of a platform with 4 or 8 CPUs, respectively. These data variable are manipulated by different instructions of floating points functions. The instructions perform loading / storing operations on these data variable.

RADIX

Among the frequent contention patterns in RADIX application shown in Table B.5. We discover that the accesses to the loop in *ran_num_init* are responsible of contention.

Table B.2: Frequent Contention Patterns in Mandelbrot Application

Platform	Frequent Pattern	Support
4 CPUs	CPU[0,2] __fpadd_parts __unpack_d __ieee754_sqrt __pack_d __muldf3 (data variables in Table B.3)	87 %
8 CPUs	CPU[0,5] __fpadd_parts __unpack_d __ieee754_sqrt __pack_d __muldf3 (data variables in Table B.4)	95 %

Table B.3: Data variable called by floating point functions in platform with 4 CPUs

```
0x10194190 0x101a42a0 0x1019c218 0x1018c108 0x1019c1f8 0x101a4280
0x10194170 0x101a42d0 0x1019c248 0x1018c0e8 0x101941c0 0x101a4288
0x1019c200 0x10194178 0x1018c0f0 0x1019c228 0x101a42b0 0x101941b4
0x1006c1ec 0x1019c23c 0x101a42c4 0x1019c230 0x1018c1b0 0x101941e4
0x101941a0 0x101a42b8 0x1018c1c0 0x1019c1f0 0x101a4278 0x1018c12c
0x101941a8 0x1018c1f0 0x1018c138 0x1019c26c 0x101a428c 0x10194168
0x1018c15c 0x101a42f4 0x1019417c 0x1019c204
```

ran_num_init function represents a loop that consist to fill the random-number array. In order to tackle this contention pattern, one possible enhancement proposed in RADIX source code application is to pin processes to processors in order to avoid migration.

LU

The discovered contention pattern in Table. B.6 is a multiple accesses simultaneously at two instructions ①, ② in Figure B.1 of *InitA* function in LU application. The first instruction ① performs multiple initialization of a matrix using *lrand48*[1] by all threads, and the second instruction ② performs the same operation on the same array index, thus, different results on different executions are obtained. This pattern is considered as a bug by developers.

Figure B.1: InitA function of LU application

```
void InitA(double *rhs)
{
…
for (j=0; j<n; j++) {
for (i=0; i<n; i++) {
…
a[ii][jj] = ((double) lrand48())/MAXRAND;//①// executed by all threads
}}
for (j=0; j<n; j++){
for (i=0; i<n; i++){
…
rhs[i] += a[ii][jj];//②// executed by all threads
}}}
```

[1] *lrand48* is a function in C *stdlib* library. The *lrand48* function uses 48-bit arithmetic to generate non-negative long integers uniformly distributed pseudo-random values.

Table B.4: Data variable called by floating point functions in platform with 8 CPUs

0x101b43b0	0x101c44c0	0x101ac328	0x1019c218	0x101bc438	0x101a42a0
0x10194190	0x1018c108	0x101a4280	0x101bc418	0x101ac308	0x1019c1f8
0x101b4390	0x101c44a0	0x10194170	0x101bc448	0x101a42b0	0x101ac310
0x101ac338	0x101a4288	0x101c44d0	0x101b43c0	0x1018c0e8	0x101bc420
0x1019c200	0x101b4398	0x101c44a8	0x1019c228	0x1019c248	0x101941a8
0x10194178	0x101bc468	0x101a42d0	0x1008c274	0x101b43e0	0x101ac300
0x101ac358	0x101bc450	0x101b43c8	0x101c44f0	0x1018c0f0	0x101941a0
0x101a4278	0x101c4498	0x1019c230	0x101ac340	0x101b4388	0x101941c0
0x101a42b8	0x101c44d8	0x10194168	0x1019c1f0	0x1018c1c0	0x101a42bc
0x101bc410	0x1019c234	0x101b43cc			

Table B.5: Frequent Contention Patterns in RADIX Application

Platform	Frequent Pattern	Support
4 CPUs	CPU[0,3] [0x10005c5c:0x10006068] ran_num_init	35 %
	CPU[0,3] [0x1000b940:0x1000b948] cpu_mp__wait	55 %
	CPU[0,3] 10038618 semaphore_pool	42 %
8 CPUs	CPU[0,7] [0x10005c5c:0x10006068] ran_num_init	39 %
	CPU[0,7] [0x1000b940:0x1000b948] cpu_mp__wait, 10038618 semaphore_pool	65 %

Table B.6: Frequent Contention Patterns in LU Application

Platform	Frequent Pattern	Support
4 CPUs	CPU[0,3] [0x10015f9c:0x10015fdc] InitA 100071ec lrand48	69 %
8 CPUs	CPU[0,7] [0x10015f9c:0x10015fdc] InitA 100071ec lrand48	78 %

Glossary

AMBA Advanced Micro-controller Bus Architecture

API Application Programming Interface

ASIP Application-Specific Instruction-set Processor

ATL Application Trace Logger

CABA Cycle Accurate Bit Accurate

CMP Chip MultiProcessor

ACMP Asymmetric Chip MultiProcessor

SCMP Symmetric Chip MultiProcessor

CPI Cycles Per Instruction

CPU Central Processing Unit

DAG Directed Acyclic Graph

DNA DNA is Not just Another OS

DS Distributed Scheduling

DSP Digital Signal Processor

DSM Distributed Shared Memory

FCA Formal Concept Analysis

FFT Fast Fourier Transform

FPM Frequent Pattern Mining

FPGA Field Programmable Gate Array

HAL Hardware Abstraction Layer

GPU Graphics Processing Unit

HAL Hardware Abstraction Layer

HCBP Hardware Counter Based Profiling

IR Intermediate Representation

ISR Interrupt Service Routines

ISS Instruction Set Simulator

LU Lower Upper

MARS Multivariate Adaptive Regression Splines

MIC Many Integrated Core

MJPEG Motion-JPEG

MPSoC Multi-Processor System On Chip

NoC Network On Chip

PC Program Counter

PCA Principal Component Analysis

RTL Register Transfer Level

SLPE Source-Level Performance Estimation

SMP Symmetric Multiple Processor

SoC System On Chip

SystemCASS SystemC Accurate System Simulator

TLM Transaction Level Modeling

UML	Unified Modeling Language	**VLIW**	Very Long Instruction Word
VA	Virtual Assembly		

List of Publications

International Conferences

[1] **Sofiane LAGRAA**, Alexandre Termier , and Frédéric Pétrot. Scalability Bottlenecks Discovery in MPSoC Platforms Using Data Mining on Simulation Traces. In *Proceedings of the 17th IEEE International Conference on Design, Automation and Test in Europe*, ***DATE 2014***, *Dresden, Germany*. ***[To Be Published]***. ***[Best Paper Award]***

[2] **Sofiane LAGRAA**, Alexandre Termier , and Frédéric Pétrot. Data mining MPSoC simulation traces to identify concurrent memory access patterns. In *Proceedings of the 16th IEEE International Conference on Design, Automation and Test in Europe*, ***DATE 2013***, *Grenoble, France*, pages 755-760, 2013. ***[Published]***

[3] **Sofiane LAGRAA**, Alexandre Termier , and Frédéric Pétrot. Automatic congestion detection in MPSoC programs using data mining on simulation traces. In *Proceedings of the 23rd IEEE International Symposium on Rapid System Prototyping*, ***RSP 2012***, *Tampere, Finland*, pages 64-70, 2012. ***[Published]***

References

[ABD+97] Jennifer M. Anderson, Lance M. Berc, Jeffrey Dean, Sanjay Ghemawat, Monika R. Henzinger, Shun-Tak A. Leung, Richard L. Sites, Mark T. Vandevoorde, Carl A. Waldspurger, and William E. Weihl. Continuous profiling: where have all the cycles gone? *ACM Trans. Comput. Syst.*, 15(4):357–390, November 1997.

[ACG+03] Adrijean Adriahantenaina, Herve Charlery, Alain Greiner, Laurent Mortiez, and Cesar Albenes Zeferino. Spin: A scalable, packet switched, on-chip micro-network. In *Proceedings of the conference on Design, Automation and Test in Europe: Designers' Forum - Volume 2*, DATE '03, pages 20070–, 2003.

[AK89] S. B. Akers and B. Krishnamurthy. A group-theoretic model for symmetric interconnection networks. *IEEE Trans. Comput.*, 38(4):555–566, 1989.

[Amd67] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pages 483–485, 1967.

[AMD02] AMD. Amd athlon processor, x86 code optimization guide. 2002.

[AS94] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *VLDB*, pages 487–499, 1994.

[AZXC11] Najla Alfaraj, Junjie Zhang, Yang Xu, and H. Jonathan Chao. Hope: Hotspot congestion control for clos network on chip. In *NOCS*, pages 17–24, 2011.

[BA97a] Leonard A. Breslow and David W. Aha. Simplifying decision trees: A survey. *Knowl. Eng. Rev.*, 12(1):1–40, January 1997.

[BA97b] Doug Burger and Todd M. Austin. The simplescalar tool set, version 2.0. *SIGARCH Comput. Archit. News*, 25(3):13–25, 1997.

[Bai90] D. H. Bailey. Ffts in external or hierarchical memory. *J. Supercomput.*, 4(1):23–35, March 1990.

[Bal69] R. M. Balzer. Exdams: extendable debugging and monitoring system. In *AFIPS*, pages 567–580, New York, NY, USA, 1969. ACM.

[BCZ07] Marc Boule, Jean-Samuel Chenard, and Zeljko Zilic. Assertion checkers in verification, silicon debug and in-field diagnosis. In *Proceedings of the 8th International Symposium on Quality Electronic Design*, ISQED '07, pages 613–620, 2007.

[BDG+00] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '00, 2000.

[BFSS01] C. Brandolese, W. Fornaciari, F. Salice, and D. Sciuto. Source-level execution time estimation of c programs. In *Proceedings of the ninth international symposium on Hardware/software codesign*, CODES '01, pages 98–103, 2001.

[BKL+00] Jwahar R. Bammi, Wido Kruijtzer, Luciano Lavagno, Edwin Harcourt, and Mihai T. Lazarescu. Software performance estimation strategies in a system-level design tool. In *Proceedings of the eighth international workshop on Hardware/software codesign*, CODES '00, pages 82–86, 2000.

[BM05] A. A. Bayazit and S. Malik. Complementary use of runtime validation and model checking. In *Proceedings of the 2005 IEEE/ACM International conference on Computer-aided design*, ICCAD '05, pages 1052–1059, 2005.

[BPG04] R. Buchmann, F. Petrot, and A. Greiner. Fast cycle accurate simulator to simulate event-driven behavior. In *Electrical, Electronic and Computer Engineering, 2004. ICEEC '04. 2004 International Conference on*, pages 35–38, 2004.

[Bra77] Achi Brandt. Multi-Level Adaptive Solutions to Boundary-Value Problems. *Mathematics of Computation*, 31(138):333–390, April 1977.

[BSPN03] Darius Buntinas, Amina Saify, Dhabaleswar K. Panda, and Jarek Nieplocha. Optimizing synchronization operations for remote memory communication systems. In *IPDPS*, pages 199.1–, 2003.

[BSsL+02] Rajeshwari Banakar, Stefan Steinke, Bo sik Lee, M. Balakrishnan, and Peter Marwedel. Scratchpad memory: A design alternative for cache on-chip memory in embedded systems. In *CODES*, pages 73–78. ACM, 2002.

[BZ08] Marc Boulé and Zeljko Zilic. Automata-based assertion-checker synthesis of psl properties. *ACM Trans. Des. Autom. Electron. Syst.*, 13(1):4:1–4:21, February 2008.

[CB92] Tien-Fu Chen and Jean-Loup Baer. Reducing memory latency via non-blocking and prefetching caches. In *Proceedings of the fifth international conference on Architectural support for programming languages and operating systems*, ASPLOS-V, pages 51–61, 1992.

[CBT+12] Patricia López Cueva, Aurélie Bertaux, Alexandre Termier, Jean-François Méhaut, and Miguel Santana. Debugging embedded multimedia application traces through periodic pattern mining. In *EMSOFT*, pages 13–22, 2012.

[CCCK11] Chih-Neng Chung, Chia-Wei Chang, Kai-Hui Chang, and Sy-Yen Kuo. Applying verification intention for design customization via property mining under constrained testbenches. In *ICCD*, pages 84–89, 2011.

[CDFR08] Peggy Cellier, Mireille Ducassé, Sébastien Ferré, and Olivier Ridoux. Formal concept analysis enhances fault localization in software. In *Proceedings of the 6th international conference on Formal concept analysis*, ICFCA'08, pages 273–288, 2008.

[CDFR11] Peggy Cellier, Mireille Ducassé, Sébastien Ferré, and Olivier Ridoux. Multiple fault localization with data mining. In *SEKE*, pages 238–243, 2011.

[CFDC11] Peggy Cellier, Sébastien Ferré, Mireille Ducassé, and Thierry Charnois. Partial orders and logical concept analysis to explore patterns extracted by data mining. In *Proceedings of the 19th international conference on Conceptual structures for discovering knowledge*, ICCS'11, pages 77–90, 2011.

[CH08] Xueqi Cheng and Michael S. Hsiao. Simulation-directed invariant mining for software verification. In *DATE*, pages 682–687, 2008.

[CLZ+11] Xuhao Chen, Jiawen Li, Zhong Zheng, Li Shen, and Zhiying Wang. Evaluating scalability of emerging multithreaded applications on commodity multicore server. In *Proceedings of the 2011 International Conference of Information Technology, Computer Engineering and Management Sciences - Volume 01*, ICM '11, pages 332–335, 2011.

[CME+95] Woo Steven Cameron, Ohara Moriyoshi, Torrie Evan, Singh Jaswinder Pal, and Gupta Anoop. The splash-2 programs: characterization and methodological considerations. In *ISCA*, pages 24–36, 1995.

[Cor06] Intel Corporation. Ia-32 intel architecture software developer's manual. *[Online]. Available:http://developers.sun.com/prodtech/cc/articles/pcounters.html.*, Accessed February 2006.

[Cor13] CoreSight. Coresight on-chip debug trace ip. 2013.

[CW10] Po-Hsien Chang and Li-C. Wang. Automatic assertion extraction via sequential data mining of simulation traces. In *ASP-DAC*, pages 607–612, 2010.

[DH01] Werner Damm and David Harel. Lscs: Breathing life into message sequence charts, 2001.

[DNAKN05] M. Dehyadgari, M. Nickray, A. Afzali-Kusha, and Z. Navabi. Evaluation of pseudo adaptive xy routing using an object oriented model for noc. In *Microelectronics, 2005. ICM 2005. The 17th International Conference on*, pages 5 pp.–, 2005.

[DT01] William J. Dally and Brian Towles. Route packets, not wires: on-chip inteconnection networks. In *Proceedings of the 38th annual Design Automation Conference*, DAC '01, pages 684–689. ACM, 2001.

[DYL02] Jose Duato, Sudhakar Yalamanchili, and Ni Lionel. *Interconnection Networks: An Engineering Approach*. Morgan Kaufmann Publishers Inc., 2002.

[EDBE12] Stijn Eyerman, Kristof Du Bois, and Lieven Eeckhout. Speedup stacks: identifying scaling bottlenecks in multi-threaded applications. In *IEEE international symposium on performance analysis of systems and software, Proceedings*, pages 145–155. IEEE, 2012.

[EEKS06] Stijn Eyerman, Lieven Eeckhout, Tejas Karkhanis, and James E. Smith. A performance counter architecture for computing accurate cpi components. *SIGOPS Oper. Syst. Rev.*, 40(5):175–184, 2006.

[EWL13] Juan Fernando Eusse, Christopher Williams, and Rainer Leupers. Coex: A novel profiling-based algorithm/architecture co-exploration for asip design. In *ReCoSoC*, pages 1–8, 2013.

[FIM04] Workshop on frequent itemset mining implementations, 2004. `http://fimi.ua.ac.be/fimi04/`.

[Fos95] Ian Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[FPsS96] Usama Fayyad, Gregory Piatetsky-shapiro, and Padhraic Smyth. From data mining to knowledge discovery in databases. *AI Magazine*, 17:37–54, 1996.

[FR01] Sébastien Ferré and Olivier Ridoux. Searching for objects and properties with logical concept analysis. In *Proceedings of the 9th International Conference on Conceptual Structures: Broadening the Base*, ICCS '01, pages 187–201, 2001.

[FR04] S. Ferré and O. Ridoux. Introduction to logical information systems. *Inf. Process. Manage.*, 40(3):383–419, 2004.

[Fri91] J. H. Friedman. Multivariate Adaptive Regression Splines. *Annals of Statistics*, 19, 1991.

[FS08] Jay Fenlason and Richard Stallman. Gnu gprof. 2003. [Accessed April 6th 2008].

[GG00] Pierre Guerrier and Alain Greiner. A generic architecture for on-chip packet-switched interconnections. In *DATE*, pages 250–256, 2000.

[GGM04] Floris Geerts, Bart Goethals, and Taneli Mielikäinen. Tiling databases. In *Discovery Science*, pages 278–289, 2004.

[GHC+09] Lei Gao, Jia Huang, Jianjiang Ceng, Rainer Leupers, Gerd Ascheid, and Heinrich Meyr. Totalprof: a fast and accurate retargetable source code profiler. In *Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, CODES+ISSS '09, pages 305–314, 2009.

[GKM82] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. Gprof: A call graph execution profiler. *SIGPLAN Not.*, 17(6):120–126, June 1982.

[GKS12] Vishal Gupta, Hyesoon Kim, and Karsten Schwan. Evaluating scalability of multi-threaded applications on a many-core platform, CiteSeerX, vol. 21, 2012.

[GMCP13] Bernard Goossens, Ali El Moussaoui, K. Chen, and David Parello. De quoi est faite une trace d'exécution ? *Technique et Science Informatiques*, 2013.

[GP09] Xavier Guerin and Frédéric Petrot. A system framework for the design of embedded software targeting heterogeneous multi-core socs. In *Proceedings of the 2009 20th IEEE International Conference on Application-specific Systems, Architectures and Processors*, ASAP '09, pages 153–160, 2009.

[GRV05] Ann Gordon-Ross and Frank Vahid. Frequent loop detection using efficient nonintrusive on-chip hardware. *IEEE Trans. Comput.*, 54(10):1203–1215, October 2005.

[GW97] Bernhard Ganter and Rudolf Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edition, 1997.

[HAG08] Yonghyun Hwang, Samar Abdi, and Daniel Gajski. Cycle-approximate retargetable performance estimation at the transaction level. In *Proceedings of the conference on Design, automation and test in Europe*, DATE '08, pages 3–8, 2008.

[HCC+11] Wim Heirman, Trevor E. Carlson, Shuai Che, Kevin Skadron, and Lieven Eeckhout. Using cycle stacks to understand scaling bottlenecks in multi-threaded workloads. In *Proceedings of the 2011 IEEE International Symposium on Workload Characterization*, IISWC '11, pages 38–49, 2011.

[HK06] Jiawei Han and Micheline Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 2006.

[HP11] Damien Hedde and Frédéric Pétrot. A non intrusive simulation-based trace system to analyse multiprocessor systems-on-chip software. In *International Symposium on Rapid System Prototyping*, pages 106–112, 2011.

[HPY00] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. *SIGMOD Rec.*, 29(2):1–12, May 2000.

[HR07] Ralf Hoffmann and Thomas Rauber. Profiling of task-based applications on shared memory machines: scalability and bottlenecks. In *Proceedings of the 13th international Euro-Par conference on Parallel Processing*, Euro-Par'07, pages 118–128, Berlin, Heidelberg, 2007. Springer-Verlag.

[HW79] J. A. Hartigan and M. A. Wong. A K-means clustering algorithm. *Applied Statistics*, 28:100–108, 1979.

[Ins] Texas Instruments. Omap™4 mobile applications platform.

[ITR07] ITRS. International technology roadmap for semiconductors - system drivers [online]. http://www.itrs.net/Links/2007ITRS/2007_Chapters/2007_SystemDrivers.pdf, 2007.

[JKLS10] Kwangok Jeong, A. B. Kahng, B. Lin, and K. Samadi. Accurate machine-learning-based on-chip router modeling. *IEEE Embed. Syst. Lett.*, 2(3):62–66, 2010.

[JSMP12] José A. Joao, M. Aater Suleman, Onur Mutlu, and Yale N. Patt. Bottleneck identification and scheduling in multithreaded applications. *SIGARCH Comput. Archit. News*, 40(1), March 2012.

[KAL] KALRAY. Kalray's mppa (multi-purpose processor array). http://www.kalray.eu/products/mppa-manycore/mppa-256/.

[KFI+12] C. Kamdem Kengne, L. C. Fopa, N. Ibrahim, Alexandre Termier, Marie-Christine Rousset, and Takashi Washio. Enhancing the analysis of large multimedia applications execution traces with frameminer. In *ICDM Workshops*, pages 595–602, 2012.

[KFK+05] Kingshuk Karuri, Mohammad Abdullah Al Faruque, Stefan Kraemer, Rainer Leupers, Gerd Ascheid, and Heinrich Meyr. Fine-grained application source code profiling for asip design. In *DAC*, pages 329–334, 2005.

[KFT+13] Christiane Kamdem Kengne, Leon Constantin Fopa, Alexandre Termier, Noha Ibrahim, Marie-Christine Rousset, Takashi Washio, and Miguel Santana. Efficiently rewriting large multimedia application execution traces with few event sequences. In *KDD*, pages 1348–1356, 2013.

[KKJ+08] Seongnam Kwon, Yongjoo Kim, Woo-Chul Jeun, Soonhoi Ha, and Yunheung Paek. A retargetable parallel-programming framework for mpsoc. *ACM Trans. Des. Autom. Electron. Syst.*, 13(3):39:1–39:18, July 2008.

[KKRL11] Sandeep Kumar, Siau-Cheng Khoo, Abhik Roychoudhury, and David Lo. Mining message sequence graphs. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 91–100, 2011.

[KKW+06] Torsten Kempf, Kingshuk Karuri, Stefan Wallentowitz, Gerd Ascheid, Rainer Leupers, and Heinrich Meyr. A sw performance estimation framework for early system-level-design using fine-grained instrumentation. In *Proceedings of the conference on Design, automation and test in Europe: Proceedings*, DATE '06, pages 468–473, 2006.

[KLPS09] Andrew B. Kahng, Bin Li, Li-Shiuan Peh, and Kambiz Samadi. Orion 2.0: a fast and accurate noc power and area model for early-stage design space exploration. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, pages 423–428, 2009.

[KND02] Faraydon Karim, Anh Nguyen, and Sujit Dey. An interconnect architecture for networking systems on chips. *IEEE Micro*, 22:36–45, 2002.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[LBH+00] M. T. Lazarescu, J. R. Bammi, E. Harcourt, L. Lavagno, and M. Lajolo. Compilation-based software performance estimation for system level design. In *Proceedings of the IEEE International High-Level Validation and Test Workshop (HLDVT'00)*, HLDVT '00, 2000.

[LC10] Rainer Leupers and Jeronimo Castrillon. Mpsoc programming using the maps compiler. In *Proceedings of the 2010 Asia and South Pacific Design Automation Conference*, ASPDAC '10, pages 897–902, Piscataway, NJ, USA, 2010. IEEE Press.

[LCM+05] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 190–200, 2005.

[LCZ01] Honghui Lu, Alan L. Cox, and Willy Zwaenepoel. Contention elimination by replication of sequential sections in distributed shared memory programs. In *PPoPP'01*, pages 53–61, 2001.

[LFS10] Wenchao Li, Alessandro Forin, and Sanjit A. Seshia. Scalable specification mining for verification and diagnosis. In *DAC*, pages 755–760, 2010.

[LKL08] David Lo, Siau-Cheng Khoo, and Chao Liu. Mining past-time temporal rules from execution traces. In *Proceedings of the 2008 international workshop on dynamic analysis: held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008)*, WODA '08, pages 50–56, 2008.

[LL73] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. In *Journal of the ACM*, pages 46–61, 1973.

[LLSV99] Marcello Lajolo, Mihai Lazarescu, and Alberto Sangiovanni-Vincentelli. A compilation-based software estimation scheme for hardware/software co-simulation. In *Proceedings of the seventh international workshop on Hardware/software codesign*, CODES '99, pages 85–89, 1999.

[LLV12] Lingyi Liu, Chen-Hsuan Lin, and Shobha Vasudevan. Word level feature discovery to enhance quality of assertion mining. In *ICCAD*, pages 210–217, 2012.

[LMK07] David Lo, Shahar Maoz, and Siau-Cheng Khoo. Mining modal scenario-based specifications from execution traces of reactive systems. In *ASE*, pages 465–468, 2007.

[LSAV11] Lingyi Liu, David Sheridan, Viraj Athavale, and Shobha Vasudevan. Automatic generation of assertions from system level design using data mining. In *MEMOCODE*, pages 191–200, 2011.

[LTP12] Sofiane Lagraa, Alexandre Termier, and Frédéric Pétrot. Automatic congestion detection in mpsoc programs using data mining on simulation traces. In *RSP*, pages 64–70, 2012.

[LTP13] Sofiane Lagraa, Alexandre Termier, and Frédéric Pétrot. Data mining mpsoc simulation traces to identify concurrent memory access patterns. In *DATE*, pages 755–760, 2013.

[LTP14] Sofiane Lagraa, Alexandre Termier, and Frédéric Pétrot. Scalability bottlenecks discovery in mpsoc platforms using data mining on simulation traces. In *DATE*, 2014.

[LXM08] Christopher LaRosa, Li Xiong, and Ken Mandelberg. Frequent pattern mining for kernel trace data. In *Proceedings of the 2008 ACM symposium on Applied computing*, SAC '08, pages 880–885, New York, NY, USA, 2008. ACM.

[LYY+05] Chao Liu, Xifeng Yan, Hwanjo Yu, Jiawei Han, and Philip S. Yu. Mining behavior graphs for "backtrace" of noncrashing bugs. In *SDM*, 2005.

[Mar06] Grant Martin. Overview of the mpsoc design challenge. In *Proceedings of the 43rd annual Design Automation Conference*, DAC '06, pages 274–279, New York, NY, USA, 2006. ACM.

[MAT09] Mubrak S. Mohsen, Rosni Abdullah, and Yong M. Teo. A survey on performance tools for openmp, 2009.

[MH89] Charles E. McDowell and David P. Helmbold. Debugging concurrent programs. *ACM Comput. Surv.*, 21(4):593–622, December 1989.

[Mic06] Sun Microsystems. Using ultrasparc-iiicu performance counters to improve application performance. *[Online]. Available: http://developers.sun.com/prodtech/cc/articles/pcounters.html*, Accessed February 2006.

[MLG05] Edu Metz, Raimondas Lencevicius, and Teofilo F. Gonzalez. Performance data collection using a hybrid approach. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-13, pages 126–135, 2005.

[MMW96] Sfi-Tr-William Macready, William G. Macready, and David H. Wolpert. What makes an optimization problem hard? *Complexity*, 5, 1996.

[MSR+07] Tipp Moseley, Alex Shye, Vijay Janapa Reddi, Dirk Grunwald, and Ramesh Peri. Shadow profiling: Hiding instrumentation costs with parallelism. In *CGO*, pages 198–208, 2007.

[MTQ07] Parth Malani, Ying Tan, and Qinru Qiu. Resource-aware high performance scheduling for embedded mpsocs with the application of mpeg decoding. In *ICME*, pages 715–718, 2007.

[NM92] Robert H. B. Netzer and Barton P. Miller. What are race conditions?: Some issues and formalizations. *ACM Lett. Program. Lang. Syst.*, 1(1):74–88, March 1992.

[Pot06] Kristin Potter. Methods for presenting statistical information: The box plot. *Hans Hagen, Andreas Kerren, and Peter Dannenmann (Eds.), Visualization of Large and Unstructured Data Sets, GI-Edition Lecture Notes in Informatics (LNI)*, pages 97–106, 2006.

[PR12] Rajendra Patel and Arvind Rajawat. A survey of embedded software profiling methodologies. *International Journal of Embedded Systems and Applications (IJESA)*, 1(2):19–40, 2012.

[PRW10] Marcin Pietron, Pawel Russek, and Kazimierz Wiatr. Loop profiling tool for hpc code inspection as an efficient method of fpga based acceleration. *Applied Mathematics and Computer Science*, 20(3), 2010.

[QLD09] Yue Qian, Zhonghai Lu, and Wenhua Dou. Analysis of communication delay bounds for network on chips. In *Proceedings of the 2009 Asia and South Pacific Design Automation Conference*, ASP-DAC '09, pages 7–12, 2009.

[Rei05] J. Reinders. *VTune Performance Analyzer Essentials: Measurement and Tuning Techniques for Software Developers*. Engineer to Engineer Series. Intel Press, 2005.

[RL10] Rohit Sunkam Ramanujam and Bill Lin. Destination-based adaptive routing on 2d mesh networks. In *ANCS*, page 19, 2010.

[RPN97] Anand Raman, Jon Patrick, and Palmerston North. The sk-strings method for inferring pfsa. In *In Proceedings of the*, 1997.

[Sam] Samsung. Samsung exynos 5 quad 5210 technical specifications.

[SBL+09a] Hassan Saneifar, Stéphane Bonniol, Anne Laurent, Pascal Poncelet, and Mathieu Roche. Mining for relevant terms from log files. In *KDIR*, pages 77–84, 2009.

[SBL+09b] Hassan Saneifar, Stéphane Bonniol, Anne Laurent, Pascal Poncelet, and Mathieu Roche. Terminology extraction from log files. In *DEXA*, pages 769–776, 2009.

[SC04a] Lesley Shannon and Paul Chow. Maximizing system performance: using reconfigurability to monitor system communications. In *FPT*, pages 231–238, 2004.

[SC04b] Lesley Shannon and Paul Chow. Using reconfigurability to achieve real-time profiling for hardware/software codesign. In *Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, FPGA '04, pages 190–199, 2004.

[SLT99] Yan Solihin, Vinh Lam, and Josep Torrellas. Scal-tool: pinpointing and quantifying scalability bottlenecks in dsm multiprocessors. In *Supercomputing*, 1999.

[SMG12] Arnab Sinha, Sharad Malik, and Aarti Gupta. Efficient predictive analysis for detecting nondeterminism in multi-threaded programs. In *FMCAD*, pages 6–15, 2012.

[SMWG11] Arnab Sinha, Sharad Malik, Chao Wang, and Aarti Gupta. Predictive analysis for detecting serializability violations through trace segmentation. In *MEMOCODE*, pages 99–108, 2011.

[Soc] SoCLib, "A modeling and simulation platform for system on chip", 2009. [Online]. Available: http://www.soclib.fr/Home.html.

[Spr02] Brinkley Sprunt. The basics of performance-monitoring hardware. *IEEE Micro*, 22(4):64–71, July 2002.

[SRA05] Koushik Sen, Grigore Roşu, and Gul Agha. Detecting errors in multi-threaded programs by generalized predictive analysis of executions. In *Proceedings of the 7th IFIP WG 6.1 international conference on Formal Methods for Open Object-Based Distributed Systems*, FMOODS'05, 2005.

[STM] STMicroelectronics. Advanced hd application processor with 3d graphics acceleration and arm cortex-a9 smp cpu.

[STM13] STMicroelectronics. Application trace logger (atl). Technical report, STMicroelectronics, 2013.

[TdRC+10] Leonel Tedesco, Thiago R. da Rosa, Fabien Clermidy, Ney Calazans, and Fernando Gehm Moraes. Implementation and evaluation of a congestion aware routing algorithm for networks-on-chip. In *SBCCI*, pages 91–96, 2010.

[TK08a] Jason G. Tong and Mohammed A. S. Khalid. Profiling tools for fpga-based embedded systems: Survey and quantitative comparison. *JCP*, 3(6), 2008.

[TK08b] Jason G. Tong and Mohammed A. S. Khalid. Profiling tools for fpga-based embedded systems: Survey and quantitative comparison. *JCP*, 3(6), 2008.

[UKA04a] Takeaki Uno, Masashi Kiyomi, and Hiroki Arimura. Lcm ver. 2: Efficient mining algorithms for frequent/closed/maximal itemsets. In *FIMI*, 2004.

[UKA04b] Takeaki Uno, Masashi Kiyomi, and Hiroki Arimura. Lcm ver. 2: Efficient mining algorithms for frequent/closed/maximal itemsets. In *FIMI*, 2004.

[VSP+10] Shobha Vasudevan, David Sheridan, Sanjay J. Patel, David Tcheng, William Tuohy, and Daniel R. Johnson. Goldmine: Automatic assertion generation using data mining and static analysis. In *DATE*, pages 626–629, 2010.

[WCGY09] Chao Wang, Swarat Chaudhuri, Aarti Gupta, and Yu Yang. Symbolic pruning of concurrent program executions. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC/FSE '09, 2009.

[WDV06] Stephan Wong, Filipa Duarte, and Stamatis Vassiliadis. A hardware cache memcpy accelerator. In *In Proc. IEEE International Conference in Field Programmable Technology*, pages 141–147, 2006.

[WG12] Chao Wang and Malay Ganai. Predicting concurrency failures in the generalized execution traces of x86 executables. In *Proceedings of the Second international conference on Runtime verification*, RV'11, pages 4–18, 2012.

[Wil09] Rudolf Wille. Restructuring lattice theory: An approach based on hierarchies of concepts. In *Proceedings of the 7th International Conference on Formal Concept Analysis*, ICFCA '09, pages 314–339, 2009.

[WKGG09] Chao Wang, Sudipta Kundu, Malay Ganai, and Aarti Gupta. Symbolic predictive analysis for concurrent programs. In *Proceedings of the 2nd World Congress on Formal Methods*, FM '09, pages 256–272, 2009.

[WLGG10] Chao Wang, Rhishikesh Limaye, Malay Ganai, and Aarti Gupta. Trace-based symbolic analysis for atomicity violations. In *Proceedings of the 16th international conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'10, pages 328–342, 2010.

[WSH93] Steven C Woo, Jaswinder P Singh, and John L. Hennessy. The performance advantages of integrating message passing in cache-coherent multiprocessors. Technical report, Stanford, CA, USA, 1993.

[WZPM02] Hang-Sheng Wang, Xinping Zhu, Li-Shiuan Peh, and Sharad Malik. Orion: a power-performance simulator for interconnection networks. In *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 35, pages 294–305, 2002.

[YH02] Xifeng Yan and Jiawei Han. gspan: Graph-based substructure pattern mining. In *ICDM*, pages 721–724, 2002.

[YYS+04] Mohamed-Wassim Youssef, Sungjoo Yoo, Arif Sasongko, Yanick Paviot, and Ahmed A. Jerraya. Debugging hw/sw interface for mpsoc: video encoder system design case study. In *DAC*, pages 908–913, 2004.

[Zak98] Mohammed Javeed Zaki. Efficient enumeration of frequent sequences. In *Proceedings of the International Conference on Information and Knowledge Management*, pages 68–75. ACM, November 1998.

[Zak00] Mohammed J. Zaki. Scalable algorithms for association mining. *IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING*, 12:372–390, 2000.

[Zak05] Mohammed Javeed Zaki. Efficiently mining frequent trees in a forest: Algorithms and applications. *IEEE Trans. Knowl. Data Eng.*, 17(8):1021–1035, 2005.

[ZB91] Marco Zagha and Guy E. Blelloch. Radix sort for vector multiprocessors. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, pages 712–721, 1991.

[ZXHW10] Jia Zou, Jing Xiao, Rui Hou, and Yanqi Wang. Frequent instruction sequential pattern mining in hardware sample data. In *ICDM*, pages 1205–1210, 2010.