POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

CPU utilization improvement of multiple-core processors through cache management and task scheduling

MAHDI MORADMAND BADIE
Département de génie électrique

Mémoire présenté en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
Génie électrique

Août 2019

© Mahdi Moradmand Badie, 2019.

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Ce mémoire intitulé :

CPU utilization improvement of multiple-core processors through cache management and task scheduling

présenté par **Mahdi MORADMAND BADIE**
en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
a été dûment accepté par le jury d'examen constitué de :

Guchuan ZHU, président
Yvon SAVARIA, membre et directeur de recherche
Jean-Pierre DAVID, membre et codirecteur de recherche
François LEDUC-PRIMEAU, membre

ACKNOWLEDGEMENTS

I would like to express my deep-felt gratitude to my supervisor, Prof. Yvon Savaria, and special thanks to my co-supervisor Prof. Jean-Pierre David, in the Department of Electrical Engineering at Polytechnique Montréal in Canada, for their advice, encouragement, enduring patience and support.

Additionally, I would like to express my appreciation to Professor Guchuan Zhu in the Department of Electrical Engineering as a leader of our project, for his support and encouragement. Also I want to say thanks a lot Professor François Leduc-Primeau for their time and effort putting on the evaluation of my thesis. Also I just wanted to say thanks to my team mates, Meng Li and Michel Gemieux, for their helps contributions and collaborations that providing me the means to complete my degree.

I would like to thanks Huawei Technologies Canada Co. Ltd and of the Natural Sciences and Engineering Research Council of Canada for their financial contributions in support of this project.

And finally, I wish to express my love and gratitude to my wife; for her understanding and endless love, through the duration of my studies.

RÉSUMÉ

De nos jours, les architectures multicœurs et multiprocesseurs sont largement utilisées dans les centres de données. Une telle utilisation fournit les performances requises pour diverses tâches, telles que le C-RAN(accès radio par info-nuagique). Le traitement du signal sans fil en bande de base(wireless baseband) pour les normes 4G et 5G désigne un ensemble de tâches qui doivent être exécutées dans un intervalle de temps spécifique. Par exemple, la pile de liaison montante(up-link stack) pour une station de base 4G virtualisée a été décomposée en plus de 1000 tâches exécutables en 5 ms. Avec la 5G, la latence cible dans un scénario de bout en bout avec utilisation à latence très faible(ultra-low latency) est de 1 ms, tandis que la complexité de calcul est d'un à deux ordres de grandeur plus élevée que celle de la 4G. Le défi à surmonter c'est de répondre à ces objectifs en terme de complexité de traitement. Pour ce faire, il est crucial de caractériser la variabilité du temps de traitement par rapport aux caractéristiques du modèle de mémoire afin de garantir un temps de traitement donné dans les grappes d'ordinateurs classiques.

En outre, la planification des tâches sur les systèmes multicœurs reste un problème ouvert. Un tel problème doit être analysé afin d'utiliser pleinement la capacité de traitement d'un système multicoeur d'un système multicoeur et d'atteindre une faible latence. Afin de remédier à l'utilisation inefficace des cœurs de processeur, un schéma d'ordonnancement des tâches basé sur la mise en file d'attente, qui se focalise sur le calcul parallèle local, est proposé. Dans cette mémoire, on introduit la gestion multi-files pour la planification dynamique des tâches afin de cibler une utilisation complète à 100% des cœurs de CPU locaux pour des tâches d'entrée suffisantes. Plusieurs simulations sont faites pour vérifier le schéma de planification des tâches proposé. Les résultats rapportés confirment sa viabilité et son efficacité.

De plus, l'utilisation de la mémoire cache est l'une des principales sources de variabilité du temps d'exécution. En outre, une gestion inefficace de la mémoire cache s'avère problématique dans les systèmes avec WCET. Une approche efficace de gestion de la cache doit prendre en compte simultanément la planification des tâches et la gestion de la cache. L'approche optimale de gestion de la cache oblige de manière critique à prendre en compte les priorités associées à toutes les tâches; la connaissance de ces priorités est essentielle pour détecter et éviter les goulots d'étranglement dans le système. Cette approche affecte des ressources suffisantes à une tâche aussi critique pour faciliter une meilleure gestion. On commence par l'introduction d'une méthode de test simple, évolutive et configurable appelée un tableau de compteurs(Array of Counters), dont le but est de caractériser les variations de temps de traite-

ment des architectures multicœurs. Cette technique aide à trouver les goulots d'étranglement. Un tel outil est utile pour l'élaboration d'un algorithme de gestion de la cache plus optimisé et amélioré.

L'objectif principal est d'améliorer l'utilisation des processeurs multicœurs en gérant mieux les ressources disponibles. De plus, les lacunes de l'approche dans la littérature sont brièvement décrites et étudiées. La puissance et l'efficacité de l'approche tableau de compteurs contribue à la recherche et à l'évaluation du WCET et à l'identification des goulots goulots d'étranglement.

ABSTRACT

Nowadays, modern multiprocessor and multicore architectures are widely used in data centers. Such usage provides the required performance for a variety of tasks, such as the C-RAN(Cloud-Radio-Access-Network). Wireless baseband signal processing for the 4G and 5G standards designates a range of tasks which must be executed in a specific time slot. For instance, the up-link stack of one 4G virtualized-base station was decomposed in more than 1000 tasks executable within 5ms. In 5G, the expected target latency for ultra-low latency use cases is 1ms in an end-to-end scenario; while the computational complexity is expected to be one to two orders of magnitude higher than that of 4G. It remains to be seen whether and how reaching such computational complexity is feasible. It is a crucial factor to characterize processing time variability besides features of memory model to guarantee a given processing time in mainstream computer clusters.

Besides, the task scheduling on multicore systems is still an open issue. Such a problem needs to be analyzed in order to fully utilize the processing capacity and to achieve low processing latency. In order to tackle the inefficient utilization of CPU cores, a queueing-based data-driven task scheduling scheme, which focuses on local parallel computing, is proposed in this thesis. This thesis introduces multi-queue management for dynamic task scheduling to target 100% utilization of local CPU cores for sufficient input tasks. Finally, the thesis entails several simulations to verify the proposed task scheduling scheme. The reported results confirm its viability and efficiency.

Moreover, cache memory usage is one of the primary sources of execution time variability. Besides, inefficient management of cache memory proves to be problematic in systems with which WCET(Worst-Case-Execution-Time) is of concern. An efficient cache managing approach needs to take both task scheduling and cache management into account simultaneously. Optimal cache-management imposes considering priorities associating with all tasks; the knowledge of such priorities is essential for detecting and avoiding system bottlenecks. Such approach proposes allocating adequate resources to such a critical task to facilitate better management. The work starts with the introduction of a simple, scalable, and configurable test method called an Array of Counters, the purpose of which is to characterize the processing time variations of multicore architectures. The technique helps to find system bottlenecks. Such help is conducive to a more optimized and enhanced cache-management algorithm.

The primary objective of this research is to enhance the utilization of multicore processors

by better managing the resources at hand. Also, the shortcomings of the state-of-the-art approaches are briefly discussed and investigated. The powerful and efficient method of Array of Counters contributes to finding and evaluating the WCET, aka bottlenecks.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS iii

RÉSUMÉ iv

ABSTRACT vi

TABLE OF CONTENTS viii

LIST OF TABLES x

LIST OF FIGURES xi

LIST OF SYMBOLS AND ABBREVIATIONS xiii

LIST OF APPENDICES xiv

CHAPTER 1 INTRODUCTION 1
- 1.1 Problem statement 1
  - 1.1.1 General definitions 2
  - 1.1.2 Data-centers and cloud computing 3
  - 1.1.3 5G communication systems and C-RAN 5
- 1.2 Research objectives and contributions 8
- 1.3 Thesis outline 9

CHAPTER 2 LITERATURE REVIEW 10
- 2.1 Introduction 10
- 2.2 Processing time variability 10
- 2.3 Cache allocation technology 11
  - 2.3.1 Cache locking technique 11
  - 2.3.2 Advantages and disadvantages of cache locking 13
- 2.4 Task scheduling 15
- 2.5 Conclusion 16

CHAPTER 3 CHARACTERIZING PROCESSING TIME VARIABILITY 18
- 3.1 Array Of Counters(AOCs): A generic scalable micro-benchmark to characterize processing time variability 18

3.1.1 Best/Worst Case Execution Time(BCET/WCET) of a program . . . 20
3.1.2 AOCs as a micro-benchmark . . . 21
3.2 Multi-level dynamic cache management . . . 21
3.3 Experiments and results . . . 26

CHAPTER 4 PRIORITY-QUEUE-BASED DATA-DRIVEN TASK SCHEDULING 32
4.1 Data-Driven task scheduling . . . 32
4.1.1 Queue-based data-driven task scheduling . . . 32
4.1.2 General definition . . . 35
4.1.3 Task allocation based on DAG . . . 35
4.1.4 Optimized queue-based data-driven task scheduling . . . 38
4.1.5 Software + Hardware task scheduling in data-driven concept . . . 43
4.2 Experiments and results . . . 45
4.2.1 The first implementation of data-driven task scheduling . . . 46
4.2.2 The optimized version of data-driven task scheduling . . . 47

CHAPTER 5 CONCLUSION AND RECOMMENDATIONS . . . 68
5.1 Advancement of knowledge . . . 68
5.2 Limits and constraints . . . 70
5.3 Future work and recommendations . . . 70
5.3.1 Multi-level dynamic cache locking . . . 70
5.3.2 Preemptive queuing-based data-driven task scheduling . . . 71
5.3.3 Refine AOCs to improve its accuracy for processing time variations on multi-core processors . . . 71
5.4 Perspective ideas for potential papers . . . 72
5.4.1 Contention-free method to reduce processing time variations of cloud applications on multi-core processors . . . 72
5.4.2 Multi-level dynamic cache allocation to reduce worst case execution time in multi-core processors . . . 72
5.4.3 Local queuing-based data-driven task scheduling on multi-core system 72

REFERENCES . . . 74

APPENDICES . . . 78

LIST OF TABLES

Table 4.1 Task Package Profile 35
Table 4.2 Data Package Profile 35
Table 4.3 Task Grouping 36

LIST OF FIGURES

Figure 3.1 CAT provides more cache space for high priorities applications 22
Figure 3.2 Example of cache capacity bitmasks 23
Figure 3.3 Multi-level dynamic cache locking. 24
Figure 3.4 The general format of the gathered data, as processing time varies according to different levels of cache and memory. 27
Figure 3.5 Various results that characterize the impact of changing the size of the array for a variable number of active logical cores. For each curve, the dotted lines stand for best/worst processing time. 28
Figure 3.6 Representative histogram of the observed processing time(1000 runs on 22 hyper-threaded cores for 1.35 MB memory block size. 29
Figure 3.7 The mean, WCET and BCET in latency observed when executing in hyper-threaded mode(blue) vs. single-threaded mode(red). 30
Figure 3.8 The latency of hyper-threaded core(blue) vs. single-threaded core(red) using AVX2 instructions. 31
Figure 3.9 The latency of hyper-threaded core for the x^2+1 function with different repetition values. 31
Figure 4.1 An example of real DAG from Huawei Co. 33
Figure 4.2 Simplified DAG of Figure 4.1 34
Figure 4.3 Queue-based data-driven task scheduling 37
Figure 4.4 The structure of priority-based queue management in a local blade. . 38
Figure 4.5 Flowchart of Quebe-based Data-Driven Task Scheduling. 39
Figure 4.6 Example of optimized version for first version. 40
Figure 4.7 addTask function parameters. 42
Figure 4.8 An example of initDag function with 13 tasks. 43
Figure 4.9 Software + Hardware Task Scheduling in the Data Driven Concept . 45
Figure 4.10 Core utilization as reported by LTTng, dark blue stands for busy state and green for wait/idle state. 46
Figure 4.11 CPU usage of real DAG with total 24 workloads, 3 concurrent workloads, and 24 active cores. 47
Figure 4.12 Optimized version of Task Scheduling vs. The first version 48
Figure 4.13 Results for Task #0. 48
Figure 4.14 Results for Task #1. 49
Figure 4.15 Results for Task #2. 49

Figure 4.16 Results for Task #3. 50
Figure 4.17 Results for Task #4. 50
Figure 4.18 Results for Task #5. 51
Figure 4.19 Results for Task #6. 51
Figure 4.20 Results for Task #7. 52
Figure 4.21 Results for Task #8. 52
Figure 4.22 Results for Task #9. 53
Figure 4.23 Results for Task #10. 53
Figure 4.24 Results for Task #11. 54
Figure 4.25 Results for Task #12. 54
Figure 4.26 1.Busy and Idle states, 4 workers and 4 workloads, from start to stop point. 59
Figure 4.27 2. Busy and Idle states, 4 workers and 4 workloads, start point. . . . 60
Figure 4.28 3. Busy and idle states, 4 workers and 4 workloads, stop point. 60
Figure 4.29 Core utilization with 13 Tasks, 4 workloads 62
Figure 4.30 Core utilization with 13 Tasks, 4 workloads 63
Figure 4.31 Core utilization with 13 Tasks, 7 workloads 63
Figure 4.32 Core utilization with 13 Tasks, 8 workloads 64
Figure 4.33 Core utilization with 13 Tasks, 15 workloads 64
Figure 4.34 Core utilization with 13 Tasks, 30 workloads 65
Figure 4.35 Core utilization with 13 Tasks, 40 workloads 65
Figure 4.36 Core utilization with 13 Tasks, 50 workloads 66
Figure 4.37 Core utilization with 13 Tasks, 50 workloads 66
Figure 4.38 Core utilization with 13 Tasks, 100 workloads 67
Figure A.1 Adding Directives. 80
Figure A.2 Solution Configuration. 81
Figure A.3 Latency Constraints. 82
Figure A.4 Loop Unrolling. 82
Figure A.5 Loop Flattening. 83
Figure A.6 Loop Merging. 83
Figure A.7 Dataflow Optimization Command. 84

LIST OF SYMBOLS AND ABBREVIATIONS

C-RAN	Cloud-Radio Access Network
GSM	Global System for Mobile communication
LTE	Long-Term Evolution
FPGA	Field Programmable Gate Array
IT	Information Technology
IoT	Internet of Things
BTS	Base Transceiver Station
DAG	Directed Acyclic Graph
OS	Operating System
AOC's	Array Of Counters
LIC	Low-Interference Configuration
HIC	High-Interference Configuration
WCET	Worst-Case Execution Time
BCET	Best-Case Execution Time
PTC	Processing Time Characterization
PTV	Processing Time Variation
CAT	Cache Allocation Technology
CAM	Cache Allocation Management
CL	Cache Locking
SCL	Static Cache Locking
DCL	Dynamic Cache Locking
COS	Class Of Services
SWCL	Software-based Cache Locking
HWCL	Hardware-based Cache Locking
MLDCA	Multi-Level Dynamic Cache Allocation
DPDK	Data Plane Development Kit
WL	Work Load
IAOC's	Interference Array Of Counters
LTTng	Linux Trace Toolkit Next Generation
HLS	High-Level Synthesis
SOC	System On the Chip
AVX	Advanced Vector Extension
SIMD	Single-Instruction-Multiple-Data

LIST OF APPENDICES

Appendix A Introduction to Vivado and Vivado HLS 78

CHAPTER 1 INTRODUCTION

Nowadays, we are in a world surrounded by accessible information and technology. We need to constantly connect and want everyone of our move to be recorded, processed, analyzed and found in the "Cloud" [1]. Thus, this project is located in the cloud world, more precisely at the data-center level. We seek to improve the processing of tasks in a Cloud Radio Access Network(C-RAN) operating environment, focused on CPU utilization and timing deadline. To do this, we want to exploit the benefits of a data-driven task scheduling, characterizing processing time variability and cache management.

Better task scheduling as well as reducing the Worst-Case-Execution-Time(WCET) in the C-RAN context would allow a more efficient and adequate use of a computing clusters. This research aims to deploy systems that consume substantially less energy and for which the costs of installations are reduced, which can reduce negative impacts on the planet.

In this introduction we will first define the problem statement concepts associated with the research project documented by this thesis. Then, we will expose the general definition and the basic concepts such as data-center and cloud computing, 5G communication systems and C-RAN. Afterward, the different research objectives followed throughout this master's degree will be listed. Finally, the organization of the different chapters and the contents of this thesis will be summarized.

1.1 Problem statement

Huawei's Radio Computing Architecture (RCA) team is actively working on various technologies related to C-RAN and it supports the research by providing the constraints of the LTE software stack and possibly other use cases. Huawei's research team (CRC, Canada Research Centre) is designing and developing a C-RAN prototype and Polytechnique worked closely with Huawei to characterize and improve the platform. This research is a joint Collaborative Research and Development(CRD) project financially supported by the Natural Sciences and Engineering Research Council of Canada. The main research goal of Huawei is to investigate new computing and communication architectures for C-RAN base station virtualization. In this regards, our problem statement consists of a two-step questioning. First, we want to study the impact of processing time variations not only to control them but also to reduce the WCET. Second, we ask ourselves which algorithm help us to increase the CPU utilization in order to get a maximum benefit of each active core. The research questions are:

1. What is the impact and source of resource utilization on the processing time variation and reducing the WCET?
2. Can we determine a scheduling algorithm to increase the CPU utilization for active cores?

1.1.1 General definitions

The number of cores on a chip increases significantly over time in modern real-time systems. Such increase translates into a corresponding increase in the complexity of the systems containing such chips. Besides, this increase results in a substantially-elevated number of tasks running on these cores. This mandates new management techniques to address the higher numbers of tasks and a higher complexity. On the one hand, the transition from uniprocessor to multiprocessors in real-time systems poses a challenge. Such a challenge, often, rises from synchronization limitation between different processors in the same or in different systems. It is evident that applying the same scheduling algorithms, similar to those used in uniprocessors, for multiprocessors undermines the performance. On the other hand, multicore processors dominate the commercial marketplace.

Applications and systems should use proper strategies concerning parallelism and functionality between cores to take maximum advantage of multicore processors technologies. In parallel computing, speed-up is generally achieved by running multiple more or less independent tasks simultaneously. Thus, a paradigm is required to represent the inter-task dependencies to utilize the computing resources efficiently.

In addition, the overall energy consumption of modern multiprocessors and multicores is considerable when they are idle. In many applications, the CPU's are idle much of the time, so power consumption when idle contributes significantly to overall system power usage. Task scheduling for multicore processors is one of the main factors determining the utilization of multicore processors. Modern task scheduling strategies in real-time systems mainly focus on uniprocessors, and some of these strategies cannot be generalized to multicore processors system efficiently. A task schedule model can often be described by a Directed Acyclic Graph(DAG). Because the DAG model is used to schedule tasks in multicore systems as well, DAGs play a central role in various proposed state-of-the-art algorithms. The DAG is a common model for task scheduling. It is a dominant model used in parallel computation. In general, a DAG model represents tasks as a graph with vertices representing dynamic functions, computation cost, edges between vertices representing dependencies, and communication cost. A critical path(the path in this study is a finite sequence of edges in a DAG

that joins a sequence of distinct nodes) of a DAG represents the longest path concerning time consumption or computation cost.

1.1.2 Data-centers and cloud computing

Sometimes it is thought that data-centers and cloud are two terms that have exactly the same meaning, unless you are in a business that is related to these services. But these are two different words. the cloud is basically Internet based, but a data-center is based on physical location. In the following we briefly describe and discuss data-centers and cloud.

1. Data-centers: The term data-center can be interpreted in several ways. First, the data-center is housed in a company and IT professionals are hired to keep an eye on it and ensure its operation. Second, the data-center can be an offsite storage center where servers and other storage components are used to make the data stored physically and virtually available.

    - Advantages: Organizations that have internal data-center access in their company have much less internet connectivity. As long as the local network is stable, data access is maintained. Remote storage also has its own advantages. If the primary location of the organization is affected by a fire, theft or flood, the company's second location remains completely intact and may be accessed.
    - Disadvantages: Keeping your data in one place makes it easier for people who are not approved to access your data physically and virtually. Depending on your organization's budget, maintaining an organization-wide data-center can cost you a great deal.

2. Cloud-Computing: Before the advent of the Internet, there was no cloud computing and it could not exist. Today, as data speeds up in the world, some people predict that data-centers will decline in 2018 and cloud computing will grow. But what is Cloud? An online storage system that is used to parse and store your data in multiple locations. With Cloud, we always make sure there is a backup of our backups. The only way to eliminate the data that exists on the cloud is to destroy the Internet itself.

    - Advantages: In today's world where communication is increasing day by day, Cloud is the ultimate solution. Services like Microsoft Office 365 and Google Drive have understood the importance of storing data online and moving to the full potential. Your organization can treat information the same way to make it always available. With online access, data will never be out of reach unless

you have internet access. The following are six common reasons organizations use cloud computing:

(a) Cost: Cloud computing eliminates the cost of purchasing software, hardware, installing and deploying site data-centers or server racks (daily electricity to power and cool them) as well as the need for IT(Information Technology) experts to manage infrastructure, which speeds up work.

(b) Speed: Most cloud computing services are self-service based on needs, so large amounts of computing resources can be provided in a matter of minutes and with just a few mouse clicks, a highly flexible business is possible and planning pressure is reduced.

(c) World scale: One of the benefits of cloud computing services is the flexible scale. In the term cloud, this means that the exact amount of IT resources (such as more or less computing power, storage and bandwidth) are delivered right when needed and from a geographic location.

(d) Productivity: Data-centers require a lot of racking, including hardware installation, software patching, and other day-to-day IT management tasks. Cloud computing eliminates the need for many of these tasks, so IT teams can spend a great deal of time on achieving important business goals.

(e) Performance: The largest cloud computing services run on a global network of secure data-centers that are constantly upgraded to the latest, fast and efficient computing hardware version. This brings many advantages to a single data-center in a large company, which includes reducing network latency for apps and saving on a larger scale.

(f) Reliability: Cloud computing makes data backup, data retrieval and business continuity easier and cheaper because it can reflect data elsewhere in the cloud provider's network.

- Disadvantages: Everything online is more vulnerable to cyber-attacks. It's very simple: a hacker tends to isolate a cloud storage rather than a data-center. Cloud is also less powerful than data-center because of its online nature. Compare Microsoft Office offline with Google Docs Online. Although Google Docs works well, it doesn't have the power of Microsoft Office that you can access offline.

In the areas of "Cloud Computing" and "High Performance Computing (HPC)", an efficient scheduling of the tasks of a process is essential in order to have the greatest possible performance. There are many literature on the subject of task scheduling, so what we can deduce,

there is not a perfect solution that has the ability to solve all the problems. Thus, instead of trying to find the ultimate scheduling solution, designers rely on algorithms or strategies, which they choose based on the characteristics of the application.

1.1.3 5G communication systems and C-RAN

Cloud-RAN or Centralized-RAN [2], is a new type of network architecture aimed at improving the systems currently used in mobile communications. C-RAN, is a mobile network architecture that is expected to be a cornerstone of 5G. C-RAN derives from traditional Radio Access Network (RAN), which was built with multiple BTS(Base Transceiver Station) across a region. A BTS, covers a small area at a time, and the system includes everything needed for wireless communications, from GSM(Global System for Mobile communication) 2G to 3G and afterwards from 3G to 4G LTE(Long-Term Evolution).

• What is the 5G?

The term 5G means "5th Generation"; this technology is the fifth generation of cellular communication. The design is so fast that it is much faster than 4G LTE technology(in 5G, the delay will be 4 milliseconds, which is much less than 20 milliseconds for 4G technology). The purpose of this new standard, however, is not merely to speed up the internet connection of smartphones. This standard will provide high-speed wireless internet everywhere and for everything including connected cars, smart homes and IoT(Internet of Things) tools.

• Why should we be eager for 5G?

1. New Realities: The widespread release of 5G makes virtual reality and augmented reality more widely used. Augmented reality allows users to access a lot of information - for example, simply identify their route; identify price tags on products and bar-codes; Virtual reality, on the other hand, will provide a completely artificial perspective. An important point in using virtual reality and augmented reality is that both require very fast internet connection.

2. High Speed: Download speeds up to 150 Mbps at 4G, while 5G speed are 10 Gbps. It's so fast that you can download a full movie in just 4 seconds.

3. Quick Response: A few seconds delay is not so important when initializing online video, but this delay is unacceptable for a self-driving car(also known as an autonomous car, driver-less car, or robotic car), where milliseconds are important. Using 4G connectivity, a smart car takes approximately 15 to 20 milliseconds to notify the smart car behind it that it has compressed the brake; this amounts is 1 millisecond in a 5G con-

nection. Overall, as download speeds have increased dramatically, the initial delay in this type of communication has been eliminated as much as possible.

In the future, the smartphones and all devices with cellular connectivity will use the 5G standard instead of 4G LTE technology.

Modern multiprocessor and multi-core architectures are widely used providing needed performance for relative tasks, such as C-RAN. Wireless baseband signal processing for the 4G and 5G standards specifies many tasks that must be executed in a particular time slot, related to the duration of Radio Frames. In [3], the up-link stack of one 4G virtualized base station was decomposed in more than 1000 tasks, all of which are expected to end within 5 ms. In 5G, the considered target latency for ultra-low latency use cases is 1 ms in an end-to-end scenario, while the computational complexity is expected to be one to two orders of magnitude higher than that of 4G. The characterization of such systems is of particular concern because their characteristics are affected by variations of the access time to shared-resources in a way that can be detrimental to C-RAN. For instance, in multicore architectures with multi-task functions, the cores compete to access the memory resources and therefore could interfere with each other in doing so, resulting in sub-optimal performance.

Improving processor utilization in order to decrease the number of active cores is the main concern for modern multiprocessor and multi-core architectures in data-centers. We investigated a method that not only exploits the advantages of the existing algorithms, but, that also points out to new ideas for improving the task scheduling effects on multiprocessors. The original concept is proposed in [4],

and Professor David had a main role to finalize the optimized version. The author contributed to the validation of this method by implementing a prototype confirming its efficiency.

The concept of Directed Acyclic Graph(DAG) related to task scheduling approach, is used in this thesis to build a task model, such that task dependency, task priority, and task WCET can collectively form priority metrics. A priority-based task scheduling list can be set up from a comprehensive analysis and by calculating the priority for each task. Then queue-based and data-driven task allocation strategies are employed to map tasks to processors. Such strategies can improve processor utilization and timing predictability using cache management. Cache management leads to allocating a more significant part of a shared cache to high priority tasks.

The WCET is dramatically increased in modern multi-core processor due to interference caused by shared resources in software and in hardware. The obvious case in software is a preemptive OS(Operating System) scheduler that could interrupt task execution. On hardware

side, shared cache, shared processing units or automatic power saving mode are hardware features that also impact the WCET. One of the most significant benefits of cache management(cache sharing) is enhancing core utilization. That is to say, when one core is not in a running state, the other core can access the whole or part of the shared cache. The benefit of cache sharing is more evident when hyper-threading used. In multicore processor cache sharing can increase efficiency and performance by reducing the number of cores needed to perform a task by improving the utilization of each core.

One of the main concerns of the Huawei team is having a framework that will target task execution in a reserved (locked) cache blocks and minimize cache misses in reserved cache memory area. The prototype computing platform uses Intel Xeon server processors. A hardware cache-locking mechanism is available in many multi-core processors but it is not yet available on the Intel server processor.

Cache locking is a useful strategy in real-time systems that improves timing predictability in cache management concept. Cache locking, if used properly, can enhance the performance of modern processors. Hence, cache locking is an essential technique not only as a means to mitigate execution time variation, but also as an a means to improve performance.

We propose a flexible priority-based multi-level dynamic cache locking approach in order to keep the balance between high and low priority tasks. Considering WCET and BCET, we reserve cache slots to given high-priority tasks. Obtaining near deterministic processing time of tasks executing on multicores processors with dynamic cache management, and reducing the WCET is the main goal of this approach.

Our strategy is a priority-based approach that incorporates a top view to decide on the number of locking slots at each entry-point and then selects the memory blocks to be locked for each level based on prioritized tasks. In effect, it is necessary to scheduled tasks to keep resource utilization balanced, which leads to shared resource management and dynamic cache locking. In effect, a task scheduling algorithm for multicore processors is critically needed in addition to an effective cache management strategy.

The relations between the two requirements of cache sharing and task scheduling is an important and challenging issue in improving utilization of multicore processors. The widespread adoption of multicore processors poses a few critical challenges both in research and in software development. Multicore CPUs will require a new generation of applications to fulfill and maximize their performance potential; applications that are specifically designed and implemented for multicore processors.

A significant challenge in cache management design is to limit the worst case processing

time and its variations. The main impact of cache misses in processing time is leading to processing time variability. If we could avoid provoking cache misses so we can control the processing time variability. Key resources that influence the processing time are the shared Last Level of Cache(LLC) and external memory. This sets a need to characterize the access time associated with each task that directly impacts processing time distribution and the WCET of relevant tasks.

We introduce a simple, scalable, and configurable test method called an Array of Counters(AOCs) to characterize the processing time variations of multicore architectures. The first idea of AOCs proposed by Professor David(thesis advisor) who had a significant role this method development and validation.

An AOCs is a tool to measure execution time variability. Besides, we should notice that AOCs have the potential of being extended to micro-benchmarks to measure and identify more memory parameters of CPUs in addition to those in the status quo version. We propose a novel procedure to characterize the impact of resource utilization and computation time of concurrent tasks on the processing time variation. The procedure leads to a better understanding of factors controlling processing time variations.

The proposed procedure consists of using AOCs where data and computation time can be set independently from each physical resource. The application is simple, scalable, and controllable, while producing measurable results concerning the average, maximum(WCET), and minimum(BCET) processing time obtained when using different levels of cache and memory. An AOCs is a set of 32-bit unsigned counters stored in various memory blocks. The array size is a parameter that can be changed to induce misses at every level of cache and memory. The simple atomic increment operation in the counters can also be replaced by a computing loop with no memory access to stretch the computation time or increase its complexity as desired.

1.2 Research objectives and contributions

In this thesis, we introduce a method to improve the utilization of multicore processors considering two main objectives: cache management and task scheduling. This research focuses on these two objectives addressing the following three fundamental challenges:

- Process-Time Variations: By introducing a novel method, the Array Of Counters (AOCs), we can characterize the impact of resource utilization and computation time of concurrent tasks the processing time variations.

- Resource Sharing: Using dynamic cache management both the instruction and data cache are considered to keep a proper balance between high and low priority tasks.

- Task Scheduling: Validate a queue-based data-driven task scheduling algorithm to increase the utilization of cores in such a way to reduce the number of active cores in order to improve each core's utilization.

1.3 Thesis outline

The outline of this thesis is as follows. In Chapter 2 the literature reviews will explain in more detail all the concepts used throughout this document while keeping a general view. In Chapter 3 will study the AOCs as a generic scalable micro-benchmark, and cache management strategy to enhance the WCET. In the Chapter 4 task scheduling will be studied. The main focus in this chapter is presenting the technique for designing task schedulers exploiting the data-driven concept. In addition, at the end of both Chapter 3 and Chapter 4, we will see the modalities of our proof of concept by presenting the plan, the test method and the experimental results obtained. Finally, we will summarize the work done throughout this thesis in Chapter 5, and discuss the results obtained. Several future works will be proposed at the end of this chapter.

CHAPTER 2 LITERATURE REVIEW

2.1 Introduction

Here, we review factors that influence processing time variability, cache management, and the task scheduling problem. In each case, the opportunities, obstacles, and challenges are also briefly discussed. The primary objective of this research is to improve the utilization of multicore processors. Existing approaches are also prone to several shortcomings, which are briefly addressed in the study.

2.2 Processing time variability

Processing Time Characterization(PTC) is a measurement method which can be used to identify the input/output behavior of systems with respect to their ability to meet some execution time targets. Via PTC, it is feasible to estimate the robustness of systems with respect to their ability to meet some timing. Also, PTC allows building models to describe the relationships between the parameters influencing processing time. Accurately measuring the effective process-time is the concern of a rich literature, [5]. Agarwal and Sharma [6] proposed a new method to compute effective process times from a data set. They estimated the mean, as well as the variance of given processing time for upcoming workloads. The results of this characterization process can be used to analyze and improve applications processing time [7, 8].

There are different sources and types of variation for processing time. The two most important classes are controlled and uncontrolled variations. The conceptualization of controlled and uncontrolled sources of processing time variation is mandatory. The key difference between the two classes is whether or not the process of interest varies significantly over time or some other condition that itself may vary over time. A "steady stable" process is the one that runs in a consistent, robust, and predictable manner so that the processing time value is essentially constant, and therefore the variability is under control. On the contrary, If the process varies over time, it may lead to some unpredictable situation concerning the resources used by the process; thus a time-variant process falls into the uncontrolled variation category. Stable processes only exhibit controlled variations [5]. In order to compare the time variations of processes, we need a flexible and straightforward benchmark. The authors in [9] proposed an algorithm to characterize task memory access on multicore architectures which is hard to implement.

To reduce the timing variation in [10], the authors propose a technique which uses some low-overhead instruction to measure cache and memory latency. Prior research such as store pre-fetching [11] have limited applicability and merely investigate several trivial sources of variations.

It is possible that some inter-cluster interference can be alleviated substantially. However, the potential risk for interference between cores on the same cluster running and using the shared resources still exists [12]. Using the Worst-Case-Execution-Time(WCET) approach that is independent of co-runner interference, which is used in [12], may affect the efficiency of the application. Besides, we should consider that real interference from co-runners depends on the scheduling technique at hand. Evaluating the WCET of the program may pose some challenges. Also, there is no guarantee to reproduce the maximum interference scenario even when we know everything about the setting of the system and of the application.

We propose a novel procedure, that can be considered as a type of microbenchmark, to characterize the impact of resource utilization and computation time of concurrent tasks on processing time variations. This procedure leads to better understanding of processing time variations particularly when a shared cache is involved. The proposed application is simple, scalable and controllable, while producing observable results with respect to the observed processing time obtained when using different levels of cache and memory. The main drawback of applications found in the current literature, is their complexity, lack of scalibility and controllability features.

2.3 Cache allocation technology

Cache locking is a means to protect access to some or all instructions or data needed by multiprocessors sharing a cache. It can be useful to improve predictability in real-time systems so it can improve the performance of modern processors if implemented in a technically sound manner. Cache locking may improve the task execution performances, and as a result, it can help reduce task processing time variability. In this regards, non appropriate using of cache locking can cause processing time variations.

2.3.1 Cache locking technique

Cache Allocation Technique(CAT) and Cache Locking(CL) are techniques that might be appropriate to all or a portion of the shared cache between processors. Many recent modern processors support cache locking, e.g., the one embedded in Intel Xeon E5-2650 V4. By choosing accurately the memory slots for locking, CAT can improve performance dramat-

ically. For instance, in the strategies proposed in [13, 14], the authors recommended two distinctive heuristic algorithms, line locking, and way locking, to improve the performance of the cache. They emphasize that CL is a an effective procedure for improving the processing time.

Cache locking methods can be grouped into two main classes, static and dynamic. In Static Cache Locking(SCL), the locked memory slots do not change during the execution time. So SCL is not a reasonable methodology when distinctive memory slots are fighting for a memory. Dynamic Cache Locking(DCL), as its name implies, can alleviate some restrictions observed with SCL. The DCL approach partitions a program into Classes of Service(COS) dynamically during processing time. DCL adjusts the locked contents at runtime, which improves the WCET in contrast with SCL. DCL gives better results as far as performance and flexibility yielded by SCL is concerned [15].

The classification of CL methods applies in both cases of software and hardware implementation. In which case they are called Software Cache Locking(SWCL) or Hardware Cache Locking(HWCL). SWCL focuses on theoretical algorithms and software coding implementation. By contrast, HWCL uses available features embedded on modern CPUs. Obviously, in HWCL, some software coding is needed, but the software complexity and implementation can reduce dramatically in comparison to SWCL.

Most existing CL strategies go for improving the WCET by utilizing SCL, as mentioned in [15]. A key issue is the simplicity of implementation of SCL versus DCL. SCL generally uses full CL, but full locking does not permit the unlocked cache slots to be shared. Not utilizing all the cache leads the under-exploited locality, which prompts a negative effect on the general WCET and can dramatically degrade performance [15]. Besides, with large programs, utilizing SCL on a small cache can lead to increased processing time.

Most of the proposed CL approaches concentrates on instruction caches rather than on data caches [16, 17], because usually during execution, the instructions remain fixed, yet the data may vary from one cycle to another [18, 19]. So applying CL for data is a more complicated problem; in the same way the majority of the work done does not address data cache, as mentioned in [15].

In the instruction cache, in order to reduce the WCET execution, Puaut and Arnaud [20] proposed a method which relies on partitioning the regions for the ordered tasks just for the instruction cache. In spite of the fact that their methodology decreased the WCET, it expanded the loading cost. In [21], Ding et al. proposed the space-share cache management approach, which assigns a part of the cache based on to task necessities and priority. Even though each running task just uses a segment of the cache, as this portion is fixed during

execution, such fixed part allocation may lead to cache line misses in DCL [15].

In order to have a good trade-off between predictability and performance, in [19], Vera et al. present a method with DCL. They used a procedure that depends on the compiler to distinguish dependencies between data segments in different code regions. They outline how to mix the cache analysis model and CL approach to ensure that all contention between running tasks can be predictable. Their results show that even though they accomplished some reduction in the WCET, they encountered some degradation in performance [15, 19]. Additionally, their methodology does not address instruction cache.

The authors of [22], Zheng and Wu, propose two methods to reduce the WCET of task. The first approach is based on picking the possible data cache. Their reported results show that data dependencies limit this methodology. The second methodology attempts to improve over the first by exploiting data structure expressed by a non-cyclic task graph. The result of the former methodology exhibits improvements in WCET but faces difficulties on data cache utilization [15, 22].

After reviewing some significant literature on cache locking method, we are going to discuss the advantages and disadvantages of cache locking, which will lead us to the proposed technique.

2.3.2 Advantages and disadvantages of cache locking

Cache locking has a few potential advantages. The first and potentially the most basic benefit that CL can bring is predictability. As referenced, SCL and DCL are utilized to anticipate task WCET and improve predictability. The second advantage of CL depends on tasks model. In such applications, when a task is in active state, the replaced cache lines should be fetched again into the cache. Such interaction causes a delay in the system which such delay causes the overhead associated with switching. CL can decrease this effect and, consequently, CL can be helpful for any system performing multiple tasks.

The last but not least benefit of cache locking is performance. There often is a trade-off between performance and predictability. One of the main objectives of cache locking techniques is to develop energy-efficient and cost-effective solutions, or in a nutshell, to obtain better performance.

Despite all the advantages, CL can cause several issues. Balancing between throughput and latency very often causes a trade-off in real-time systems. It should be noted that kind of processors under study, can affect the efficiency of the CL approach.

Multicore processors and Graphics Processing Units(GPUs) are two classes of architectures

that can benefit from CL. The move from uni-processor to multi-processor systems poses critical challenge particularly with respect to power utilization. Along these lines, in multicore processors, in contrast with single-core ones, CL implementation is a more complex problem.

As referenced previously, SWCL and HWCL methodologies require programming support, but that can also benefit from hardware support in present processors. Since some hardware features are not accessible on all processors, this makes CL more complex to implement, which as a result, lead some designer to deal with CL by software only.

DCL and SCL both involve various degrees of implementation complexity. A flexible loop-based dynamic cache locking technique was proposed in [23] . It was shown that the proposed technique improves very significantly the WCET, while preserving predictability, in contrast with previous strategies. Their implementation is complex and takes into account instruction caches and does not address hardware locking techniques, which make implementation much more straightforward.

In [24], a proposed semi-partitioned technique addresses the task relocation between cores in multiple tasks applications utilizing CL to optimize cache performance and energy consumption. Their methodology demonstrates a few improvement in cache performance, but leads to communication overload, which is not promising in real-time systems. In contrast, in [14, 21] the authors utilizes SCL without relocating tasks between cores; in any case, the locking pattern cannot change during processing. References [14,21] present how to wrap up instruction cache analysis, but not data cache. It is of interest that in [14], the authors show 10 percent improvement over previous techniques.

In [25], the authors propose a procedure for improving the performance of data caches and their energy consumption. To validate that procedure, they experimented different strategies with various benchmarks. These experiments showed 20 percent miss-rate decrease, which improved the energy efficiency by 20% [15,25]. Although performance has a main role in many real-time systems, predictability, which is one of the main advantages of CL, should not to be disregarded. Additionally, we should consider the instruction and data CL combined and employ the dynamic methodology as much as possible.

Ding et al., in [23], studied CL and showed that performance degradation can be as high as 100-150 extra cycles of latency. Hyperthreading leads to memory latency [26] that CL could reduce. These two aspects oppose each other. Subsequently, finding a fair trade-off is challenging in the current multicore systems. So finding the appropriate strategy reasonably and practically seem to rely more on objective-based rather than performance-based, or predictability-based techniques.

Considering all the literature, we propose a Multi-Level Dynamic Cache Allocation (MLDCA) algorithm for shared caches to reduce the WCET. One of the most important goals of our approach is to reasonably allocate cache lines to each core(processor), under given constraints. Unlike all previous cache-locking approaches, like [27], that considers the longest path(the path in this study is a finite sequence of edges in DAG concept which joins a sequence of distinct nodes); our algorithm focuses on a sub-critical path. Besides, our method takes into account not only the instruction cache but also the data cache. Inclusion of data cache compounds the complexity of the problem significantly.

2.4 Task scheduling

The increase in the number of cores on modern multi-core processors increases the structural complexity of such systems. Moreover, this increase in the number of cores in a processor has substantially increased the number of tasks that can be handled; therefore, making the task scheduling difficult in such systems. On the other hand, the transition from uni-processor to multi-processor in real-time systems [28] poses significant challenges. The source of such challenges, is often associated with synchronizing the different processors. Applying scheduling algorithms specifically designed for uni-processors is not ideal for multi-processors. Research shows that this typically reduce the overall performance.

Task scheduling for multi-core processors plays an essential role in the performance of real-time systems. At this time, large data-centers are critical to the prosperity of many companies. The construction and maintenance of data-centers are very costly as they use a very large amount of power and energy. Also, managing data-centers efficiently is a technical as well as a financial challenge.

The authors of [29] propose a task scheduling model for multi-core systems which relies on the DAG approach. Minimizing scheduling length is the primary objective, not only in [29], but in most algorithms. On the other hand, load balancing between multiple cores is a second challenge. Regarding the comparison with related work, one sees that the model of algorithm and implementation has advantages in scheduling length; an example of which is discussed in this paper [30]. Furthermore, the authors propose an algorithm based on task duplication. The results show that they obtain nearly optimal solutions in very large time; however, the improvement performance is small compared with soutions obtained with a Genetic Algorithms(GA). The reported implementation is also quite complicated and it cannot meet timing deadlines.

The task scheduling algorithm can be divided into two categories, static task scheduling, and

dynamic task scheduling. The static scheduling algorithms are more straightforward and they have a lower overhead in comparison to dynamic scheduling. Random-based search and heuristics-based [6] are two kinds of basic static scheduling algorithm. The algorithm for random search includes GA [31], annealing algorithm, and local-search approach [32].

The primary focus of most recently published task scheduling strategies is on uni-core processor [29], which are not expendable to multi-core processors system. Optimal task scheduling of resources in real-time systems is a typical NP-hard problem [22]. So for this reason, task scheduling for minimizing the overall execution time of workflow application, and reducing the number of running cores have gained wide attention, [33].

In [34], the authors consider online-scheduling of multiple workflows submitted in a time period. Their approach is to maximize the utilization of resources. A limitation of that method is that it does not guarantee satisfaction of workflow deadlines. Task scheduling for multi-core systems has two requirements [35], load balancing and processor utilization. Most of the time, there is a conflict between the two requirements. Generally, tasks need to migrate from one processor to another to guarantee load balancing; on the contrary, due to the processor dependency, the load balancing may be violated with task migration. Finding a good trade-off between these two requirements is a big challenge in multi-core systems.

The heuristic algorithms proposed to solve that problem are generally classified into three categories: list-based task scheduling algorithm, cluster-based task scheduling algorithm, [36] and task-duplication-based algorithm [32].

List based scheduling algorithms, which are illustrated in [32, 37], investigates if a task fits into the gap period between scheduled tasks or not. Cluster-based scheduling algorithms break down into two steps the mapping and scheduling a task [32]. In the first step, the tasks are mapped into different groups, in the mapping stage, based on specified policy. In the second step, the same processor allocates to the same tasks. The mapping and scheduling steps are solved by complex algorithms without guarantee to meet the timing deadlines, which is our concern in this research.

In [38], the authors propose an algorithm for load balancing. The objective of this approach is reducing data locality, data-transferring overhead, as well as making some degree of balance between the two.

2.5 Conclusion

With an increasing use of multi-core processors, an effective task scheduling strategy to solve the DAG-based problems has been a hot issue. Many researches have been conducted on

task scheduling. However, the current task scheduling strategies still have some drawbacks. As mentioned, improving processor utilization and reducing the implementation complexity, are two main concerns of existing data-centers. The main concern for modern data-centers is more on power and energy rather than speed. However, we focus on both timing deadlines and increasing the number of unused processors.

The priority-queue-based data-driven task scheduling method is a unique approach explored in this research to reduce the number of active cores and improve each core utilization via the task scheduling concept. In addition, we investigate some aspects of processing time variability as well as of cache management, which can be used as part of future works.

CHAPTER 3 CHARACTERIZING PROCESSING TIME VARIABILITY

As discussed in the previous chapter, task scheduling for multi-core processors while maintaining a high resource utilization alongside and meeting the specified timing deadline are critical requirements for achieving high performance. In brief, this work takes into account the effects of processing time variability on the multi-core processor.

3.1 Array Of Counters(AOCs): A generic scalable micro-benchmark to characterize processing time variability

The key idea behind the AOCs, proposed in this thesis, is to provide a micro-benchmark which can characterize the performance of multi-core and multi-processor systems. The benchmark and characterization method was designed to meet the following criteria: simplicity, scalability, and stretchability.

1. *Simplicity*: The basic algorithm is based on simple atomic operations such as one or more additions. Simplicity helps to predict the behavior of the processor in its interactions with memory. It is of interest that processing time of basic operations may have no impact if they execute faster than memory access time.

2. *Scalability*: The algorithm should be scalable such that the benchmark can spread over a multi-processor, a multi-core, or combinations of such systems. Management of the functions and data sent to the cores is facilitated by using the Intel Data Plane Development Kit(DPDK)cite50 library.

3. *Controllability/Stretchability*: Memory usage of the algorithm must be stretchable, so it can at will, and in a fully controllable way, exceed the capacity of every memory hierarchy level. In the beginning, the benchmark data would fit in the L1 cache, and it would migrate to L2, L3, and finally to the DDR4 external memory. The stretchability allows measuring the impact on the performance of each cache level and DDR4 memory. The stretchability allows to measure the impact on performance of each cache level and DDR4 memory.

In this research, a stretch factor(e.g., Scale = 1.5) is used to expand the AOCs data footprint gradually. In the beginning, the algorithm starts with an array of size 24 Bytes, and at each step, the dataset grows to 150% of its previous size in 34 steps(24 B, 36 B $\ldots 24 \times 1.5^{34} =$ 22 MB).

Algorithm 1 Array Of Counters

```
1:  procedure MAIN FUNCTION
2:      N ← Number of counters;
3:      cMax ← Maximum counting value;
4:      iMax ← Maximum number of iterations;
5:      bMax ← Maximum number of memory block size;
6:      Scale ← stretch factor;
7:      Processing Time ← 0;
8:      counter ← 0;
9:      for i = 1, i ≤ bMax, i++ do
10:         for j = 1, j ≤ iMax, j++ do
11:             Time Measurement: T1
12:             for m = 1, m ≤ cMax, m++ do
13:                 for n = 1, n ≤ N, n++ do
14:                     counter[m][n] = counter[m][n] + 1;
15:                 end for
16:             end for
17:             Time Measurement: T2
18:             Processing Time(j,i) = T2 − T1;
19:         end for
20:         N ← N × Scale;
21:     end for
22:     return Processing Time;
23: end procedure
```

The details of the AOCs algorithm goes as follows(shown in Algorithm 1). The AOCs can run simultaneously on each core or thread of a many-core architecture. The inner loop increments a single counter on the array and goes on to the next one(for example in 24 B we have 6 counters, 32 bits = 4 bytes so 24/4 = 6). The second inner loop enforces the maximum counting value of each counter.

These two inner loops are surrounded by time measurement and the outer loop repeats this procedure for iMax times. In order to have a fair comparison when hyper-threading is used, we split the computation load into two different threads(one for each logical core), each thread processing $N/2$ counters. It is possible to execute AOCs single-threaded and hyper-threaded at the same time. In all cases, the average processing time we report is the total time divided by the total number of counter increments. As addressed in the following, AOCs, as a timing analysis method, is a useful tool to estimate worst/best/average processing time. This helps us ordering the tasks according to some constraints and priority, and then assigning them to cache slots with the proper cache locking strategy.

3.1.1 Best/Worst Case Execution Time(BCET/WCET) of a program

To exploit the BCET of an application, an AOCs was configured in a mode which we call a "controlled mode". The entire application, in this mode, is assigned to a core, assuming that all the other cores are in idle mode and all the software and hardware are always available for the running core. Such an arrangement leads to no interaction with the other parts of the system. To do so, we implemented a platform for the Intel Xeon E5-2650 V4 processor.

Via this platform, we can do the following;

1. Isolate the target core on which the program is running. An isolated system would suffer no interference from the rest of the system. Also, such implementation is feasible with the concept of atomic operation, which is a means through which no other instruction can interrupt the running operation

2. Handle all operations regarding the system with a master core in order to minimize the interference as much as possible.

3. Enforce some operations, which can be available for a user to configure based on which test is to be conducted; relevant operations include: activate/deactivate the data/instruction cache, invalidate the data/instruction cache, flush the data cache.

With these configuration methods, low interference and high interference options can be implemented. A Low-Interference Configuration(LIC) is defined when, the data cache is enabled, the shared cache is set in "interleaved" mode, and the instruction cache is enabled so it means assuming a minimum interference on all the shared resources(software and hardware services) happen for a given set of input data to produce its output. So this mode helps us to have an estimation for BCET. On the other hand, a High-Interference Configuration(HIC) happens when there is a maximum interference on all the shared resources. In this case, HIC is defined when the data cache is disabled, the shared cache is set in "sequential" mode, and the instruction cache is disabled. This mode gives a good understanding of WCET.

Using the LIC and HIC, we can illustrate the impact on variability of execution time for the application. These two configurations are useful to evaluate the performance in general on any system like, the Intel Xeon E5-2650 V4 processor to find the WCET and BCET, which is desirable in this research.

To exploit the WCET of each application, an AOCs is configured in "uncontrolled mode." This mode is built to create the WCET conditions for each application. So the execution time of all running programs can be suspended for a while because of interference between

one program with another. Such mode can be useful to estimate an upper-bound of the WCET of a program.

Unlike the controlled mode, which shuts down all the CPUs, in uncontrolled mode, we enforce a small program of Interference Array Of Counters(IAOCs) to run onto all the other CPUs. IAOCs can be configured to saturate the shared resources. To obtain the maximum benefit of IAOCs, we strongly recommends running IAOCs before starting the analysis program and stopping the IAOCs upon finalization of the program.

Often, there is no guarantee to reproduce the maximum interference scenario throughout the experiments, even thought one knows adequate parameters about the setting of the system and application.

The importance of knowing the WCET using AOCs helps us to find the critical path, bottlenecks of the system, and the source of interference or variation. In all cases, a proper cache locking strategy the can be used to protect operations of high priority tasks.

3.1.2 AOCs as a micro-benchmark

A micro-benchmark is used to identify many CPUs memory parameters such as L1/L2/L3 cache size, L1/L2/L3 cache access time, page size, memory access time, etc. With the help of AOCs, we measure the access time required to access different cache layers and memory for different block-sizes and strides. AOCs provided a framework for time measurement that allows to characterize the impact of block-size, stride, and scale-factor. So AOCs allow computing the execution time for varying block sizes and strides. Also, this lets us compute cache miss-rate or hit-rates for each cache level and memory separately which is the objective of the micro-benchmark concept.

3.2 Multi-level dynamic cache management

Cache locking is a useful strategy in real-time systems to improve timing predictability. Cache locking can improve the performance of modern processors if used properly. Hence, cache locking is crucial as it is a source of processing time variation; however, it can also improve performance and reduce time variation. Generally, there are two categories of cache locking, static cache locking, and dynamic cache locking.

The Cache Allocation Technology(CAT) features provide more cache space which is available for high priority applications as shown in Figure 3.1. CAT allocate the cache to tasks with the help of the Dynamic Cache Locking(DCL) approach, during runtime to further optimize

the performance of the high priority applications versus the low priority applications.

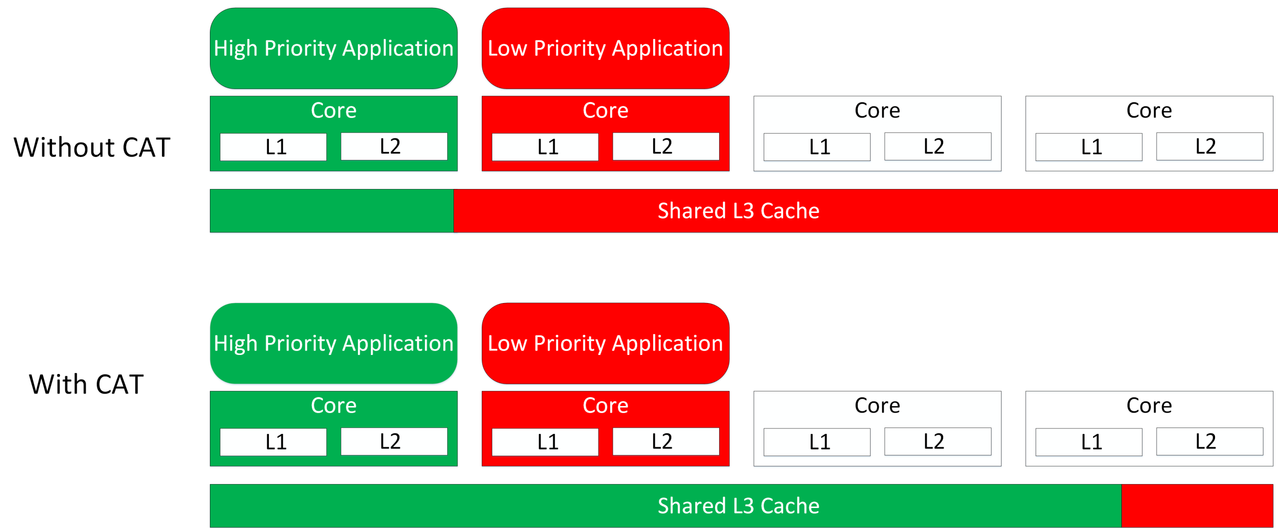

Figure 3.1 CAT provides more cache space for high priorities applications

The main objective of Cache Allocation Technology(CAT) is to activate resource allocation based on application need or Classes of Service(COS). The processors shows COS into which applications can be set. CAT for each applications could set limits based on classes to which it belongs. Each COS can be arranged by bitmasks which specify the limits, possible overlap and isolation between classes [39].

Sample cache capacity bitmasks for a bitlength of 20 (which is for Xeon E5-2650 V4 processor) are shown in Figure 3.2.

	M1	M2	M3	M4	...	M17	M18	M19	M20	
COS 0	1	1	1	1	...	0	0	0	0	Isolated bitmasks
COS 1	0	0	0	0	...	1	0	0	0	
...	0	0	0	0	...	0	0	0	0	
COS14	0	0	0	0	...	0	1	0	0	
COS15	0	0	0	0	...	0	0	1	1	

	M1	M2	M3	M4	...	M17	M18	M19	M20	
COS 0	1	1	1	1	...	1	1	1	1	Overlapped bitmasks
COS 1	0	0	1	1	...	1	1	1	1	
...	0	0	0	0	...	1	1	1	1	
COS14	0	0	0	0	...	1	1	1	1	
COS15	0	0	0	0	...	0	0	1	1	

Figure 3.2 Example of cache capacity bitmasks

All '1' mixes are permitted(for example FFFFH, 0FF0H, 003CH, and etc.). Always a mask bit set to '1' indicates that a specific COS can assign into the cache subset related to that bit. On the other hand, '0' in a mask bit indicates that a COS cannot assign into the represented cache subset.

Figure 3.2 demonstrate 2 instances sets of cache capacity bitmasks. The second example demonstrates overlapped scenario, which permit some low-priority occupy a share slot with high priority ones. The first case shows different non-overlapped dividing scenario. As an issue of programming approach for priority issue, COS0 regularly considered and arranged as the highest priority COS. COS1, COS2 to COS15 are in the next priorities respectively [39].

Considering WCET and BCET, we propose to reserve cache slots to given tasks. Obtaining near deterministic processing time of tasks executing on multi-cores processors with dynamic cache management, while reducing WCET, is the primary goal of this part.

We are proposing a flexible priority-based Multi-Level Dynamic Cache Allocation(MLDCA) algorithm for shared caches to reduce the WCET of tasks. Unlike all previous cache locking approaches, such as [27], that considered the longest path just for the instruction cache, we propose an algorithm which considers the longest path for both the instruction and data

caches. In contrast with [27] which exclusively deals with the longest path, our algorithm considers the sub-critical path for several of the longest path. It is necessary to ensure scheduled tasks keep resource balance, which leads to shared resource management and dynamic cache locking.

Such a scheme is a useful technique to improve timing predictability in real-time systems. Therefore, a task scheduling algorithm for multicore processors is mandatory besides proper cache management.

In theory, the use of a multi-level system not only speeds up cache locking but also produces better locking than the traditional single-level locking. Multi-level systems work well for two main reasons: As depicted in Figure 3.3, the first one is that with increasing granularity(Coarsening as first step), we can cover a substantial number of the longest path on a coarsening phase. The second reason why multi-level systems work well is that the cache allocation phase becomes much more powerful in this context. The proposed Algorithm is illustrated with more details in Algorithm 2. During the un-coarsening phase, the cache allocation of the coarsest graph is projected onto the next level fine graph.

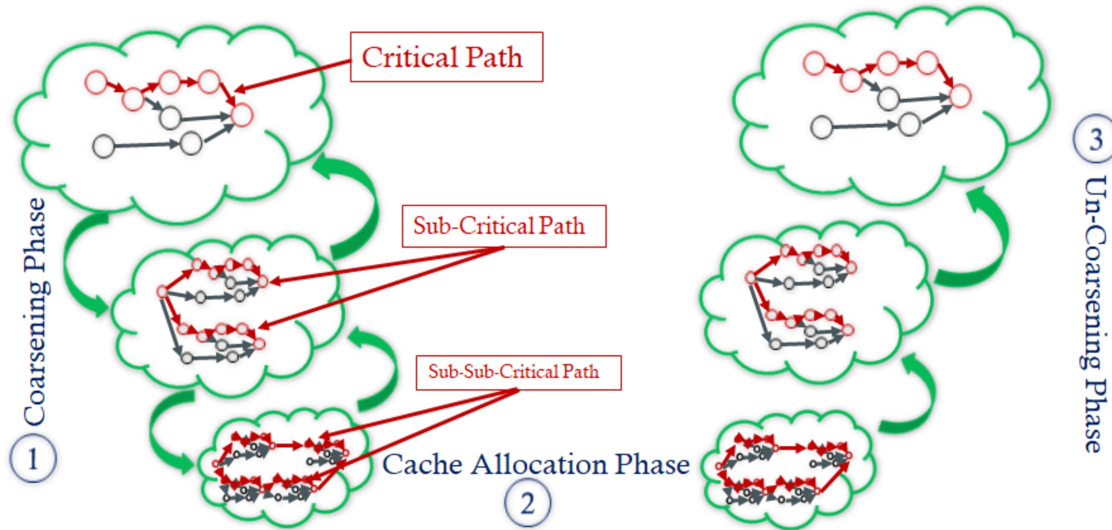

Figure 3.3 Multi-level dynamic cache locking.

In Algorithm 2, the proposed technique finds a set of cache slots at each level for locking, in order to minimize the WCET. Thus, the most critical paths are guaranteed to be locked in the cache and execute at the earliest time. The multiple levels are processed sequentially like the nested loops. In each level, the DAG(sub-DAG) with the longest path(s) is chosen. Then, the available cache slots are selected for this path(s). This is performed iteratively,

and it stops when there is no free space for locking or no critical path for selection.

Algorithm 2 Multi-Level Dynamic Cache Allocation

```
1: procedure Main Function
2:     Graph ← The Adjacency Matrix;
3:     Repeat ← Number of repetition of Coarsening steps, default = 3;
4:     YList allLongestPaths ← new YList();
5:     YCursor pathCursor ← LongestPaths.kLongestPathsCursor((graph, edgeCostsDP, startNode,
   endNode);
6:     while (!sharedcache.empty) do
7:         for (i = 1, i ≤ Repeat, i++) do
8:             final EdgeList firstPath ← pathCursor.current();
9:             final double costsOfFirstPath ← calculateCostsForPath(firstPath, edgeCostsDP);
10:            allLongestPaths.add(firstPath);
11:            pathCursor.next();
12:            while (pathCursor.ok()) do
13:                EdgeList currentPath ← (EdgeList)pathCursor.current();
14:                double currentCosts ← calculateCostsForPath(currentPath, edgeCostsDP);
15:                if !(currentCosts > costsOfFirstPath) then
16:                    allLongestPaths.add(currentPath);
17:                    pathCursor.next();
18:                else
19:                    break;
20:                end if
21:            end while
22:        end for
23:    end while
24: end procedure
```

With the help of proposing an approach, Multi-Level Dynamic Cache Management, the shared cache blocks can dynamically be allocated to the same cache block. Also, the cache allocation based on high and low priority tasks leads to a significant reduction in the task WCET. The proposed methods will be implemented and characterized shortly.

Finally, we believe that a proper cache locking approach needs to take both task scheduling and cache locking into account simultaneously. We thus see the potential for significant further improvements on joint task scheduling and cache locking that is hopefully open to further exploration in the future.

3.3 Experiments and results

This section will present the AOCs results obtained and the methods used through our experiments and tests. The proposed method also allows characterizing the effects of intra-core parallelism using SIMD(Single Instruction and Multiple Data) as well as the effects of core-level parallelism that can be exploited through multi-threading and hyper-threading technology.

In particular, we observed that the gain offered by the Advanced Vector Extension 2(AVX2) slightly depends on data size when data is in the cache and, as expected, we also observed that hyper-threading is only interesting when data are not entirely in the cache.

Hence the proposed method enables a better understanding and measurement of the task processing time variations with specific data size and number of processing cores; the method also suggests how to deal with them in the context of real-time low latency applications running over typical data-center processors.

So refining this method is according to our objectives can be useful to improve its accuracy, flexibility, and capacity to observe the critical features of the processing time variability. Some related open question are left for future work.

Figure 3.4 shows the general form of the data gathered from line 17 of Algorithm 1 when performing experiments with the AOCs. The shape of the result curves is very similar for different scenarios using hyper-threading or SIMD features of the processor. ‘X’ is the starting point size of the AOCs as its size increases in steps specified by the stretch factor to reach the bending point ‘Y’. Point ‘Y’ is where the data set cannot fit into the cache anymore and is migrated to the external RAM. Horizontal lines are labeled with the different cache levels and memory. They correspond to respective average processing time associated with a single atomic operation of the AOCs that comprise one read, one write and one addition when data is found in cache and RAM respectively. In each figure, the worst and the best execution time is depicted with the dotted line.

There are two important things to notice for this illustrative curve:

- The position of the jump(Y) depends on the number of physical cores used in the test. When more physical cores are used, the jump position is observed for smaller values. This happens because several processors of equal speed share a fixed size cache.

- The magnitude of the jump is related to both the processing time difference between caches and RAM and the number of physical cores.

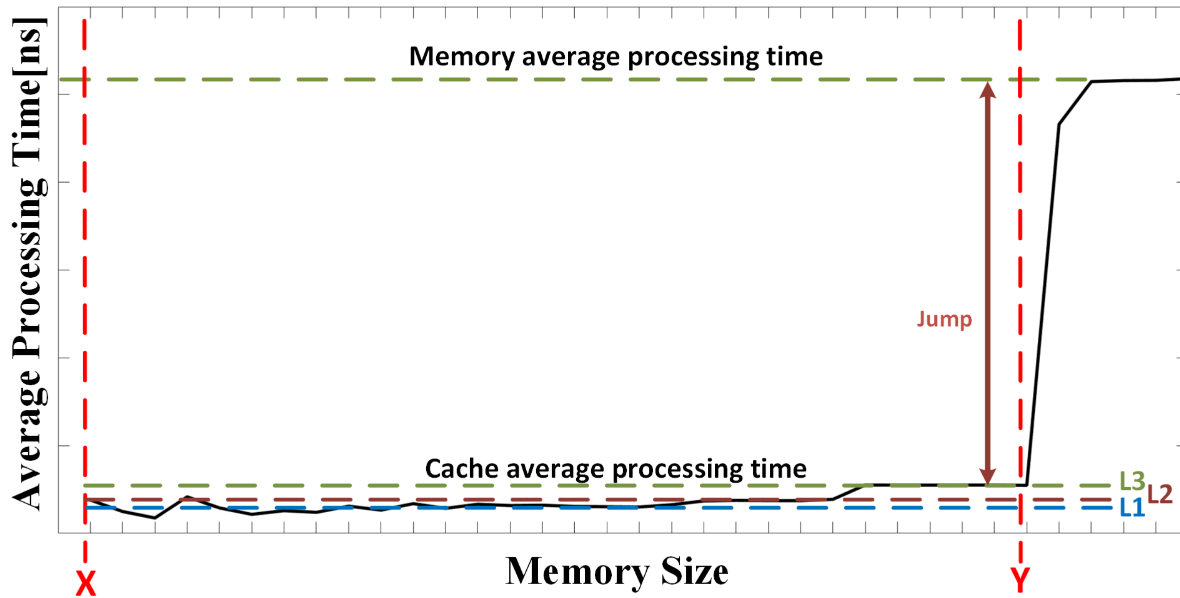

Figure 3.4 The general format of the gathered data, as processing time varies according to different levels of cache and memory.

The tests were conducted using the Ubuntu 16.04 GNU compiler running on the Intel Xeon E5-2650 V4 processor, with 12 physical cores(24 logical cores with hyper-threading). The processors are clocked at 2.2 GHz with 33.75 MB of cache spread over different levels(L1, L2, and L3) connected to a 32 GB DDR4 memory.

Figure 3.5 characterizes multicore performance related to cache and RAM. It shows that using more physical cores forces the bending point to move towards smaller array size values, due to increased resource sharing by more cores, while the magnitude of the jump increases significantly from 0.78 ns for 2 logical cores to 1.08 ns for 4 logical cores, 1.63 ns for 6 logical cores, 2.26 ns for 8 logical cores, 2.82 ns for 10 logical cores, and 3.37 ns for 12 logical cores. The increase of the magnitude of the jump quantifies the impact of the lower performance of the RAM compared to the L3 cache when trying to serve more cores and a controlled time variation reflects a stable and consistent pattern of variation over time.

The dotted lines show the worst and best case execution time for each core. The Worst-Case Execution Time(WCET) is vital to ensure meeting timing deadlines. On the other hand, we are interested in Best-Case Execution Time(BCET) as a reference to assess code performance quality.

Figure 3.6 shows the distribution of the processing time for a single memory size(1000 runs on 22 hyper-threaded cores for 1.35 MB memory block size). The mean processing time is

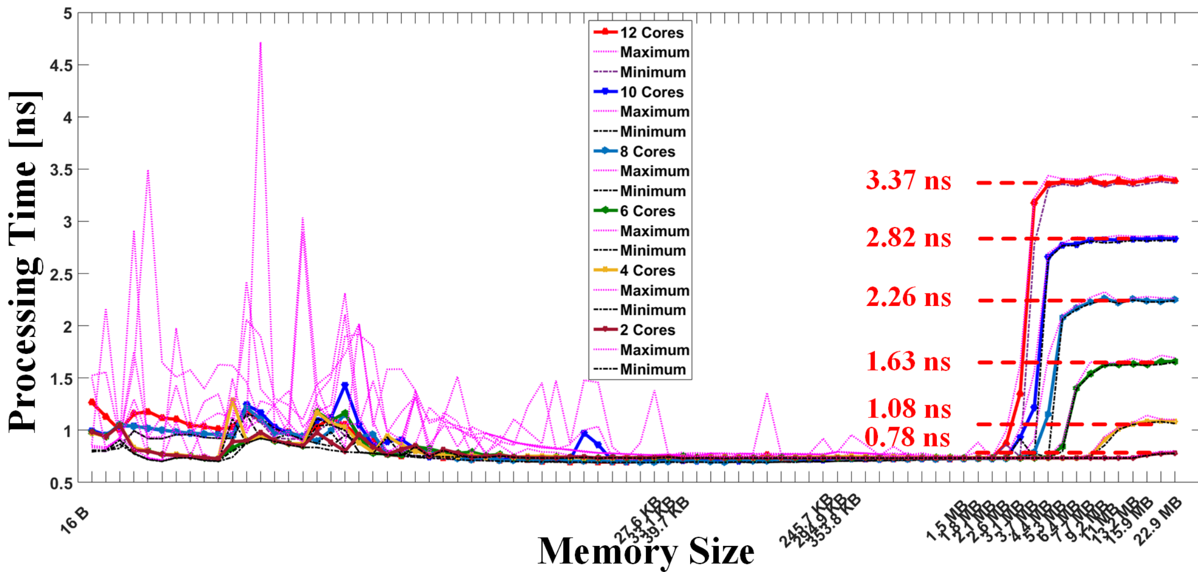

Figure 3.5 Various results that characterize the impact of changing the size of the array for a variable number of active logical cores. For each curve, the dotted lines stand for best/worst processing time.

0.74 ns. It is interesting to notice that most of the points represent a processing time lower than 0.75 ns, which represents only 1% of variation from the mean value. Actually, only 10 points(0.05% of 22000) have a processing time greater than 0.75 ns. This observation happened to all tested memory blocks 35 blocks in this experiment, which started from 24 B and finished at 22.17 MB.

Figure 3.7 shows average, worst and best execution time, and compares hyper-threaded and single-threaded executions of AOCs. In this figure, all 12 physical cores of the Xeon processor handle the AOCs tasks, with 12 of them executing in hyper-threaded mode and 5 executing single-threaded. As apparent in Figure 3.7, the processing time when executing from DDR4 memory did increase in single-threaded mode compared to hyper-threaded operation. For example, the 3.08 ns observed for single-threaded execution was reduced by 18 percent to 2.51 ns with hyper-threaded execution. But this is not true about the cache.

In Figures 3.5, 3.7, 3.8 and 3.9 the dotted lines determine the worst and best execution time around average time, which shows a low variation of WCET and BCET with respect to average.

The second-generation Intel Advanced Vector Extensions(AVX2) instructions is a means to exploit parallelism that provides more efficient memory access in both load and store sequences. Such usage reduces the amount of hardware control logic by a factor of 8 when

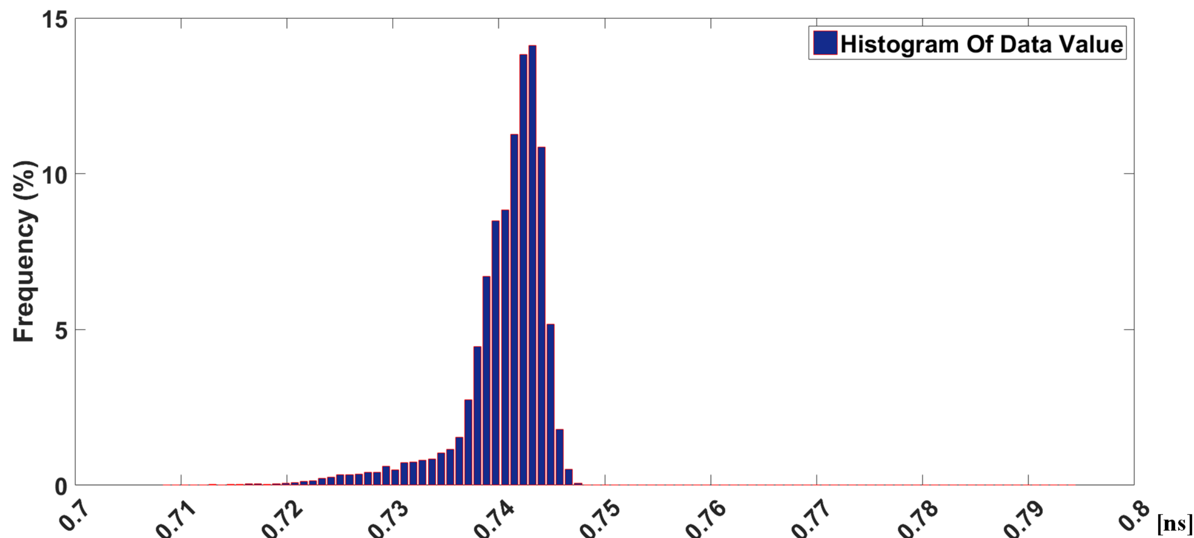

Figure 3.6 Representative histogram of the observed processing time(1000 runs on 22 hyper-threaded cores for 1.35 MB memory block size.

processing 32-bit integer numbers. Figure 3.8 shows the result with AVX2 activated. It is observable that vectorized access significantly decreased the processing time variation when data is in the cache compared to Figure 3.7, while it remained the same when executing from DDR4 memory.

As shown in Figure 3.8, the cache processing time is 0.27 ns for hyper/single-threaded execution with AVX2. Thus, processing time is reduced by almost 62 percent(0.71 ns to 0.27 ns) when executing from the cache using hyper/single-threading in combination with AVX2. As indicated, combining hyper-threading and intra-core vectorization gave better results compared to vectorization alone. By contrast, the processing time variation when executing from RAM is not improved using AVX2 vectorization with this application. Thus, executing AOCs is fast enough to exploit all the memory bandwidth with or without vectorization.

Based on these results, an application crossing that boundary that way would a throughput reduction when in the cache(0.27 ns) vs. when in the RAM that can exceed a factor of 10(2.58 ns).

As long as the execution time increases, the average/worst/best processing time increases when executing either from cache or RAM. The execution time eventually dominates the read and write latencies. As shown in Figure 3.9 with the green curve that corresponds to a repetition factor of 10, the average processing time remains steady when executing from cache or external memory for such a repetition factor. Such an arrangement shows that if the processing time of the inner loop is large enough, the access time difference of the cache

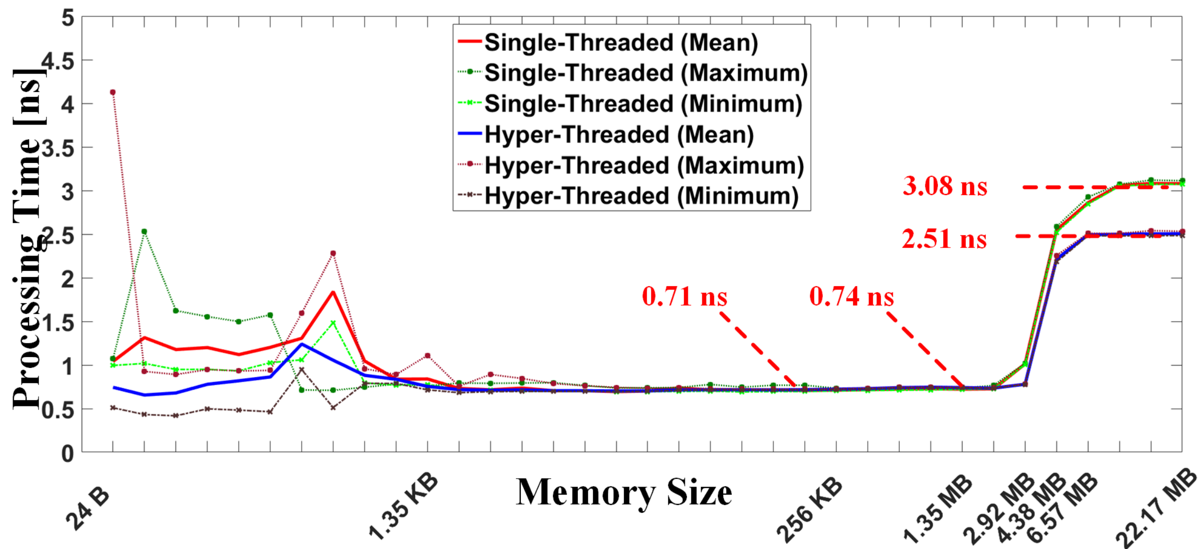

Figure 3.7 The mean, WCET and BCET in latency observed when executing in hyper-threaded mode(blue) vs. single-threaded mode(red).

and RAM is masked and is not visible in the results. As in previous figures, the dotted lines stand for maximum and minimum execution time(BCET).

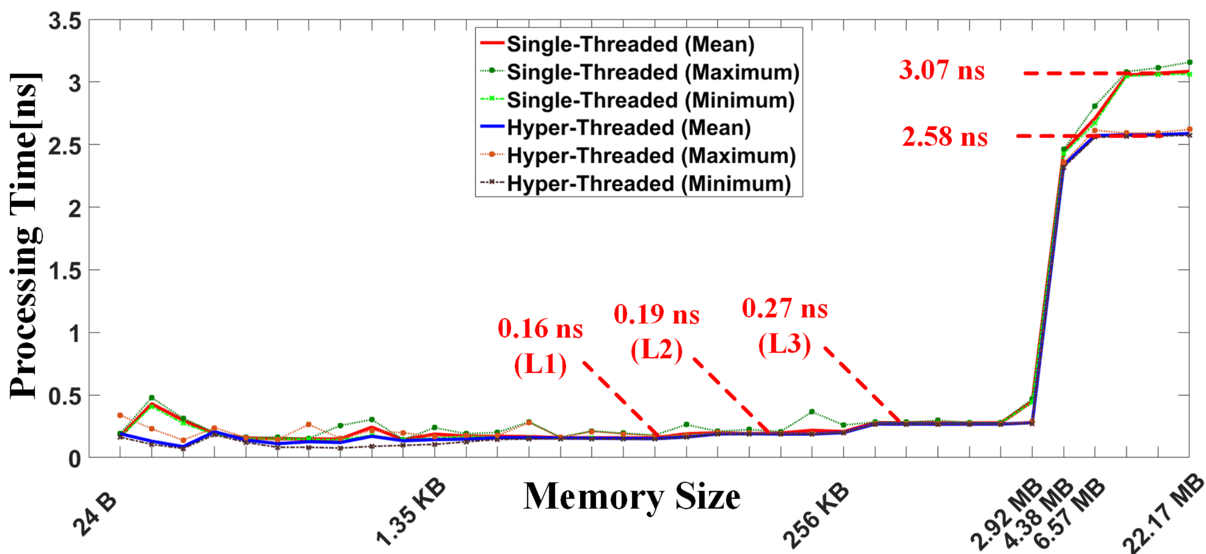

Figure 3.8 The latency of hyper-threaded core(blue) vs. single-threaded core(red) using AVX2 instructions.

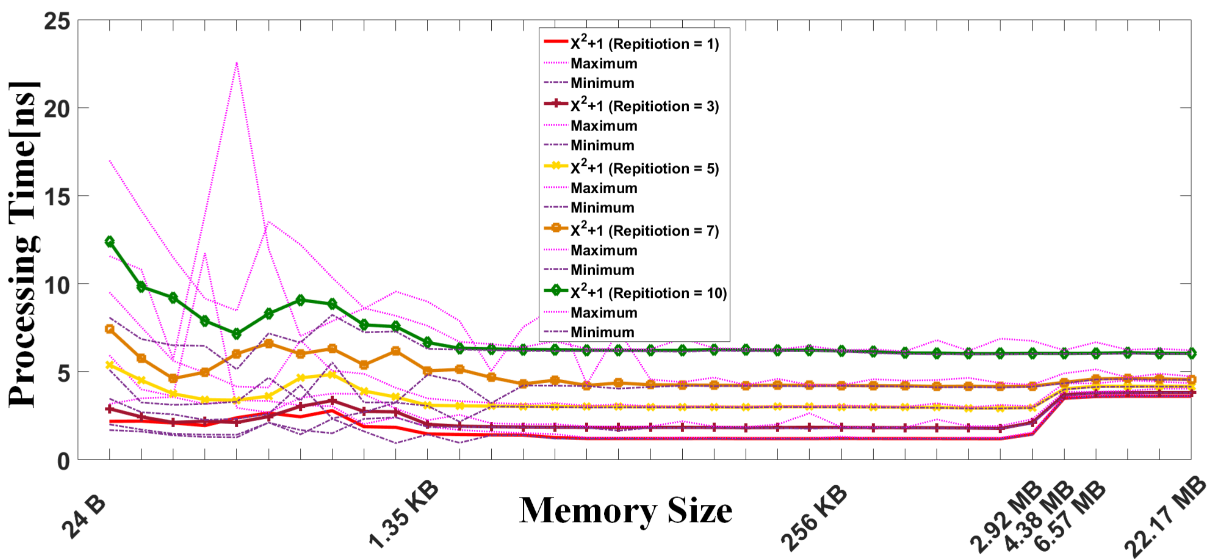

Figure 3.9 The latency of hyper-threaded core for the $x^2 + 1$ function with different repetition values.

CHAPTER 4 PRIORITY-QUEUE-BASED DATA-DRIVEN TASK SCHEDULING

4.1 Data-Driven task scheduling

Scheduling is addressed by assuming that we have a multi-tasked processor in which one core is a dedicated master. A scheduler can facilitate several goals, e.g., maximizing performance, minimizing latency, maximizing fairness. These goals often conflict(e.g., performance versus latency) in reality, so a task scheduler should be implemented suitably. Improving utilization of CPU cores with a queue-based data-driven task scheduling algorithm can reduce the processing time of concurrent workloads, which is one of our primary goal.

Because multicores derive their power from inter-core parallelism, a successful multicore application must schedule tasks and exploit parallelism to keep the cores busy. This challenge is to employ task-scheduling strategies so that the method improves the utilization of the cores to ideally 100 percent.

4.1.1 Queue-based data-driven task scheduling

In parallel computing, speed-up is achieved by running multiple independent tasks simultaneously. Thus, a paradigm is required to represent the inter-task dependency in order to utilize computing resources efficiently. A Directed Acyclic Graph(DAG) is commonly used for this purpose, which is a dominant model used in parallel computation.

The data-driven mechanism is used for task classification criteria and then a multi-queue architecture is introduced to handle these tasks in different classes. In addition, data-driven guarantees that each task in a ready queue can be executed as soon as allocated.

A data-driven task scheduling algorithm is supposed(in the first definition of project) to run on a general platform, which is composed of 6 blades. Each of the blades includes two CPUs and each CPU includes 12 physical cores, which can run at 3GHz. In the first step, a static scheduling approach will be studied based on given DAGs.

In general, a DAG model represents tasks as a graph, with vertices representing dynamic functions/computation cost, and edges between vertices representing dependencies along with communication cost.

A critical path of a DAG is defined as the longest path in terms of time consumption or computation cost. Based on the description of DAGs, a queue-based-data-driven task scheduling

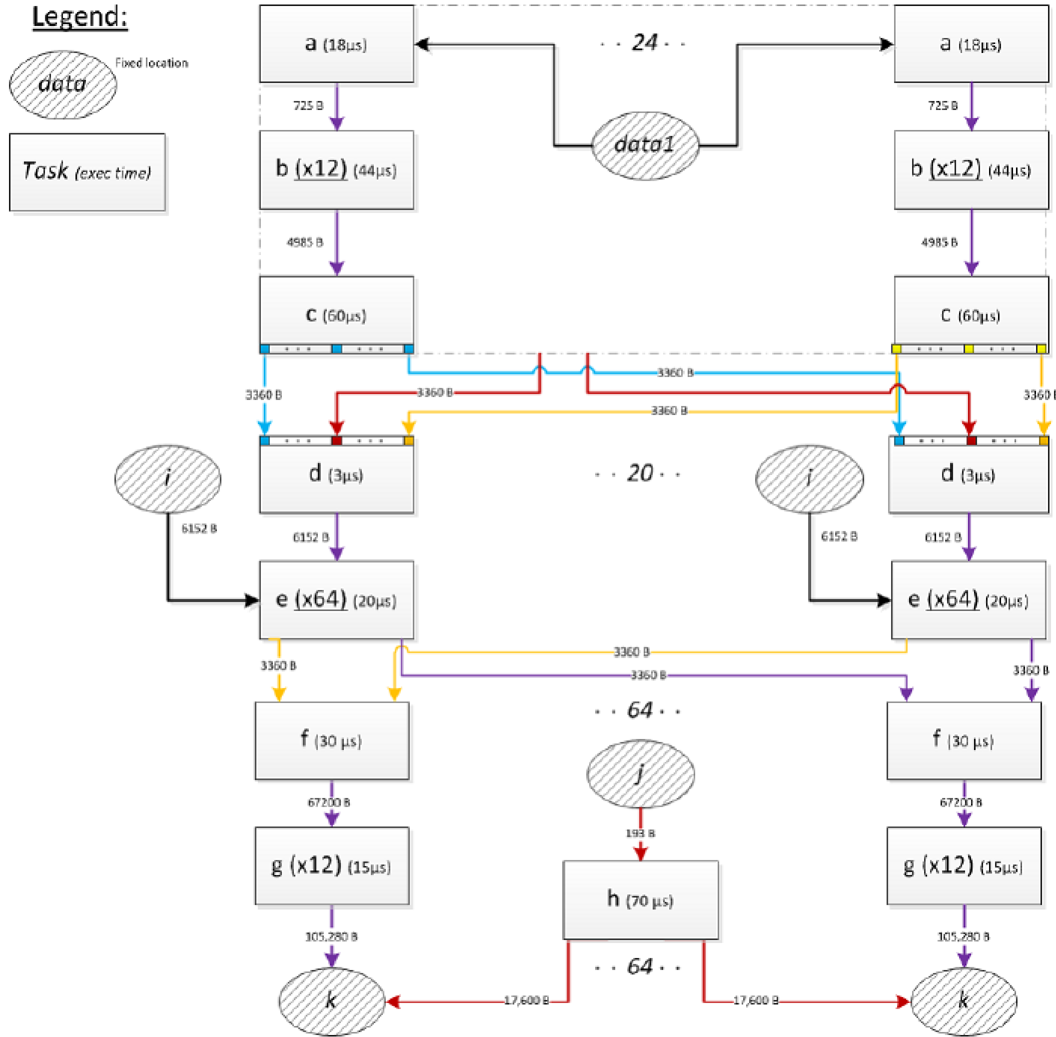

Figure 4.1 An example of real DAG from Huawei Co.

algorithm is proposed and implemented in this section, which schedules individual tasks until their data inputs are available.

For example for a given real DAG(Huawei Co. project) as shown in Figure 4.1. It is composed of multiple tasks, which can be classified in multiple layers according to the a dependency and functionality. For example, the first layer is all task a's, which wait for the data from the master blade or other slave blades(for simplicity we consider that all tasks at the same level have the same function, but in reality they can have the different functionality). The tasks a's execution time are 18 us.

Furthermore, none of the task 'b' can be processed as far as it receives all output data from the corresponding tasks 'a' as parents input. In this case, we call the task 'a' parent tasks

of the task 'b' and the task 'b' is the child task of the task 'a'. The numbers on the edge between tasks a and tasks b are stand for the band-width, in this case is 725 Byte. Similarly, any task 'c' cannot start until it receives output data from all task 'b's. Then each task 'b' is a parent task for any task 'c' and each task 'c' is a child task for any task 'b'. For better understanding Figure 4.1 can be simplified to Figure 4.2.

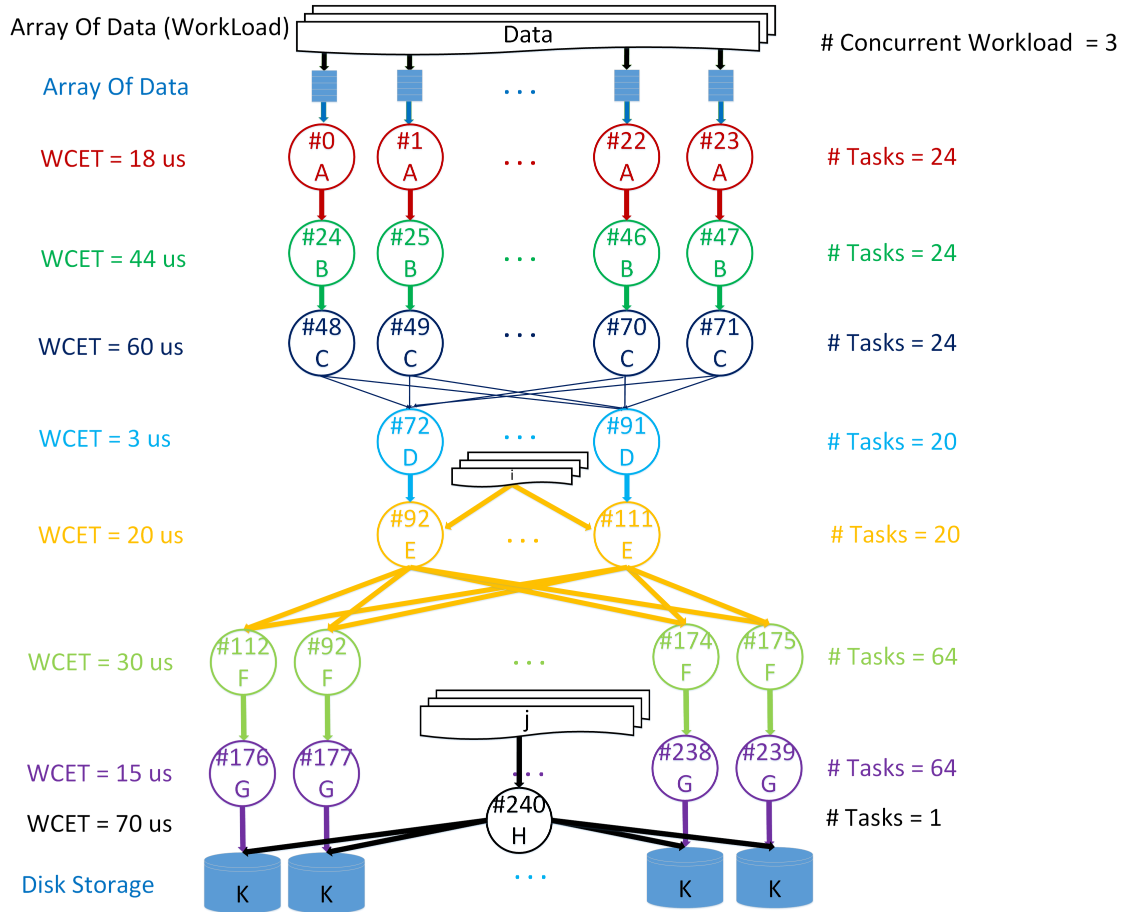

Figure 4.2 Simplified DAG of Figure 4.1

The proposed data-driven task scheduling has two main phases: Task Registration(or Task Allocation) and Data Transmission. In the task registration phase, each blade is configured with task package profiles(Table 4.1) to be executed. In the task profile, the Task ID, Task Type, Task WCET, the corresponding DAG ID are given. Besides each task profile should be associated with a list indicating the parent tasks, its child tasks, and destination blade(s) for the generated data if dependencies exist. After registration, all tasks enter the waiting mode, and a task starts to run when all the required data packets are received.

In the data transmission phase, the master blade initiates the processing by distributing the

input data packets 4.2 for the first layer tasks of DAGs and then each blade exchanges data packets according to the information stored in task profiles to trigger the task execution. For one DAG, it corresponds to multiple workloads. Thus, Workload ID is required in the data packet for task mapping and priority determination if priority-based scheduling is incorporated.

4.1.2 General definition

Defining the task package is necessary to perform task registration and data transmission for communication between blades. In the following statements, general definitions are given for both package types.

Table 4.1 Task Package Profile

Task ID	Task Type	Child Task ID List	Parent Task ID List	Task WCET	Blade ID	DAG ID

For the task package profile, there are ten labels, as shown above. Specifically, the task function can be defined by the Task Type, with which the operation can be changed easily for any task. The WCET stands for the worst-case execution time, which can be used to determine the timing deadlines. Parent tasks are the tasks which the current task depends on. In other words, the execution of the current task requires the outputs of all parent tasks.

Similarly, the output of the current task is the input of its child task(s). Since there may exist multiple DAGs to be executed, DAG ID is required to provide identification. Note that the definition is extendable during implementation due to practical requirements.

Table 4.2 Data Package Profile

Task ID	Workload ID	DAG ID	Data

For the data package profile, there are four fields: Task ID, Workload ID, DAG ID, and Data field. Since the scheduling is driven by data, the information is required to determine the Task ID, DAG ID, and Workload ID for scheduling. Furthermore, Data can be considered as an array of data for each task.

4.1.3 Task allocation based on DAG

As shown in Figure 4.1, and 4.2, a balanced DAG is used as input. Then a 0-1 adjacency matrix can be given by considering the direct dependency between tasks. This DAG can be

represented as a square matrix of order ‘K’, where ‘K’ is the total number of tasks. Based on the matrix, we can know the depth of the DAG and its critical path of the DAG can also be obtained. Furthermore, the parallelism of the DAG can also be obtained, which is defined as “the maximum number of tasks that can be processed in parallel”.

Suppose that the adjacency matrix is A, let $A_i = 0$ and $A_{i-1} \neq 0$, where i ϵ N, i $\leq$ K. Then ‘i’ is the depth of DAG. It is a formal and generic technique to get the depth of DAG. By having the depth of the DAG, it is known how many layers that the tasks within the DAG can be divided into. Next, the parallelism property of the DAG shown in Figure 4.1 can be extracted. By solving the matrix generated based on the DAG shown in Figure 4.1, the tasks can be divided into 7 groups and there is no dependency within each group. The seven groups are listed as follows:

Table 4.3 Task Grouping

Group 1	Group 2	Group 3	Group 4	Group 5	Group 6	Group 7
25 Tasks	24 Tasks	24 Tasks	20 Tasks	20 Tasks	64 Tasks	64 Tasks
Task a’s and Task h	Task b’s	Task c’s	Task d’s	Task e’s	Task f’s	Task g’s

For task ‘h’, there is no dependency with other tasks. Then it can be individually scheduled. Based on the grouping and the symmetric(or balanced) characters, the critical path can be obtained as follows :

Task ‘a’ → Task ‘b’ → Task ‘c’ → Task ‘d’ → Task ‘e’ → Task ‘f’ → Task ‘g’.

Then the minimum execution time that can be achieved is 2099 us without considering communication costs.

Suppose that there are 6 blades and each one with 12 physical cores. One core is dedicated to local scheduling, and then there are 11 cores available in each blade for data processing. For the first group, it is composed of all ‘a’ tasks, and the size of the group is 24. This means that 24 tasks should be dispatched into multiple blades. As two blades are not enough for data processing, based on our assumption, then 3 blades are used for task processing, e.g., blade1, blade2, and blade3.

In a data-driven structure, each input workload will generate an instance of a DAG, which corresponds to a set of tasks. In general, four queues are used to facilitate task scheduling. They are, waiting queue, ready queue, running queue, and finish queue. A task can be moved from one queue to another one when the state changes. With the help of these queues, the

system supports scheduling tasks with data dependencies specified by an arbitrary DAG.

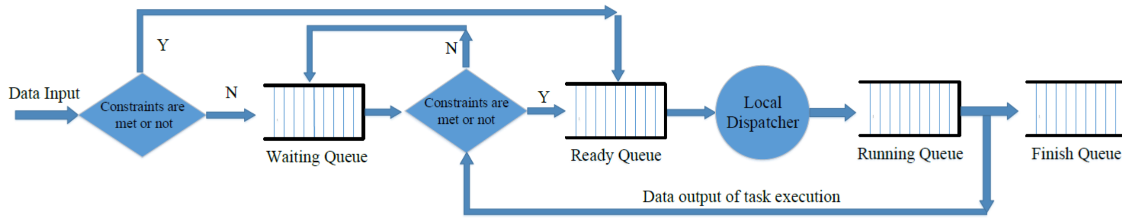

Figure 4.3 Queue-based data-driven task scheduling

Tasks that have received part of their inputs reside in the waiting queue. When the input constraints are met, a task is moved from the waiting queue to the ready queue. The local dispatcher checks the states of local cores. When there is an available core, it picks up one task from the ready queue for execution if the ready queue is not empty. If the data output of a task is an input of tasks in the waiting queue, the output is fed back to trigger a possible execution, as shown in Figure 4.3

We further assume that the data packet with smaller Workload ID and smaller Task ID will be released earlier by the master core. In such a case, the data packet with smaller Workload ID and Task ID will be assigned higher priority. Then priority-based queue management can be shown in Figure 4.4.

Since the running queue and finish queue remain the same as they have priorities, only the waiting queues and ready queues are shown in Figure 4.4. The local queues are classified into three types based on the workload number(Workload ID). Specifically, they are high priority queue, medium priority queue, and low priority queue. Due to the specifications and limitations of data-driven mechanism, a task in a queue cannot be dispatched into a local core for processing until all its input data packets arrive. Therefore, each priority queue is composed of the entry queue, waiting queue, and ready queue. The entry queue receives the data packet and then verifies whether the input constraints are addressed or not. If all the constraints are met, the corresponding task is in a ready queue for processing. Otherwise, the corresponding task will be stored in the waiting queue until all the input data packets are received. The flowchart of the current approach is depicted in Figure 4.5.

The detailed priority-based data-driven scheduling algorithm is shown in Algorithm 3.

The implemented algorithm so far is a non-preemptive algorithm which does not meet the timing deadlines and just used to propose an analytical model for analyzing the data-driven technique based on the DAG. This model gives a good view of the performance of the technique and helps us better understand it from a theoretical perspective. The next steps, as

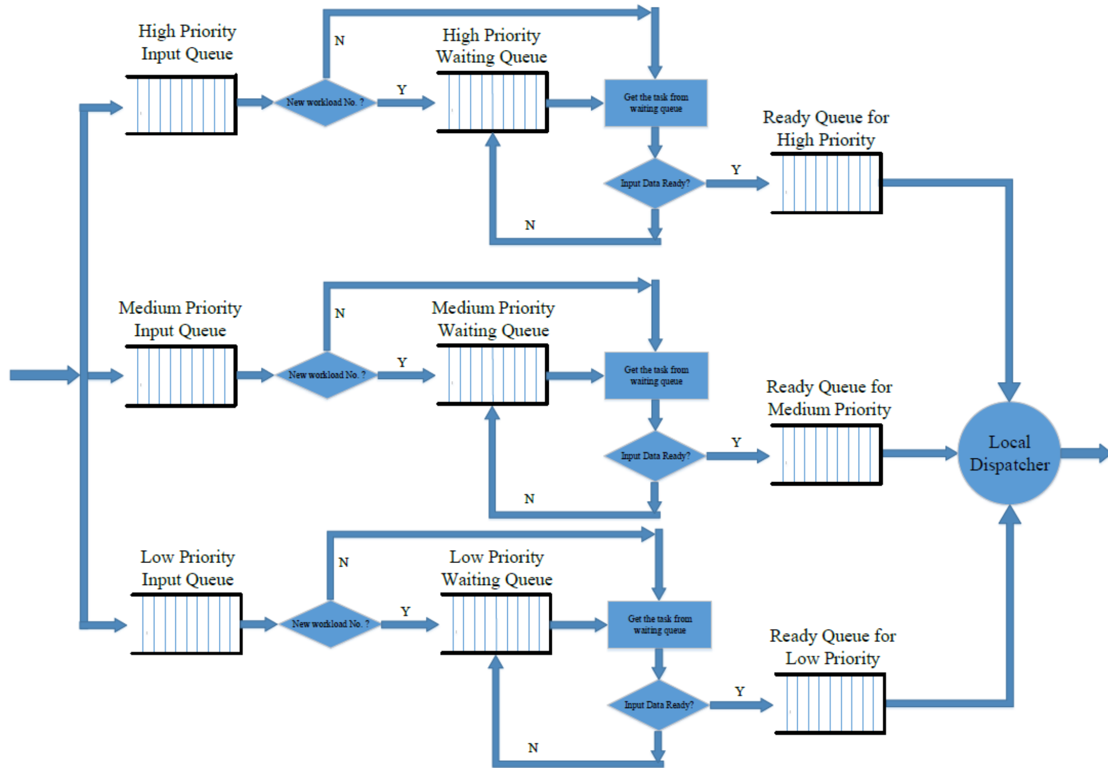

Figure 4.4 The structure of priority-based queue management in a local blade.

future work, is, first, implement the fit and light-weight preemptive scheduling algorithm(a combination of Bestfit and Short-Job-First algorithms) in such a way that the method meets the timing deadlines. Second, comparing the experimental results(a real complicated DAG) with the theoretical expectation. The available scenario is based on the availability of current framework with 144 cores. This 144 comes from, 6 blades that each blade includes two CPU, and each CPU includes 12 physical cores, which can run at 3 GHz.

4.1.4 Optimized queue-based data-driven task scheduling

The optimized queue-based data-driven task scheduling take into account not only to reduce the number of functions, calling function and number of queues, but also to improve the performance of the CPU cores and bring them to the desired level of 100% efficiency.

As shown in the Algorithm 4, which is the optimized version of the Algorithm 3, using a simple example, we describe the implementation in further detail.

In the DAG below, the research includes several tasks, each of which can start according to the readiness of the parent's task.

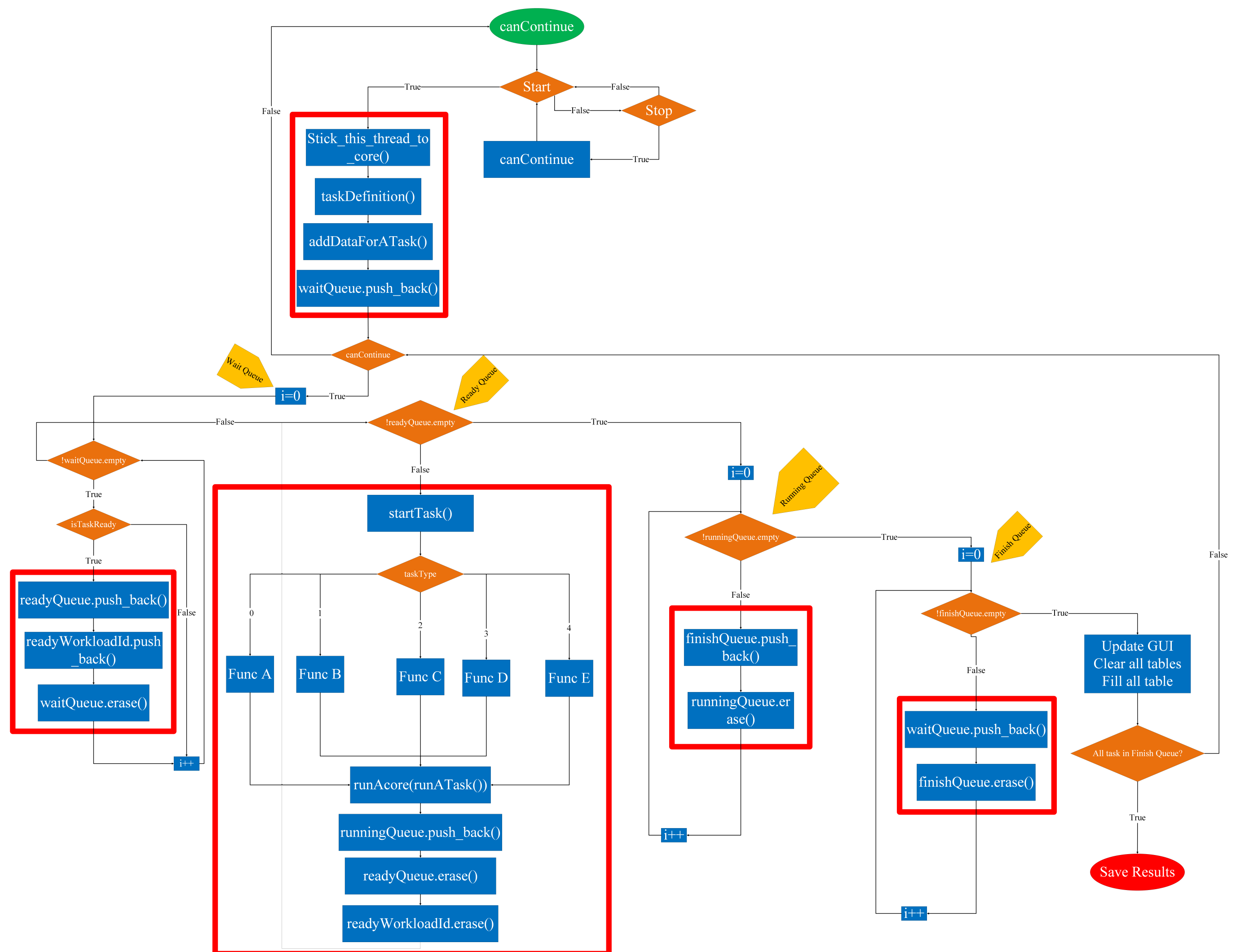

Figure 4.5 Flowchart of Quebe-based Data-Driven Task Scheduling.

The following parameters are considered to initialize the program.

```
#define CORE_MAX = 4;              // The number of cores
#define WORKLOAD_MAX = 8;          // The number of Workloads
#define DEEP = 2;                  // Size of task FIFO for each core
#define CHILD_MAX = 4;             // The maximum number of Child
#define TASK_CONUT_MAX = 13;       // Total Number of Tasks
#define READY_LOOP_DEEP = 26;      // The size of Ready Queue
```

We explain the problem in this manner. According to Figure 4.6 there are 13 tasks(including

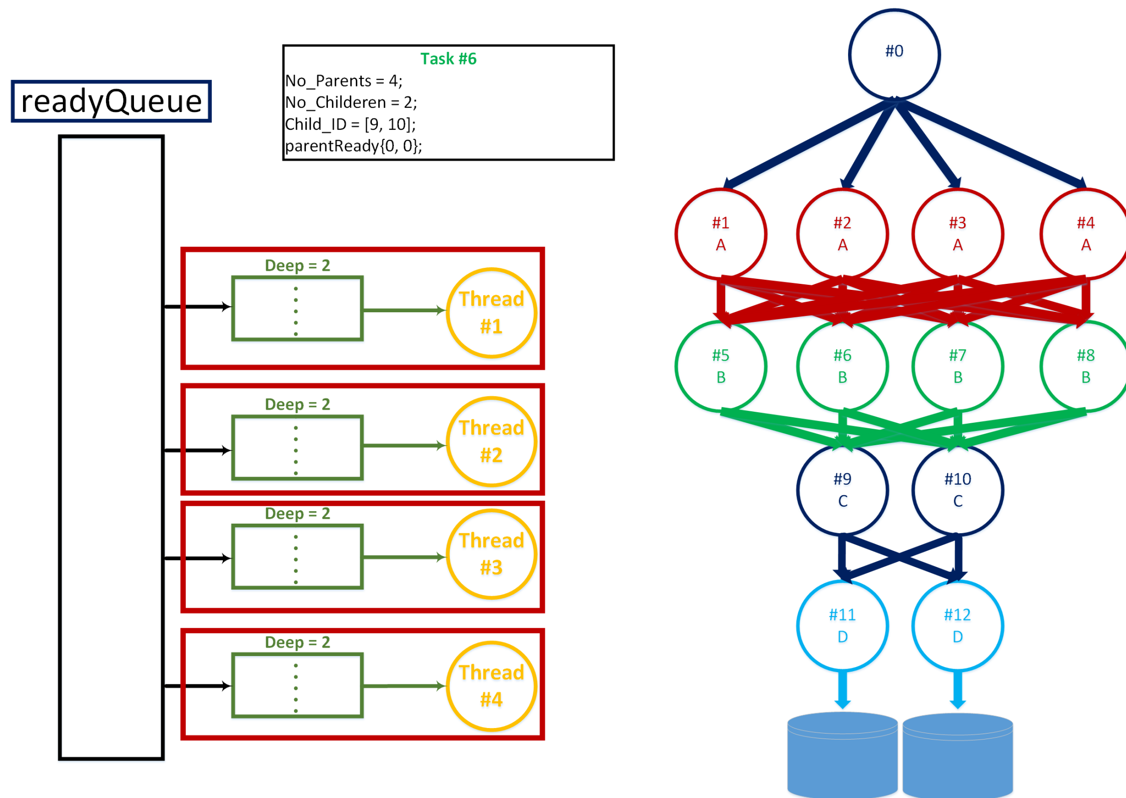

Figure 4.6 Example of optimized version for first version.

Task #0 which is the parent of first layer tasks), and 4 cores(CORE_MAX) are considered. We could use these cores simultaneously; each of which cores preceded by itself a queue(FiFo) called assignedQueue with a length of, for example, DEEP = 2, which is empty at first.

Tasks that are ready to enter the cores are placed within these queues, respectively. On the other hand, we have a large FIFO called readyQueue. This queue contains all the tasks that are ready to run, which means that its parents' data is ready. In principle, how the tasks can enter into the assignedQueue from readyQueue is a problem that will be studied later(the final scheduler). At the moment, the tasks with the priority entering to readyQueue move into the assignedQueue queues by calling the allocateTask function. So we have an array of queues called assignedQueue and a long queue called readyQueue;

– int assignedQueue[CORE_MAX][DEEP] = (1,0), (2,0), (3,0), (4,0);
– int readyQueue[MAX_TASK_CONUT];

For each task, the following information is available;
– int parentCount; // The number of parents
– int childCount; // The number of children

– int child_task_ID[CHILD_MAX]; // The ID of each child
– int parentReady[WORKLOAD_MAX];// The number of parents done for a given workload

The program initialize with function init_DAG;
– int init_DAG(DAG *dag); // initialize the DAG

In the beginning, the DAG is initialized with calling the init_DAG function. Afterward, Task #0, which in fact, is the mother or root of all the first-level DAG is ready. Subsequently, we look at all the children of this task and then increase the amount of parentReady for all children of each WorkLoad(this is done with the command taskDone(0, 1, 0) at the beginning of the main function that does not need to call the allocateTask function.). Therefore, parentReady parameter switches the numbering of Tasks#1 to Task#4 from zero to one. Since the parentCount is 1 for all 4 tasks, the 4 tasks enter the readyQueue.

Now we have to look at which worker are in place in their assignedQueue for new tasks. Because we are in the beginning, we can add up to 2 tasks to each assignedQueues(since DEEP parameter is set to 2). However, since tasks 5 through 8 are not ready to start, they cannot enter the readyQueue.

Now using the allocateTask function, we are assigning tasks to each assignedQueue which are empty, respectively. Several DEEP parameters for each assignedQueue are added after the assignment. Also, the relationship between tasks cores and workloads(WL), are printed as output. As soon as each assignedQueue is not empty(DEEP> 0), we can read the corresponding queue from the beginning of each queue, and then send it to the corresponding core.

Whenever each of the tasks 1 to 4 is completed by the worker, a taskDone call and is added one to parentReady for each child's task. The process continues to run until the parentReady parameter of each child's task equals to its parent's number; in such a case, the child's task moves to the readyQueue.

– int taskDone(int task_ID, int WORKLOAD_ID, int Core_ID)
– for all child: increment parentReady[workload_ID]
– for all child: if (parentReady[WORKLOAD_ID] == parentCount)

if one or several child tasks are ready, add them to the readyQueue.

– assignedQueue[Core_ID] = assignedQueue[Core_ID] - 1;
– addNewTaskToThisCore;

The taskDone function is called as soon as a worker comes to an end; while entailing the parameters of int task_ID, int workload_ID, int Core_ID. After calling the taskDone, parentReady increases all related children by one (parentReadyp[child_task_ID][workload_ID]

= parentReady[child_task_ID][workload_ID] + 1).

The method checks whether every child is eligible to enter the readyQueue. In case, they were entered into the readyQueue, if the number of parentReady of each child was equal to the number of the parents of that child, the allocateTask function transfers the corresponding task from the readyQueue to the assignedQueue list. Such a process reduces the DEEP parameter by one.

Now, if there are more than one queue of the worker whose DEEP is less than the set value(in this case is 2), a new task for it can be sent to the queue, which has more space. The allocateTask function performs the following tasks when called.

– allocateTask(int task_ID, int WORKLOAD, int processor_ID)
– assignedQueue[processor_ID]++
– print the allocation

This function increases the number of members in the queue. The function also keeps track of tasks and their workloads, as well as their respective assigned worker. In the Figure A.7 the definition of addTask function depicted. As illustrated, the assumption is that each processor core has different processing speeds. Assuming variant speeds for different processors makes the simulation more realistic. For example, core#1 is 5 times faster than core#0; core#2 is 2 times faster than core#1; core#3 is 2 times faster than core#2.

```
void addTask(int task_ID, int parentCount, int child_task_ID[], int childCount, int processingTime)
double core_speed[CORE_MAX] = {1.0, 5.0, 10.0, 20.0};
```

Figure 4.7 addTask function parameters.

The definition of initDag with the help of an example illustrated in Figure 4.8. Based on the this function the DAG, as an simple example illustrated in Figure 4.6, is interpreted for the program.

```
void initDag()
{
    int ch0[] = { 1, 2, 3, 4}; addTask( 0, 0, ch0, 4, 1000);

    int ch1[] = { 5, 6, 7, 8}; addTask( 1, 1, ch1, 4, 2000);
    int ch2[] = { 5, 6, 7, 8}; addTask( 2, 1, ch2, 4, 200);
    int ch3[] = { 5, 6, 7, 8}; addTask( 3, 1, ch3, 4, 20000);
    int ch4[] = { 5, 6, 7, 8}; addTask( 4, 1, ch4, 4, 1200);

    int ch5[] = { 9, 10}; addTask( 5, 4, ch5, 2, 300);
    int ch6[] = { 9, 10}; addTask( 6, 4, ch6, 2, 3000);
    int ch7[] = { 9, 10}; addTask( 7, 4, ch7, 2, 130);
    int ch8[] = { 9, 10}; addTask( 8, 4, ch8, 2, 300000);

    int ch9[] = { 11, 12}; addTask( 9, 4, ch9, 2, 400);
    int ch10[] = { 11, 12}; addTask( 10, 4, ch10, 2, 400000);

    int ch11[] = {}; addTask( 11, 2, ch11, 0, 500000);
    int ch12[] = {}; addTask( 12, 2, ch12, 0, 500);
}
```

Figure 4.8 An example of initDag function with 13 tasks.

4.1.5 Software + Hardware task scheduling in data-driven concept

The main goal of implementing data-driven strategies is not only to make optimal use of available resources but also to achieve better performance.

In this regard, the propose algorithms were simulated and implemented. Here, we are looking to optimize the software-based algorithm of the current models using the combination of software and hardware. The reason for using hardware is not only to take advantages from parallelization, but also to meet the deadlines and to have deterministic processing time.

The Zynq-7000 family, which integrates a complete ARM Cortex-A9 MPCore processor-based system on a 28 nm FPGA for system architects and embedded software developers, provides a hardware environment for developing and evaluating designs of interest.

Zynq is a class FPGA types. The internal; structure of the Zynq comprise an ARM microprocessor along with FPGA fabrics. This is the simplest combination, and some Zynq devices may comprise many more features as we will describe in the examples below. Wherever we need to process parallel information (FPGA) alongside a micro-controller or microprocessor, we can use these kind of chips.

“The Xilinx Vivado High-Level Synthesis(HLS) tool transforms a C specification into a reg-

ister transfer level(RTL) implementation that one can synthesize into a Xilinx Field Programmable Gate Array(FPGA), like Zynq family. One can write C specifications in C, C++, or SystemC, and the FPGA provides a massively parallel architecture with benefits in performance, cost, and power over traditional processors." [40] In Appendix A, we investigate different aspects of Vivado HLS.

One of the most critical phases of HLS is scheduling. Scheduling determines which operations occur during each clock cycle based on:

1. Length of the clock cycle or clock frequency

2. Time it takes for the operation to complete, as defined by the target device

3. User-specified optimization directives

If the clock period is longer or a faster FPGA is targeted, more operations are completed within a single clock cycle, and all operations might complete in one clock cycle. Conversely, if the clock period is shorter or a slower FPGA is targeted, high-level synthesis automatically schedules the operations over more clock cycles, and some operations might need to be implemented as multi-cycle resources. This critical phase makes it possible for us to reduce the clock rate in hardware in comparison to the software level and to compensate this shortage as much as possible.

Using Vivado HLS, the program is converted into a hardware code, and the initial results, including the system clock and the use of logical cells, LUTs, memory usage and etc. are reported.

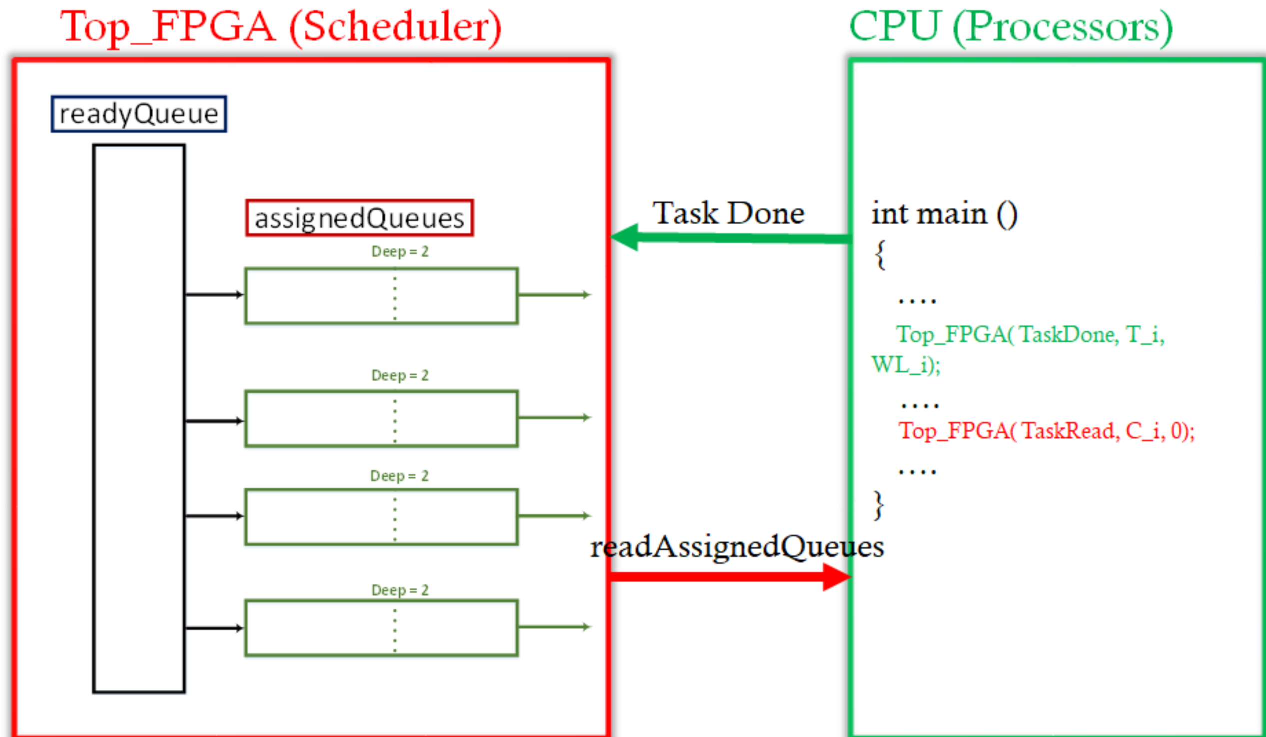

Figure 4.9 Software + Hardware Task Scheduling in the Data Driven Concept

An implementation of a hardware scheduler is usually capable of supporting only one scheduling algorithm. The hardware can support a narrow range of applications, which works well under the same scheduling scheme. Unlike software components, a hardware unit is less flexible and more difficult to modify after implementation.

As a result, we can say hardware solutions are frequently avoided; however, if the hardware scheduler is configurable to support several scheduling algorithms, then the hardware solutions become more flexible. So not specifically for the sake of speed but to enhance the performance and to improve the predictability of real-time systems, the hardware approach is preferred. Using hardware(like FPGA) in parallel processing is preferred over using the software since the speedup is achievable by running multiple tasks simultaneously.

4.2 Experiments and results

The sub-sections below will present the results obtained and the methods used through our experiments and tests.

4.2.1 The first implementation of data-driven task scheduling

Figure 4.2 shows the structure of the DAG used in the following experiments. In the experiments, the 8 workloads are fed into the DAG. Figure 4.10 shows this scenario with 4 active cores. The performances of the cores are traced using LTTng. “LTTng is a system software package for correlated tracing of the Linux kernel, applications, and libraries” [41]. As shown in Figure 4.10, the core utilization is close to 100% for each worker in this program. Close-To-Complete CPU usage is our primary result.

For better and more accurate analysis, as well as more accurate testing of the utilization of the cores, more tests with more accurate tuning features are needed. The potential future work would ideally include such evolved and extended features.

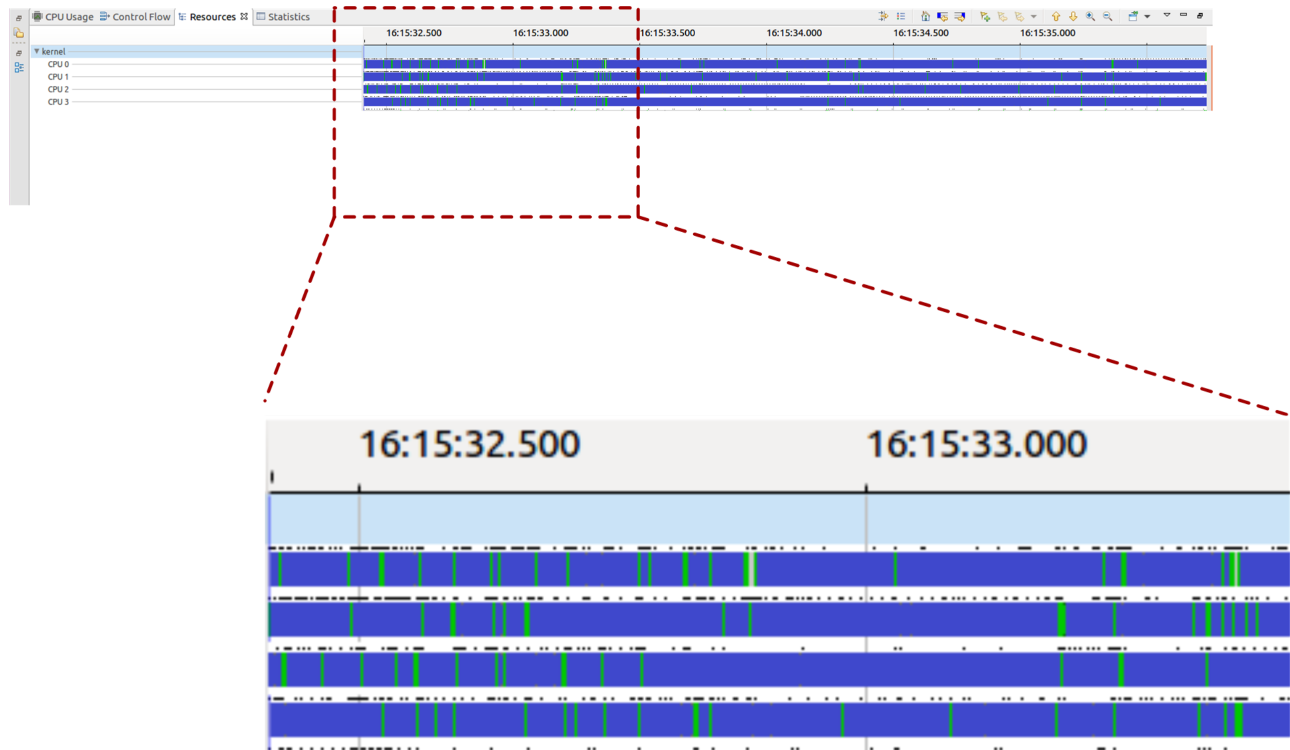

Figure 4.10 Core utilization as reported by LTTng, dark blue stands for busy state and green for wait/idle state.

Under the limitation of having 3 concurrent workloads served at any given time, the results traced by LTTng show that the relative performance is nearly 100%, as shown in Figure 4.1.

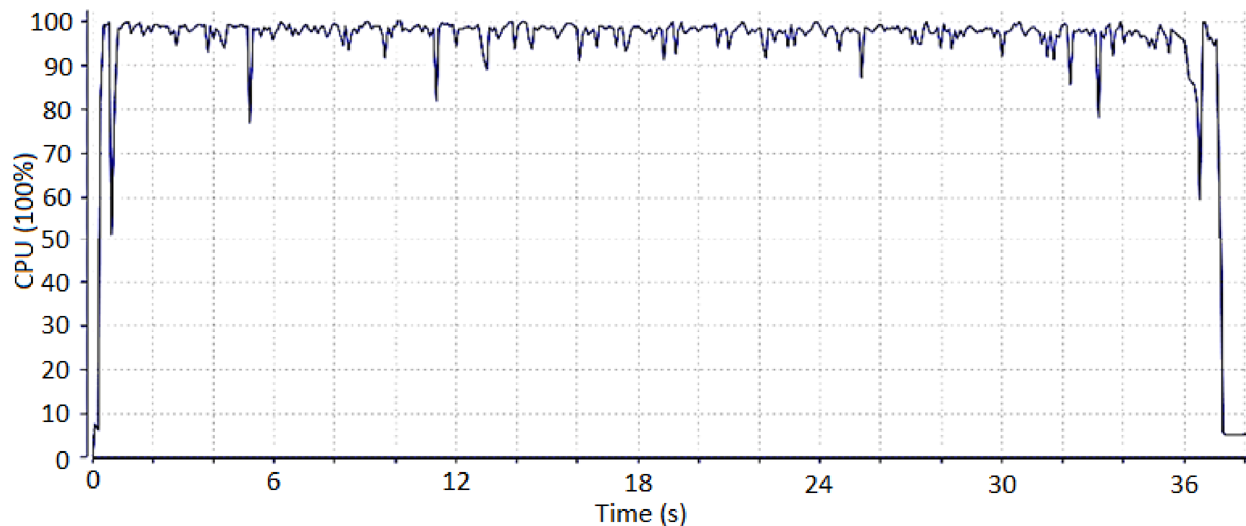

Figure 4.11 CPU usage of real DAG with total 24 workloads, 3 concurrent workloads, and 24 active cores.

4.2.2 The optimized version of data-driven task scheduling

As shown in the profiler, reported in Figure 4.12, the preliminary results of the first implementation versus the previous version are compared. As shown in, Figure 4.12, not only the number of functions, number of calling functions and number of queues are less used in the current version, but also the functions used are operating almost optimally with the net performance approaching 100% most of the time.

Figure 4.13 to Figure 4.25 show two comparisons: the method of assigning each task to each core in a different workload(8 workloads from 0 to 7 in this instance), as well as the amount of processing time of each core for each task. For example, in Figure 4.13, the first column, WL_0, C_0 shows that Task#0 of Workload#0 is running on Core#0 and takes 1,136,951(ns) execution time. The last column also shows that Task#0 of Workload#7 is running on Core#2 and takes 164,865(ns) time to complete.

In this scenario, the assumption is that each processor core has different processing speeds. Assuming variant speeds for different processors makes the simulation more realistic.

For example, core#1 is 10 times faster than core#0; core#2 is 10 times faster than core#1; core#3 is 10 times faster than core#2, and so on.

To simulate the processing time of each task, we use the usleep() function, which has an input value for each task. The input values depend on the target task at any given time. For

Functions	# of Calls	Instruction Read per call	
addTask	12/13	20123	76
addToReadyQueue	12/13	4122	18
removeFromReadyQueue	12/13	348	8
addToAssignedQueue	--/13	--	50
removeFromAssignedQueue	--/13	--	28
allocateTask	37/13	518	75
addToWaitQueue	12/--	7585	--
addToRunningQueue	12/--	5333	--
removeFromWaitQueue	12/--	10195	--
removeFromRunningQueue	12/--	8993	--

Figure 4.12 Optimized version of Task Scheduling vs. The first version

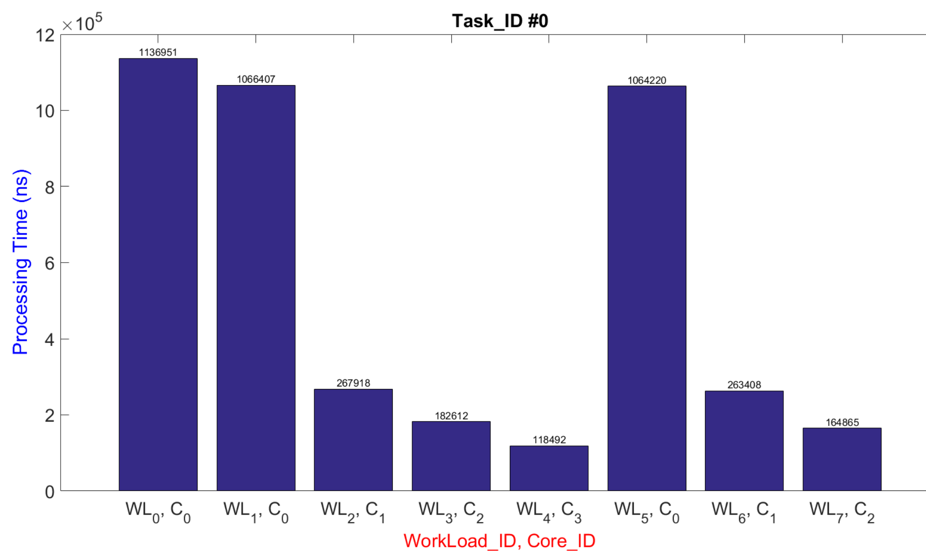

Figure 4.13 Results for Task #0.

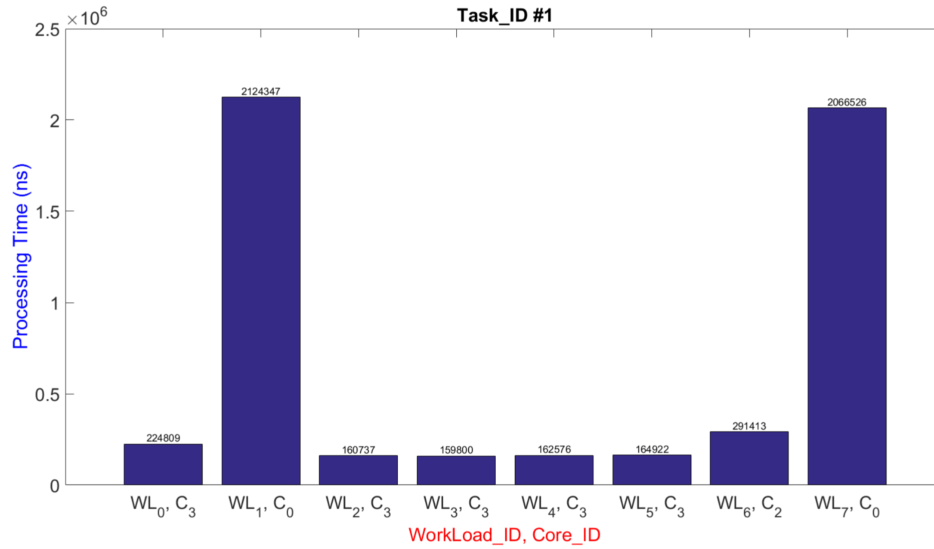

Figure 4.14 Results for Task #1.

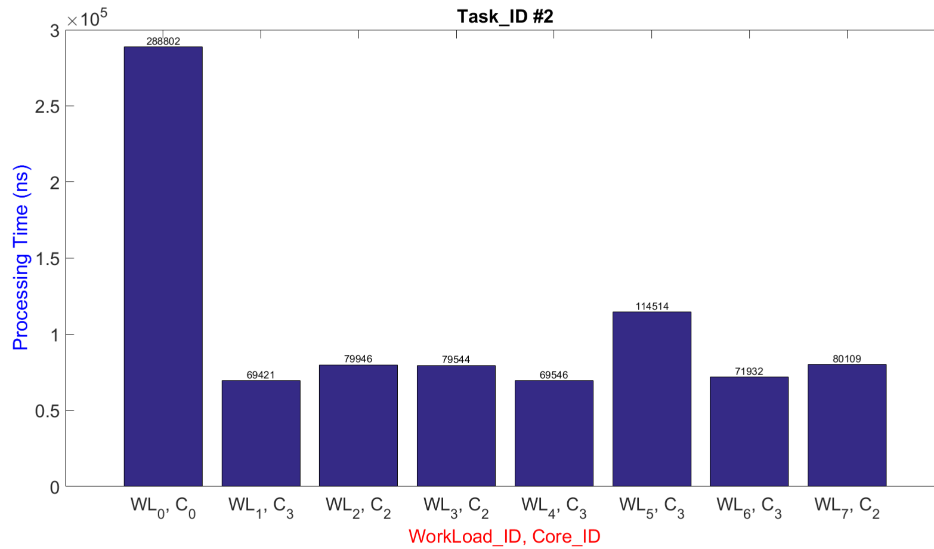

Figure 4.15 Results for Task #2.

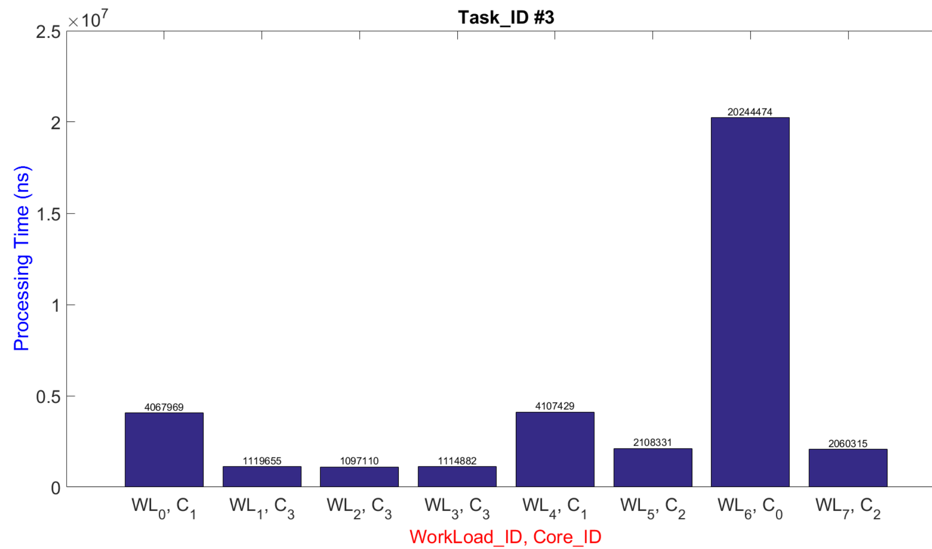

Figure 4.16 Results for Task #3.

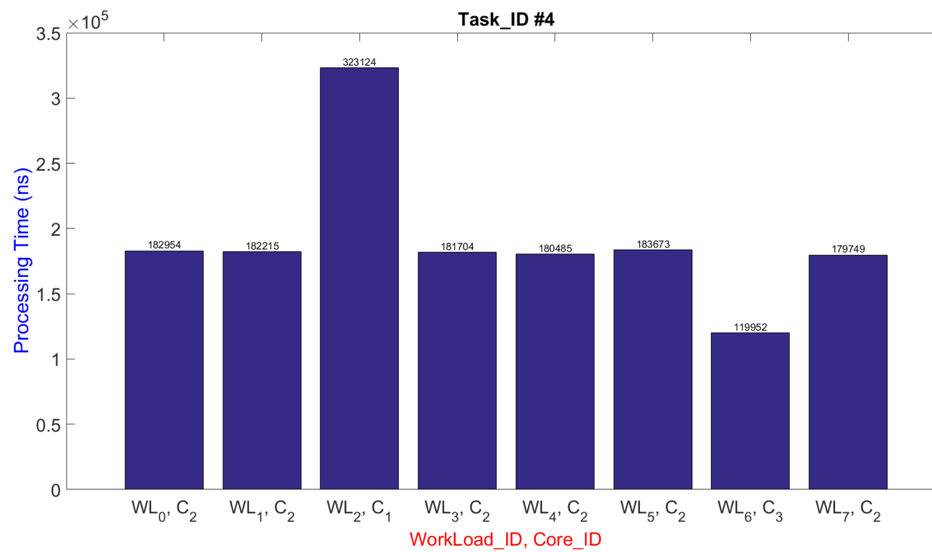

Figure 4.17 Results for Task #4.

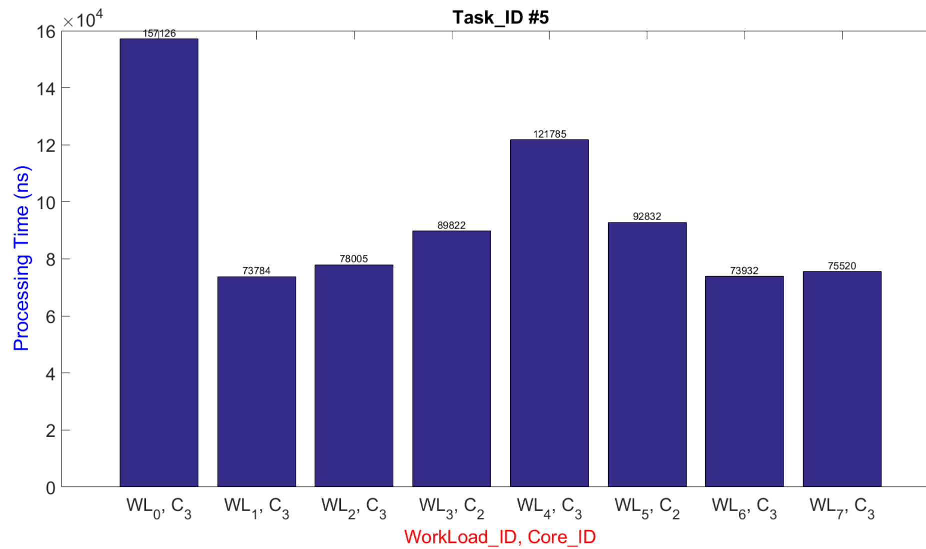

Figure 4.18 Results for Task #5.

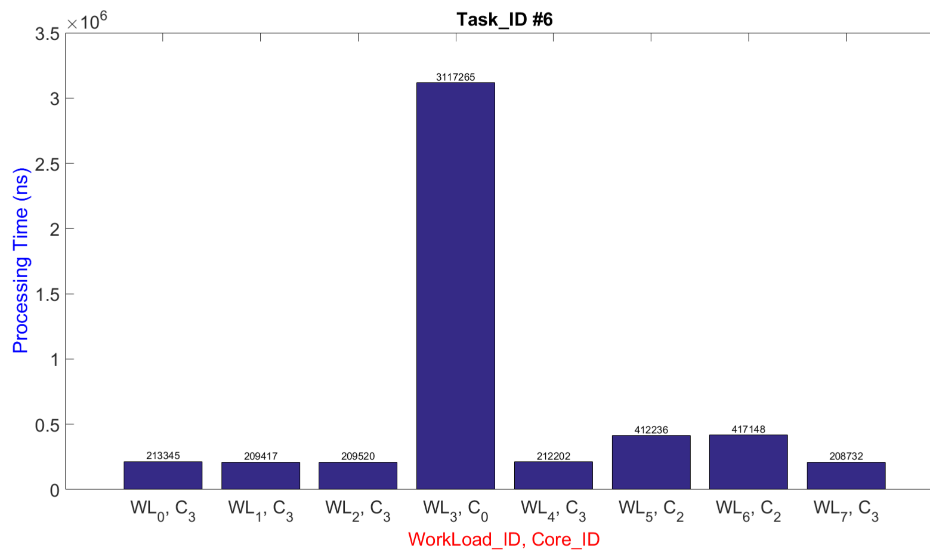

Figure 4.19 Results for Task #6.

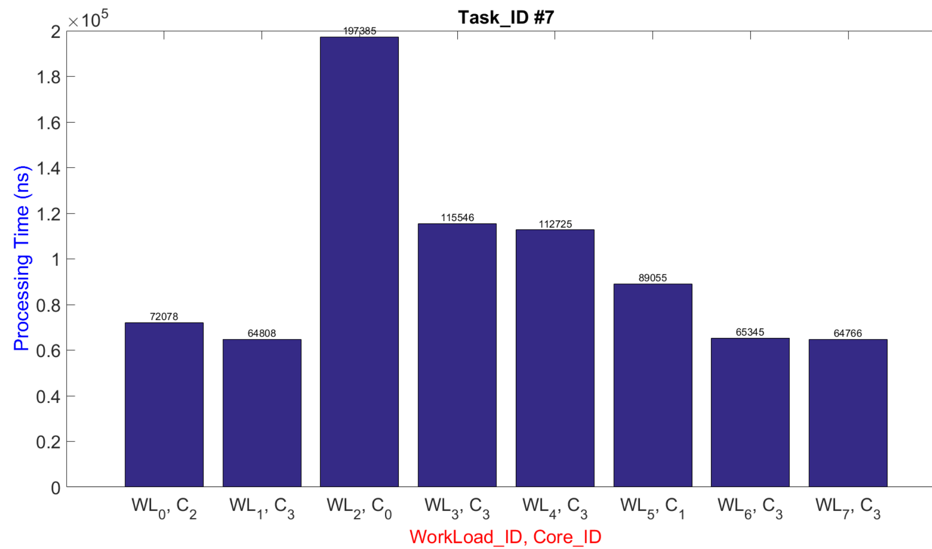

Figure 4.20 Results for Task #7.

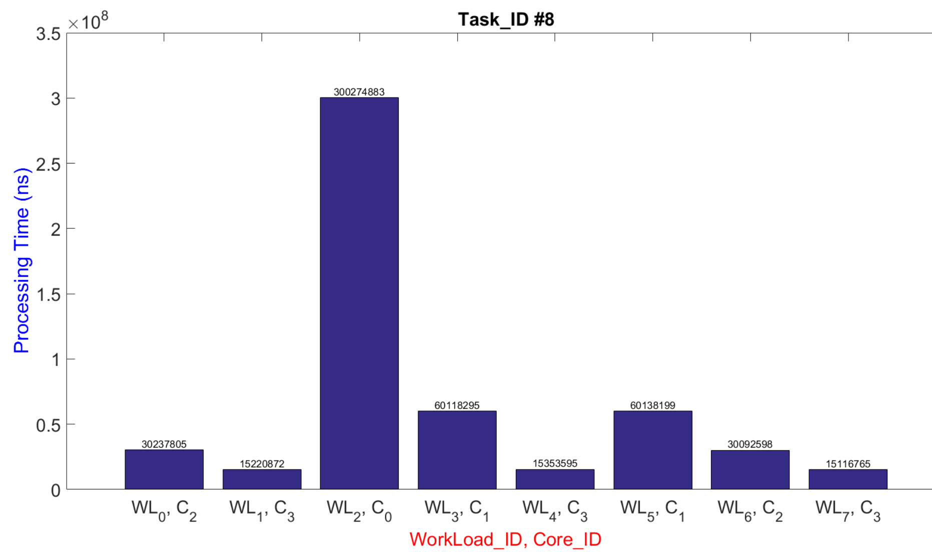

Figure 4.21 Results for Task #8.

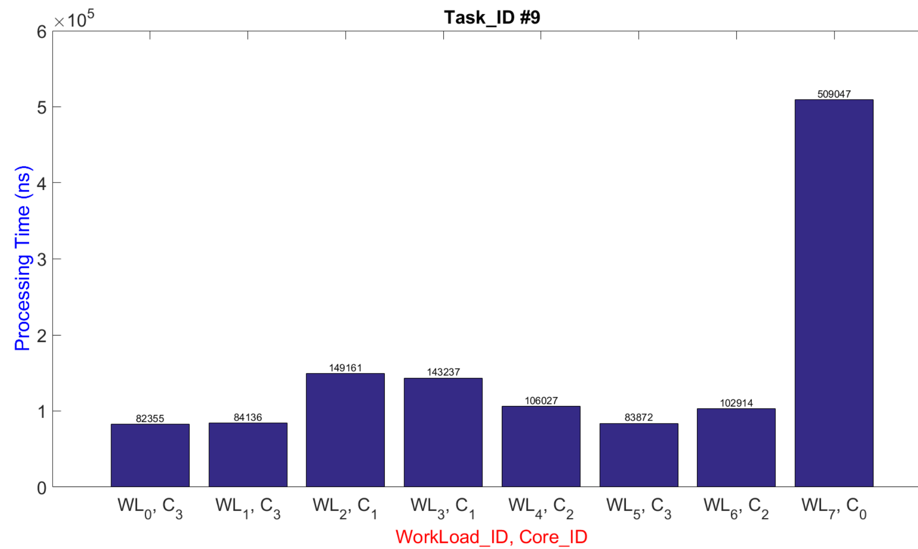

Figure 4.22 Results for Task #9.

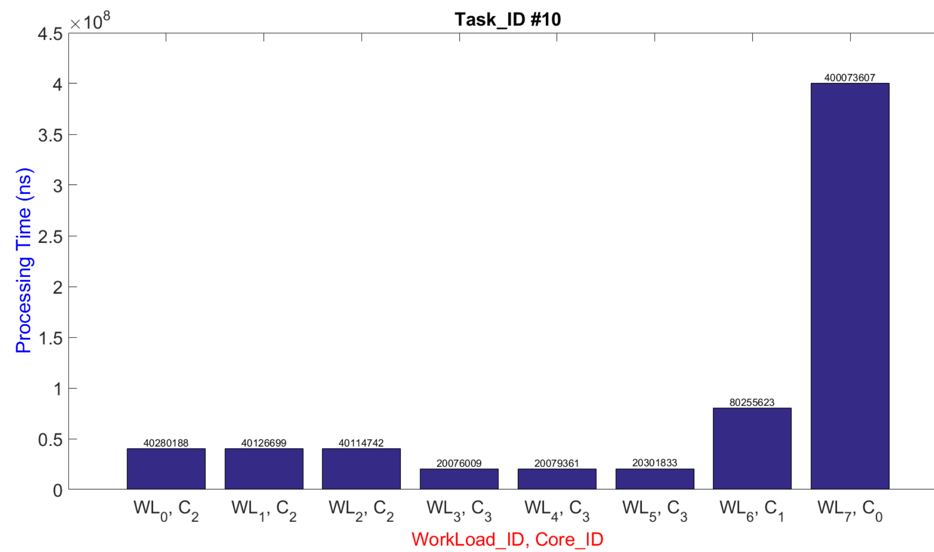

Figure 4.23 Results for Task #10.

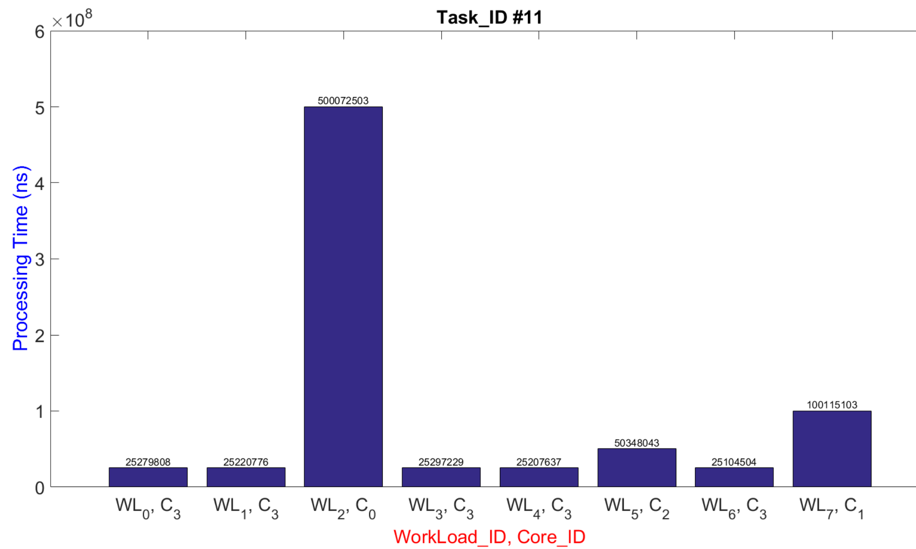

Figure 4.24 Results for Task #11.

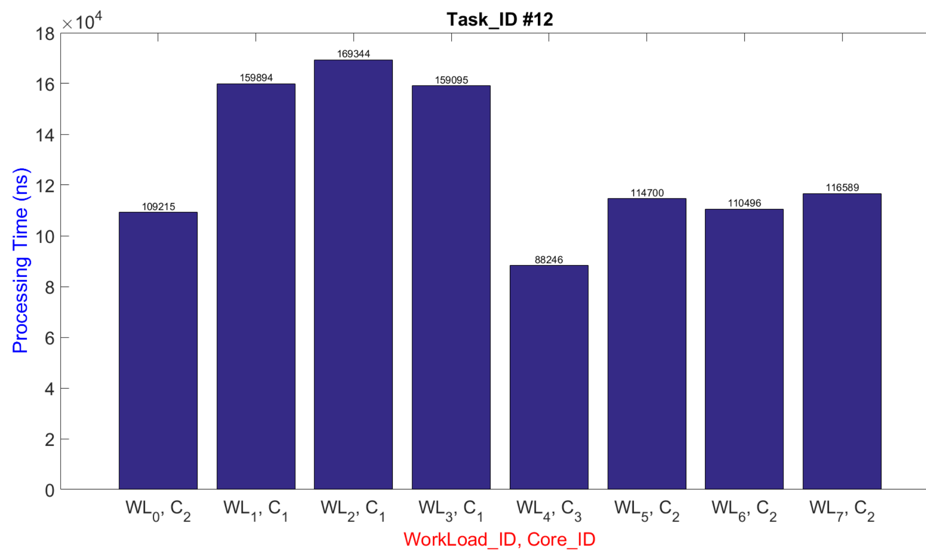

Figure 4.25 Results for Task #12.

example, for the Task#0, the input value of 1000 represents the processing time equivalent to 1000 microseconds. Finally, the usleep() function is likely to evolve into more elaborate versions in future follow-up research works.

Algorithm 3 Priority-Queue-Based Task Scheduling Data-Driven

Inputs :

taskDefinition : Registration of DAG includes : task_ID, task_layer, task_type, task_WCET, DAG_type, parents_task_ID, child_task_ID;

CORE_MAX : Define the maximum number of core involved in this DAG;

testNumber : Define the test number;

Outputs :

task_input : The data input of each task;

task_output : The data output of each task;

taskTime : Shows that each task meet or violate the WCET;

```
1:  if (startbuttom) then
2:      stick_this_thread_to_core(); // Assigns main thread to core 0;
3:      taskDefinition();         // Call the taskDefinition just here
4:      addTask();                // Start adding;
5:      SendFirstData;            // Send the first Woarkloads data to the tasks in layer #0;
6:      addDataForATask();
7:      waitQueue.push_back();    // Adds all tasks to the wait queue
8:      while (canContinue) do
9:          WaitQueue;            // Wait state to the Ready state, if all data of parents is
    received then task is ready.
10:         while (!waitQueue.empty) do
11:             if (isTaskReady) then
12:                 readyQueue.push_back();
13:                 readyWorkloadId.push_back();
14:                 waitQueue.erase();;
15:             end if
16:         end while
17:         ReadyQueue;           // Ready to Runnig state, Finds the highest priority task
    and find a free core to run the task.
18:         while (!readyQueue.empty()) do
19:             FindsHighestPriorityTask;  // Find a task in a core.
20:             startTask();               // Sets start state of task and runs it
21:             switch (task_type)
22:             case 0
23:                 Function_A;
24:             end case
25:             case 1
26:                 Function_B;
27:             end case
28:             case 2
29:                 Function_C;
30:             end case
```

```
31:             case 3
32:                 Function_D;
33:             end case
34:             case 4
35:                 Function_E;
36:             end case
37:             end switch
38:             runACore(runATask());  // Runs a task in a new thread, assign it to a core
39:             runningQueue.push_back();
40:             readyQueue.erase();
41:             readyWorkloadId.erase();
42:         end while
43:         RunnigQueue;                 // Running state to Finish state.
44:         while (!runningQueue.empty) do
45:             finishQueue.push_back();
46:             runningQueue.erase();
47:         end while
48:         FinishQueue;                 // Finish state to Wait state.
49:         while (!runningQueue.empty) do
50:             waitQueue.push_back();
51:             finishQueue.erase();
52:         end while
53:         if (AllTaskInFinishQueue) then
54:             Save Results;
55:             break;
56:         end if
57:     end while
58: end if
```

Algorithm 4 Optimized Task Scheduling

Inputs :

```
#define CHILD_MAX // Maximum number of children for a task;
#define WORKLOAD_MAX // The maximum number of DAG processed simultaneously
#define DEEP 2 // Size of assignedQueue for each core
#define CORE_MAX 4 // The number of cores
```

```
1:  for All Task do
2:      int nb_parents; // The number of parents
3:      int nb_children; // The number of children
4:      int child_ID[MAX_CHILDREN]; // the ID of each child
5:      int parent_ready[MAX_WORKLOAD]; // the number of parents that are done for
        a given WORKLOAD
6:  end for
7:
8:  int assignedQueue[MAX_CORE]=0, 0; // The number of tasks already assigned to each
    core for each workload
9:  int readyQueue[MAX_CORE]=0; // The Queue to keep the ready tasks.
10: int initDAG(DAG *dag) // initialize the DAG (each task with said parameters)
11:
12: int taskDone(int task_ID, int WL_ID)
13:     Update the status of the child tasks:
14:     for all child: increment parent_ready[WORKLOAD]
15:     if one or several tasks are ready, add them to the ready set
16:
17: for All Task do
18:     if parent_ready[WORKLOAD] == nb_parents then
19:       assigned_queue[processor_ID]–;
20:       Decrement the number of tasks assigned to processor_ID;
21:     end if
22: end for;
23:
24: if one or more core have less than DEEP task(s) in their queue then
25:       send them new tasks.
26: end if
27: Apply schedule algorithm.
28: allocate_task(int task_ID, int WL_ID, int Core_ID)
29:     Assigned_queue[Core_ID]++;
30:     Print the allocation;
31:
```

To get a better understanding of how the cores work, we compiled the results manually in a simulated environment(Modelsim). The task scheduling software, which is written in C language, generates several task-related characteristics. Another program, written in Matlab uses the mentioned output to later pass on to the next unit. The entire process until this stage takes place automatically. The Modelsim intakes the outputs of the Matlab program (VHDL_Generator.m). The VHDL program models the characteristics relative to the focus tasks.

As shown in Figure 4.26 through 4.28, the cores are in the busy and idle state. Each core performs a particular task of a distinct workload at a predefined processing time.

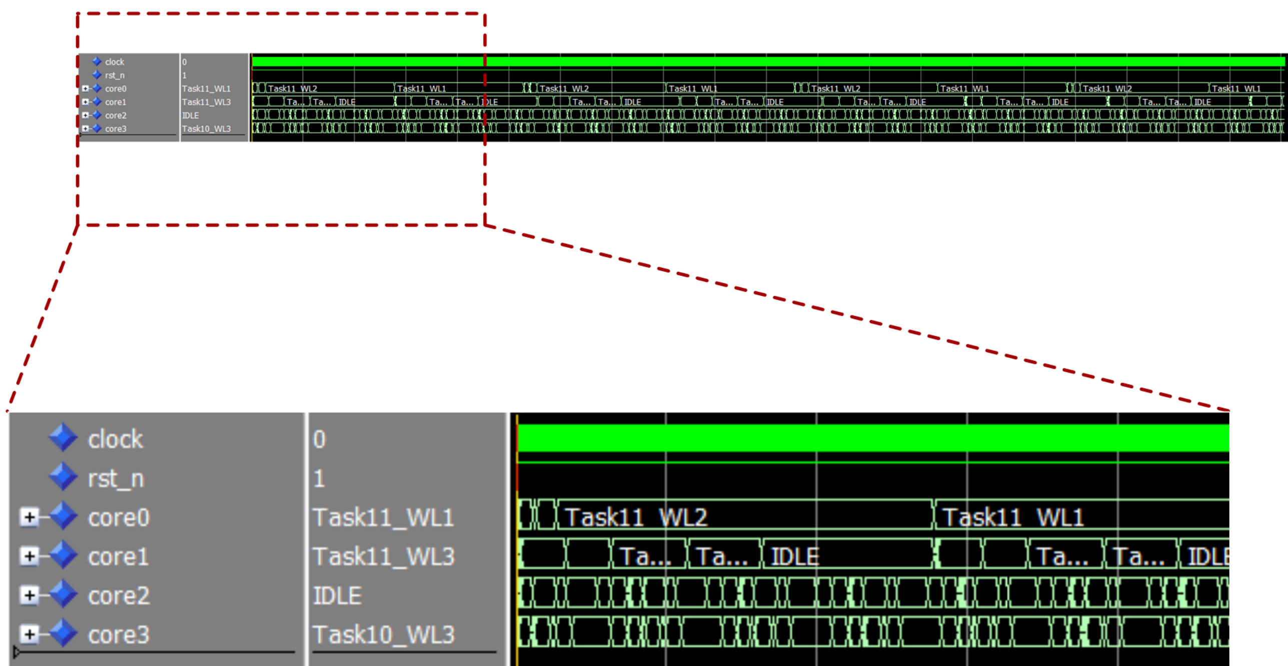

Figure 4.26 1.Busy and Idle states, 4 workers and 4 workloads, from start to stop point.

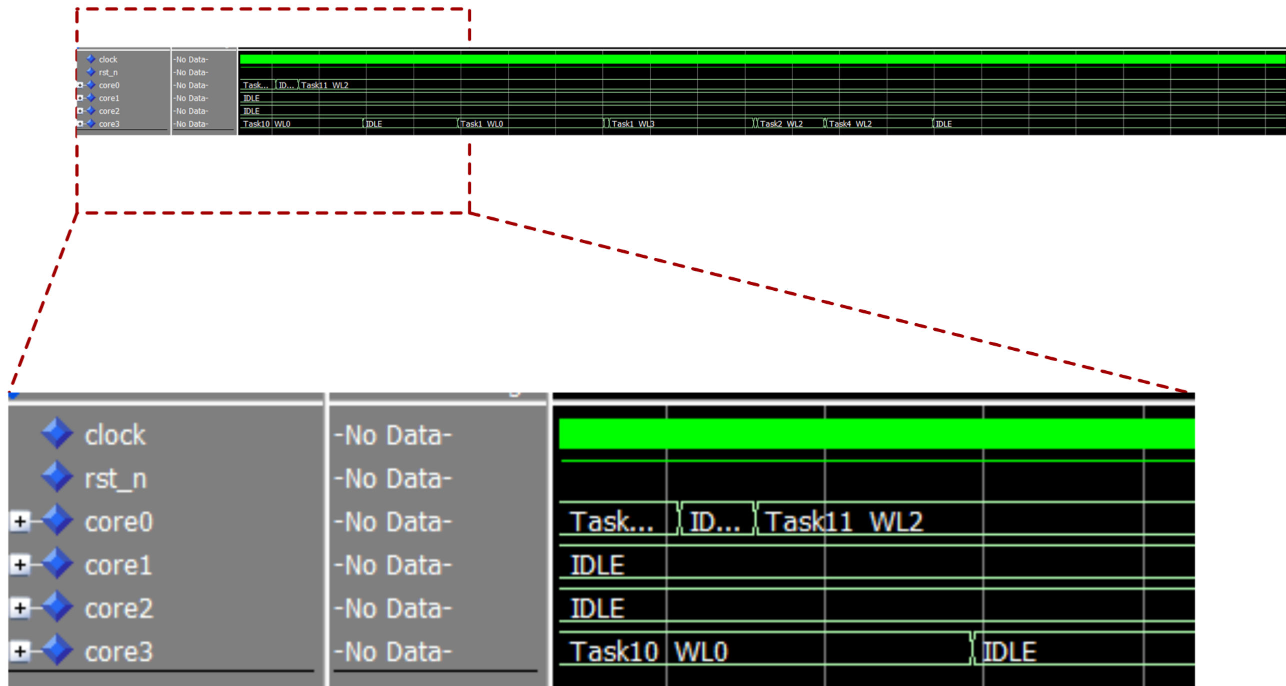

Figure 4.27 2. Busy and Idle states, 4 workers and 4 workloads, start point.

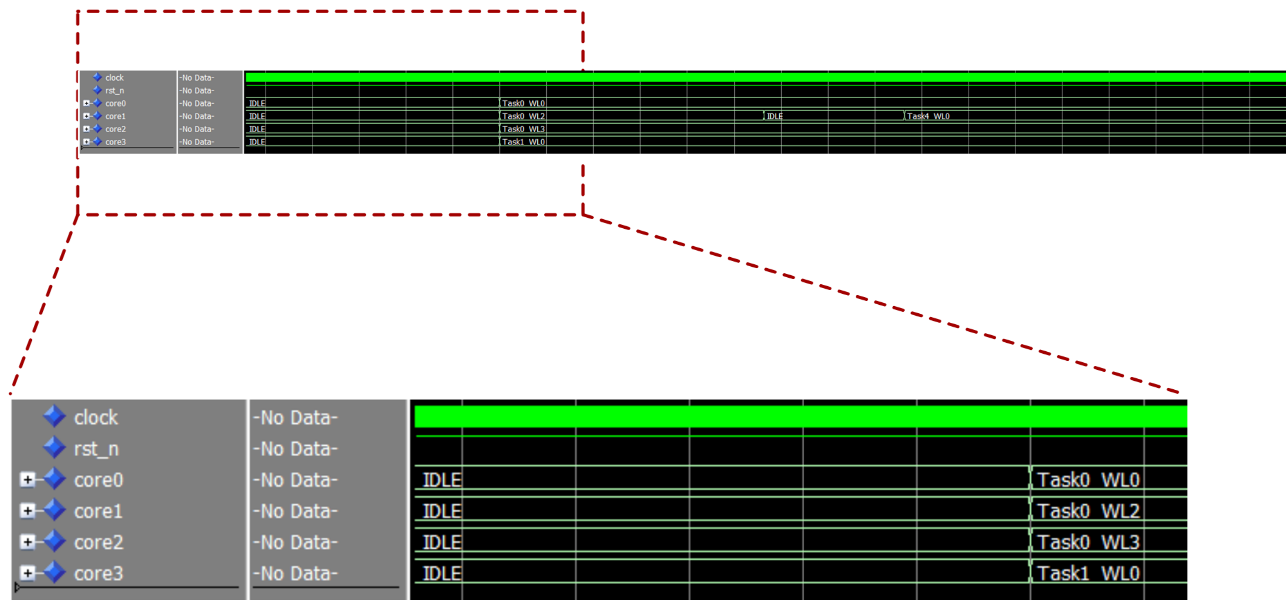

Figure 4.28 3. Busy and idle states, 4 workers and 4 workloads, stop point.

Figure 4.29 to Figure 4.38 show the method of assigning tasks to cores, focusing on measuring

core utilization and core idle time. This scenario includes 4 active cores and 4 workloads.

For example, in Figure 4.29, the first column $Core_0$(99.86%), shows that the $Core_0$ is in active mode(at 99.86% performance level) during the start-to-stop time of the application with 4 coming workloads. Also, the $Core_1$, $Core_2$, and $Core_3$ are active with performance levels of 99.83%, 99.82%, and 99.9% respectively.

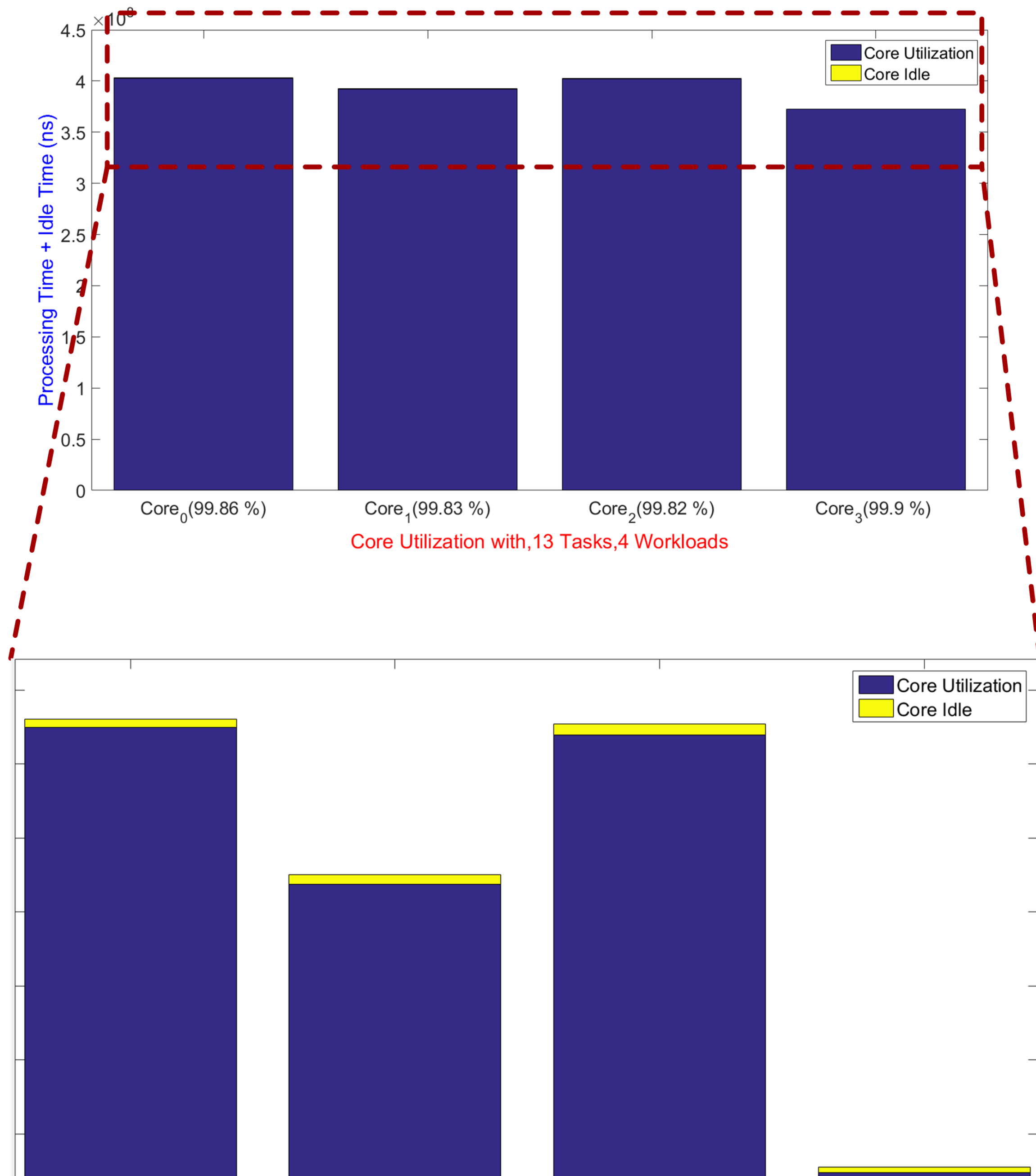

Figure 4.29 Core utilization with 13 Tasks, 4 workloads

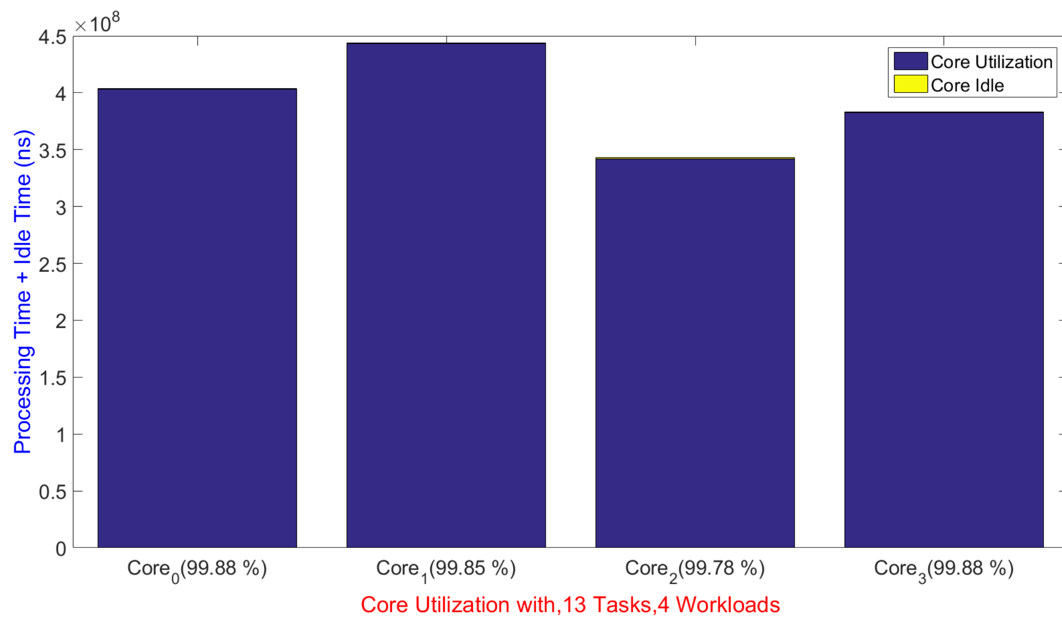

Figure 4.30 Core utilization with 13 Tasks, 4 workloads

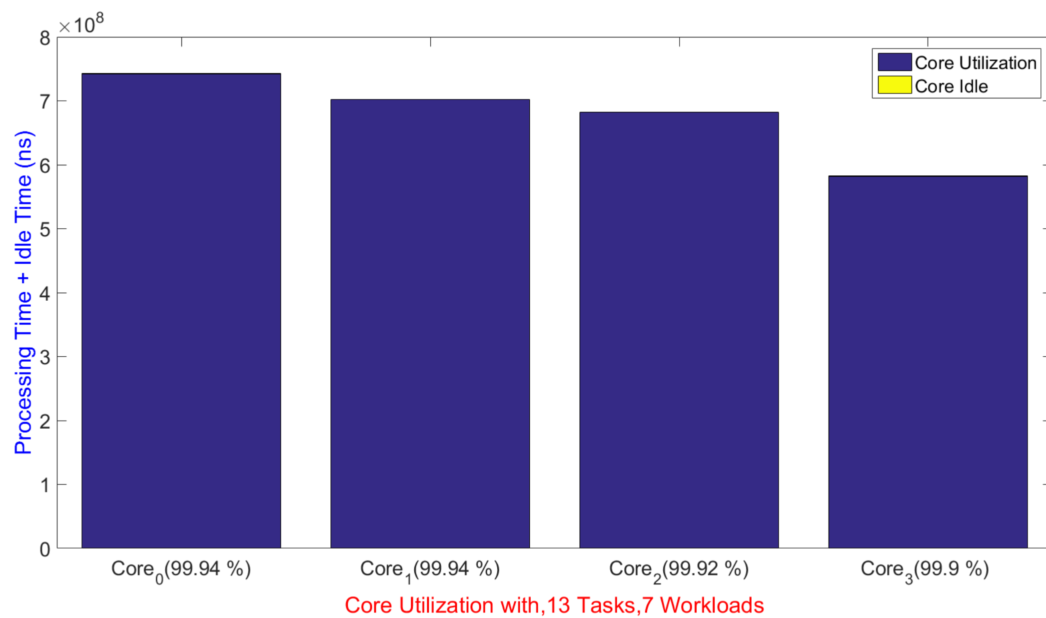

Figure 4.31 Core utilization with 13 Tasks, 7 workloads

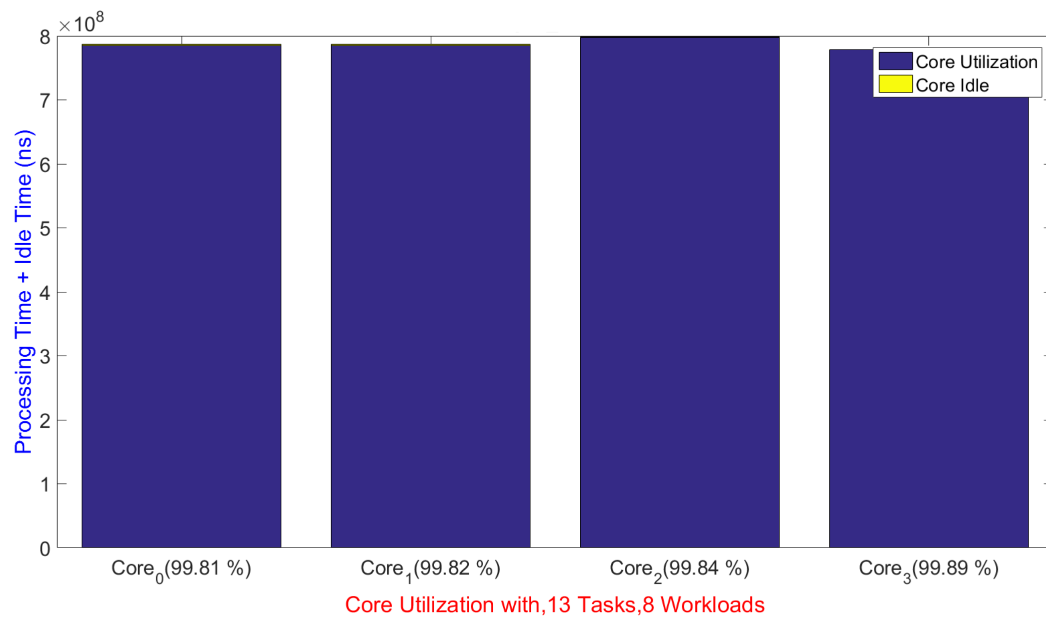

Figure 4.32 Core utilization with 13 Tasks, 8 workloads

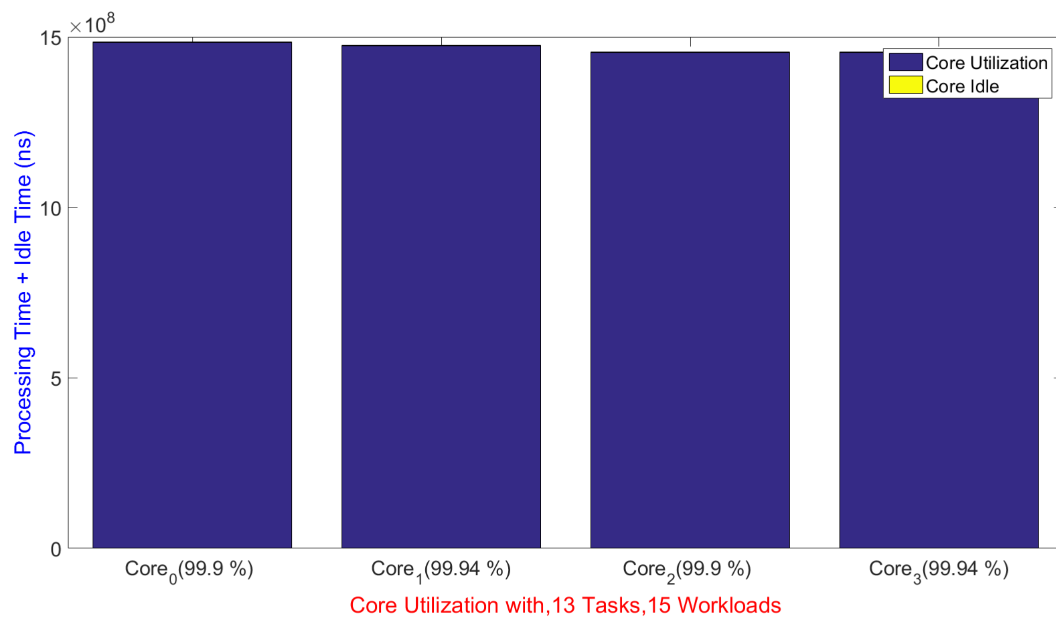

Figure 4.33 Core utilization with 13 Tasks, 15 workloads

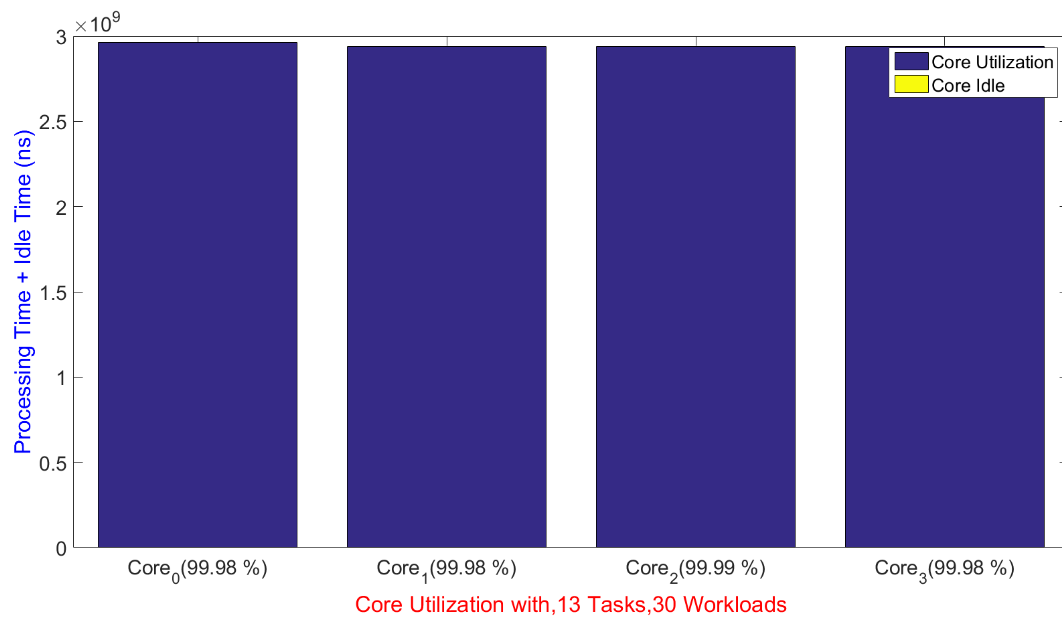

Figure 4.34 Core utilization with 13 Tasks, 30 workloads

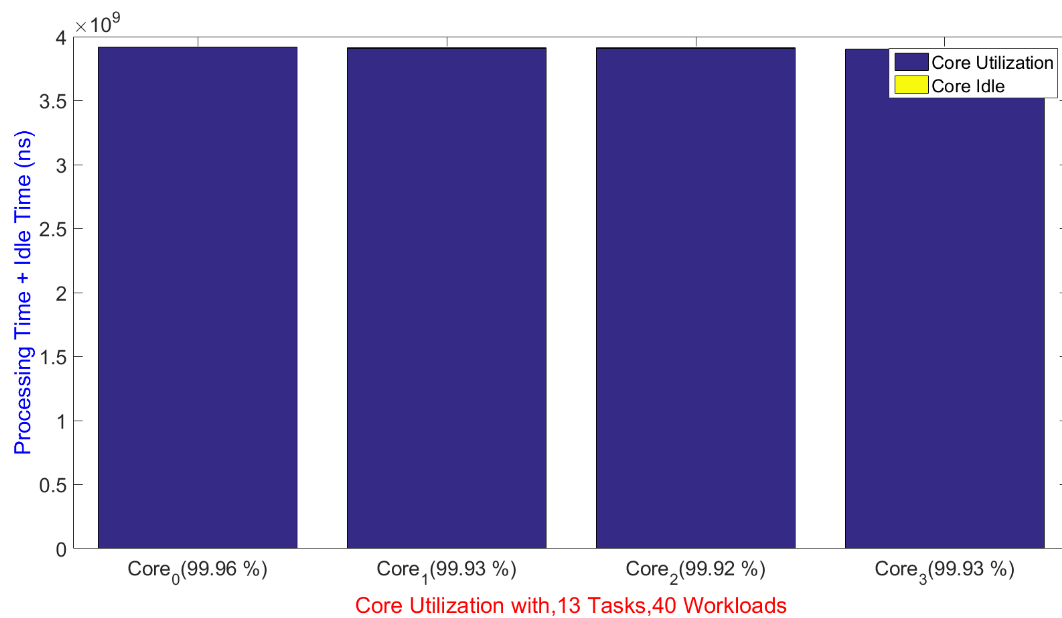

Figure 4.35 Core utilization with 13 Tasks, 40 workloads

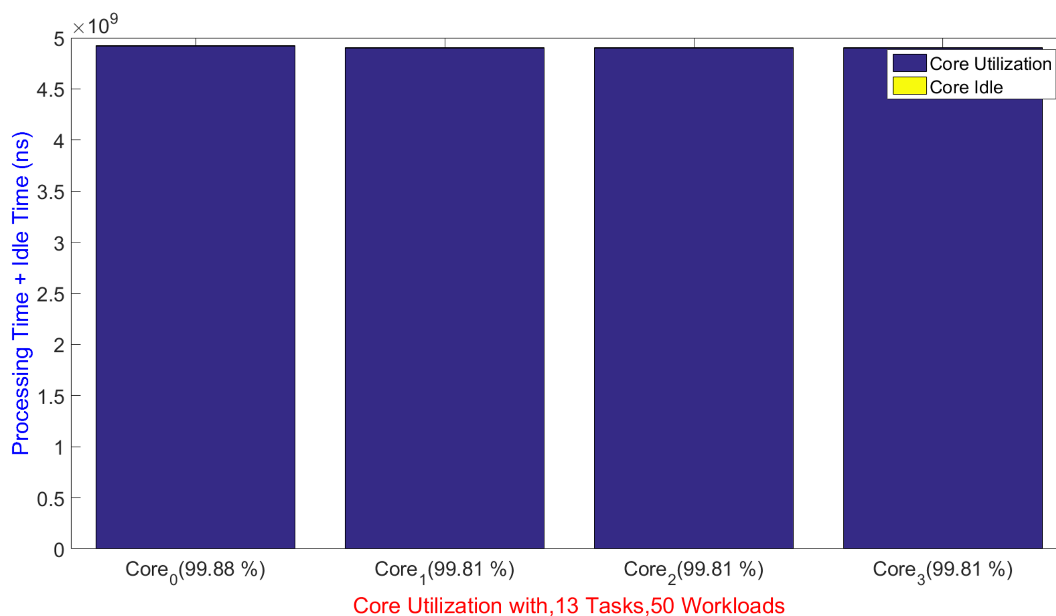

Figure 4.36 Core utilization with 13 Tasks, 50 workloads

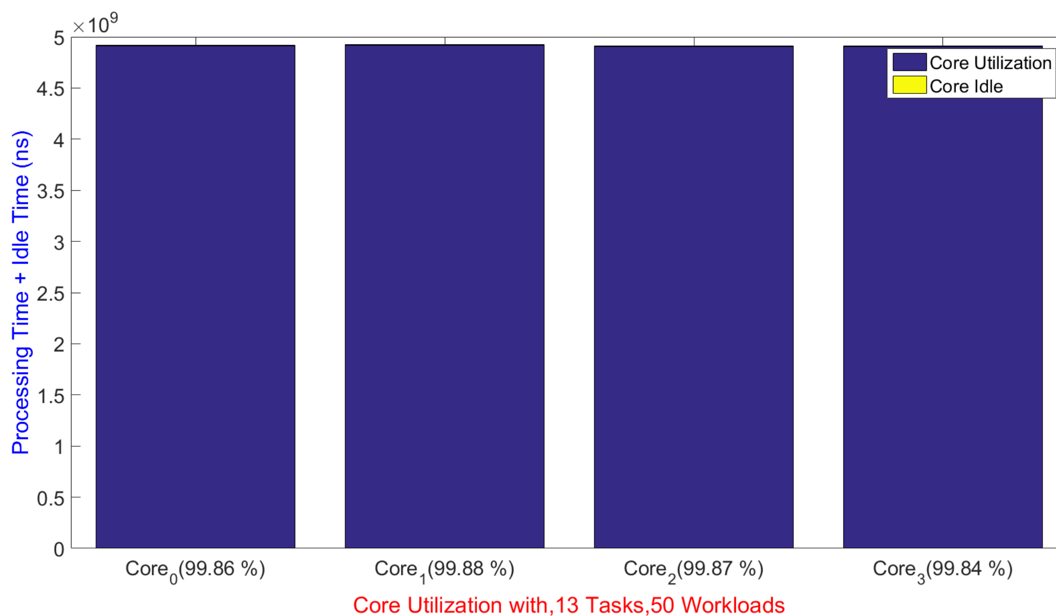

Figure 4.37 Core utilization with 13 Tasks, 50 workloads

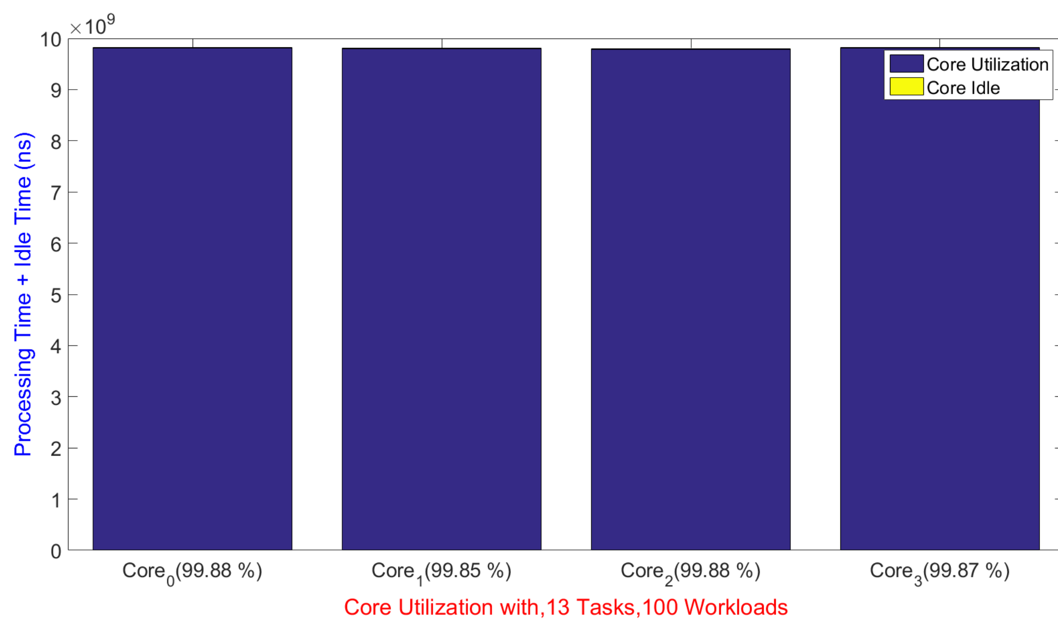

Figure 4.38 Core utilization with 13 Tasks, 100 workloads

CHAPTER 5 CONCLUSION AND RECOMMENDATIONS

This chapter summarizes the contributions of the thesis while identifying its limits and constraints. It also allows us to identify new paths of research, as well as to make recommendations for the future.

5.1 Advancement of knowledge

Through this thesis, we have studied many concepts. During this research, the literature was first explored to identify issues and trends in the world of cloud and more precisely at the data-center level. Thanks to a broad literature review, we have been able to study several methods that can improve the processing of tasks in a Cloud Radio Access Network(C-RAN) operating environment. Possible improvements relate to CPU utilization and probably to meet timing deadlines. In this thesis, we introduced a method to improve the utilization of multicore processors considering two main objectives: cache management and task scheduling. Based on these two objectives, we have addressed processing time variations, resource sharing and task scheduling in details.

As the number of cores on one chip increases significantly with multi-processor technology developments, methods are needed to ensure that throughput is high while providing a fair level of service to all workloads. Therefore, in this situation, means for resource-sharing and core utilization become necessary to guarantee particular measures of performance.

Characterizing processing time variations to perform a given task on mainstream processors is of particular interest to system designers who must comply with real-time constraints, such as the one-millisecond end-to-end radio frame processing latency of 5G networks.

In this research, our team proposed a simple, scalable, and controllable benchmark for characterizing the processing time variations due to the different layers of memory, as well as means of exploiting parallelism. The main idea proposed by Professor David and the author contributed to its validation by implementing a prototype confirming its efficiency. Moreover, we have conducted various tests to observe the worst-case processing time variations. AOCs help us find the bottlenecks(critical path) of the system for proper memory allocation.

By applying the proposed benchmark on an Intel Xeon E5-2650 V4 processor, we can characterize AOCs' behavior under various circumstances. In particular, we observed that the gain offered by the Advanced Vector Extension 2(AVX2) slightly depends on the data-size when data is in the cache and, as expected, that hyper-threading is only interesting when

the cache does not hold the entire data set.

Cache management, as one of the objectives in this thesis, is a useful technique in real-time systems to improve timing predictability. Effective cache management can improve the performance of modern processors. A good cache management implementation is important as such management can be a source of processing time variation on the one hand, and a means for improving performance and reducing variations, on the other.

We proposed a flexible priority-based Multi-Level Dynamic Cache Allocation(MLDCA) algorithm for shared caches to reduce the WCET of tasks. Unlike the other approaches that considered the longest path just for the instruction cache, we proposed an algorithm which considers the longest path for both the instruction and data caches. MLDCA is a suitable technique to obtain high efficiency and predictability.

Finding optimized-enough approach requires taking into account both task-scheduling and cache-locking simultaneously. Besides, to have an optimized-enough cache-allocation strategy, characterizing processing-time for finding the critical-path is a prerequisite factor.

Task scheduling for multi-core processors plays a central role in the performance of real-time systems. These days, big companies run large data-centers as processing, managing, and maintaining such a large pool of data is a business necessity. The construction and maintenance of data-centers are very expensive; also, it consumes a significant amount of power and energy.

So managing such a data-center efficiently is a paramount problem. There are different ways to measure the performance of data-centers from the viewpoint of scheduling quality, like throughput, total elapsed time, etc. The primary goal of the most data-centers is based on increasing throughput and increasing the number of unused machines; such a strategy paves the way for potential power savings.

In order to address the CPU utilization problem, our team proposed a queueing-based data-driven task scheduling algorithm. In detail, a data-driven technique is applied for task classification. Subsequently, a multi-queue architecture is introduced to handle such tasks. The original queuing concept was proposed by our team leader, Professor Zhu. A member of the team, Li Meng, had a significant contribution to finalize the prototype. Professor David also had a main role to finalize the developments and validation.

The proposed scheduling algorithm guarantees that each distributed task is ready for execution, and the task waiting time is eliminated. The performance of the proposed algorithm is evaluated on a developed platform. The results confirm that the developed algorithm is feasible and effective. It is worth noting that this work primarily focuses on improving CPU

utilization. From a broader view, the completion time of tasks is reduced as the utilization of CPU cores has been improved.

5.2 Limits and constraints

AOCs provided a framework for time measurement that allows to characterize the impact of block-size, stride, and scale-factor. So AOCs allow computing the execution time for varying block sizes and strides. AOCs still have some shortcomings to become effective micro-benchmarks. With the help of AOCs, we can upgrade the framework to micro-benchmarks in order to compute cache miss-rate/hit-rates, L1/L2/L3 cache size/cache access time, page size and memory access time for each cache level and memory separately, which are the objectives of the micro-benchmark concept.

Regarding task-scheduling, we often guarantee that the mainstream parallel programming models ensure data-dependence agreement, by making the parent-task to provide the required data to the child tasks. Data dependency agreement requires representing the tasks in a controlled-and-dependent manner. Such requirements impose topological challenges and constraints in the implementation and programming phase of the approach.

This approach becomes even more pertinent for an application whose control-and-data-dependency graph is more complicated. We remedy this problem by adopting the priority-queue-based data-driven task scheduling method and will improve it via a preemptive algorithm by considering the timing deadline.

5.3 Future work and recommendations

The tests performed in my research work were meant to be proofs of concept. Future works and recommendations aim to improve the knowledge associated with the results obtained and especially to move forward in the study of a real-time cloud computing.

5.3.1 Multi-level dynamic cache locking

An effective cache locking approach needs to take both task scheduling and cache locking into account simultaneously; therefore, for the future work, we would delve more deeply into dynamic cache management which proposed as MLDCM. On the other hand, having an efficient task scheduling algorithm with an accurate test framework is our fundamental need which leads to utilizing improvement of multicore systems.

5.3.2 Preemptive queuing-based data-driven task scheduling

Tasks are often associated with different priorities when assigned to the cores. It is necessary to run a high-priority task before another task with a lower priority. As the priority in our approach is timing deadline, hence, implementing the current scheduling model while considering the preemptive scheduling factor is a crucial step.

Such a strategy helps us meet a timing deadline and optimal use of each core. A good preemptive algorithm is a combination of Shortest-Job-First(SJF) and Best-Fit(BF) which is suitable for the cache slots. It occurs when a new upcoming process enters the ready queue that has an expected execute time, which is less than the process remaining time that is currently running on the CPU. Such an approach will be replaced with the current scheduling, which is not a preemptive algorithm.

Using one core as a master is a crucial need for each processing system. In high performance systems allocating one core as master limits the core's role to managing rather than processing. Therefore, using the management core for processing, like the other cores, increases performance in real-time systems.

A master-slave division of responsibilities creates processing limitations. One way to address such challenges is to conduct relevant processing on an external entity such as an FPGA. Such assignment is conducive to further saving of resources as well as speeding up the control of related tasks.

So, our approach to fully utilize all available resources is moving the manager agent to the FPGA. As in our available platform, we have the embedded FPGA beside our CPUs, so such moving is applicable.

5.3.3 Refine AOCs to improve its accuracy for processing time variations on multi-core processors

The refined version of Array Of Counters(AOCs) as a micro-benchmark constitutes a promising future work. Refined AOCs apply to improve its accuracy, flexibility, and capacity to observe the essential features, such as interference leading to processing time variability. This refined version could replace simple time consuming functions by real functions, like a FFT or a Decoder. On the other hand, refined AOCs let us compute cache misses or hit rates for each cache level and memory separately; which in a way is the objective of micro-benchmark approach. The proposed method enables a better understanding and measurement of the task processing time variations with specific data size and the number of processing cores.

5.4 Perspective ideas for potential papers

5.4.1 Contention-free method to reduce processing time variations of cloud applications on multi-core processors

Cloud Radio Access Networks(C-RANs) is a promising means to implement 5G wireless networks. A significant challenge in their design is to limit the worst-case processing time and its variations. This paper will introduce the contention-free method to reduce the critical source of interference of the processing time variability. The new method uses the Interference Array Of Counters(IAOCs) to achieve more precise details regarding the cause of processing time variations. The proposed method gives a better understanding of the task processing time variations than the status quo state-of-the-art solutions. This method shows how to deal with processing time interfering in the context of real-time low latency applications running over typical data-center processors.

5.4.2 Multi-level dynamic cache allocation to reduce worst case execution time in multi-core processors

Caches dramatically improve the performance of modern processors. However, caches introduce timing unpredictability in real-time systems, which are the leading cause of execution time variability. The Worst-Case Execution Time(WCET) in such systems, is an essential metric for schedulability analysis. Many modern processors support cache locking mechanism, e.g., Intel Xeon E5-2650 V4. So by carefully selecting the memory blocks to lock, cache locking can substantially improve performance. We are proposing a Multi-Level Dynamic Cache Allocation(MLDCA) algorithm for shared caches to reduce the WCET of tasks. Unlike all the previous cache locking approaches that took into account exclusively instruction cache, we propose an algorithm which employs the longest path for both instruction cache as well as data cache. On the other hand, unlike the status-quo state-of-the-art algorithm that only considers the longest path, in the proposed algorithm, the sub-critical path for a number of the longest path is considered.

5.4.3 Local queuing-based data-driven task scheduling on multi-core system

Nowadays, multi-core systems are extensively employed in high-performance computing. Many algorithms have been proposed to enhance system performance by load balancing or concurrent scheduling to reduce the needed execution time of applications. Such a task scheduling requires in-depth analysis to utilize the processing capacity fully and to achieve low processing latency. To tackle the inefficient CPU utilization, a queuing-based data-driven

task scheduling scheme, which focuses on local parallel computing, is proposed in this paper. In the proposed scheme, multi-queue management is introduced for dynamic task scheduling to target 100% utilization of local CPU cores for a sufficient number of input tasks.

REFERENCES

[1] Yonghua Lin, Ling Shao, Zhenbo Zhu, Qing Wang, and Ravie K Sabhikhi. Wireless network cloud: Architecture and system requirements. *IBM Journal of Research and Development*, 54(1):4–1, 2010.

[2] China Mobile. C-RAN: the road towards green RAN. *White Paper, ver*, 2, 2011.

[3] Michel Gémieux, Yvon Savaria, Guchuan Zhu, and Jean-François Frigon. Towards LTE physical layer virtualization on a COTS multi-core platform with efficient scheduling. In *New Circuits and Systems Conference (NEWCAS), 2016 14th IEEE International*, pages 1–4. IEEE, 2016.

[4] Meng Li, Chao Chen, Guchuan Zhu, and Yvon Savaria. Local queueing-based data-driven task scheduling for multicore systems. In *2018 IEEE 61st International Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 897–900. IEEE, 2018.

[5] William F Guthrie. NIST/SEMATECH engineering statistics handbook. 2010.

[6] Tarun Agarwal, Amit Sharma, A Laxmikant, and Laxmikant V Kalé. Topology-aware task mapping for reducing communication contention on large parallel machines. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 10–pp. IEEE, 2006.

[7] Mary Natrella. Nist/sematech e-handbook of statistical methods. 2010.

[8] NGMN Alliance. Ngmn 5G white paper. *Next Generation Mobile Networks, White paper*, 2015.

[9] Renato Mancuso, Roman Dudko, Emiliano Betti, Marco Cesati, Marco Caccamo, and Rodolfo Pellizzoni. Real-time cache management framework for multi-core architectures. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*, pages 45–54. IEEE, 2013.

[10] Daniel Molka, Daniel Hackenberg, Robert Schone, and Matthias S Muller. Memory performance and cache coherency effects on an Intel nehalem multi-processor system. In *Parallel Architectures and Compilation Techniques, 2009. PACT'09. 18th International Conference on*, pages 261–270. IEEE, 2009.

[11] Julio Sahuquillo and Ana Pont. Impact of reducing miss write latencies in multiprocessors with two level cache. In *Euromicro Conference, 1998. Proceedings. 24th*, volume 1, pages 333–336. IEEE, 1998.

[12] Vincent Nélis, Patrick Meumeu Yomsi, and Luís Miguel Pinho. The variability of application execution times on a multi-core platform. In *16th International Workshop on Worst-Case Execution Time Analysis*, 2016.

[13] Daniel Hackenberg, Daniel Molka, and Wolfgang E Nagel. Comparing cache architectures and coherency protocols on x86-64 multicore SMP systems. In *Proceedings of the 42th Annual IEEE/ACM International Symposium on microarchitecture*, pages 413–422. ACM, 2009.

[14] Yun Liang and Tulika Mitra. Instruction cache locking using temporal reuse profile. In *Proceedings of the 47th Design Automation Conference*, pages 344–349. ACM, 2010.

[15] Sparsh Mittal. A survey of techniques for cache locking. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 21(3):49, 2016.

[16] Tiantian Liu, Minming Li, and Chun Jason Xue. Minimizing WCET for real-time embedded systems via static instruction cache locking. In *Real-Time and Embedded Technology and Applications Symposium, 2009. RTAS 2009. 15th IEEE*, pages 35–44. IEEE, 2009.

[17] Tiantian Liu, Minming Li, and Chun Jason Xue. Instruction cache locking for multi-task real-time embedded systems. *Real-Time Systems*, 48(2):166–197, 2012.

[18] Sascha Plazar, Jan C Kleinsorge, Peter Marwedel, and Heiko Falk. Wcet-aware static locking of instruction caches. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, pages 44–52. ACM, 2012.

[19] Xavier Vera, Björn Lisper, and Jingling Xue. Data cache locking for tight timing calculations. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(1):4, 2007.

[20] Isabelle Puaut and Alexis Arnaud. Dynamic instruction cache locking in hard real-time systems. In *Proc. of the 14th Int. Conference on Real-Time and Network Systems*, 2006.

[21] Huping Ding, Yun Liang, and Tulika Mitra. Integrated instruction cache analysis and locking in multitasking real-time systems. In *Proceedings of the 50th Annual Design Automation Conference*, page 147. ACM, 2013.

[22] Wei Zheng, Chao Xu, and Wen Bao. Online scheduling of multiple deadline-constrained workflow applications in distributed systems. In *Advanced Cloud and Big Data, 2015 Third International Conference on*, pages 104–111. IEEE, 2015.

[23] Huping Ding, Yun Liang, and Tulika Mitra. Wcet-centric dynamic instruction cache locking. In *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, pages 1–6. IEEE, 2014.

[24] Mayank Shekhar, Abhik Sarkar, Harini Ramaprasad, and Frank Mueller. Semi-partitioned hard-real-time scheduling under locked cache migration in multi-core systems. In *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*, pages 331–340. IEEE, 2012.

[25] Tosiron Adegbija and Ann Gordon-Ross. Phase-based cache locking for embedded systems. In *Proceedings of the 25th edition on Great Lakes Symposium on VLSI*, pages 115–120. ACM, 2015.

[26] John Picchi and Wei Zhang. Impact of L2 cache locking on GPU performance. In *SoutheastCon 2015*, pages 1–4. IEEE, 2015.

[27] Wenguang Zheng and Hui Wu. Wcet: aware dynamic instruction cache locking. In *ACM SIGPLAN Notices*, volume 49, pages 53–62. ACM, 2014.

[28] Ya-Shu Chen, Han Chiang Liao, and Ting-Hao Tsai. Online real-time task scheduling in heterogeneous multi-core system-on-a-chip. *IEEE Transactions on Parallel and Distributed Systems*, 24(1):118–130, 2013.

[29] Xuanxia Yao, Peng Geng, and Xiaojiang Du. A task scheduling algorithm for multi-core processors. In *Parallel and Distributed Computing, Applications and Technologies (PDCAT), 2013 International Conference on*, pages 259–264. IEEE, 2013.

[30] Xiaozhong Geng, Gaochao Xu, Xiaodong Fu, and Yuan Zhang. A task scheduling algorithm for multi-core-cluster systems. *JCP*, 7(11):2797–2804, 2012.

[31] Hu Chen, Wenguang Chen, Jian Huang, Bob Robert, and Harold Kuhn. Mpipp: an automatic profile-guided parallel process placement toolset for SMP clusters and multi-clusters. In *Proceedings of the 20th annual international conference on Supercomputing*, pages 353–360. ACM, 2006.

[32] Apostolos Gerasoulis and Tao Yang. *A Comparison of Clustering Heuristics for Scheduling DAGs on Multiprocessors*. Rutgers University, Department of Computer Science, Laboratory for Computer Science Research, 1991.

[33] Haluk Topcuoglu, Salim Hariri, and Min-you Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE transactions on parallel and distributed systems*, 13(3):260–274, 2002.

[34] Michael R Gary and David S Johnson. Computers and intractability: A guide to the theory of NP-completeness, 1979.

[35] Canturk Isci, Alper Buyuktosunoglu, Chen-Yong Cher, Pradip Bose, and Margaret Martonosi. An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget. In *Proceedings of the 39th annual IEEE/ACM international symposium on microarchitecture*, pages 347–358. IEEE Computer Society, 2006.

[36] Hidehiro Kanemitsu, Masaki Hanada, and Hidenori Nakazato. Clustering-based task scheduling in a large number of heterogeneous processors. *IEEE Transactions on Parallel and Distributed Systems*, 27(11):3144–3157, 2016.

[37] Hesham El-Rewini, Theodore G Lewis, and Hesham H Ali. *Task scheduling in parallel and distributed systems*. Prentice-Hall, Inc., 1994.

[38] Ke Wang, Kan Qiao, Iman Sadooghi, Xiaobing Zhou, Tonglin Li, Michael Lang, and Ioan Raicu. Load-balanced and locality-aware scheduling for data-intensive workloads at extreme scales. *Concurrency and Computation: Practice and Experience*, 28(1):70–94, 2016.

[39] Intel Intel. IA-32 architectures software developer's manual, volume 3b: System programming guide. *Part*, 1:64, 2007.

[40] *Vivado Design Suite User Guide(Tutorial); High-Level Synthesis*.

[41] Pierre-Marc Fournier, Mathieu Desnoyers, and Michel R Dagenais. Combined tracing of the kernel and applications with LTTng. In *Proceedings of the 2009 linux symposium*, pages 87–93, 2009.

APPENDIX A INTRODUCTION TO VIVADO AND VIVADO HLS

FPGAs are often used to build high-speed telecommunication devices, industrial and commercial devices which are very sensitive and fast, military equipment and expenditures benefit like this. FPGAs range from a few thousand to a couple of million gates inside of which the number of FPGA gateways has a significant impact on its price. The Xilinx Vivado Design Suite is a powerful software for FPGA design. The FPGA is derived from the Field Programmable Logic Gate Array, and the speed of implementing logical functions is very high. In the case of FPGA applications, it can be said that FPGAs can be used to test HDL designs, and if desired, the results of the ASIC chip-set.

Vivado Design Suite is designed for large system design and is an environment based on the use of IP-Core and systems. In terms of the speed of various stages, it is roughly 4 times faster than ISE. The fundamental difference between the Vivado Design Suite and ISE is the speed where the program is implemented on the FPGA which is faster than ISE.

This software suite is used to synthesize and analyze programs written in HDL languages, and is actually a new and upgraded version of the ISE suite of software. Xilinx Vivado Design Suite HLx Edition is a powerful Xilinx software which presented to design Xilinx Series 7 FPGAs and later. This software is presented in its past versions with ISE software, but now it has been independently provided with many features.

Also, artificial intelligence algorithms that are used to embed, fitting the circuit and wiring are more efficient. That is, the circuit formed within the FPGA is better in terms of delays and many other parameters.

Key Features of Xilinx Vivado Design Suite HLx Edition:

- Has an environment similar to the ISE environment
- An environment based on the use of IP Core (IP Block Design)
- Support for multi-core systems
- Increase the design speed by about 4 times
- 20% better design density
- Increased integration speed
- Integrated user interface for design and simulation

- Simulator Vivado Simulator Internal program tool (equivalent to impact program)
- An internal analysis and troubleshooting tool called Logic Analyzer (Chipscope equivalent)
- Hardware debugging
- Compatible with different versions of Windows
- Vivado HLS tool for high-level programming in C, C ++ and SystemC
- SDK tool for developing application code for ARM processors Zynq chips and Microblaze processors

High-Level Synthesis (HLS) help us to create and RTL implementation from the C, C++ or SystemC source code. Also it extract control and dataflow from these source codes. Afterwards it can implemet the desired design based on user constraints which we call here directives. With the help of HLS many implementation are possible to do from the same source which lead to smaller, faster and optimal design, which all these factors enable exploration of desired design.

The process of hardware design has changed somewhat over the past few years, and it has also begun to develop FPGA chips using C++. Vivado HLS software Xilinx is specifically designed for this purpose. The difference is that Vivado supports VHDL, Verilog, SystemC, but Vivado HLS supports C++ and C properly.

Vivado only supports the Xilinx serial-7 FPGAs, and the former series didn't add to it. The Vivado software needs everything you need to design FPGA and integrate all the tools seamlessly, core generator tools, I/O planing, timing tools, power tools, embedded tools, simulations and etc.

For convenience, Vivado is composed of several layouts. After synthesis, you can open the design project in different layouts, such as I/O Planning, clock planning, floorplanning, timing analysis debug, and continue your designing. Each of these tools has a complex set of features.

The most important feature of Vivado is the multi-core programing. ISE was very annoying because of its single-core function! When you perform operations like synthesis, you can not do anything else. While the Vivado environment, which is very complete and integrates all the tools together, is fully multi-core and you can use several tools at the same time.

The most important issue is the synthesis and implementation process itself. The ISE writes wrongly that are executing dual-cores, but in reality it is not true, and there is no improvement in the many cores that the ISE refers to. The implementation speed of ISE is either too

low for the single-core or its old and original algorithm, and this is a very annoying problem when working with large FPGAs. A simple Ethernet core for Virtex7 with ISE takes about 15 minutes to be implemented with a 3.5 GHz Core-i7 processor while this core with Vivado lasts about 7 minutes.

Syntax of constraints changed in the Vivado-based design, and the UCF file does not use anymore. The Vivado use XDC instead of UCF, which has a different syntax. But it does not need to be involved with the XDC at the beginning. The various Vivado tools themselves updated the constraint file. Constraints can be defined both for synthesis and for implementation separately. In fact, with more detail, you can define different constraints. It also can have several constraint groups for different implementation conditions or even for a specific file or core can define a constraint. What is seen when using the core and even some are read only! Meanwhile, from Series 7 onwards, the software draws heavily on constraints and does not let to leave everything to the default state, this also applies to the ISE, for example, you must define all I/O standards. Editor and Schematic Environment finally in Vivado became a professional designer.

Vivado HLS has a number of way to improve the performance which are generally, automatic/default optimization, latency directives and pipelining. Also Vivado HLS support techniques to remove performance bottlenecks such as manipulating loops and partitioning or reshaping the arrays. Most of the optimization in Vivado HLS is performed via using directives.

There are two ways to place directive in Vivado HLS, in the directive file or into source code. Figure A.1

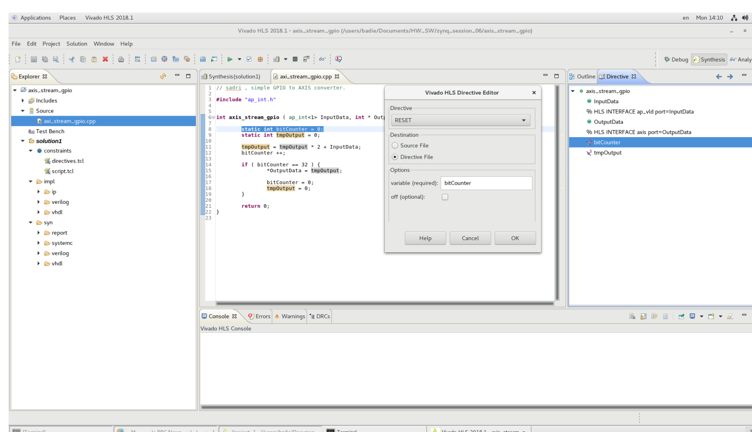

Figure A.1 Adding Directives.

Each of configuration can be set as a solution, which can be useful to save time for future

tests. Figure A.2

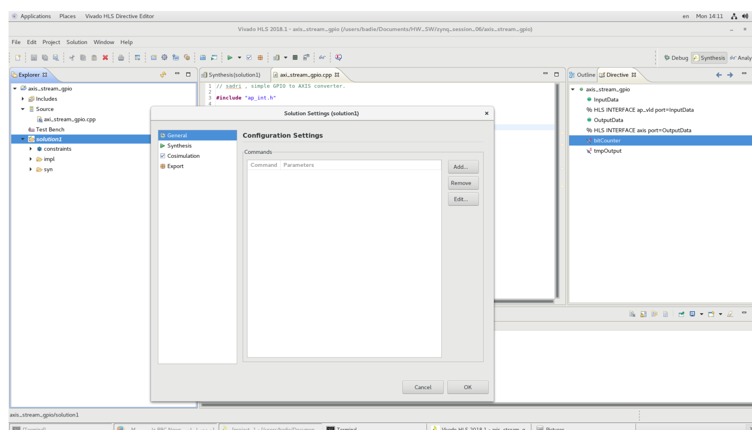

Figure A.2 Solution Configuration.

Familiarity with two concepts is important to better understanding the concept of performance. 1) Design Latency which is the number of cycle it takes to output the results and 2) Desing Throughput which is the number of cycle between new inputs. So in the absence of any concurrency the latency is the same as throughput. On the other hand, pipelining can be used via Vivado HLS for higher throughput, like pipelining functions and loops. In the following first we discuss optimizing the latency and after throughput.

Vivado HLS will minimize the latency by default. Notice that the throughput is prioritized above latency. Also Vivado HLS automatically take advantage of the parallelism.

- Vivado HLS try to minimize latency by allowing functions to operate in parallel.
- Vivado HLS will not schedule loops to operate in parallel by default and user should set this configuration.
- Vivado HLS try to minimize latency by allowing the operation to force run in parallel which didn't met within functions and loops.

Latency Constraints can define a minimum/maximum latency for each location Figure A.3.

Also the latency directives can be used in function, loops and regions as well.

By default loops are rolled. Loop can be unrolled if their indices statically determine at elaboration time. Unrolled loops can reduce latency Figure A.4.

As the fully unrolling loops can create a lot of hardware so loop can be partially unrolled. Vivado HLS can automatically flatten nested loop too Figure A.5.

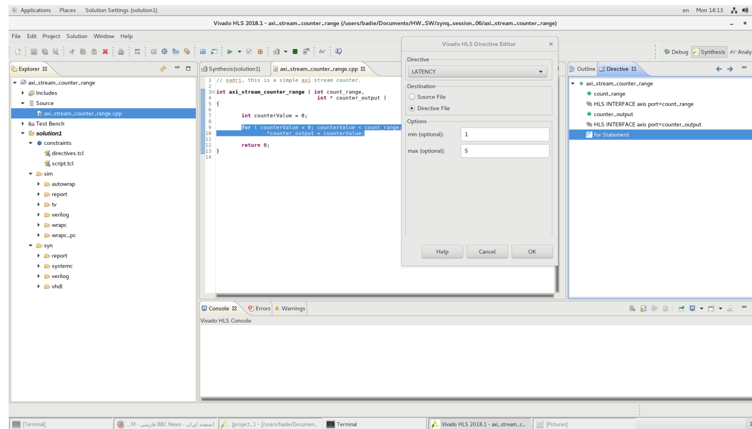

Figure A.3 Latency Constraints.

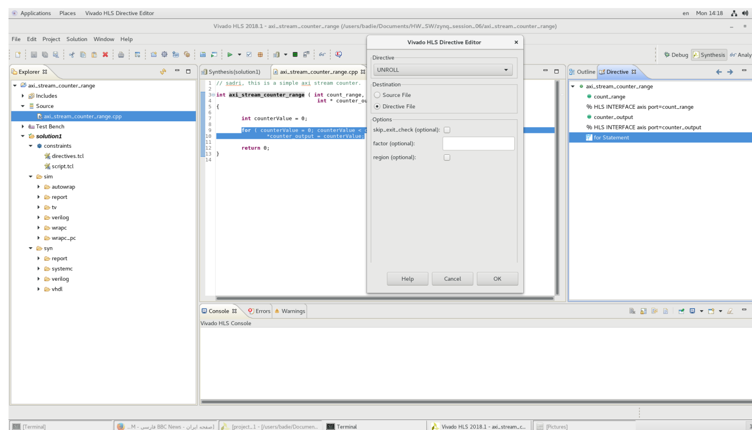

Figure A.4 Loop Unrolling.

One of the other directive is loop merging, which Vivado HLS can automatically merge loops together Figure A.6.

Whenever a design with multiple functions gives to Vivado HLS, it will schedule the design (latency and throughput) then it can automatically optimize the dataflow for throughput. One of the impressive optimize method is function pipelining, another one is loop pipelining. Vivado HLS try to unroll all loops nested via the PIPELINE directive (which is called Pipelining and Function/Loop Hierarchy). This approach may not succeed all time for different reason, such as lead to unacceptable area or create a lot of hardware. Another option in pipelining is Flushing. Pipeline can optionally be flushed when there is no more data for all existing results and the pipeline can be flushed out. By default is to stall all existing values in the pipeline.

Considering all of the above, sometimes we can not use pipeline and we can not take advantage

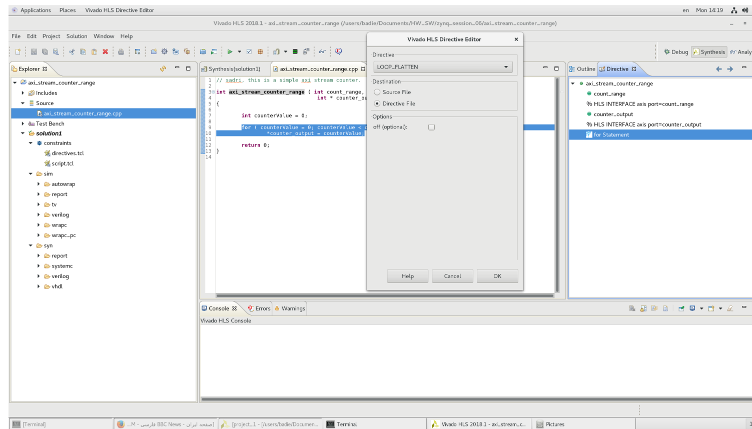

Figure A.5 Loop Flattening.

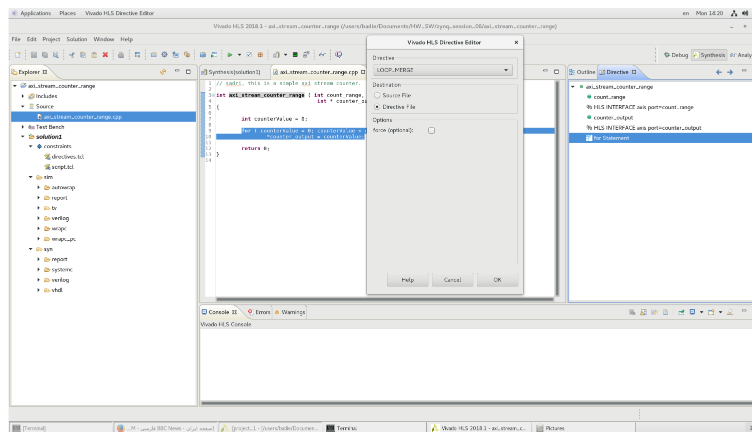

Figure A.6 Loop Merging.

for optimization in this way, such as;

- Pipelining functions unrolls all loops; Loops with variable bounds cannot be unrolled and this will prevent pipelining.
- Feedback within the code will prevent or limit pipelining.
- Resource Contention may prevent pipelining; Can occur within input and output ports/arguments.

Dataflow optimization can be used at the top-level function and allows blocks (functions or loops) of code to operate concurrently. This will be happen with placing the channels between the blocks to maintain the data-rate. On the other hand the dataflow optimization may have an area overhead, such as additional memory blocks which are added to the design Figure A.7.

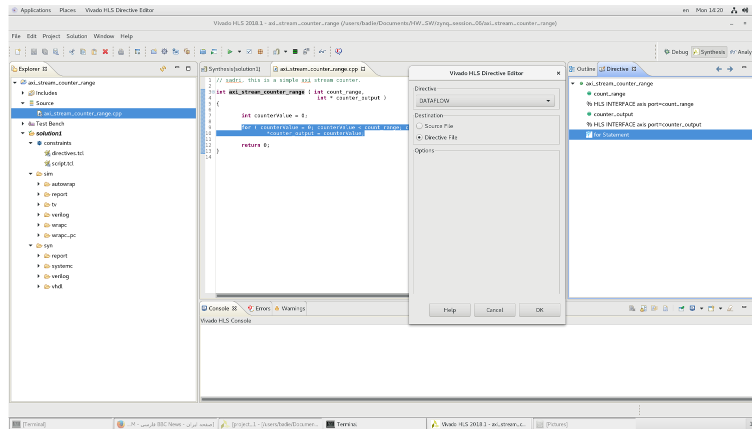

Figure A.7 Dataflow Optimization Command.

At the end it is a good idea to compare, Dataflow versus Functions and Loops;

- Dataflow Optimization;
  - Dataflow optimization is "coarse grain" pipelining at the function and loop level.
  - Increases concurrency between functions and loops.
  - Only works on functions or loops at the top-level of the hierarchy which cannot be used in sub-functions.
- Function & Loop Pipelining;
  - "Fine grain" pipelining at the level of the operators (*, +, etc.)
  - Allows the operations inside the function or loop to operate in parallel.
  - Unrolls all sub-loops inside the function or loop being pipelined.
    - Loops with variable bounds cannot be unrolled which can prevent pipelining.
    - Unrolling loops increases the number of operations and can increase memory and run time.