نموذج ترخيص

أنا الطالبة: عائشة فالح فناطل المسلم أمنح الجامعة الأردنية و/ أو من تفوضه ترخيصاً غير حصري دون مقابل بنشر و / أو استعمال و / أو استغلال و / أو ترجمة و / أو تصوير و / أو إعادة إنتاج بأي طريقة كانت سواء ورقية و / أو إلكترونية أو غير ذلك رسالة الماجستير / الدكتوراه المقدمة من قبلي وعنوانها.

Evaluation of Popular Multicore Design Alternatives Using Configuration Dependent Analysis.

وذلك لغايات البحث العلمي و / أو التبادل مع المؤسسات التعليمية والجامعات و / أو لأي غاية أخرى تراها الجامعة الأردنية مناسبة، وأمنح الجامعة الحق بالترخيص للغير بجميع أو بعض ما رخصته ليا.

اسم الطالب: عائشة فالح فناطل المسلم

التوقيع:

التاريخ: ٣١/٥/٢٠١٧

EVALUATION OF POPULAR MULTICORE DESIGN ALTERNATIVES USING CONFIGURATION DEPENDENT ANALYSIS

By

Aieshah Faleh Almaslam BanySakher

Supervisor

Dr. Gheith Ali Abandah, Prof

This Thesis was submitted in Partial Fulfillment of the Requirements for the Master's Degree of Computer Engineering and Networks

School of Graduate Studies
The University of Jordan

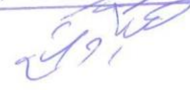
May, 2017

تعتمد كلية الدراسات العليا
هذه النسخة من الرسالة
التوقيع .... التاريخ ....

COMMITTEE DECISION

This Thesis/Dissertation (Evaluation of Popular Multicore Design Alternatives Using Configuration Dependent Analysis) was Successfully Defended and Approved on Tuesday May 2, 2017.

Examination Committee	Signature
Prof. Gheith A. Abandah (Supervisor) Prof. of Computer Engineering	
Dr. Andraws A. Swidan (Member) Prof. of Computer Engineering	
Dr. Basel A. Mahafzah (Member) Prof. of Computer Science	
Dr. Essam A. AlQaralleh (Member) Assoc. Prof. of Computer Engineering Princess Sumaya University for Technology	


تعتمد كلية الدراسات العليا
هذه النسخة من الرسالة
التوقيع.......التاريخ.......

DEDICATION

To my caring parents, Abo Osama and Um Osama

To my loving husband, Hani

To my lovely aunt, Um Hani

To my clever kids, Bandar and Balsam

ACKNOWLEDGEMENTS

First and foremost, glory and praise be to Allah, the Almighty, for providing me with the strength, patience, and for guiding me through all the difficulties to carry out this work.

I would like to express my gratitude to everyone who helped me during the thesis starting with endless thanks and sincere gratitude for my advisor Prof. Gheith Abandah who didn't keep any effort in encouraging me and providing me with valuable advice to be better each time.

I would like to appreciate the support received from Dr. Wim Heirman and Dr. Trevor E. Carlson, from Intel ExaScience Lab, and Ghent University; they answered all my questions about various features of Sniper simulator, and suggested me to choose the suitable options in multicore configuration files. Also, many thanks for the Spanish researcher, Marco Antonio Pérez García, for his help in working with Sniper multicore simulator.

In addition, I would like to thank my family: my parents, my sister and my brothers who encouraged and fully supported me spiritually throughout this thesis and my life in general. They were always supporting and encouraging with their best wishes. Special thanks to my lovely husband, Hani. He was always there cheering me up and stood by me through the good and bad times. Also, many thanks for my caring aunt, Um Hani, she provided lovingly care for my children in the hard times. I really appreciate her support.

Finally, I would like to thank my lovely manager Kefah Al-Etan and my lovely friends in my work, they made a suitable environment for my study during my master road.

To those who were mentioned, or indirectly contributed to this research and not mentioned, your kindness means a lot to me. You have all made a huge impact on who I am today, and for that; I am forever grateful.

Aieshah Bany Sakher, 2017

TABLE OF CONTENTS

Subject **Page**

COMMITTEE DECISION ..... **ii**

DEDICATION ..... **iii**

ACKNOWLEDGEMENTS ..... **v**

TABLE OF CONTENTS ..... **v**

LIST OF TABLES ..... **vii**

LIST OF FIGURES ..... **viii**

LIST OF ABBREVIATIONS ..... **xi**

ABSTRACT ..... **xiii**

CHAPTER 1: INTRODUCTION **1**

1.1 Background and Motivation 2

1.2 Research Contributions 4

1.3 Research Methodology 5

1.4 Thesis Outline 6

CHAPTER 2: LITERATURE REVIEW **7**

2.1 Introduction 8

2.2 Multicore Overview 8

2.3 Multicore Simulators 12

2.4 Multicore Benchmarks 16

2.5 Multicore Studies 18

CHAPTER 3: METHODOLOGY AND TOOLS **28**

3.1 Introduction 29

3.2 Overview 29

3.3 Multicore Design Alternatives 29

3.3.1 Core 2 / Dunnington Microarchitecture 33

3.3.2 Nehalem / Gainestown Microarchitecture 34

3.3.3 Haswell Microarchitecture 35

3.3.4 Xeon Phi / Knights Corner (KNC) Microarchitecture 35

3.4 Experimental Setup 36

3.4.1 Host Machine 36

3.4.2 Operating System, Compiler, and Libraries 36

3.5 Sniper Multicore Simulator 37

3.6 Benchmarks 40

3.6.1 SPLASH2 Benchmark Suite 40

3.6.2 PARSEC Benchmark Suite 42

3.7 Performance Evaluation Metrics 43

3.8 Validation 44
CHAPTER 4: RAW AND NORMALIZED EVALUATIONS 47
4.1 Introduction 48
4.2 Raw Comparison 49
4.2.1 Total Execution Time 54
4.2.2 L2 Cache Miss Rate 57
4.3 Normalized Comparison 60
4.3.1 Total Execution Time 62
4.3.2 Average Core IPC 65
4.3.3 CPI Cycle Stack 67
4.3.4 Average Core Utilization 94
4.3.5 Power Consumption 98
4.3.6 Cache Coherence Protocol 101
CHAPTER 5: THESIS CONCLUSION AND FUTURE WORK 104
5.1 Introduction 105
5.2 Conclusion 105
5.3 Future work 109
APPENDIX A: USAGE INSTRUCTIONS 110
APPENDIX B: CONFIGURATION FILES 115
REFERENCES 129
ABSTRACT(Arabic) 137

LIST OF TABLES

No.	Table Caption	Page
3.1	Multicore design features for the four commercial Intel's server processors.	31
3.2	Two input sets for thesis studied applications.	43
3.3	Validated Nehalem core configuration.	44
3.4	Validated benchmarks and input sets.	45
3.5	IPC results of our validation and the results of Sniper validation paper (2014).	46
A.1	The required libraries.	110
A.2	The names of the studied benchmarks applications and input sets.	114
B.1	Sniper configuration files for all studied design alternatives.	115

LIST OF FIGURES

No.	Figure Caption	Page
2.1	Multicore processors classifications.	10
3.1	Dunnington/ Core 2-based microarchitecture.	34
3.2	Gainestown/ Nehalem microarchitecture.	34
3.3	Haswell microarchitecture.	35
3.4	Xeon Phi Coprocessor microarchitecture.	35
3.5	Validation of Sniper simulator.	45
4.1	Multi-socket Dunnington based microarchitecture.	50
4.2	Multi-socket Gainestown based microarchitecture.	50
4.3	Haswell-based microarchitecture.	51
4.4	Xeon Phi-based microarchitecture.	51
4.5	Local and remote communications in multi-socket Gainestown microarchitecture.	53
4.6	The two common NoC; a) 2D mesh topology, b) ring topology.	53
4.7	Execution times for running eight benchmark applications with small input size over the four multicore design alternatives.	54
4.8	Execution times for running eight benchmark applications with large input size over the four multicore design alternatives.	55
4.9	Various memory types and their corresponding data rates.	56
4.10	L2 miss rates for running eight benchmark applications with small input sizes over the four multicore design alternatives.	58
4.11	L2 miss rates for running eight benchmark applications with large input sizes over the four multicore design alternatives.	59
4.12	Normalized multi-socket Dunnington based microarchitecture.	60
4.13	Normalized multi-socket Gainestown based microarchitecture.	61
4.14	Normalized Haswell-based microarchitecture.	61
4.15	Normalized Xeon Phi based microarchitecture.	62
4.16	Execution times for running eight benchmark applications with small input size over the four normalized multicore design alternatives.	64

4.17	Execution times for running eight benchmark applications with large input size over the four normalized multicore design alternatives.	64
4.18	Average core IPC for running the eight benchmark applications with the small input sizes over the four normalized multicore design alternatives.	66
4.19	Average core IPC for running the eight benchmark applications with the large input sizes over the four normalized multicore design alternatives.	67
4.20	An example of Haswell simulation output a) simple b) detailed c) single component.	69
4.21	CPI stack and IPC over time for running SPLASH2-FFT benchmark application with small input size over the four normalized multicore design alternatives.	72
4.22	CPI stack and IPC over time for running SPLASH2-FFT benchmark application with large input size over the four normalized multicore design alternatives.	73
4.23	CPI stack and IPC over time for running SPLASH2-Radix benchmark application with small input size over the four normalized multicore design alternatives.	75
4.24	CPI stack and IPC over time for running SPLASH2-Radix benchmark application with large input size over the four normalized multicore design alternatives.	76
4.25	CPI stack and IPC over time for running SPLASH2-Lu.cont benchmark application with small input size over the four normalized multicore design alternatives.	78
4.26	CPI stack and IPC over time for running SPLASH2-Lu.cont benchmark application with large input size over the four normalized multicore design alternatives.	79
4.27	CPI stack and IPC over time for running SPLASH2-Cholesky benchmark application with small input size over the four normalized multicore design alternatives.	81
4.28	CPI stack and IPC over time for running SPLASH2-Cholesky benchmark application with large input size over the four normalized multicore design alternatives.	82
4.29	CPI stack and IPC over time for running PARSEC-Blackscholes benchmark application with small input size over the four normalized multicore design alternatives.	84
4.30	CPI stack and IPC over time for running PARSEC-Blackscholes benchmark application with large input size over the four normalized multicore design alternatives.	85
4.31	CPI stack and IPC over time for running PARSEC-Canneal benchmark application with small input size over the four normalized multicore design alternatives.	88

4.32 CPI stack and IPC over time for running PARSEC-Canneal benchmark application with large input size over the four normalized multicore design alternatives. 89

4.33 CPI stack and IPC over time for running PARSEC-Fluidanimate benchmark application with small input size over the four normalized multicore design alternatives. 90

4.34 CPI stack and IPC over time for running PARSEC-Fluidanimate benchmark application with large input size over the four normalized multicore design alternatives. 91

4.35 CPI stack and IPC over time for running PARSEC-Swaptions benchmark application with small input size over the four normalized multicore design alternatives. 92

4.36 CPI stack and IPC over time for running PARSEC-Swaptions benchmark application with large input size over the four normalized multicore design alternatives. 93

4.37 Average core utilization for running the eight benchmark applications with the small input sizes over the four normalized multicore design alternatives. 97

4.38 Average core utilization for running the eight benchmark applications with the large input sizes over the four normalized multicore design alternatives. 97

4.39 Average runtime dynamic power for running the eight benchmark applications with the small input sizes over the four normalized multicore design alternatives. 99

4.40 Average runtime dynamic power for running the eight benchmark applications with the large input sizes over the four normalized multicore design alternatives. 99

4.41 Kiviat chart for performance evaluation for the four normalized multicore design alternatives. 100

4.42 MESIF versus MESI Protocol 101

4.43 Average core IPC for running 4 SPLASH2 benchmark applications with large input sizes over the four studied multicore design alternatives with MESI/MESIF cache coherence protocols. 102

4.44 Average core IPC for running 4 PARSEC benchmark applications with large input sizes over the four studied multicore design alternatives with MESI/MESIF cache coherence protocols. 103

LIST OF ABBREVIATIONS

AMD	Advanced Micro Devices
ARM	Advanced RISC Machine
CMP	Chip Multi-Processor
CPU	Central Processing Unit
DRAM	Dynamic Random Access Memory
DDR	Double Data Rate
DVFS	Dynamic Voltage and Frequency Scaling
ECC	Error Correcting Code
FSB	Front Side Bus
FLITs	FLow control unITs
HB	Higher is Better
HJM	Heath Jarrow Morton
HPC	High-Performance Computing
HT	Hyper-Threading
ILP	Instruction Level Parallelism
IPC	Instruction Per Cycle
IWC	Instructions Window Centric
JIT	Just in Time
L1	Level 1 cache
LB	Lower is Better
LLC	Last Level Cache
MC	Memory controller
MESIF	Modified Exclusive Shared Invalid Forward protocol
MIC	Many Integrated Core
MIPS	Million Instructions Per Second

MOESI	Modified Owned Exclusive Shared Invalid protocol
NoC	Network on Chip
NUCA	NonUniform Cache Access
NUMA	Non-Uniform Memory Access
OS	Operating System
PARSEC	Princeton Application Repository for Shared-Memory Computers
QEMU	Quick Emulator
QPI	Quick Path Interconnect
RAM	Random Access Memory
RISC	Reduced Instruction Set Computing
RMS	Recognition Mining Synthesis
ROI	Region of Interest
SIMD	Single Instruction Multiple Data
SMP	Symmetric Multiprocessor
SMT	Simultanious Multithreading
SPEC	Standard Performance Evaluation Corporation
SPLASH	Stanford ParalleL Applications for Shared memory
SoC	System on Chip
TBB	Threading Building Block
TDP	Thermal Design Power
TLB	Translation Look-aside Buffer
TLP	Thread Level Parallelism
VLIW	Very Long Instruction Word

EVALUATION OF POPULAR MULTICORE DESIGN ALTERNATIVES USING CONFIGURATION DEPENDENT ANALYSIS

By
Aieshah F. Bany Sakher

Supervisor
Dr. Gheith A. Abandah, Prof

ABSTRACT

Multicore processor architectures have been gaining increasing popularity in recent years in high-performance computing (HPC) domain. Many designs are proposed and many commercial multicore processors are introduced. It is important to evaluate the common design alternatives using representative multithreaded applications. Performance evaluation helps the programmers in tuning and developing future parallel applications and helps designers in developing multicore architectures that efficiently run parallel applications.

The purpose of this thesis is to evaluate alternative multicore and many-core designs, identify strengths and weaknesses in current processors, and identify design aspects that have a high positive impact on such processors and areas that need further investigation and improvement. This thesis presents a micro-architectural simulation using the Sniper simulator, a fast and accurate multicore simulator, to evaluate four common Intel server multicore processors (Xeon brand). Two of them are bus-based multi-socket architectures: Dunnington/Core2-based and Gainestown/ Nehalem-based microarchitectures. The others are network on chip (NoC) based architectures: 2D mesh and bidirectional ring interconnection networks. They are Haswell and Xeon Phi based processors. They were chosen because they cover a wide range of recent multicore design options.

In this research, we have chosen eight representative parallel applications from two benchmark suites: Princeton Application Repository for Shared-Memory Computers (PARSEC) and Stanford ParalleL Applications for Shared memory (SPLASH2). We have conducted many raw and normalized experiments for the four designs with two problem sizes of each of the selected applications. In these experiments, we used a comprehensive collection of performance evaluation metrics to facilitate trade-off evaluations. These metrics are execution time, average instructions per cycle (IPC), average core utilization, and power consumption. Also, we analyzed benchmarks cycles per instructions (CPI) cycle stacks changes over time.

We do a normalized comparison where the multicore design alternatives are put on the same technological level with similar component sizes and speeds. The normalized comparison better exposes the performance differences due to microarchitecture main features; like memory hierarchy organization, network interconnection topology, and cache coherent protocols, rather than the underlying technology and component sizes. Discussions of raw and normalized experiments and comparative analysis are included in this thesis.

We found that the normalized Haswell exhibits better execution time (69 ms) and system throughput (1.39 IPC) averaged over the eight multithreaded benchmarks for the large data set. This relatively high performance is due to its architectural features: private level 2 cache, large level 3 shared non-uniform cache access (NUCA), and high-speed core-to-core communication through the bidirectional ring NoC. On the other hand, it relatively consumes large power (52.3 watts).

The nXeon Phi architecture shows lower power consumptions (48.14 watts), but designers should do further research in developing its memory components. Xeon Phi suffers relatively from larger CPI loss in memory intensive applications (1.27 IPC) leading to larger execution times (77.25 ms on average). Most of the performance hit is from the off-chip DRAM access and smaller on-core units.

We concluded that the bus-based microarchitectures are no longer able to meet the requirements of new HPC workloads due to the obvious weakness in their handling of the synchronization and communication overheads, which for sure will increase in future many-core architectures. The normalized Gainestown and normalized Dunnington have average execution times of 74 ms and 73 ms, respectively, and have average throughput of 1.33 IPC and 1.31 IPC, respectively. Also, their power consumptions are 50.14 and 49.99 watts on average, respectively.

Finally, we have shown that the Modified Exclusive Shared Invalid Forward (MESIF) cache coherence protocol enhances the multicore performance, compared with the older MESI protocols. Normalized Dunnington has speedup of 1.028x, normalized Gainestown has 1.027x, normalized Xeon Phi has 1.01x, and normalized haswell does not benefit from MESIF protocol because it has only one socket and all cores share the same NUCA cache.

CHAPTER 1: INTRODUCTION

1.1 Background and Motivation

Multicore architecture is the current step in processor evolution; it is a special kind of a multiprocessor on a single chip (Olukotun, Kunle, et al. 1996). With this advent of multicore processors and their widespread use in the commercial and scientific fields, extensive research areas were open. Thousands of researchers are working on multicore processor designs.

Many design options are in front of the designers because of many system parameters. Vendors and designers have shown an interest in increasing the number of cores and cache sizes. However, this may not be the best performance solution because it imposes high communication overheads. Other options exist in choosing the cache coherence protocols, cache hierarchy, cache associativity degree, cache write locality, chip interconnection networks, and clock speeds.

One of the biggest challenges in effectively using the multicore architectures is tuning the system parameters to have the optimized performance and to determine which option has the largest impact on performance without increasing the energy consumption (Abandah, G.A 1998).

Some scientists try to enhance the performance of multicore processors by exploiting thread level parallelism (TLP) in the applications and tuning these applications to suit the multicore platforms and achieve better performance. In fact, to take the full advantages of multicore architectures, the programmers need to know the characteristics of their parallel applications and how they behave on the multicore systems. The programmers also need to know enough information about the system's performance characteristics like the system's strengths and weaknesses (Abandah and Davidson 1998).

A major challenge with design development phase is the ability to analyze and optimize performance for multicore systems. Computer architects need performance analysis tools and workload characterization methodologies to understand the behavior of existing and future workloads in order to design and optimize future hardware. However, the study of parallel applications performance on alternative system configurations will support future designs of multicore systems by allowing the designers to modify the multicore configurations to achieve higher performance.

There are various system configurations in multicore processor, that include many parameters like the number and speed of processor cores, the number and size of cache memory levels (L1, L2, L3, or more), cache type (private or shared), cache coherence protocols (MESI, MSI, MESIF, etc.), write-through and write-back techniques, cache associativity (Direct or Set associative mapping), and core interconnection networks.

With the emergence of efficient commercial multicore architectures, it is important to find the weaknesses and strengths of the current processors. This involves using appropriate metrics to evaluate the performance of multicore processors like bandwidth and latency (Eeckhout, et al. 2010).

It is important to understand how various designs of multicore processors perform with the current parallel applications by characterizing such applications on various high-end multicore alternatives. We need to find out where and when the system is weak or strong. Our study gets its importance from the fact that it characterizes the parallel applications on multicore systems depending on the system configuration. This evaluation will lead to developing and tuning the multicore parameters to run efficiently on parallel applications with higher performance.

1.2 Research Contributions

In this thesis, we have the following contributions:

- **Evaluating and calibrating common multicore systems,** by using raw and normalized microarchitectural simulations.
- **Comparing the behavior of wide range of multithreaded benchmarks,** analyzing their behavior due to different microarchitectures and different input data sets, and concluding some interesting information about (dis)similarly properties of them.
- **Facilitating trade-off evaluations,** by using comprehensive and representative multicore performance metrics. These metrics are execution time, average instructions per cycle (IPC), average core utilization, and power consumption. Also, CPI cycle stacks changes over time.
- **Finally, identifying multicore designs strengths and weaknesses,** concluding system design features that have significant impact on system performance, and presenting future work directions.

1.3 Research Methodology

The methodology of our thesis consists of the following stages:

- **Investigating and survey for various design options of recent multicore processors**, in order to select few representative multicore design alternatives.
- **Investigating and survey for different multicore simulators**, where this simulator should be able to efficient simulate different multicore design options; can evaluate the performance of these alternatives by determining the systems strength and weakness, and must be flexible and easy to modify system configurations for normalization issue.
- **Investigating the available benchmarks parallel applications**, and selecting a representative set of them for further study. These applications should be representative and cover several types of multithreaded workloads.
- **Implementing raw experiments of each multicore designs**, and gathering the results of these experiments, then analyzing them, so we can determine the best multicore design performance.
- **Implementing normalization experiments**, by normalizing the values or types of not interested design parameters, so we can determine the real impact of any determined performance factors, and the system strength and weaknesses.
- **Conclusion and future works**, where the conclusion and results analysis of this work will be presented and future research direction will be discussed.

1.4 Thesis Outline

This thesis contains five chapters that describe the development of the whole research work. The rest of the thesis is organized as follows:

Chapter two presents a survey of some related work. It includes multicore processors features, performance evaluation techniques, recent and common multicore processors simulators, and multithreaded benchmarks.

Chapter three summarizes our methodology for common multicore performance evaluation, describes case study multicore design alternatives and used multithreaded applications, and then, describes Sniper multicore simulator, host machine, setup environment, and used performance metrics.

Chapter four presents the results of raw and normalized comparisons, presents performance evaluation and results analysis to measure the impact of multicore processor parameters.

Chapter five presents the main conclusions regarding the thesis's methodology, the strengths and weaknesses points of multicore design alternatives depending on simulation results. Additionally, it presents some proposed future work.

CHAPTER 2: LITERATURE REVIEW

2.1 Introduction

This chapter presents a survey of some related work. It includes multicore processors overview, multicore simulators, multithreaded benchmarks, multicore performance evaluation metrics, and a survey of multicore studies in the high performance computing (HPC) domain.

2.2 Multicore Overview

Multicore processors trend is the new trend in computer architecture domain. It can be defined as replicating multiple independent processor cores and implementing multiprocessing in a single physical package called a chip. However, if all cores fit into a single processor socket then it is called Chip Multi-Processor CMP (Barbic, J. 2007). The first multicore processor is IBM Power 4 in 1996 (Tendler, et al. 2002), which has two high-performance microprocessor cores on a single silicon chip. In the past, the trend was to add more components and more cababilities on one die. In Fact, manufacturers cannot do this forever, because of the limitation of improvements of a single core. Many components suffer from communication overheads. Also, it is difficult to make single-core clock frequencies much higher because of the heat problem, design difficulties, and verification. Hence, server farms need expensive air-conditioning.

On the other hand, most of the new applications are multithreaded, because there is a general trend in computer architecture for shifting towards more parallelism: instruction level parallelism (ILP) and thread level parallelism (TLP) (Barbic, J. 2007). In ILP, the parallelism is at the machine-instruction level; the processor can reorder, pipeline instructions, split Instructions into microinstructions, do aggressive branch prediction, etc.

TLP employs parallelism on a big scale, however, the server can serve each client in a separate thread (Web server, database server, etc.). In addition, they employ TLP in desktop applications (Blake, G. et al., 2010). Computer researches predicted that "Anything that can be threaded today will map efficiently to multicore" (Barbic, J. 2007).

After the first publication about the multicore processors by Kunle Olukotun et al. (1996). Thesse researchers, the pioneers of multicore processors, argued that multicore computer processors are likely to make better use of hardware than existing superscalar designs, the multicore processors became very hot topic in the computer engineering (Barbic, J. 2007).

Chip designers continue evolution to increase the number of cores, which is leading to the many-core architectures. However, a many-core processor is one in which the number of cores is large enough that traditional multi-processor techniques are no longer efficient. It can be in the range of several tens of cores and is likely to require a network on chip (NoC). Many-core architecture is a special type of multicore processors. See Figure 2.1 that presents a classification of multicore processors. If all cores are identical, the system is called homogeneous multi-core system and if they are not identical, it is called heterogeneous multi-core systems. Also, like single-processor systems, the cores in multi-core systems may implement architectures such as superscalar, very long instruction word (VLIW), vector processing, single instruction multiple data (SIMD), or multithreading. An example of many core processors is Intel Xeon Phi Knights Corner and Knights Landing processors which contain 60+ cores connected with ring topology network or 2D mesh network respectively. Nowadays, Intel introduced a max number of 72 cores with 4way simultaneous multi-threading (SMT) on the high-end Xeon Phi line processors.

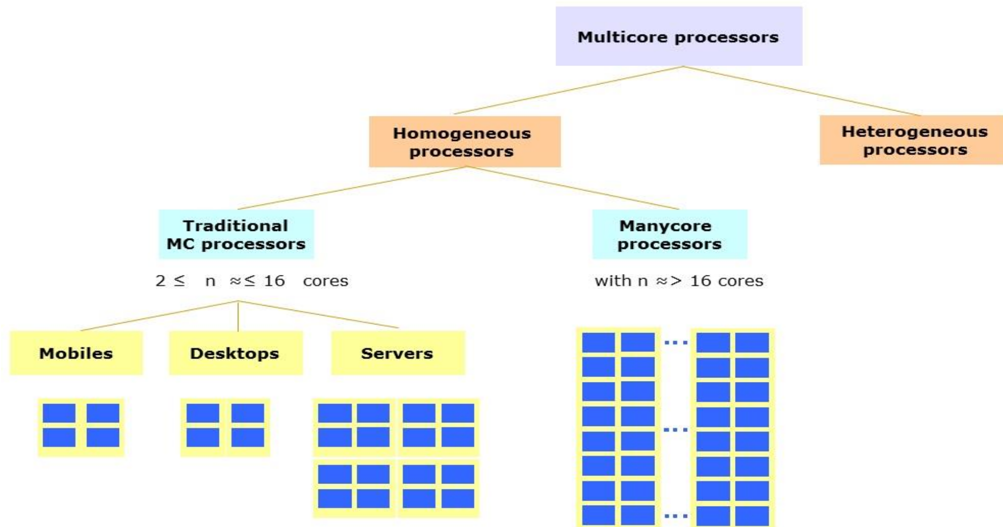


Figure 2.1 Multicore processors classifications[1].

Interconnect systems in multicore systems have gradually evolved from simple busses to more scalable NoC. NoC is a communication centric interconnection approach that provides a scalable infrastructure to interconnect multiple cores and sub-systems (i.e., memory controllers, I/O ports) in a system on chip (SoC) (Bhople, et al. 2013). Common network topologies to interconnect cores include: conventional bus-based, ring, 2-dimensional mesh as an example of NoC and crossbar networks (Lecler, et al. 2011). The bus topology used in processors with front side bus (FSB), which means that several resources use the same communication channel (e.g., Intel Nehalem microarchitecture). In the ring topology, which was used in high-end Intels desktop and server multicore processors, the resources are connected to each other in a ring (ex. Intel corei7 and Xeon E5 v3). Every core and resource is connected to its two neighbours, all communication with other resources then should pass through the neighbours. In the mesh topology any node can communicate with all other nodes in the system. The 2D-mesh topology is a type of mesh network in which nodes form a two-dimensional grid where each node is connected to the four adjacent notews. The cores at the edges have only two or three connections since they do not have more adjacent cores (Bhople, et al. 2013).

1. http://slideplayer.com/slide/9016026/

Common cache coherence protocols include Modified Exclusive Shared Invalid Forward (MESIF) protocol, which is recently used by Intel's multicore processors, they added forward (F) state to the previous MESI protocol. Advanced Micro Devices (AMD) processors use the Modified Owned Exclusive Shared Invalid (MOESI) protocol that benefits from the added owned (O) state to the original MESI protocol (Tiwari, A. 2014). The two protocols support cache to cache transfers in order to efficiently transfer data between caches instead of expecting information from the main memory. The added F state in the MESIF is an optimization to the MESI protocol, where a read request for data in the "shared" state is serviced by one of the sharers of the data instead of waiting for the data to come from the main memory. Where the added O state in the MOESI protocol lets the caches to share data which are dirty as long as one of the sharers takes the responsibility of owning the data. Requests for the shared data will be satisfied by the owner (Tiwari, A. 2014).

Intel and AMD are the two most common vendors in desktops, workstations, and server's processors. On the other hand, Advanced RISC Machine (ARM) is the leader in mobile processors and embedded systems (Furber, S. B. 2000) (Jarus, et al. 2013). Recent multicore processors differ in many features; like cache parameters, allocation /replacement policies, and write policies. Moreover, the number of levels in the cache hierarchy can be two, three, and four with specifications on the type of inclusion policy. Even more, multicore processors differ in the configuration of some or all levels of the cache hierarchy to be shared or private amongst the multiple threads and cores.

2.3 Multicore Simulators

Computer architecture research is mainly driven by simulation in HPC domain (Ricco A. 2013). There are many simulators that are used to evaluate multicore processors in the design phase. In this Section, we review the most popular multicore simulators and report most of their features.

The sniper multicore simulator[1] is an open source and licensed execution-driven simulator. It is based on the interval core model and the Graphite simulation infrastructure, so it is a fast and accurate simulation. It is fast because it is based on a Pin on-the-fly instrumentation tool. It allows a range of flexible simulation options like in-order and out-of-order (OoO) cores when exploring different homogeneous and heterogeneous multi-core architectures. It supports time simulation for multithreaded and multi-programmed workloads and shared-memory applications with 10s to 100+ cores, at a high speed when compared to existing simulators. The sniper simulator has been validated for real hardware of Intel Nehalem and Intel Core2 with error accuracy about 11%, it supports modern Linux-OS (Redhat EL 5,6, Debian Lenny+, Ubuntu 10.04-14.04+, etc.) (Carlson, T. et al., 2011) (Carlson, T. et al., 2014a) (Carlson, T. et al., 2014b) (Florea, et al. 2014). The sniper simulator is explained with higher details in Chapter Three.

1 http://marss86.org/~marss86/index.php/Home
1. https://github.com/mit-carbon/Graphite
2. https://groups.google.com/forum/#!forum/graphite-sim

The MARSS simulator[2] (Micro-ARchitectural and System Simulator for x86-64 based Systems) is an open-source, cycle-accurate, full system simulator, Quick Emulator (QEMU) based, full-system emulation environment with models for the chipset and peripheral devices. It supports detailed models for coherent caches, bus based and on-chip interconnections networks, and MESI or MOESI cache coherent protocols. MARSS has its root on MPTLsim that is the multicore version of PTLsim simulator. Also, it runs in user-space only, without any need for root access or the installation of any kernel module (Patel, et al 2011).

The Graphite simulator[3] is an open source, distributed parallel simulator. It can explore dozens, hundreds or thousands of cores. And also, it is capable of accelerating simulation by distributing simulated cores across multiple Linux machines. Graphite[4] is the root of the Sniper simulator (Miller, et al. 2010).

The CMP$im simulator, a Pin-based on-the-fly multi-core cache simulator, is a flexible multicore simulator and uses a dynamic binary instrumentation tool as an alternative to trace-driven and execute-driven approach. It is a memory system simulator that characterizes memory performance of x86 workloads on multicore processors. CMP$im is fast, fully configurable and can gather detailed cache performance statistics. Users can model any kind of cache hierarchy and supports multi-cores and multi-threaded cores, but the disadvantage of this simulator is the lack of speculation and out-of-order execution (Jaleel, et al. 2008).

1 http://marss86.org/~marss86/index.php/Home
1. https://github.com/mit-carbon/Graphite
2. https://groups.google.com/forum/#!forum/graphite-sim

Zsim[1], is an open-source and a Pin-based simulator, like Graphite, CMP$im, and Sniper simulators. It is fast and accurate microarchitectural simulation system of thousand-core systems. Its main goal is to focus on memory hierarchies and large, heterogeneous systems. It supports detailed core models (including OoO cores) with instruction driven timing models. It supports complex workloads, including multi-programmed, client-server, and managed-runtime applications, without the need for full-system simulation (Sanchez, et al. 2013).

The Manifold[2] simulator, a parallel simulation framework for multicore systems, is a full system, open source simulator, and supports parallel and serial simulations that is transparent to the users. It supports Stanford ParalleL Applications for Shared memory (SPLASH2) and Princeton Application Repository for Shared-Memory Computers (PARSEC) benchmark suites and it has integrated library of power, thermal, reliability, and energy models. It is defined as flexible and scalable simulator. However, Manifold's component-based design provides the user with the ability to easily replace a component with another for efficient exploration of the design space (Wang, et al. 2011) (Wang, et al. 2014).

The ESESC[3], a fast multicore simulator using time-based sampling, is an open source, very fast simulator that supports heterogeneous multicores, with detailed power, thermal, and performance models for modern out-of-order multicores. It supports multicore homogeneous and heterogeneous configurations, various memory hierarchies, and on-chip memory controller. It can model power and temperature in addition to performance and their interactions (Ardestani, et al.2013) (Ardestani, et al.2014).

1. https://github.com/s5z/zsim
2. http://manifold.gatech.edu/
3. http://masc.soe.ucsc.edu/esesc/

The Hornet[1], a parallel, open source and cycle-level multicore simulator, is based on an ingress-queued wormhole router NoC architecture. It is highly configurable, scalable, accurate multicore simulation in the 1000-core range. Hornet permits tradeoffs between perfect timing accuracy and high speed with very good accuracy (Lis, et al. 2011) (Ren, et al. 2012).

Gem5[2] is a simulation infrastructure introduced from the merger of the best aspects of the M5 and GEMS simulators. M5 provides a highly-different configuration infrastructures, multiple ISAs, and diverse central processor unit (CPU) models. GEMS simulator complements these features with a detailed and flexible multicore system features. Such features include memory system and support for multiple cache coherence protocols and interconnect models except MESIF. Although Gem5's simple CPU models are much faster than their detailed counterparts, they are still considerably slower than binary translation-based simulators (Binkert, et al. 2011).

Finally, Carlson et al. (A,2014) concluded that the good simulator should have simulation infrastructure that has many important requirements; particularly:

- **Efficiency**, both in time and space, by only simulating relevant parts of the benchmark in detail, avoiding long warm-up time; and occupying a small disk footprint for storing workloads.
- **Accuracy**: simulation results should be representative of running the complete workload.
- **Reproducible**: the unit of work must be fixed across architectures to allow for valid comparisons to be made; workloads must be easily shareable while guaranteeing (mostly) identical simulation results.

1. http://csg.csail.mit.edu/hornet/
2. http://gem5.org/Main_Page

2.4 Multicore Benchmarks

Modern processors are designed as a SoC, which can execute many independent threads in parallel. Hence, the performance measurement should be done on multithreaded benchmarks.

There are many general proposed benchmark suites such as Standard Performance Evaluation Corporation (SPEC), SPEC CPU2006 suite (SPEC, 2014), which is a collection of compute-intensive applications and is a representative of scientific and engineering applications. These applications are serial programs that are not suitable for studies of multicore platforms.

SPLASH2 suite (Woo, et al., 1995) is a collection of multithreaded applications, which is representative of scientific, engineering, and graphic applications. These applications are widely used in the HPC domain.

PARSEC suite (Bienia, et al., 2008) is a collection of multithreaded commercial and new applications in recognition, data mining, and synthesis (RMS) (Dubey, 2005), which is representative of animation, media processing, computer vision, enterprise servers, and computational finance applications.

The PARSEC benchmark suite is often used in studies related to multicore processors. Bienia, et al. (2008) characterized PARSEC benchmarks to show that their benchmark suit has diverse types of multi-threaded behaviors. Bhattacharjee and Martonosi (2009) characterized the translation look-aside buffer (TLB) behavior of the PARSEC benchmark applications. Contreras and Martonosi (2008) characterized a subset of PARSEC benchmark applications that were compiled with Intel threading building block (TBB) on AMD dual-core processors to determine the sources of overhead within the TBB. The recent threading library TBB is a C++ template library developed by Intel

for writing software programs that take advantage of multi-core processors. These entire tools target helping the programmers to develop efficient parallel applications.

Dey, et al. (2011) characterized PARSEC benchmark applications to measure the effect of shared resource contention on performance. They classified resource contention into intra-application contention, which is the contention among threads from the same application, and inter-application contention, which is the contention among threads from different applications.

Natarajan and Chaudhuri (2013) characterized a set of multi-threaded applications selected from the PARSEC, SPEC openMP, and SPLASH to understand the last level cache (LLC) behavior of multi-threaded applications. They proposed a generic design that introduces sharing-awareness in LLC replacement policies. They showed that their design could significantly improve the performance of LLC replacement policies.

Despite that SPLASH was released at the beginning of the 1990s for the HPC domain, it is widely used beside PARSEC in the recent multicore research (Shi and Khan, 2013), (Shriraman et al., 2013), (Krishna et al., 2013).

Bienia, et al. (2008b) showed that SPLASH and PARSEC complement each other in terms of the diversity of working set size, cache miss rate, and distribution of instructions. Heirman, et al. (2011) characterized a set of multithreaded benchmarks from SPLASH2, PARSEC, and Rodinia suites in order to understand scaling bottlenecks in multi-threaded workloads. They concluded that the three benchmark suites cover similar areas in the workload space.

Mohammad and Abandah (2016) used SPLASH2 besides PARSEC in their multicore applications characterization independently on hardware configurations. They used eight applications from the two benchmark suites, which were selected as they represent a wide range of multicore applications.

Researchers prefer doing their simulations on the parallel region of the benchmark called the region of interest (ROI) (Southern, G. 2016), i.e. Sniper simulator has the feature of running all simulation on just the ROI of the multithreaded benchmarks. Furthermore, it can eliminate the initialization (warm up) and finalization transient times of any simulation, which could give more accurate results on max processor parallelization performance and give hints on max run-time peak power (Jha, et al. 2017).

2.5 Multicore Studies

The multicore studies branched out in more than one direction in studying the multicore processors. Researchers proposed and discussed various design options like the number and speed of chip cores, the interconnection networks, cache hierarchies, and cache coherence protocols. They study how these issues affect the important metrics of performance of multicore processors like throughput, execution time, energy, CPU clock speed, memory bandwidth, inter-core communication overhead, and scalability ability.

Shukla, et al. (2015), in their literature survey, conclude that all studies attempt to address some isolated issues and some common issues. Less research is available about the correlation between multicore performance parameters, or about obtaining the performance issues when many parameters influence each other.

Some researchers tried generally to summarize many multicore processor performance parameters. Ubal, et al. (2007) proposed Multi2sim tool: a simulation framework to evaluate multicore-multithreaded processors for the analysis of multicore architecture performance. They studied three major performance elements of multicore architecture: processor cores, memory hierarchy, and the interconnection network. By this tool, they can graphically show how cores execute threads, how many cores are idle, and how the interconnection network is utilized by all cores for the inter-core communication.

Blake, et al. (2009) studied five major attributes common among multicore architectures and discussed the tradeoffs for each attribute in the context of actual commercial products. These areas were application domain, power/performance, processing elements, memory, and accelerators/integrated peripherals.

Other studies are about the multicore interconnection networks. Bononi, et al. (2007) study four NoC topologies; ring, 2d mesh, spidergon and unbuffered crossbar. They found that ring forces some packets to follow longer paths than other topologies although that mesh has more channels for data transfer. Also, they found that ring and spidergon have the best performance of elapsed execution cycles while mesh and crossbar perform worse than expected. In percentage, the difference between the performances, spidergon and ring behave equivalently being 3.3% faster than mesh and 6.2% faster than crossbar. And by considering the total buffer size for the 12-node architectures, they note that mesh and crossbar have less buffering memory: 204 flits for mesh and 0 for crossbar vs. 288 flits for ring and 216 for spidergon.

Mohanty, et al. (2013) had concluded that the evaluation of performance is dependent on the internal network, e.g., ring network and a hybrid network. They used the metrics execution time and speed-up to show the performance of ring network and a hybrid network.

Ingle, et al (2013) studied the performance of mesh topology of NoC architecture using Source routing algorithm. They observed that topology and routing algorithm are two key features which distinguish various NoC platforms. And, 2D mesh topology is one of the most frequently mentioned topologies for an NoC design due to its natural layout mapping onto an SoC. and because of its network Scalability and the use of a simple routing algorithm.

Some researchers try to analyze only one processor architecture. Molka, et al. (2015) studied cache coherence protocol and memory performance of the Intel Haswell architecture. In addition, Molka, et al. (2009) have pointed out some of the fundamental details of the Intel Nehalem microarchitecture with its integrated memory controller, quick path interconnects, and non-uniform memory access (NUMA) architecture. They used benchmarks to measure the latency and bandwidth between various locations in the memory subsystem.

Rolf (2009) also studied Intel’s Nehalem architecture and has summarized the major improvements to the architecture of Intel’s previous multicore architectures with a special focus on the memory organization and cache coherency scheme.

Rahman, R. (2013) studied the Intel Xeon Phi architecture and presented tools and guides for the application developers.

Jarus, et al. (2013) presented performance evaluation and energy efficiency of high-density HPC platforms based on Intel, AMD and ARM processors. They discussed the trade-off that could exist between computing and power efficiency.

Another point of view, some researchers create new hardware techniques. Kakoullie, (2012) created a hot spot router, which is responsible for the data exchange among the cores in the multicore processor. He showed that the hot spot has major improvements in the performance of multicore architecture.

Duarte, et al. (2010) proposed the Accelerator scheme. With the help of this scheme, data movement between the main memory and the cache memory can be increased. It can improve the performance of multicore architecture. They showed that this scheme has a power enhancement just in case of copy data from memory to cache and not suitable in the case of real-time applications.

Cache coherence protocols have a high impact on multicore performance. Therefore, many researches concentrate on this hot topic. Tiwari, et al. (2014) used the GEM5 simulator and SPLASH benchmark to compare the performance of cache coherence protocols on multicore architectures such as snoopy and directory protocols. Snoopy coherence is studied with Modified MOESI coherence protocol and directory coherence is studied with MI, MESI TWO LEVEL, MESI THREE LEVEL, MOESI, and MOESI TOKEN coherence protocols.

Martin, et al (2012) concluded that cache coherence protocols can be affected by many factors including parallel programming communication and synchronization. Marty, (2008) contributed a hierarchical coherence protocol, directory CMP, that uses two directory-based protocols bridged together to create a highly scalable system. He compared this protocol with token CMP and extended the later to create a multiple-CMP

system. His simulation results showed that the token CMP has better performance than directory CMP. He also proposed a new cache coherence protocol that exploits a ring's natural round robin order.

Other researchers studied and analyzed cache and memory hierarchy in multicore processors. Ramasubramanian, et al., (2011) used M5sim tool for analyzing cache memory performance and have found that cache memory plays a crucial role in deciding the performance of the multicore system.

Jaleel, et al. (2006) characterized LLC memory behavior of parallel bioinformatics data mining workloads on multicore processors. They concluded that shared last-level cache memory is better than private last-level cache memory for high-performance systems.

Tudor and Young, (2011) concluded that memory contention is a big issue in the performance of the multicore architecture. The cache misses depend +on the problem size. If it is small, then there are fewer cache misses. Their model has many limitations but it is useful when there are large memory requirements.

Zhou, et al. (2009) proposed a concept of performance fairness metric depending on management mechanism. They also designed an adaptive hardware mechanism for enforcing performance fairness on the shared cache.

Some studies evaluate different multicore designs like a single chip and superscalar multiprocessor. Chaturvedi, et al. (2013) compared a single chip multiprocessor design with the dynamically scheduled superscalar processor. They used GEM5 full system functional simulator extended with multi-facet GEMS. Their results show that the single chip multiprocessor performs 50–100% better than the wide superscalar processor with the applications that have full parallelism. On applications that

can not be parallelized, the superscalar processor performs marginally better than one processor of the multiprocessor architecture.

Fasiku, et al. (2014) worked on performance evaluation studies on AMD dual core and Intel dual core processors to find which of processor has better execution time and throughput. They studied the architecture of AMD and Intel dual core processors and used SPEC CPU2006 benchmarks suite to measure their performance. The results of overall execution and throughput time measurement showed that the execution time of CQ56 Intel Pentium Dual-Core processor is about 6.62% faster than AMD Turion II P520 dual-core processor while the throughput of Intel Pentium dual-core processor is 1.06 times higher than AMD Turion (tm) II P520 dual core processor. They concluded that Intel Pentium dual-core processors exhibit better performance probably due to the following architectural features: faster core-to-core communication, dynamic cache sharing between cores, and smaller size of level 2 cache.

There is also another research comparison of memory write policies for multicore cache coherent systems. Pierre, et al. (2012) showed that write-through-invalidate protocols are a possible and simple solution to maintain coherency and this protocol performs very well compared with a classic write-back-MESI protocol in both execution time and generated traffic.

Some computer scientists saw the solution with working on improvements on parallel programming and applications. Shukla, et al. (2015) concluded that the development of parallel programming is useful in the growth of multicore architectures. They said that operating system (OS), scheduling algorithms, and memory management should be developed for the multicore architectures.

Eduardo et al. (2014) proposed a new thread mapping technique to optimize the communication overhead. They also proposed algorithms to dynamically migrate the threads. Using the NAS parallel benchmarks and with a producer-consumer benchmark, they observed the importance of dynamic mapping over static mapping.

Several studies have proposed different techniques to characterize shared memory behavior of several types of parallel applications on multicore platforms. Pan et al, (2014) used a set of benchmarks from the PARSEC benchmark suite to evaluate their newly creative model, which can be used to predict the private cache misses of a multi-threaded application for different cache sizes. This approach can be used to guide program optimizations to improve utilization of the private cache.

Woo, et al. (1995) used configuration dependent analysis to characterize several aspects of the SPLASH benchmark suite. Abandah and Davidson (1998, a) proposed a Configuration Independent Analysis Tool (CIAT) to characterize configuration independent characteristics such as memory access instructions, concurrency, communication patterns, and sharing behavior of shared-memory applications on a varying number of processors. Abandah, (1998) proposed Configuration Dependent Analysis Tool (CDAT) to characterize memory behaviors such as cache misses and false sharing that depend on configuration parameters such as cache block size. CDAT is a simulator that has memory, cache, bus, and interconnection models. By using a configuration file, users can specify a system configuration through specifying the coherence protocol, size and speed of system components, and processors and memory banks interconnections.

Mohammed and Abandah, (2015) developed CIAT tool to characterize shared memory multithreaded applications on recent common multicore processors. They

proposed an on-the-fly, configuration-independent characterization approach for characterizing the inherent communication characteristics of multicore applications. Recent research goes far energy and power-aware systems. They proposed techniques for power consumption estimation. Priya, et al (2016) also presented a survey of different techniques to improve the energy consumption of multicore processors. They considered parameters in a survey like dynamic energy, area, throughput, performance, lifetime, harmonic mean instruction per cycle, miss rate and latency.

Heinrich, et al, (2017) present an extension of the SimGrid simulation toolkit that addresses these challenges. They firstly introduce a model for application energy consumption that supports dynamic voltage/frequency scaling (DVFS) of simulated processors. Secondly, they discuss means to account for coarse-grain memory effects in multi-core architectures. The advantages of their approach, compared to cycle-level simulators, are faster simulation run times and enhanced scalability with, retained excellent accuracy if the target platform is correctly modeled.

Jho, et al (2017) proposed a two-tier hierarchical power management methodology to exploit per tile voltage regulators and clustered last-level caches. In addition, they included a novel thread migration layer that (i) analyzes threads running on the tiled many-core processor for shared resource sensitivity in tandem with core, cache and frequency adaptation, and (ii) co-schedules threads per tile with compatible behavior. On a 256-core setup with 4 cores per tile. They showed that adding sensitivity-based thread migration to a two-tier power manager improves system performance by 10% on average (and up to 20%) while using 4× less on-chip voltage regulators. It also achieves a performance advantage of 4.2% on average (and up to 12%) over existing solutions that do not take DVFS sensitivity into account.

As we explained previously, the first aspect of multicore development is the microarchitectural simulation, processors designers predict the processor performance by using some efficient performance metrics (Eeckhout, et al. 2010); some of them indicates the overall system performance. However, the most used metric is the system latency which indicates the total execution time needed for execution of any selected program/task or many simultaneous programs executed together. The other metric is the system throughput, which is a measure of how many units of a specific unit of workload/instructions the processor can process in a given amount of time. The other specific performance metrics that affect overall performance include branches, caches, and dynamic random access memory (DRAM) misses. The time loss due these misses can be shown by using CPI stack (Eeckhout, et al. 2010).

The processor power and energy consumption are the main constraints for modern high-performance multicore systems. Some simulators give us hints on the change of performance over time, ex. Sniper simulator can show the change of CPI stack and consumed power over time (Jho, et al. 2017).

Our study differs from prior works in that we use configuration dependent characterization technique to characterize the multithreaded applications depending on a specific configuration of some system components. We will evaluate four common multicore processors from state of the art multicore systems.

However, we will analyze the configuration dependent characteristics of multithreaded applications (SPLASH2 and PARSEC benchmark suites) on different multicore platforms. Our study will use Sniper, fast, accurate, and efficient multicore simulator to determine the major system factors that have a large impact on system performance.

We use representative performance evaluation metrics for analysis; i.e. execution time, average core IPC, CPI stack changes over time, average core utilization, cache misses and finally, average run time power. We also study the changes on performance due change of cache coherency protocol MESI to MEISF.

System designers can benefit from our benchmark applications characteristics analysis to investigate much larger design space in the early design stages of their designs. On the other hand, software researchers and developers can benefit from using cycle stacks, they can easily identify the performance bottleneck of an application on a particular platform and study how application behavior changes with varying hardware configurations, while computer architects can use cycle stacks to optimize different architectures (Heirman, et al. (2011).

CHAPTER 3: METHODOLOGY AND TOOLS

3.1 Introduction

This chapter summarizes our methodology for performance evaluation of multicore design alternatives, and characterizing multi-core applications on these designs. Then, describes Sniper simulator and the study metrics that are used to evaluate these multicore design systems.

3.2 Overview

The methodology relies on choosing four commercial multicore designs that cover different options of multicore processor parameters and choosing a set of benchmarks, which are representative of multithreading applications based on the recent related studies. Then, we determine best performance parameters that have the main impact on multicore processors performance. The Sniper simulator is a fast and accurate system-based simulator cooperative with the Pin dynamic instrumentation tool which instruments the multithreaded applications dynamically during the simulation and sends application characteristics to Sniper that analyze them. Sniper outputs many result files like the CPI stack and Power. It gives us visualization results which explain how CPI changes over time.

We perform micro-architectural simulation using the Sniper x86-64 simulator on four multicore design alternatives based on commercial Intel's server processors.

3.3 Multicore Design Alternatives

In our study, we investigated the available commodity multicore processors in order to determine their important features. We concentrated on the differences that cover important multicore processor issues, which face designers in the processor design phase.

The main issues are multicore interconnection network (Bus-based, or NoC-based 2D mesh (Tilera like), and ring topology), the memory hierarchy issues specified by the number and type of the core caches, private, shared, or non-uniform cache access (NUCA) caches, and the cache coherence protocols (we have intended to study the MESI and MESIF protocols).

We have chosen four representative multicore processors from Intel's server multicore list, which cover all the previously mentioned design issues. We have performed micro-architectural simulation and tradeoff between optimum performance and power consumption. We also investigated their strengths and weaknesses points. By so, we can recommend useful multicore design features.

In this section, we discuss their design options and touch on their main features. We choose Intel's multicore server processors (Xeon brand X86-64 processors) in our study. They have the same microarchitecture with the same line desktop-grade multicore processors. However, they have some advantages over desktop processors, like limited power consumption due to lower clock rates (since servers run more tasks in parallel than desktops do), their multi-socket capabilities, higher core counts, larger cache sizes that support Error-correcting code memory (ECC RAM), and more multiprocessing capabilities.

Table 3.1 shows multicore design features for the four commercial Intel's server processors. The full details of their features are in Appendix B. These four processors are:

- [1]Intel Xeon Processor X7460 (Dunnington or Core 2 codenamed microarchitecture), Sep 2008
- [2]Intel Xeon Processor X5550 (Gainestown or Nehalem-EP codenamed microarchitecture), Jan 2009
- [3]Intel Xeon Phi coprocessor (Knights Corner codenamed microarchitecture), Nov 2012
- [4]Intel Xeon Processor E5-2667 v3 (Haswell-EP codenamed microarchitecture), Sep 2014

Table 3.1. Multicore design features for the four commercial Intel's server processors.

	[1]Intel Xeon Processor X7460	**[2]Intel Xeon processor X5550**	**[3]Intel Xeon Phi Coprocessor 5110P**	**[4]Intel Xeon processor E5-2667 v3**
Code Name	Dunnington/ Core 2-based	Gainestown/ Nehalem-based	Knights Corner (KNC)	Haswell
Launch Date	Q3'08	Q1'09	Q4'12	Q3'14
Lithography	45 nm	45 nm	22 nm	22 nm
# of Cores	6	4	60	8
Processor Base Frequency	2.66 GHz	2.66 GHz	1,05 Ghz	3.20 GHz
Total LLC	16 MB L3	8MB L3 per socket	30 MB L2	20 MB NUCA L3
Bus Speed	1066 MHz FSB	6.4 GT/s QPI	5 GT/s QPI	9.6 GT/s QPI
Link bandwidth (Bus BW)	8 GB/s for two directions	25.6 GB/s for two directions	256 GB/s for two direcion	80 GB/s for two directions
TDP	130 W	95 W	225 W	135 W
Dispatch width	4micro operations	4 micro operations	2 micro operations	4 micro operations
Reorder buffer	96 entries	128 entries	32 entries	192 entries
Branch predictor Mispredict penalty	Pentium M 15 cycles	Pentium M 8 cycles	Pentium M 5 cycles	Pentium M 14 cycles
# of QPI Links	No QPI	2 between sockets	2 between tiles	2 between tiles
Interconnection network	Bus-based network / FSB	Bus-based network / QPI	NoC-based/ 2D-mesh network	NoC-based/ bi-directional ring network

1https://ark.intel.com/products/36947/Intel-Xeon-Processor-X7460-16M-Cache-2_66-GHz-1066-MHz-FSB
2https://ark.intel.com/products/37106/Intel-Xeon-Processor-X5550-8M-Cache-2_66-GHz-6_40-GTs-Intel-QPI
3 https://ark.intel.com/products/71992/Intel-Xeon-Phi-Coprocessor-5110P-8GB-1_053-GHz-60-core
4 http://ark.intel.com/products/83361/Intel-Xeon-Processor-E5-2667-v3-20M-Cache-3_20-GHz

D-TLB **Size** **Associativity**	 0 1	 64 4	 0 1	 64 4
I-TLB Size **Associativity**	0 1	128 4	0 1	128 4
S-TLB **Size** **Associativity**	 0 1	 512 4	 0 1	 1024 4
L1 features: private/shared **Size** **Associativity** **Data access time** **Tags access time**	Private L1 x 6cores 32 KB L1-D 8 way L1-I 8 way 3 cycles 1 cycles	Private L1 x 8cores 32 KB L1-D 4 way L1-I 8 way 4 cycles 1 cycles	Private L1 x 60 cores 32 KB L1-D 4 way L1-I 8 way 3 cycles 1 cycles	Private L1 x 8cores 32 KB L1-D 8 way L1-I 8 way 3 cycles 1 cycles
L2 features: private/shared **Size** **Associativity** **Data access time** **Tags access time**	Shared by 2 cores x 3 3072 KB 12 way 14 cycles 3 cycles	Private cache x 8 256 KB 8 way 8 cycles 3 cycles	Private cache x 60 512 KB 8 way 22 cycles 5 cycles	Private cache x 8 256 KB 8 way 8 cycles 3 cycles
L3 features: private/shared/ NUCA cache **Size** **Associativity** **Data access time** **Tags access time**	Shared by 6 cores 16384 KB 16 way 96 cycles 10 cycles	Shared by 4 cores x 2 8192 KB 16 way 30 cycles 10 cycles	No L3 cache	NUCA cache shared by 8 cores 8192 KB 16 way 30 cycles 10 cycles
DRAM : **Number of MC** **Per controller bandwidth** **Latency**	 1 MC per 6 cores 2.5 GB/s 173 ns	 2 MC per 8 cores 7.6 GB/s 45 ns	 8 per 60 cores 32 GB/s 80 ns	 1 per 8 cores 68 GB/s 45 ns
Memory type	DDR2	DDR3	GDDR5	DDR4
Vdd	1.6 volts	1.2 volts	1.05 volts	1.2 volts
Cache coherence protocol	MESI	MESI	MESI	MESI inside socket MESIF in case of multi-socket

1https://ark.intel.com/products/36947/Intel-Xeon-Processor-X7460-16M-Cache-2_66-GHz-1066-MHz-FSB
2https://ark.intel.com/products/37106/Intel-Xeon-Processor-X5550-8M-Cache-2_66-GHz-6_40-GTs-Intel-QPI
3 https://ark.intel.com/products/71992/Intel-Xeon-Phi-Coprocessor-5110P-8GB-1_053-GHz-60-core
4 http://ark.intel.com/products/83361/Intel-Xeon-Processor-E5-2667-v3-20M-Cache-3_20-GHz

3.3.1 Dunnington / Core 2-based Microarchitecture

Intel Xeon X7460 is based on Intel Core 2 series, coden amed Dunnington, Intel's first multicore which was introduced on 15 September 2008. It features 45 nm technology node running at 2.66 GHz. Figure 3.1 shows the die microarchitecture containing a six-core design that contains three Core 2 dies put in one chip and 16 MB shared level three cache. It features 1066 MHz FSB. Dunnington supports double data rate memory (DDR2-533 MHz), and have a maximum thermal design power (TDP) below 130 W (see Table 3.1 for other design features).

The purpose of this study is to do a performance evaluation of different commercial multicore processors dependent on the main system performance factors. The number of cores per multicore per socket is known, that has a direct relation on performance. By so, we fixed the number of cores to all design alternatives to eight cores. The first performance evaluation technique is doing a raw comparison, for that purpose, we made some minor modifications to the Dunnington microarchitecture. Hence, in Dunnington microarchitecture, we use eight cores Intel Xeon X7460 based architecture, 2 sockets with two memory controller MC, one MC for each socket that will double the memory bandwidth and the L3 total size. Figure 3.1 shows the 6 core Dunnington-based system.

1. http://www.hardwarezone.com.sg/feature-intels-cpu-roadmap-nehalem-and-beyond
2. https://www.bjorn3d.com/2008/11/intel-core-i7-920-nehalem/

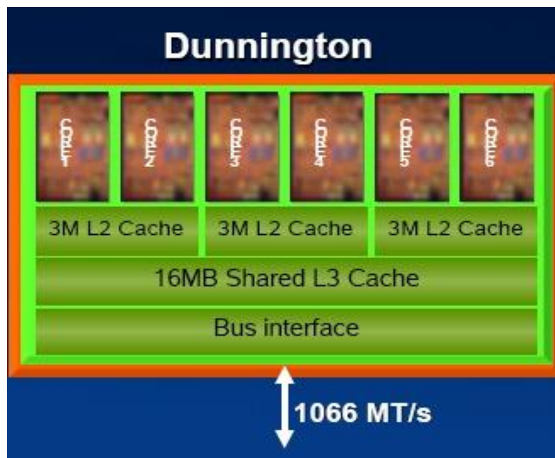


Figure 3.1 Dunnington microarchitecture[1].

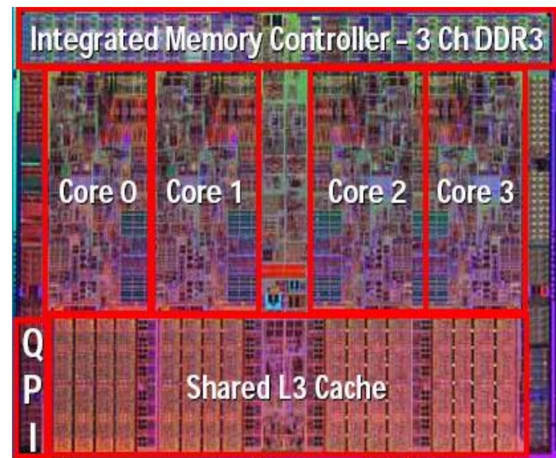


Figure 3.2 Gainestown microarchitecture[2].

3.3.2 Gainestown / Nehalem-based Microarchitecture

Xeon X5550 Core microarchitecture is based on the Nehalem microarchitecture, which used 45 nm manufacturing technology. Figure 3.2 shows a Core 2 die shot that features 4 cores sharing 8MB cache and supports multi-socket. Each core has two levels of private caches and works at 2.66 GHz. Intel Core i7 is the first processor released with the Nehalem architecture. The server version for Nehalem has performance improvements over the previous server processors. They mainly rely on using integrated memory controller IMC that uses 3 channels of DDR3 and using QuickPath interconnect (QPI) running at 6.40 GT/s. QPI is a new point-to-point processor interconnect replacing the legacy front side bus (FSB). Other advantages are the support of simultaneous multithreading by the multiple cores, hyper-threading (HT) of two threads per core. Additionally, Nehalem has fewer branches miss-predict penalty cycles, equal to eight cycles; Core2 has 15 cycles miss-predict penalty. See Table 3.1 for more Nehalem features. Also, here we will use two sockets, 4 cores each, to reach the required eight cores.

1. http://www.hardwarezone.com.sg/feature-intels-cpu-roadmap-nehalem-and-beyond
2. https://www.bjorn3d.com/2008/11/intel-core-i7-920-nehalem/

3.3.3 Haswell Microarchitecture

Haswell microarchitecture was introduced in September 2014. Figure 3.3 shows the Haswell die shot. It features 22 nm technology node running at 3.20 GHz. Xeon processor E5-2667 v3 consists of eight cores sharing an L3 NUCA cache (a distributed shared LLC) connected in a ring topology. However, logically there are indeed a single NUCA cache and a single tag directory that are shared by all cores. However, each physical slice handles a distinct set of cache blocks. So, all slices can operate completely independently from each other (see Table 3.1 for other features).

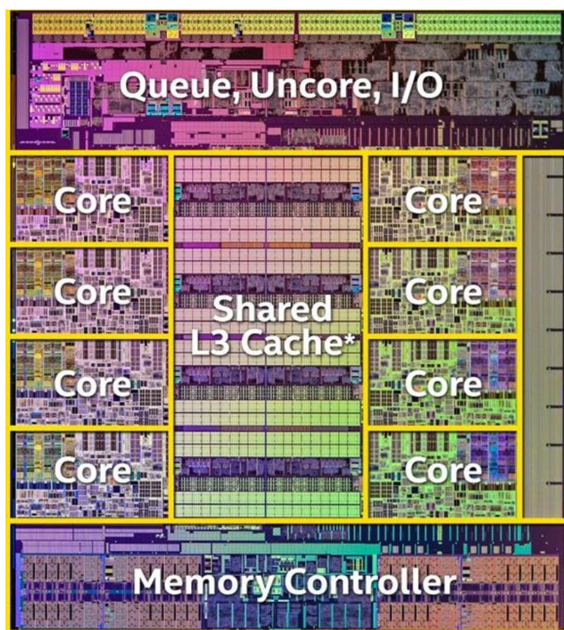


Figure 3.3 Haswell Microarchitecture[1].

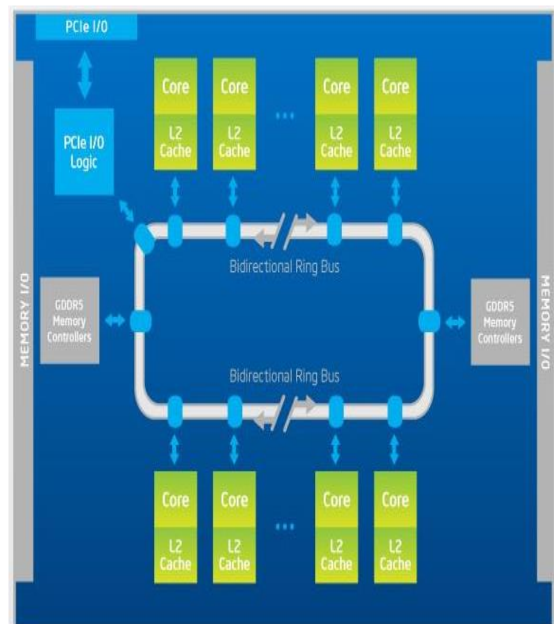


Figure 3.4 Xeon Phi Microarchitecture[2].

3.3.4 Xeon Phi Knights Corner (KNC) Microarchitecture

Intel introduced the first commercial product of the Xeon Phi line/ Knights Corner (KNC) in November 2012 that belongs to the many integrated core (MIC) design space. This product contains many Intel CPU cores combined in a single chip by a high bandwidth Bi-directional ring topology.

1. https://www.pcper.com/reviews/Processors/Haswell-E-Intel-Core-i7-5960X-8-core-Processor-Review
2. http://gray.biji.us/xeon-architecture/

Figure 3.4 shows the block diagram for a KNC die, which targets the highly parallel workloads. Each core has private small L1/L2 caches, directory-based coherency with MESI protocol, and high memory bandwidth (see Table 3.1 for more details). The main purpose of the Intel Xeon Phi coprocessor is offloading the main processor for doing the heavy computations. Designers classify the Intel Xeon Phi coprocessor as Symmetric Multiprocessor (SMP) with shared uniform access memory. However, each core has access to all memory at the same priority.

For our study target, as mentioned in the previous section, we choose eight cores for all multicore alternatives to be equally evaluated, and also, Sniper doesn't model ring topology without NUCA cache. Therefore, we modify the network to be Tilera-like 2d mesh network (4x2 2d mesh size). Actually, the 2D mesh network is used by Intel in the next generation Xeon Phi line (KNL). So, it is preferable to study how it affects the system performance. The new design is explained further in Chapter 4.

3.4 Experimental Setup

This section represents all environmental requirements for installing Sniper multicore simulator, describes its features and validation method.

3.4.1 Host Machine

We do all simulations with Sniper multicore simulator on HP ProBook 4530s laptop which features Core i3-2310M multicore processor, 64-bit operating system, 4 GB RAM, and 320 GB HDD.

3.4.2 Operating System, Compiler, and Libraries

We have installed Sniper multicore simulator on Linux 2 Ubuntu 14.04.3 LTS 64 bit. Sniper needs a special environment to successfully works, like the GNU Compiler

Collection (GCC 4.8.2), Python library 2.7.6, Pin tool 2.14-71313 (Dynamic Binary Instrumentation Tool), Perl, Perl base, and Perl modules version 5.14.2-21.

After setting the environment and libraries, we have installed the latest version of Sniper multicore simulator 6.1 (more details in Section 3.5). Then, we installed and built the benchmarks. Sniper is compatible with SPLASH2 and PARSEC benchmark suites, which are described in the next section.

3.5 Sniper Multicore Simulator

By reviewing the literature searching for an efficient multicore simulator, we choose Sniper multicore simulator for its valuable features. In this section, we will explain how it works and what its features are. Sniper is an execution-driven simulator that uses functional-first simulation with timing feedback based on the Pin dynamic instrumentation framework and the Graphite simulation infrastructure (Miller et al. 2010). It implements parallel simulation by keeping threads synchronized using a quantum-based barrier synchronization with a quantum of 100ns. Each thread in the benchmark application is pinned to its own simulated core. Sniper is a user-space simulator, hence, it does not model the operating system nor a scheduler, although emulation of some aspects that impact performance, such as system call overhead, have been added.

Sniper is designed for fast and accurate simulation and makes a good tradeoff between accuracy and speed. It is validated against multicore two-socket Intel Core 2 processor with an average error of 25% against the real hardware (Carlson et al., 2011), at simulation speed of several million instructions per second (MIPS). It was enhanced by introducing the instruction window-centric (IWC) core model which is used and

validated on Nehalem processors against real hardware (Carlson et al., 2014a). It shows good accuracy with an average single-core error of 11.1% and a maximum of 18.8% for the IW-centric model with only 1.5 slowdown factor and is more accurate compared to interval simulation. Eyerman et al. (2009) proposed the interval core simulation.

One of the key features of Sniper simulator is its utility in unicore and system-level studies because it gives more details than the typical one-IPC models. However, this happens by “jumping” between miss events called intervals. Miss events include branch misprediction and cache misses. So, there is an added benefit for the interval core model.

Sniper can generate CPI stacks that show the number of cycles lost due to different characteristics of the system, like the cache hierarchy or branch predictor or interconnection network. Therefore, Sniper offers a better understanding of each component’s effect on total system performance. By so, we could use it to characterize applications on different designs (Al-Manasia et al., 2015).

Modern multicore and many-core designs show high numbers of core counts and more cache hierarchies’ complexity. Instruction window-centric (IWC) core model was introduced to support these new configurations and can simulate new many-core designs. Cycle accurate simulation can give more accurate results, but also tends to be slow. This model limits the number of configurations that can be evaluated. Hence, resulting in a large simulation bottleneck. Interval simulation provides a middle ground that is needed for fast simulation of complex many-core processors while still providing accurate results (Carlson et al., 2014a). IW- centric take new core features in details making it easy to implement dispatch stage and reorder buffer (ROB) in the out of order (OoO) core model, in addition to the previous interval configurations. Sniper supports a wide range of flexible simulation options for exploring different homogeneous and heterogeneous

multi-core architectures, different types of workloads like multi-threading and multi-program workloads, and support parallel applications like OpenMP, etc. Sniper runs SPLASH2, PARSEC, Rodinia, and SPEC OMP. It is compatible with modern Linux OS (Redhat EL 5,6/Debian Lenny+/Ubuntu 10.04-15.04+/etc.) and supports DVFS scaling and integrate with MCPAT for generating a Power and energy results. Sniper uses barrier synchronization with a 100ns quantum to minimize simulation error by decreasing synchronization periods (Carlson, et al., 2014a).

Sniper uses the timing model feedback instrumented by the Pin dynamic instrumentation framework (Luk et al., 2005), which is available on Linux and Windows. It is just in time (JIT)-based dynamic instrumentation tool. It instruments single and multiple threaded applications and it supports different types of processors including Intel's instruction set IA-32bit, IA-64 bit (Intel, 2017).

One of the Sniper key features is its integration with MCPAT; an integrated power, area, and timing modeling framework for multicore architectures. It is an analytical modeling framework that gives an estimation of power and area consumption, like CPI stacks. Sniper has high-quality visualization power results plotted over time, leading to better understanding of individual runs (Ahn et al., 2013), (Ahn et al., 2009).

Sniper implements snooping coherency between caches on a socket (or a tile in the NoC based configuration) and directory-based coherency across sockets/tiles. In addition, it supports MSI/MESI/MESIF protocols (which applies to both snooping and directory-based protocols). The MESIF protocol is always used across the socket/tile because the LLC is always inclusive in Sniper so it provides the data and the Forward state is not needed. Sniper supports different core interconnection network, multi-socket bus-based and NoC architectures. The two types of networks will be explained in Chapter four.

3.6 Benchmarks

Almost all the new multicore processors have abilities to run applications using a high number of threads in parallel by the multithreading features. Thus, it is very important to choose a representative multithreaded workload when evaluating multithreaded processor designs. Applications should cover the compute intensive and memory intensive applications and belong to the well-scaling benchmarks and poorly scaling benchmarks. However, after studying all the available workload types in current practice in computer architecture research and development, we chose eight multithreaded applications belonging to the two representative multicore benchmark suites; SPLASH2 and PARSEC (M. Sultan and G. Abandah, 2015). To make the analysis meaningful, we use two input sets (small and large data sets). Benchmarks in general are executed with eight threads on our eight core processors. Each thread pinned to a core. We run each benchmark to completion and report many performance metrics like total execution time, average IPC per core, processor utilization, and energy consumption. Simulation speed for all benchmarks in our research is around 2 MIPS, which allows us to complete the simulation of a typical benchmark used in this study in around 1 to 3 hours on a modern dual core (i3) host machine.

3.6.1 SPLASH2 Benchmark Suite

Radix is a sorting algorithm that carries out one iteration on radix *r* digits of the keys, which are a series of integers. In each iteration, a processor sorts its assigned keys and creates a local histogram. After that, the local histograms are accumulated into a global histogram. Finally, each processor uses the global histogram to permute its keys into a new array for the next iteration. Radix does not have floating point operations. It is integer kernel application (Mohammad and Abandah, 2015) (Bienia, e al. 2008).

LU is a kernel benchmark that decomposes a dense square matrix into the product of a lower triangular and an upper triangular matrix. The n×n matrix is divided into a N×N array of B×B blocks, where n=NB. The blocks are divided among the processors and each processor updates its blocks. To reduce communication, a2-D scatter decomposition is used to assign blocks to processors (Mohammad and Abandah, 2015) (Bienia, e al. 2008).

FFT is a one-dimensional kernel of the radix-6 steps Fast Fourier Transform (FFT) algorithm that is optimized to minimize inter-processor communication. The dataset is organized as a number of $\sqrt{n} \times \sqrt{n}$ matrices, which are distributed, in a neighboring set of rows, on the processors and assigned to each processor's local memory. The all-to-all inter-processor communication occurs in three matrix transpose steps. Each processor transposes a neighboring sub-matrix of $\sqrt{n}/p \times \sqrt{n}/p$ from each other processor. To avoid high contention, each processor starts by transposing a submatrix from the next processor (Mohammad and Abandah, 2015) (Bienia, e al. 2008).

Cholesky is a kernel benchmark that decomposes a sparse matrix into the product of a lower triangular matrix and an upper triangular matrix by using blocked Cholesky decomposition. As LU, it divides a sparse matrix into blocks that are divided among the processors and each processor updates its blocks (Mohammad and Abandah, 2015) (Bienia, e al. 2008).

3.6.2 PARSEC Benchmark Suite

Canneal is a kernel benchmark that uses a cache-aware Simulated Annealing (SA) algorithm to minimize routing cost of a chip design. The SA algorithm is a generic probabilistic metaheuristic for locating a good approximation to the global minimum of a given function in a large search space.

Canneal simulates putting elements on a chip with minimum routing cost. Like Radix benchmark, Canneal is an integer kernel application (Mohammad and Abandah, 2015) (Bienia, e al. 2008).

Blackscholes is an Intel recognition, data mining, and synthesis (RMS) application. It calculates the prices for a portfolio of European options analytically by using the Black-Scholes partial differential equation solution. It partitions the portfolio work among the threads and processes them simultaneously (Mohammad and Abandah, 2015) (Bienia, e al. 2008).

Fluidanimate is an Intel RMS application that uses an extension of the smoothed particle hydrodynamics approach to simulate an incompressible fluid for interactive animation purposes. Fluidanimate partitions the work among the threads and each thread handles its portion and interacts with the other threads to handle shared work (Mohammad and Abandah, 2015) (Bienia, e al. 2008).

Swaptions is an Intel RMS application that uses the Heath Jarrow Morton (HJM) framework to price a portfolio of swaptions. The HJM framework describes how interest rates evolve for risk management and asset liability management for a class of models. Its central insight is that there is an explicit relationship between the drift and volatility parameters of the forward-rate dynamics in a no-arbitrage market. Swaptions uses Monte Carlo simulation to compute the prices (Mohammad and Abandah, 2015) (Bienia, e al. 2008).

We use two input sets for each benchmark application, small input set and large input set to make the study meaningful. Table 3.2 shows the input sets for all applications.

Table 3.2 Two input sets for thesis studied applications.

Benchmark application	Small input size	Large input size
SPLASH2-FFT	64 K points	1 M points
SPLASH2-Radix	256 K integers	2 M integers
SPLASH2-Lu.cont	256 x 256	512 x 512
SPLASH2-Cholesky	tk15.0 file	Tk29.0 file
PARSEC-Canneal	100.000 elements, 32 temperature steps	200.000 elements, 32 temperature steps
PARSEC-Blackscholes	4 K options	16 K options
PARSEC-Fluidanimate	5 frames, 35 K particles	5 frames, 100 K particles
PARSEC-Swaptions	16 swaptions, 10,000 simulations	32 swaptions, 20.000 simulations

3.7 Performance Evaluation Metrics

Choosing suitable multicore performance metrics helps in evaluating design alternatives in system software and architecture in the multicore era. Performance metrics are classified into two sets: user-oriented metrics like the response time (simulation total execution time) that indicates how long it takes to do a task, and system-oriented metrics like throughput that focuses on the total work done per unit of time. Here, we use average instructions per cycle (IPC) the inverse of cycles per instruction (CPI). We use the visualization CPI stacks to interpret the change of CPI characterization over time, and we use the utilization metric (U %), mainly, the total processor utilization.

To take the power consumption into account, we use average runtime power consumed from the MCPAT tool integration. Therefore, designers can make tradeoffs and optimize performance within an allocated power budget.

The performance of multicore design alternatives is evaluated in two techniques; a raw comparison which compares original processors features without any modifications, and normalized comparison (Abandah et al., 1998) which normalizes the technology-related features and focuses on three main parameters in multicore design: memory hierarchy, interconnection network, and cache coherence protocol.

The normalized comparison better exposes the performance differences due to microarchitecture main features rather than the underlying technology and component sizes.

3.8 Validation

The Sniper paper (Carlson, et al. 2011) and its validation journal version (Carlson, et al. 2014a) validated Sniper multicore simulator on Intel Core 2 and Nehalem real hardware, respectively. For validation issue, we have validated Sniper multicore simulator version 6.1 by reproducing their validation work (Carlson, et al. 2014a) on Intel Xeon X5550 (Nehalem codename). See Table 3.3 for the validated Nehalem core configuration. We reproduce 39 simulations of 13 applications from SPLASH2 benchmark suite (see Table 3.4 for SPLASH2 benchmark applications and its input sets).

Table 3.3 Validated Nehalem core configuration.

Component	Configuration
Processor	1 and 2 sockets, 4 cores per socket
Core	2.66GHz, 4-way dispatch, 128-entry ROB
Branch predictor	8 cycles penalty
L1-I	32KB, 4 way, 4 cycle access time
L1-D	32KB, 8 way, 4 cycle access time
L2 cache	256KB per core, 8 way, 8 cycle
L3 cache	8MB per 4 cores, 16 way, 30 cycle
Main memory	QPI, 12.8GB/s per direction

Carlson, et al. (2014a) used 3 types of core models: IW-Centric that supports reorder buffer ROB, Interval for in order execution, and One-IPC on a single core and large input size of SPLASH2 suite. They compare simulation results with their collected results from running the same applications on real hardware. We reproduced the same simulations' results by taking the SPLASH2 applications and simulate them using the three core models. By so, we got the 39 results shown in Table 3.5.

We tried to match exact configurations, but we keep in mind that any absolute numbers we found in the validation paper are for a specific version of Sniper, with very specific application binaries and command lines, etc. Also, Nehalem core model in the current version of Sniper has some improvements to suite the recent processors like instruction extensions, and branch prediction techniques.

Table 3.4 Validated benchmarks and input sets.

SPLASH2 Benchmarks	Input Set
Barnes	32,768 particle
Cholesky	tk29.O
FFT	4M points
Fmm	32,768 particles
Lu.cont	1,024×1,024 matrix
Lu.ncont	1,024×1,024 matrix
Ocean.cont	1,026×1,026 ocean
Ocean.ncont	1,026×1,026 ocean
Radiosity	-room
Radix	1M integers
Raytrace	car –m64 –a4
Raytrace_opt	car –m64 –a4
Water.nsq	2,197 molecules
Water.sp	2,197 molecules

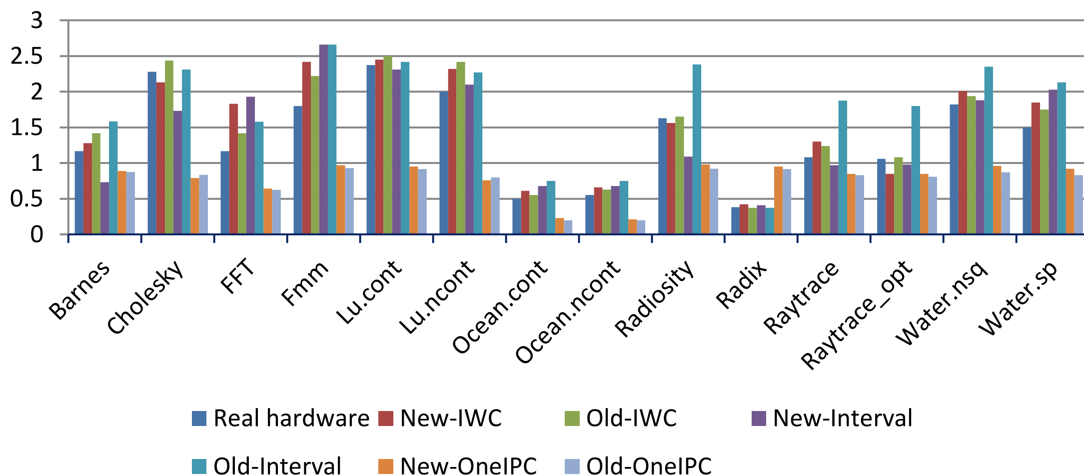


Figure 3.5 Validation of Sniper multicore simulator, the first column for real hardware. Each two consecutive columns are from the new and old results from the same core model.

Table 3.5 IPC results of our validation and the results of Sniper1 validation paper 2014.

SPLASH2 Applications	**IWC results**		**Interval results**		**IPC results**		**Real hardware results**
	Old	**New**	**Old**	**New**	**Old**	**New**	
Barnes	1.42	1.28	1.58	0.73	0.88	0.89	1.166
Cholesky	2.44	2.13	2.31	1.73	0.83	0.79	2.281
FFT	1.416	1.83	1.58	1.93	0.63	0.64	1.166
Fmm	2.22	2.42	2.66	2.66	0.93	0.97	1.80
Lu.cont	2.5	2.45	2.42	2.31	0.92	0.95	2.375
Lu.ncont	2.42	2.32	2.27	2.10	0.80	0.76	2.00
Ocean.cont	0.55	0.61	0.75	0.68	0.20	0.23	0.5
Ocean.ncont	0.63	0.66	0.75	0.68	0.20	0.21	0.55
Radiosity	1.65	1.56	2.38	1.09	0.92	0.98	1.63
Radix	0.38	0.42	0.38	0.41	0.92	0.95	0.38
Raytrace	1.24	1.30	1.88	0.97	0.83	0.85	1.08
Raytrace_opt	1.08	0.85	1.80	0.98	0.81	0.85	1.06
Water.nsq	1.94	2.01	2.35	1.88	0.87	0.96	1.82
Water.sp	1.75	1.85	2.13	2.03	0.83	0.92	1.50

The comparison between the old and new IPC results in Figures 3.5 showed the similarity results view. In addition, that the results of recent simulations tend to be more accurate than the old results relative to hardware results in most of the simulations. Hence, the results agree in general with negligible differences and the same results relations.

CHAPTER 4: RAW AND NORMALIZED EVALUATIONS

4.1 Introduction

In this chapter, we present the results of our comparative study of the four commodity multicore processors. The purpose of this study is to evaluate alternative multicore and many core designs, and identify system strengths and bottlenecks in current processors. Also, we determine design aspects that have high positive impact on such processors, we identify areas that need further investigation and improvement.

We used the Sniper simulator to evaluate the four common Intel's server multicore processors (Xeon brand). Recall that two of them are bus based multi-socket architectures (Dunnington/Core2 based and Gainestown/Nehalem based microarchitectures). The others are NoC based architectures (2d mesh and bidirectional ring interconnection network). They are state of the art multicore and many core processors (Haswell and Xeon Phi based processors). Although the four multicore processors share many similarities, they have significant differences that lead to large performance differences.

For fair comparison, we used eight multi-threaded benchmark applications with small and large input sets. Four of these applications are from SPLASH benchmark suite, which are Radix, FFT, LU, and Cholesky. The other four are from PARSEC benchmark suite, which are Canneal, Blackscholes, Fluidanimate, and Swaptions. The eight benchmarks have interesting differences (Mohammad and Abandah 2016).

We used total execution time (Ex. time) in milliseconds, instructions per cycle (IPC), and cycles per instruction (CPI) stack changes over time. In addition to the average thread utilization, we used average dynamic run time power (P) in Watts.

The next section presents the raw comparison, and Section 4.3 presents the normalized comparison.

1. https://www.slideshare.net/IntelSoftwareBR/numa-i-step2014
2. http://slideplayer.com/slide/7090758/

4.2 Raw Comparison

This section presents the raw comparison where we use Sniper configuration files that select components of the same size and speed as those used in the four case-study processors. The four processors design alternatives are explained in Chapter 3. Figures 4.1, 4.2, 4.3, and 4.4 show the four case-study multicore design alternatives as plotted by Sniper multicore simulator. These figures show the multi-core processors networks, memory hierarchy organizations that contain number of caches and their sizes, and number of memory controllers.

Parameterizing the configuration files of the Dunnington and Gainestown were easy because there are detailed publications (Carlson, et al. 2011), (Carlson, et al. 2014a), and (Rolf, Trent. 2009). Also, parameterizing the Xeon Phi configuration file was easy because it is supported as an open source example from the Sniper multicore simulator and also there are detailed publications, e.g., (Rahman, R. 2013). The last Haswell configuration is the hardest design to collect all of its detailed features. We used few specifications that are provided from Intel home page, and some related publications, e. g., (Molka, et al. (2015).

1. https://www.slideshare.net/IntelSoftwareBR/numa-i-step2014
2. http://slideplayer.com/slide/7090758/

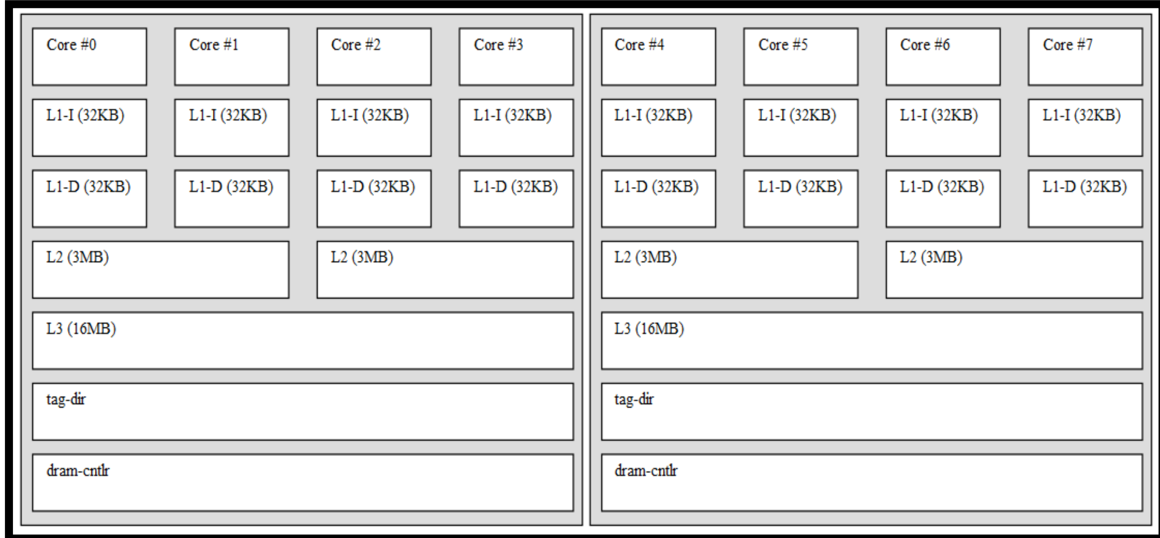


Figure 4.1 Multi-socket Dunnington based microarchitecture (Sniper simulator output).

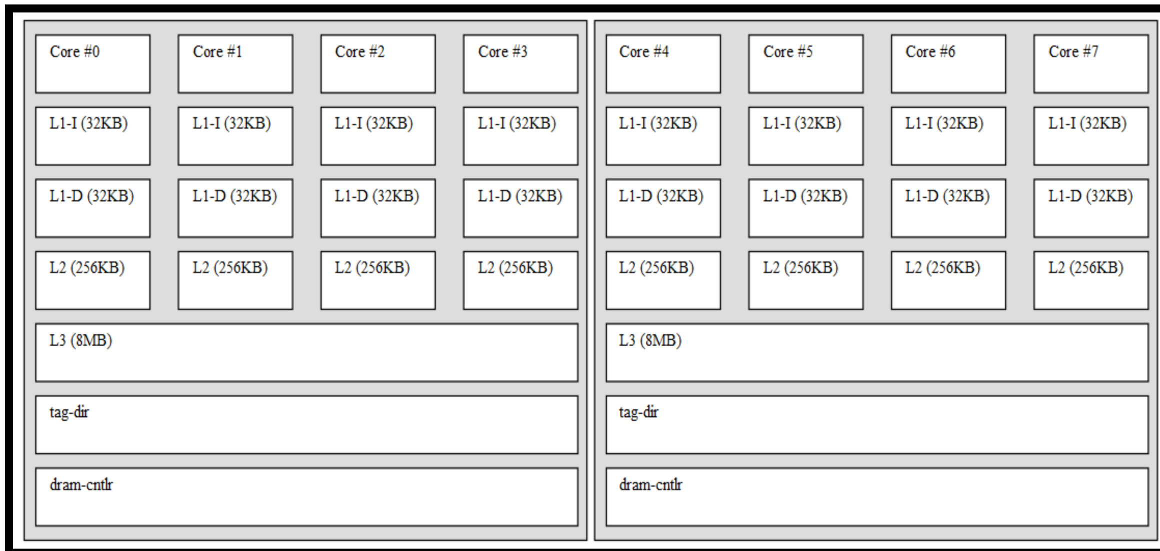


Figure 4.2 Multi-socket Gainestown based microarchitecture (Sniper simulator output).

1. https://www.slideshare.net/IntelSoftwareBR/numa-i-step2014
2. http://slideplayer.com/slide/7090758/

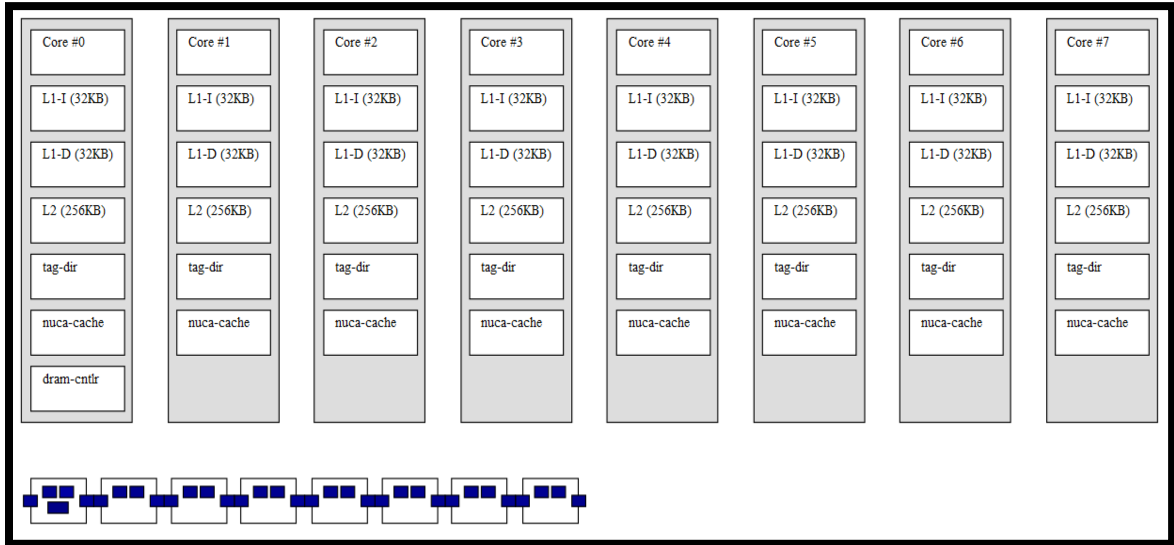


Figure 4.3 Haswell based microarchitecture (Sniper simulator output).

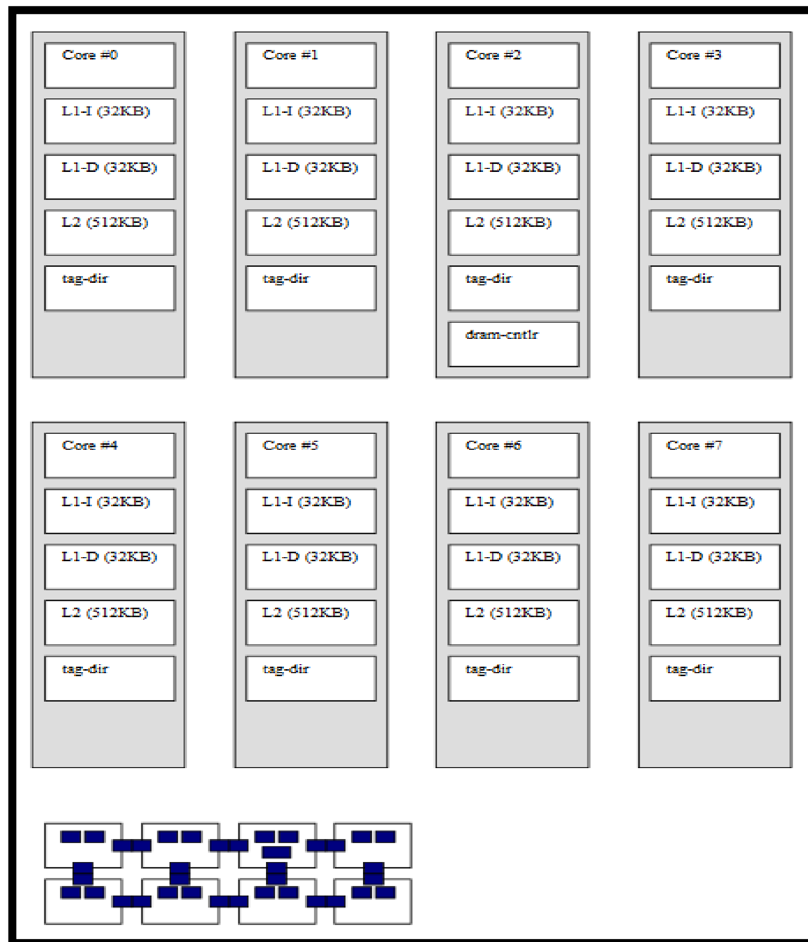


Figure 4.4 Xeon Phi based microarchitecture (Sniper simulator output).

1. https://www.slideshare.net/IntelSoftwareBR/numa-i-step2014
2. http://slideplayer.com/slide/7090758/

Some of the factors that affect multicore performance are:

- Number, type and size of cache levels.
- Core interconnection network and network link bandwidth.
- Number of memory controllers, memory type, memory link bandwidth and speed, and number of memory channels per memory controller.
- Technology node.
- Branch predictor type and penalty.
- TLB size and associativity.
- Processor base frequencies.

The overall processor performance has a positive relationship with processor clock frequency. The following equation is the CPU performance equation.

$$\frac{Seconds}{Program} = \frac{Instructions}{Program} \times \frac{Clocks}{Instructions} \times \frac{Seconds}{Clock}$$

The performance equation describes the three main factors of any processor performance: which are, in order, instruction count (IC), clocks per instruction (CPI), and clock time (CT). Processor frequency is the reciprocal of clock time. Hence, higher processor frequency generally gives lower execution time. Therefore, higher performance.

The network in multicore processors is conceptually between the different LLC and DRAM. We can configure processor network in two ways: either as a multi-socket system where each socket has a shared LLC and locally connected DRAM. In this case, the network models the QPI interface. This mode is used in the Gainestown configuration which models an Intel Nehalem-like system. Figure 4.5 shows how local and remote communications happen over the two QPI links. Although Gainestown and Dunnington use buses to connect sockets together, but Dunnington uses legacy FSB.

1. https://www.slideshare.net/IntelSoftwareBR/numa-i-step2014
2. http://slideplayer.com/slide/7090758/

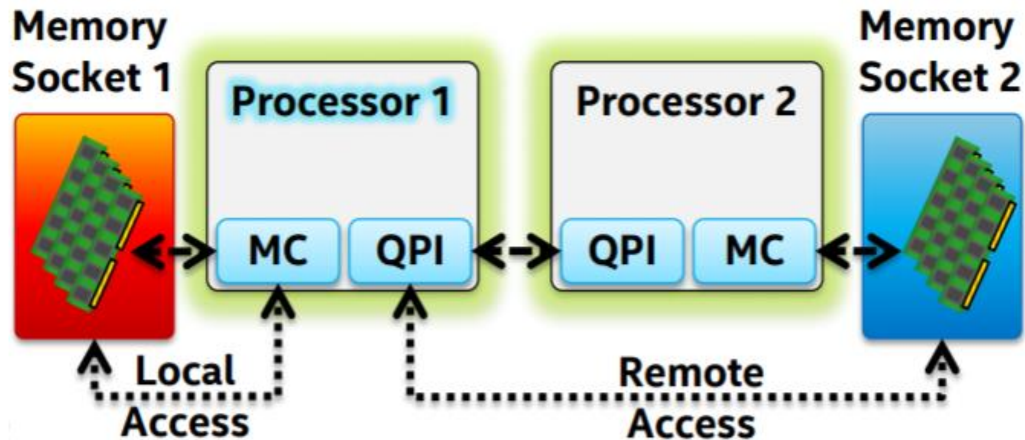


Figure 4.5 Local and remote communications in multi-socket Gainestown microarchitecture[1].

The second network type is where we have a single processor chip with multiple LLCs connected through an NoC, such as an Intel Xeon Phi Knights Corner implementing 2D mesh NoC with L2 as LLC. The second example is Haswell architecture that implements bi-directional ring NoC over L3 NUCA cache slices. Figure 4.6 shows the two NoC topologies, 2d mesh and ring network. The squares in graphs represent processing elements (PE) or cores and the black circles represent routers that are responsible of routing packets between cores.

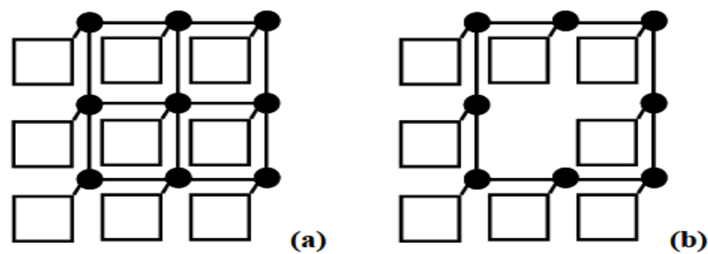


Figure 4.6 The two common NoC; a) 2D mesh topology, b) ring topology[2].

1. https://www.slideshare.net/IntelSoftwareBR/numa-i-step2014
2. http://slideplayer.com/slide/7090758/

4.2.1 Total Execution Time

This subsection presents and analyzes the user perspective performance metric and the first order performance insight; total execution time (Ex. time) in milliseconds. Figure 4.7 and Figure 4.8 show the total execution time for running the eight multithreaded applications on the four multicore alternatives with the small and large input data sets (Table 3.2).

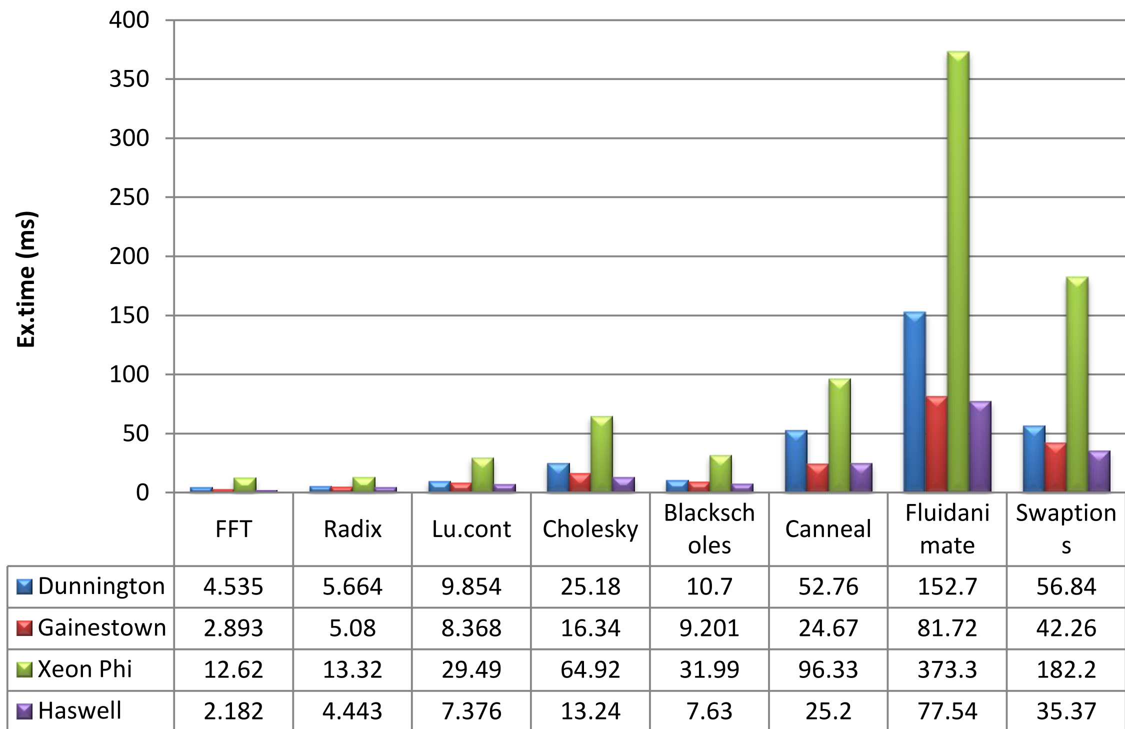
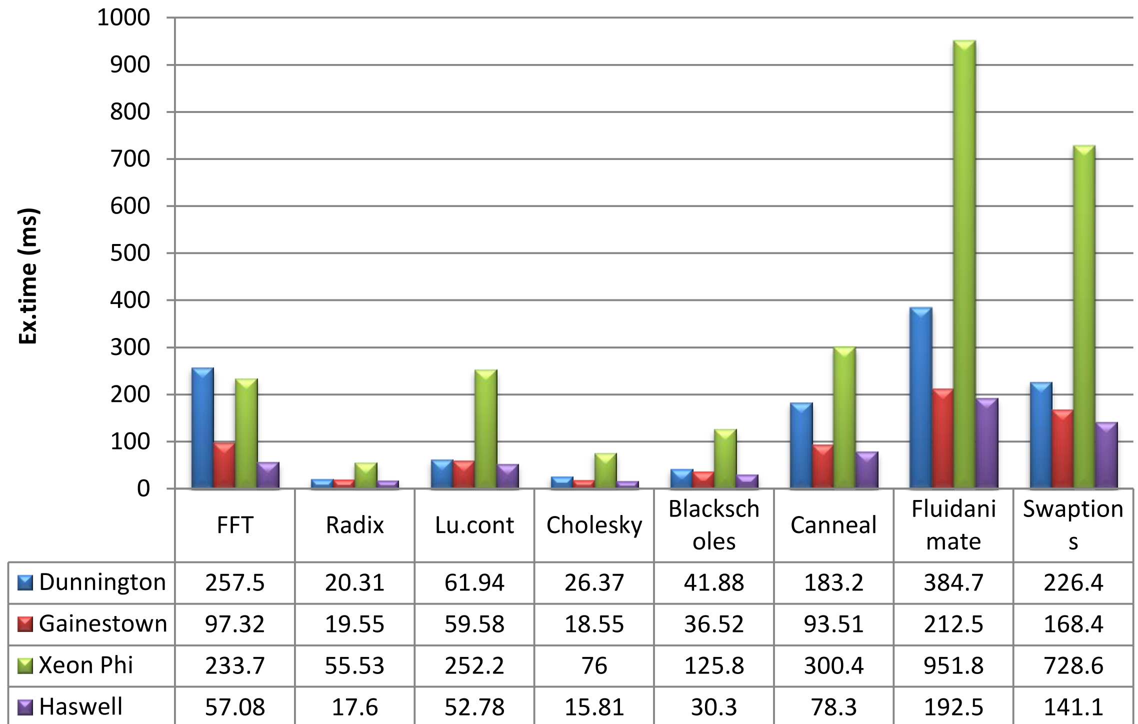


Figure 4.7 Execution times for running eight benchmark applications with small input size over the four multicore design alternatives.



	FFT	Radix	Lu.cont	Cholesky	Blackscholes	Canneal	Fluidanimate	Swaptions
Dunnington	257.5	20.31	61.94	26.37	41.88	183.2	384.7	226.4
Gainestown	97.32	19.55	59.58	18.55	36.52	93.51	212.5	168.4
Xeon Phi	233.7	55.53	252.2	76	125.8	300.4	951.8	728.6
Haswell	57.08	17.6	52.78	15.81	30.3	78.3	192.5	141.1

Figure 4.8 Execution times for running eight benchmark applications with large input size over the four multicore design alternatives.

We see big differences in performance amongst all multicore alternatives. Also, we see how benchmark applications execution times depend on the workload data set. There is positive relationship between the application input set size and the application execution times. Haswell design shows high performance in all multithreaded applications, due to its high component speeds and sizes. It’s the most recent technology relative to the other systems. Haswell has the highest clock rate (3.2 GHz) and highest bus speed (9.6 GT/s). The bus based systems, Gainestown and Dunnington have the same frequency (2.66 GHz), but Gainestown has higher bus speed than Dunnington. Xeon Phi has the minimal processor frequency (1.05 GHz), but its communication speed is higher than the other two bus based systems.

In fact, the data transfer rates and memory bandwidth are highly dependable on the memory type. Figure 4.9 shows common memory types and their corresponding data rates. Furthermore, each multicore microarchitecture supports several types of DRAM memory and has different number of channels. So, the max memory bandwidth varies depending on these two factors. Dunnington has three channels of double data rate memory (DDR2-533 MHz) for each socket, Gainestown has four channels of DDR3 for each socket, Haswell has four channels of DDR4, and Xeon Phi has two channels of the new graphic memory (GDDR5) for one memory controller. GDDR5 memory type supports high memory bandwidth and is suitable for graphics and HPC. Also, the lack of TLB buffer makes it less efficient in terms of throughput and latency in Dunnington and Xeon Phi microarchitectures. The TLB improves overall CMP performance (Lustig, et al. 2013) (Mittal, S. 2016).

DDR data transfer rate:
DDR 266 : 2.1 GB/s
DDR 333 : 2.6 GB/s
DDR 400 : 3.2 GB/s

DDR2 data transfer rate:
DDR2 533 : 4.2 GB/s
DDR2 667 : 5.3 GB/s
DDR2 800 : 6.4 GB/s

DDR3 data transfer rate:
DDR3 1066 : 8.5 GB/s
DDR3 1333 : 10.6 GB/s
DDR3 1600 : 12.8 G MB/s
DDR3 1866 : 14.9 G MB/s

DDR4 data transfer rate:
DDR4 2133 : 17 GB/s
DDR4 2400 : 19.2 GB/s
DDR4 2666 : 21.3 GB/s
DDR4 3200 : 25.6 GB/s

Figure 4.9 Various memory types and their corresponding data rates[1].

It's clear that Xeon Phi has the least performance. Although it is designed for HPC and low memory latency, but the target performance of its design is aggregated over many core architecture (60+ cores). Its low frequency, small L1 and L2 cache sizes, and the lack of L3 cache are main reasons behind this low performance.

The two medium performance values are for the two bus-based systems. In fact, in all multithreaded applications, Gainestown shows better performance than Dunnington.

1. https://en.wikipedia.org/wiki/DDR_SDRAM

One important reason is the high speed quick path interconnect (QPI) instead of Dunningtons FSB. Although that Dunnington has larger cache L2 and lower miss rates as shown in Figures 4.10 and 4.11. Gainestown hides these misses better by its high-speed communications using high speed QPI.

There is an exceptional case of the Canneal benchmark application with small input data set. Gainestown has execution time better than Haswell with small difference value. This can be interpreted due to the number of L3 cache sharing cores. In Gainestown there is 8 MB L3 cache shared by four cores for each socket, but Haswell has 8 MB L3 cache shared by eight cores. Hence, with small data set size of Canneal, most of instructions and data fit into L1 and L2 caches and the rest data lies in L3 cache. Thus, most of the work in Gainestown is done locally, there is small ratio of remote memory or DRAM accesses. Haswell has fewer available L3 banks and hence more misses and more DRAM accesses.

4.2.2 L2 Cache Miss Rate

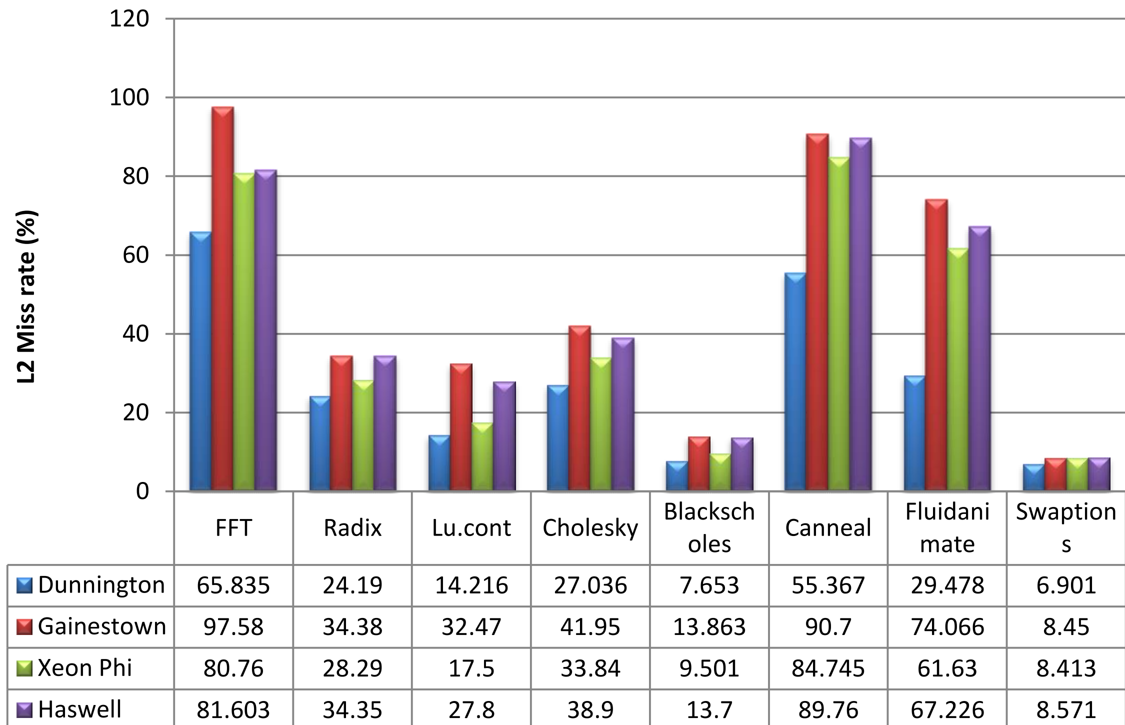
Figure 4.10 and Figure 4.11 show L2 miss rates for the four multicore design alternatives when running the eight benchmark applications with small and large input data sets, respectively. We observe that the size and sharing property are playing a large impact on processor performance.

The best performance design, from L2 miss rate point of view, is Dunnington processor, due to its largest L2 cache size (3072KB), despite it is shared by two cores. Xeon Phi comes in the second rank as it has larger cache size (512 KB private L2) than Haswell and Gainestown (256 KB L2).

Although Haswell has L2 miss rates fewer than Gainestown processor, they have the same L2 cache sizes. We think that the larger ROB size in Haswell leads to more data space

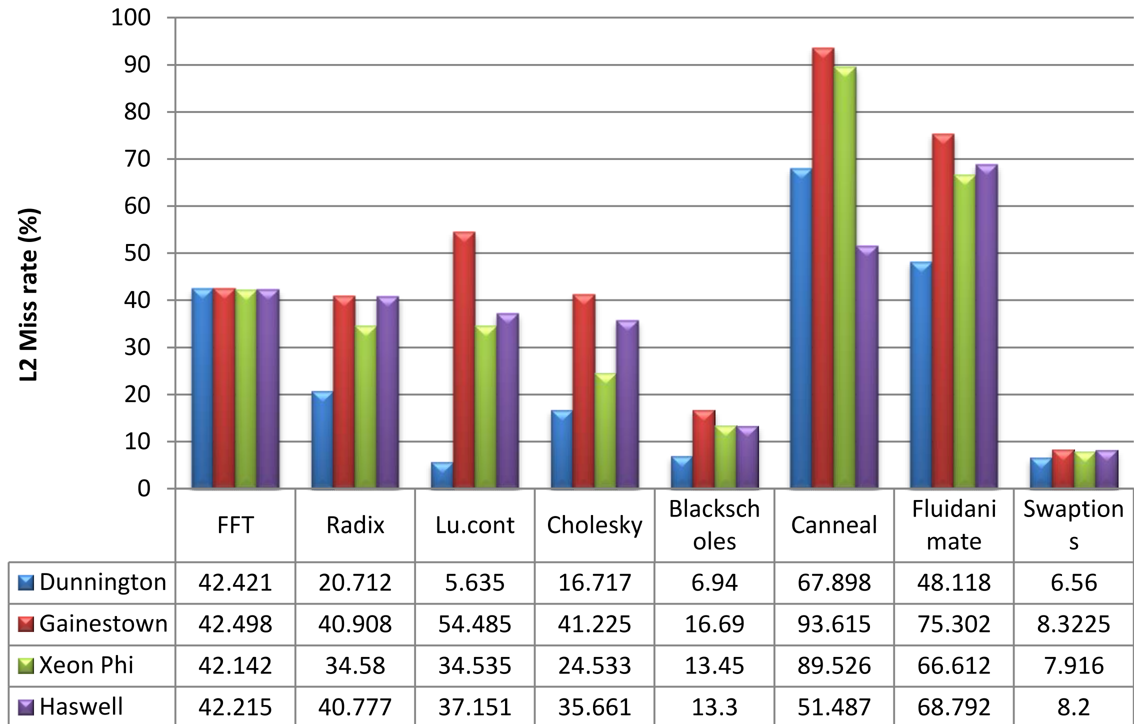
locality hits in the L2 cache. Although the L2 cache misses affects the over all performance, but Haswell and Gainestown hides these misses by their higher clock rates, higher parallelizing features, and higher speed of QPIs.

Most benchmarks behave similiraly, but Swaptions benchmark has low miss rates in the four multicore alternatives with the small and large data sets. Swaptions is a highly memory intensive application and is a data streaming workload where there is a large working set and little data reuse. Sniper simulator gives the number of L2 accesses and the number of total instructions. By so, we can compute the probability of accesses to the L2 cache. In Swaptions benchmark the average probablility of L2 accesses over all cores is very small. As an example, the L2 accesse propability in Haswell design is equall to 0.0039 with large data set.



	FFT	Radix	Lu.cont	Cholesky	Blacksch oles	Canneal	Fluidani mate	Swaption s
Dunnington	65.835	24.19	14.216	27.036	7.653	55.367	29.478	6.901
Gainestown	97.58	34.38	32.47	41.95	13.863	90.7	74.066	8.45
Xeon Phi	80.76	28.29	17.5	33.84	9.501	84.745	61.63	8.413
Haswell	81.603	34.35	27.8	38.9	13.7	89.76	67.226	8.571

Figure 4.10 L2 Miss rates for running eight benchmark applications with small input size over the four multicore design alternatives.



	FFT	Radix	Lu.cont	Cholesky	Blacksch oles	Canneal	Fluidani mate	Swaption s
Dunnington	42.421	20.712	5.635	16.717	6.94	67.898	48.118	6.56
Gainestown	42.498	40.908	54.485	41.225	16.69	93.615	75.302	8.3225
Xeon Phi	42.142	34.58	34.535	24.533	13.45	89.526	66.612	7.916
Haswell	42.215	40.777	37.151	35.661	13.3	51.487	68.792	8.2

Figure 4.11 L2 Miss rates for running eight benchmark applications with large input size over the four multicore design alternatives.

4.3 Normalized Comparison

This section presents the normalized comparison where we use Sniper configuration files that preserve the architectural and network differences, but put the four multicore design alternatives on the same technological level, i.e., same network speeds and same component sizes and speeds. The configuration files configure Sniper to simulate the four derived systems: nDunnington, nGainestown, nXeon Phi, and nHaswell. We select modern component sizes and speeds like those used in Haswell (Table 3.1). The four derived systems, unlike the original Haswell, have high memory bandwidth as Xeon Phi processors. See Figures 4.12, 4.13, 4.14, and 4.15 for further detail about the cache sizes and normalized design features.

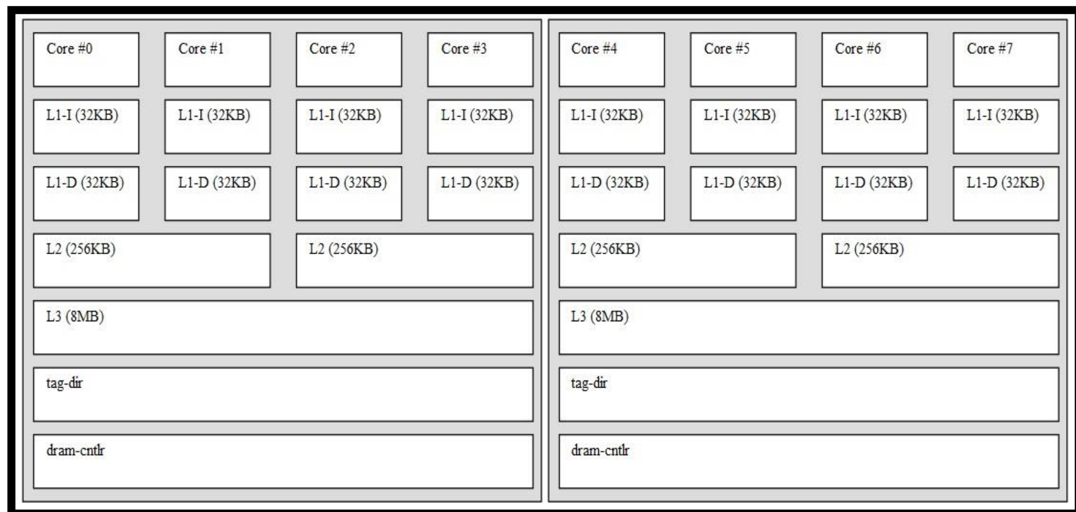


Figure 4.12 Normalized multi-socket Dunnington based microarchitecture (Sniper simulator output).

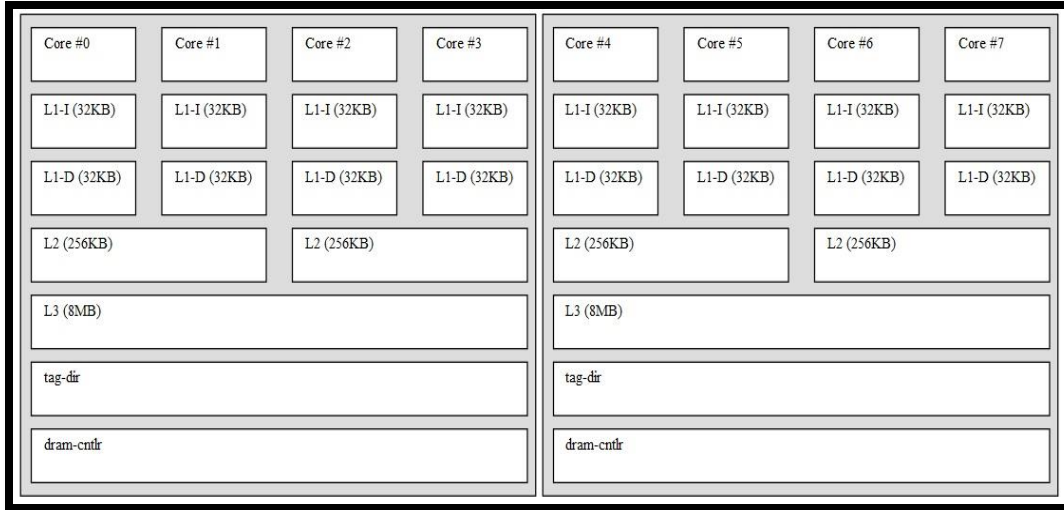


Figure 4.13 Normalized multi-socket Gainestown based microarchitecture (Sniper simulator output).

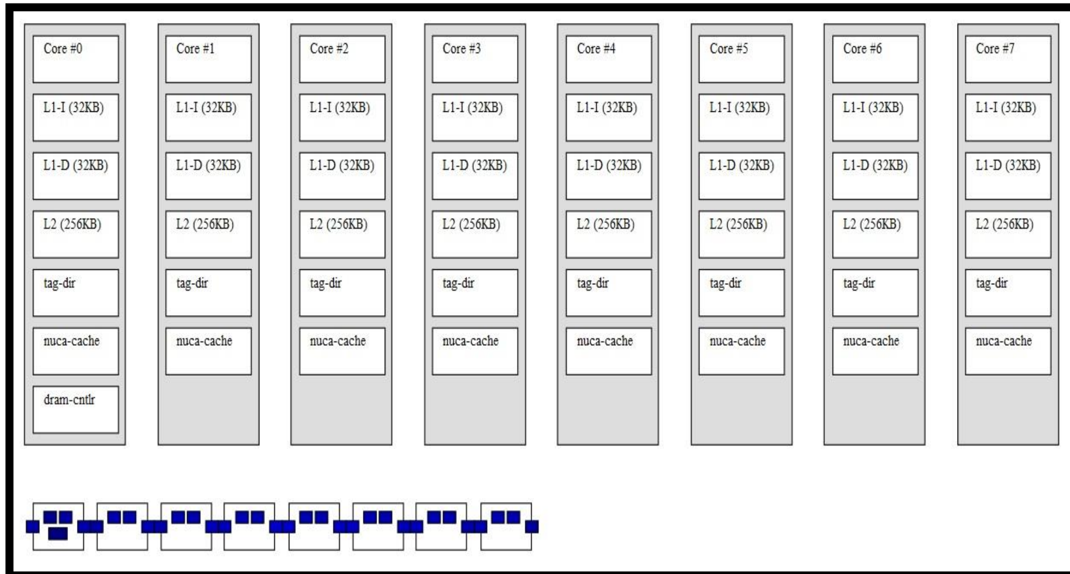


Figure 4.14 Normalized one socket Haswell based microarchitecture (Sniper simulator output).

The network in Figure 4.14 is bidirectional ring topology as plotted by the Sniper simulator. However, the two sides of the network graph are connected via QPI links to perform the ring.

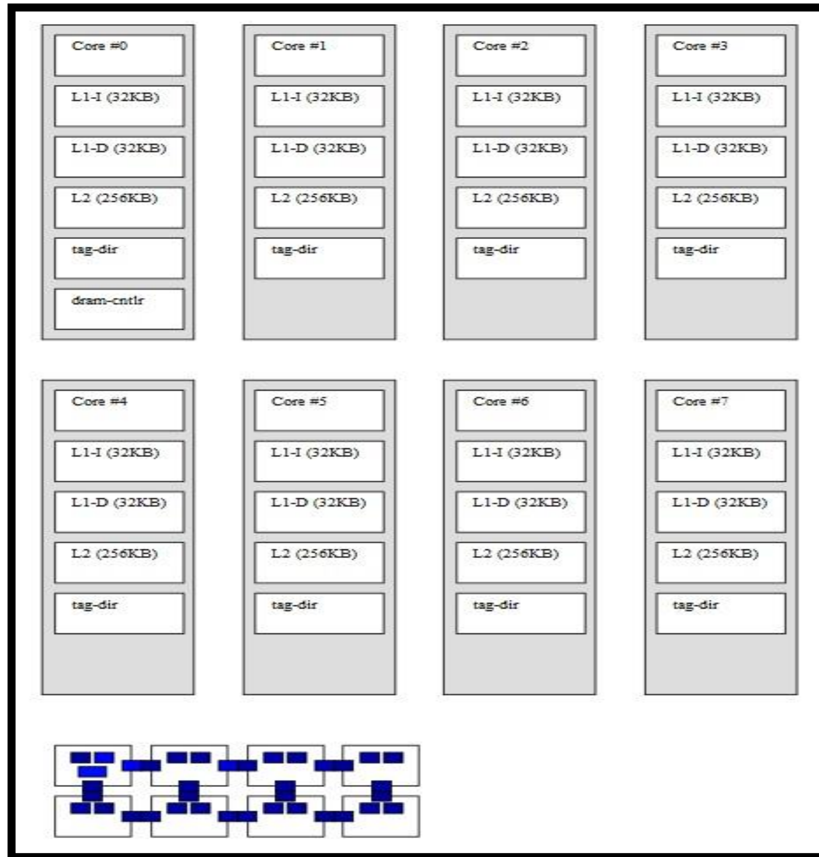


Figure 4.15 Normalized one socket Xeon Phi based microarchitecture (Sniper simulator output).

The following subsections present the performance evaluation results using performance metrics for their normalized comparison with analysis. It is worth mentioning that we used the multithreaded application characterization results concluded by Mohammad and Abandah, (2015) in analyzing the behaviours of the multithreaded applications on these multicore design alternatives.

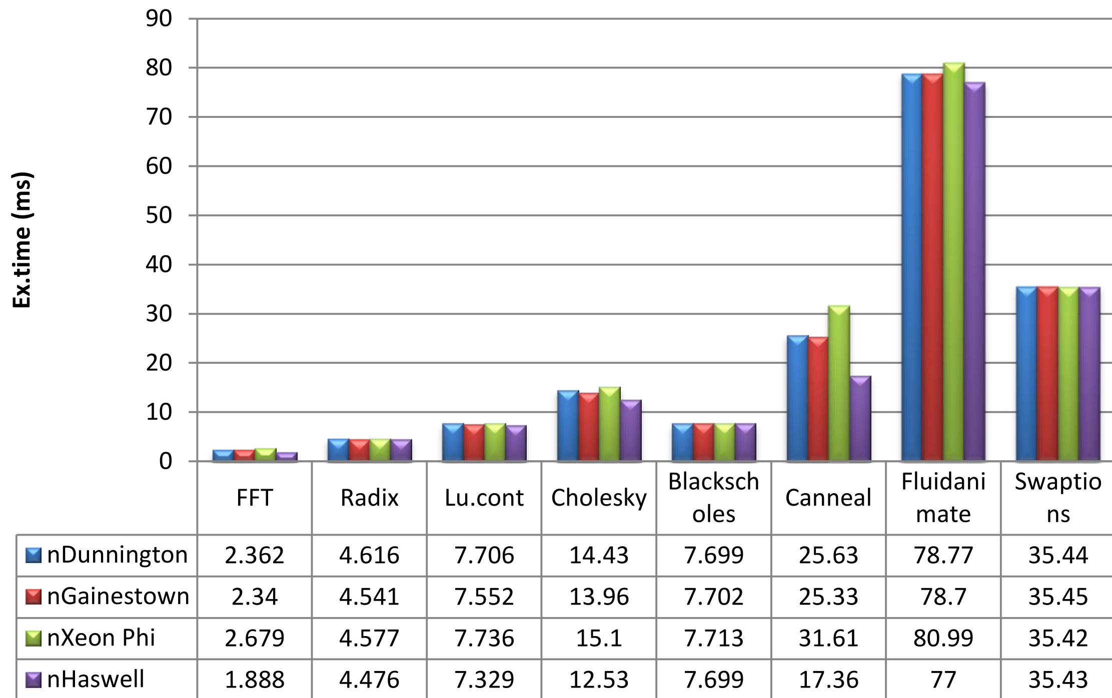
4.3.1 Total execution Time

This subsection presents and analyzes the user perspective performance metric; total execution time in milliseconds. Figure 4.16 and Figure 4.17 show the total execution times for running the eight benchmark applications with small and large input sizes over the four normalized multicore design alternatives, respectively.

Comparing the results shown in Figures 4.16 and 4.17 gives a general view of the overall system performance for the four normalized systems. Another advantage for this comparison is identifying the impact of changing the data set input size on performance for all the applications from SPLASH2 and PARSEC suites. As application problem size is scaled up, the execution time increases.

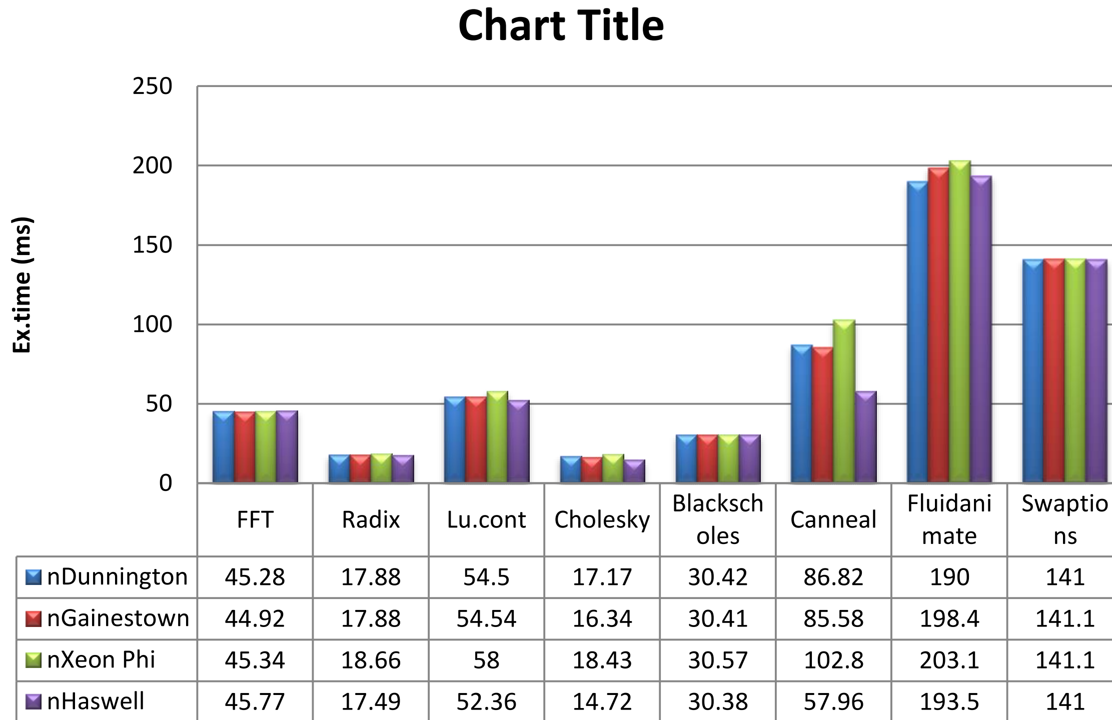
The nHaswell time shown in Figure 4.16 is the best performance for six benchmarks; FFT, Radix, Lu.cont, Cholesky, Canneal, and Fluidanimate. The system of worst performance is the nXeon Phi in most cases because it lacks L3 cache and due to the time spent on the routing protocol. However, the 2D mesh spent is relatively more time in computing the shortest path because of many paths possibilities. The two multi-socket bus based systems (nDunnington and nGaineston) behave very close to each other with minor differences. Further investigation and evaluation for the multicore design alternatives are done through evaluatinging the CPI stack for all multithreaded workloads in Section 4.3.3. Also, we evaluate the multithreaded benchmarks behaviors.

Most multithreaded benchmarks have small differences between execution times of the four multicore designs except Canneal benchmark. It shows big differences in the execution time of simulation over the four designs. However, Canneal is a memory intensive application, it has around 70% of memory contribution on the CPI stack (will be discussed later in the CPI stack section). The mem-DRAM contribution is the reason behind the large memory access latency. Also, the multicore designs have various memory designs, such as different cache levels, private or shared caches, and existence of NUCA cache.



	FFT	Radix	Lu.cont	Cholesky	Blackscholes	Canneal	Fluidanimate	Swaptions
nDunnington	2.362	4.616	7.706	14.43	7.699	25.63	78.77	35.44
nGainestown	2.34	4.541	7.552	13.96	7.702	25.33	78.7	35.45
nXeon Phi	2.679	4.577	7.736	15.1	7.713	31.61	80.99	35.42
nHaswell	1.888	4.476	7.329	12.53	7.699	17.36	77	35.43

Figure 4.16 Execution times for running eight benchmark applications with small input size over the four normalized multicore design alternatives.



	FFT	Radix	Lu.cont	Cholesky	Blackscholes	Canneal	Fluidanimate	Swaptions
nDunnington	45.28	17.88	54.5	17.17	30.42	86.82	190	141
nGainestown	44.92	17.88	54.54	16.34	30.41	85.58	198.4	141.1
nXeon Phi	45.34	18.66	58	18.43	30.57	102.8	203.1	141.1
nHaswell	45.77	17.49	52.36	14.72	30.38	57.96	193.5	141

Figure 4.17 Execution times for running eight benchmark applications with large input size over the four normalized multicore design alternatives.

4.3.2 Average Core IPC

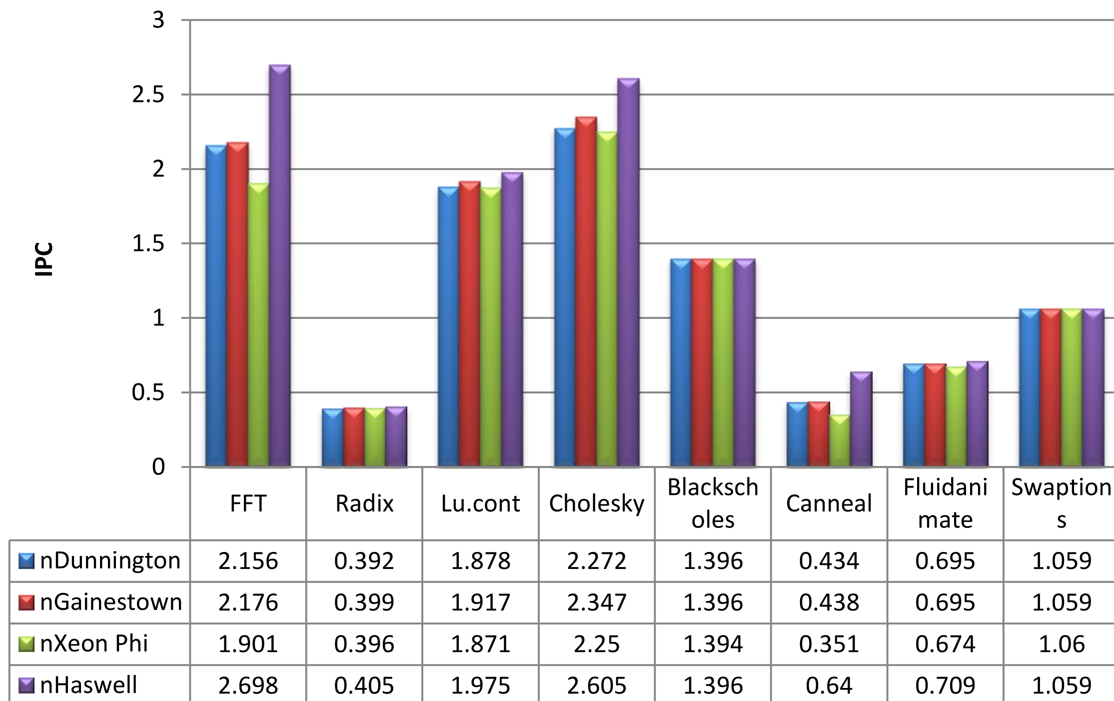
The average number of instructions per clock cycle, or IPC, is a function of the machine and program. The IPC or its reciprocal CPI is a system throughput metric (Eeckhout, et al. 2010). The CPI depends on the actual instructions appearing in the program, for example, a floating-point intensive application might have a higher CPI than an integer-based program. Also. It depends on the processor features. When each instruction takes one cycle, CPI or IPC = 1. The IPC can be <1 due to memory stalls and slow instructions. The IPC can be > 1 on machines that execute more than 1 instruction per cycle (superscalar).

Figures 4.18 and 4.19 show the average IPC for running the eight benchmark applications with the small and large input sizes over the four normalized multicore design alternatives. The normalized Haswell (nHaswell) with its ring topology and NUCA L3 cache shows high throughput IPC over all benchmarks with large and small input sizes. Multi-socket nGainestown with its QPI interconnect comes in the second performance position. The nDunnington has IPC performance less than nGainestown, because of the sharing property of L2 every two cores in nDunnington, where that decreases the L2 cache availability and increases cache misses. The least performance is the nXeonPhi processors. The evaluation of IPC or CPI stacks performance metrics are explained briefly in the next Section.

The two figures show large differencies in IPC among the eight benchmarks, these differencies relate to the high differencies in cache misses in each benchmark. As an example, the average L2 miss rate for Cholesky is about 54% for nDunnington but it is about 7.8% for the same multicore design, and so for the others. Radix benchmark has very close IPC values over multicore design alternatives, high percent of CPI loss due to compute time (close to

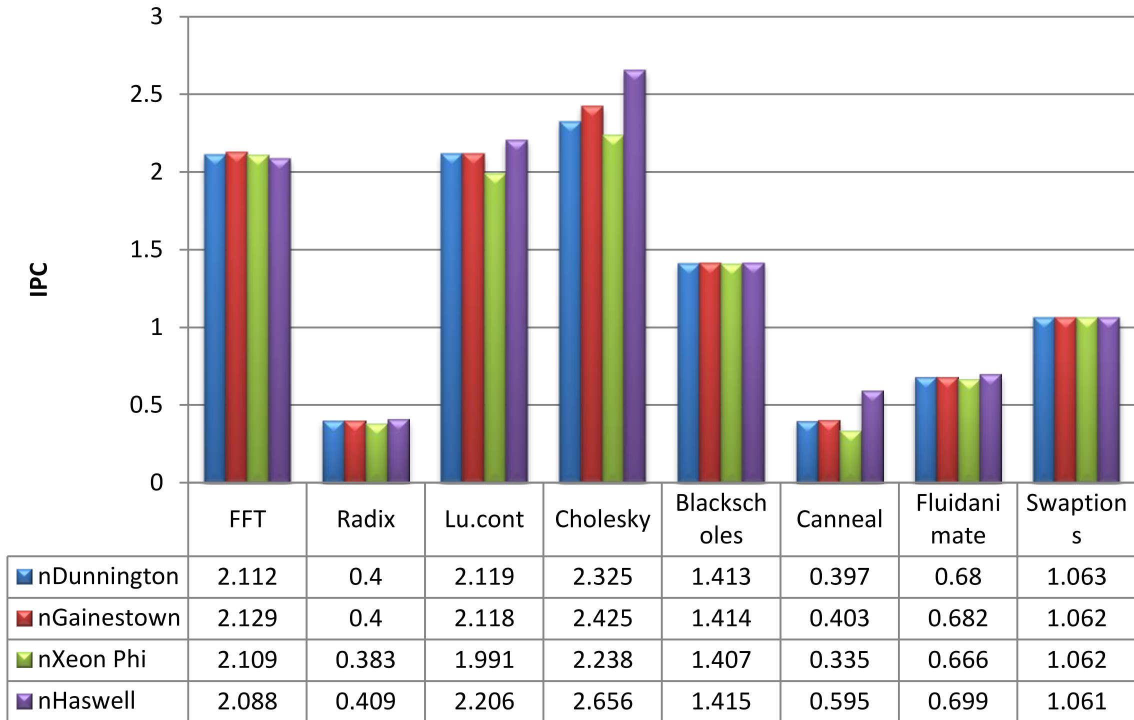
95% CPI), and negligable synchronization contribution (around 0.05%). Mohammad and Abandah, (2015) mentioned that Radix has a small sharing degree where 90% of shared data are shared with only one thread. Because we configure all designs at the same in-core configurations, most of the designs have the same time spent due to computation components.

FFT, Radix, Blackscholes, Fluidanimate, and Swaptions show very close IPC values over the four design alternatives. This happens becuase they are mostly compute intensive applications, like FFT, Radix and Blackscholes. As all designs put at the same computation technological components, the show same time spent to compute. The Fluidanimate and Swaptions have the same behavior even they are highly memory intensive applications because they have large working memory sets but without data reuse or minimum communication slack.



	FFT	Radix	Lu.cont	Cholesky	Blackscholes	Canneal	Fluidanimate	Swaptions
nDunnington	2.156	0.392	1.878	2.272	1.396	0.434	0.695	1.059
nGainestown	2.176	0.399	1.917	2.347	1.396	0.438	0.695	1.059
nXeon Phi	1.901	0.396	1.871	2.25	1.394	0.351	0.674	1.06
nHaswell	2.698	0.405	1.975	2.605	1.396	0.64	0.709	1.059

Figure 4.18 Average core IPC for running the eight benchmark applications with the small input sizes over the four normalized multicore design alternatives.



	FFT	Radix	Lu.cont	Cholesky	Blackscholes	Canneal	Fluidanimate	Swaptions
nDunnington	2.112	0.4	2.119	2.325	1.413	0.397	0.68	1.063
nGainestown	2.129	0.4	2.118	2.425	1.414	0.403	0.682	1.062
nXeon Phi	2.109	0.383	1.991	2.238	1.407	0.335	0.666	1.062
nHaswell	2.088	0.409	2.206	2.656	1.415	0.595	0.699	1.061

Figure 4.19 Average core IPC for running the eight benchmark applications with the large input sizes over the four normalized multicore design alternatives.

4.3.3 CPI Cycle Stack

The CPI stack equals 1/IPC, and depends on performance for one specific architecture. However, the CPI components can provide a breakdown in base components CPI, and gives more insight than looking at IPC alone. Hence, we use CPI stacks to evaluate performance. By cycle stacks we can understand and analyze performance of multi-threaded workloads over different microarchitectures. CPI stacks can quantify where the cycles have gone, and provide more information than raw event rates, such as miss rates of the memory hierarchy and branch predictors. A cycle stack is typically plotted as a stacked bar with the different components showing the relative contribution of each component to overall performance. The main benefit of a CPI cycle stack is that it provides quick insight into the major performance bottlenecks, which hints towards optimization opportunities. This is particularly interesting for analyzing parallel software and hardware performance.

By analyzing how the cycle stacks change with changing the different processor designs, one can understand whether designing bottlenecks come from synchronization overhead, poor performance in the memory hierarchy, load imbalance, etc. Figure 4.20 is an example to show how Sniper can visualize performance CPI stack over time. The shown output is from simulation of Haswell-like design running Canneal benchmark with large data input set. Figure 4.20(a) shows the simple CPI stack (compute, memory, branch, and synchronization contribution on system CPI). It also shows how CPI changes corresponds to the IPC. Figure 4.20(b) shows detailed CPI stack which we used to interpprent the simple CPI stack. The base CPI is typically shown at the bottom of the CPI stack and represents useful workdone. The other CPI components, which reflect 'lost' cycle opportunities due to miss events such as branch mispredictions, and cache and TLB misses, are stacked on top of each other. Figure 4.20(c) demonstrates Sniper ability to focus on a single CPI component contribution.

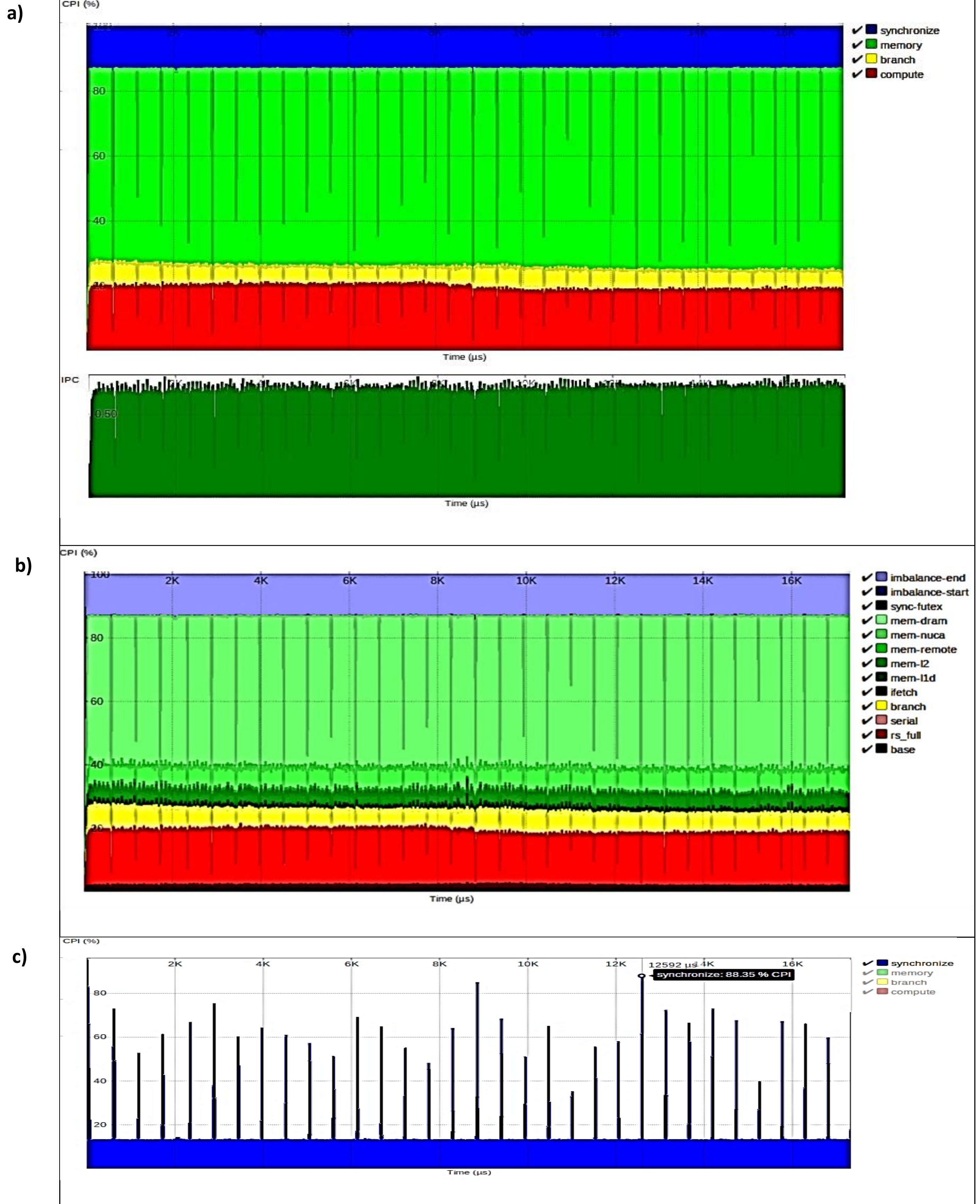


Figure 4.20 An example of Haswell simulation output a) simple b) detailed c) single component.

In the following CPI stack graphs, from Figure 4.21 to Figure 4.36, we present the multithreaded benchmarks CPI stack changes over time and correlating these CPI downbreaks with the IPC changes over time. We used detailed CPI stack graphs and single components contribution graphs as explained in the previous example in Figure 4.20. Unfortunately, due to space limitation, we can not include the detailed and single components graphs. They are large data statistics outputted from 64 simulations. But include all simulation results on compact disk (CD) that is attached with this thesis.

Figure 4.21 shows **FFT** benchmark CPI stack with small input size. FFT is a compute and memory intensive application. The CPI stack shows how the computation (in red) has significant ration in the CPI stack. The time spent on computation doesn't change over all designs (around 0.21 CPI). However, we fixed all in-core and branch prediction specifications. The other key design bottleneck is memory components. It is the reason for performance changes over the four case-study designs. nHaswell has the smallest execution time (1.88 ms) and largest IPC. The worst case is with nXeon phi microarchitecture (2.679 ms). The memory contribution in nXeon takes place in more misses in CPI. The lack for L3 stresses maximize the need for off-chip DRAM accesses. The medium performance of the two bus based systems put them in the second and third performance ranking levels (2.105 ms and 2.138 ms) for nGainestown and nDunnington, respectively. We can point the more role for private L2 cache in nGainestown over shared L2 cache of nDunnington. The more L2 cache available per core, the more hit rates. Also, the more IPC.

Figure 4.22 shows the second **FFT** IPC stack with the large input data set, we can see that in all applications as the problem scaled up the memory contributions increases, because the parallel applications stresses more the memory hierarchy which results in a significant fraction of time spent on cache misses and off-chip DRAM accesses.

The other interesting conclusion is when the FFT problem scaled up the synchronization overhead becomes very small percent. Mohammad and Abandah, (2015) presented that the number of synchronization calls per 10^6 memory accesses generally decreases because these synchronization calls either are at fixed points of the code and they do not increase as the problem size increases. From FFT CPI stacks, we showed that the synchronization contribution becomes dominant factor in three positions in the execution time; in the start, middle and final. But with more than of three quarters of the time execution, synchronization has negligible contribution on CPI loss. As the problem scaled up, FFT behaves unexpectedly. The L3 cache accesses becomes overhead for memory contribution. The needed data were lied in DRAM. In addition, FFT has small sharing degree, 100% of shared data are shared by one thread Mohammad and Abandah, (2015). Therfore, nHaswell has the largest CPI losses due to off tile-LLCs L3 contribution. nXeon on the other hand, has high memory cycle stalls due to the lack of upper cache levels. Therefore, more off-chip DRAM accesses. In the case of bus based systems. They are the more suitable for FFT like benchmarks. Gaining 2.390 IPC and 2.377 IPC for nGainestown and nDunnington processors, respectively. The IPC drop for nDunnington because sharing property of L2 cache.

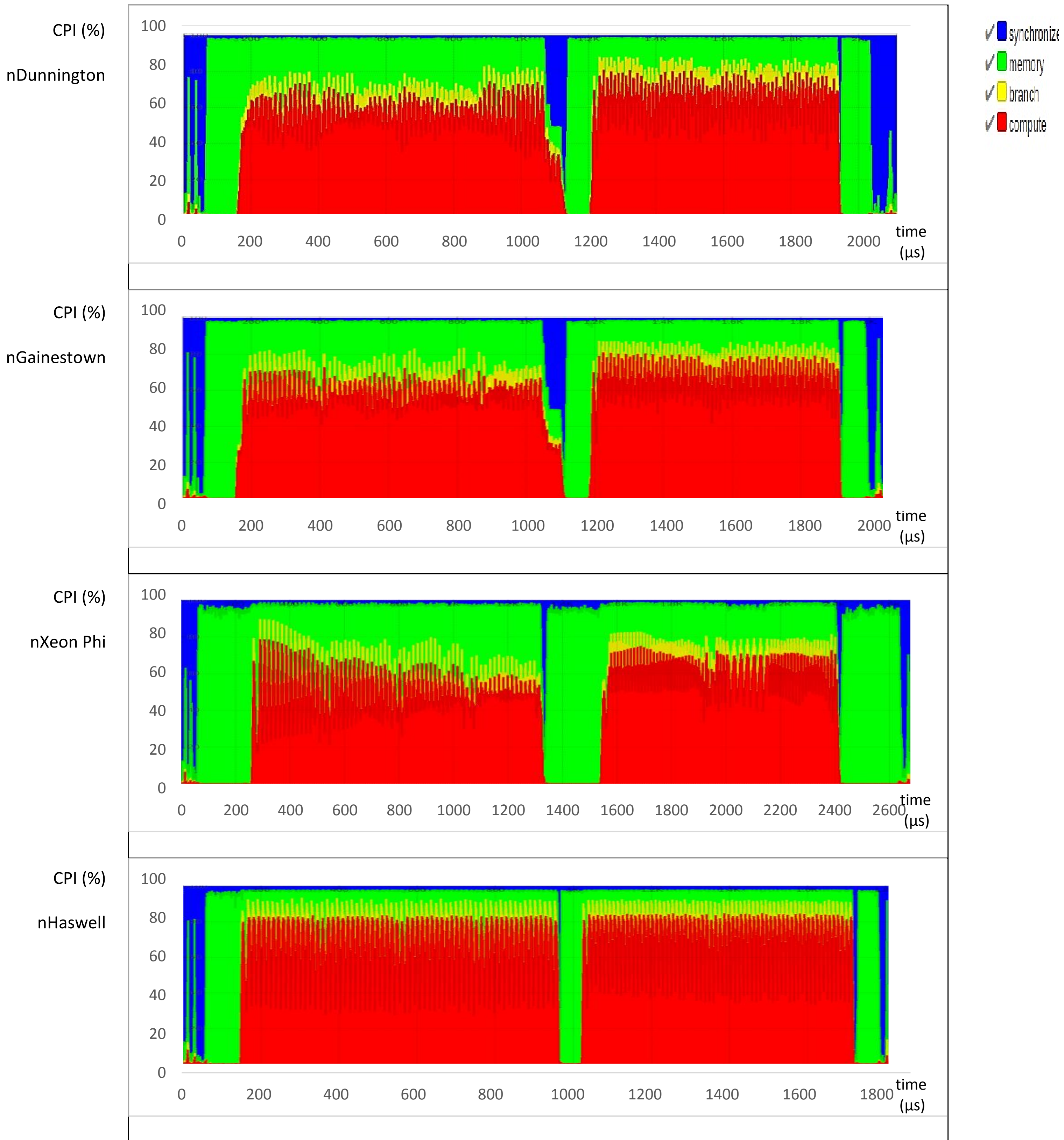


Figure 4.21 CPI stack over time for running SPLASH2-FFT benchmark application with small input size over the four normalized multicore design alternatives.

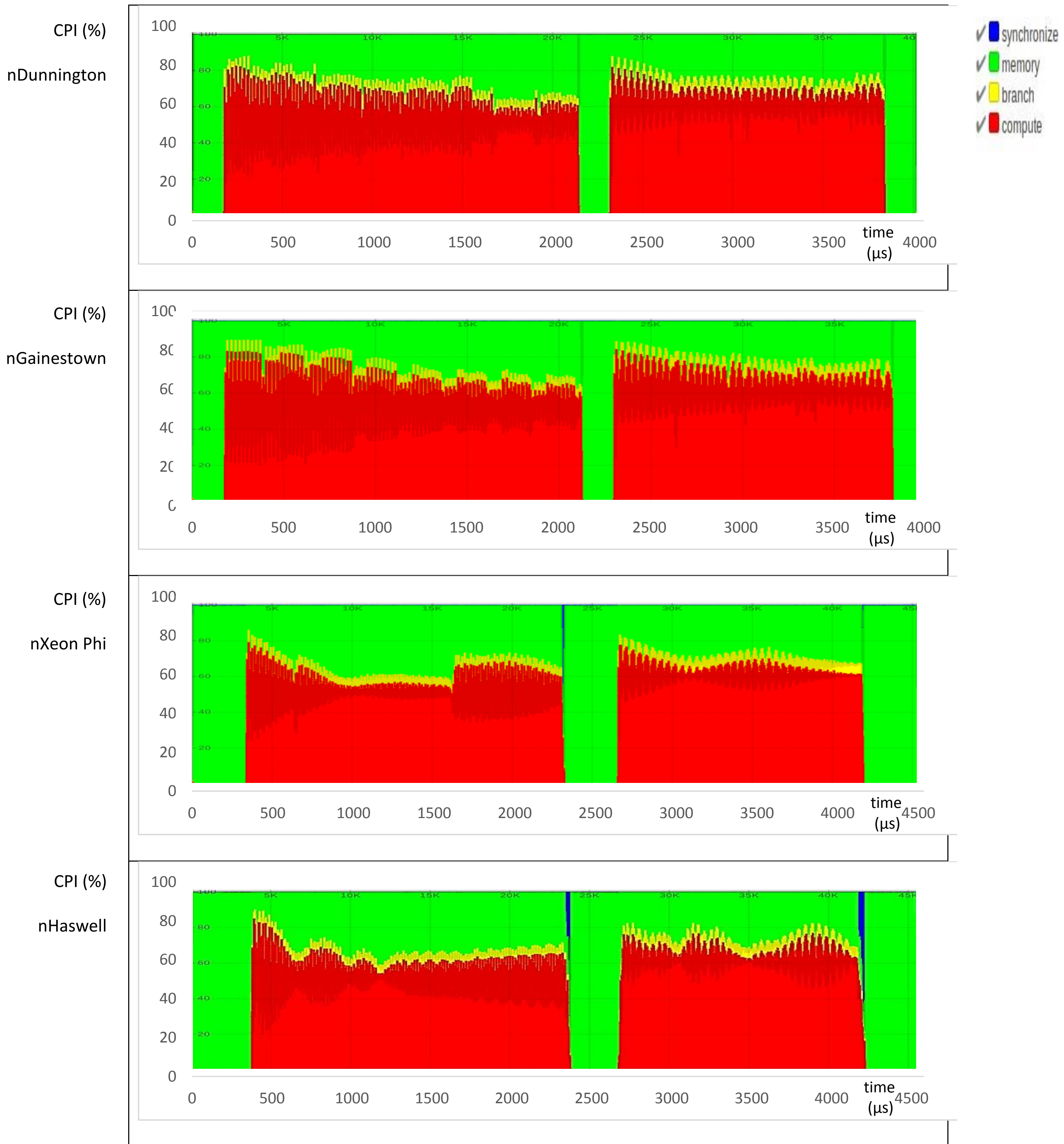


Figure 4.22 CPI stack over time for running SPLASH2-FFT benchmark application with large input size over the four normalized multicore design alternatives.

Radix benchmark CPI stacks are shown in Figures 4.23 and 4.24. The design bottleneck is the integer and floating-point computation (24% Floating-point operations), high percent of CPI loss due to compute time (close to 95% CPI), Radix has negligable synchronization contribution (around 0.05%). Mohammad and Abandah, (2015) mentioned that Radix has a small sharing degree where 90% of shared data are shared with only one thread. Because we configure all designs at the same in-core configurations, most the designs have the same time spent due to computation components. But at the last quarter of simulation, Radix transfers to become memory intensive application (near 86% CPI). nXeon spent more highest (relatively) time in waiting for memory stall cycles, therefore, it needs 4.577 ms execution time. On the other hand, nHaswell consumes the smallest Execution time equal to 4.476 ms. It needs smallest time processing memory operations. nDunnington and nGainestown also, have medium execution times (4.63 ms and 4.53 ms respectively). nDunnington has more memory contribution. Its mem-L2 contribution is close to 1.2 % CPI on average at the end of execution time, instead of nGainestown mem-L2 contribution about 1.15 % CPI.

Radix is the same as FFT benchmark, in case of large problem size. The synchronization calls do not increase when number of memory accesses increases. Therefore, memory contribution increases and synchronization decreases. All system designs exhibit the same 0.33 IPC in the most of their execution times. But, at the end of simulation, the memory components contribute the CPI stack and be a key role in varying the total overall performance. The design systems have the same performance order as small problem size. nHaswell, nGainestown, nDunnington and nXeon, we ordered them from the best performance to the worst.

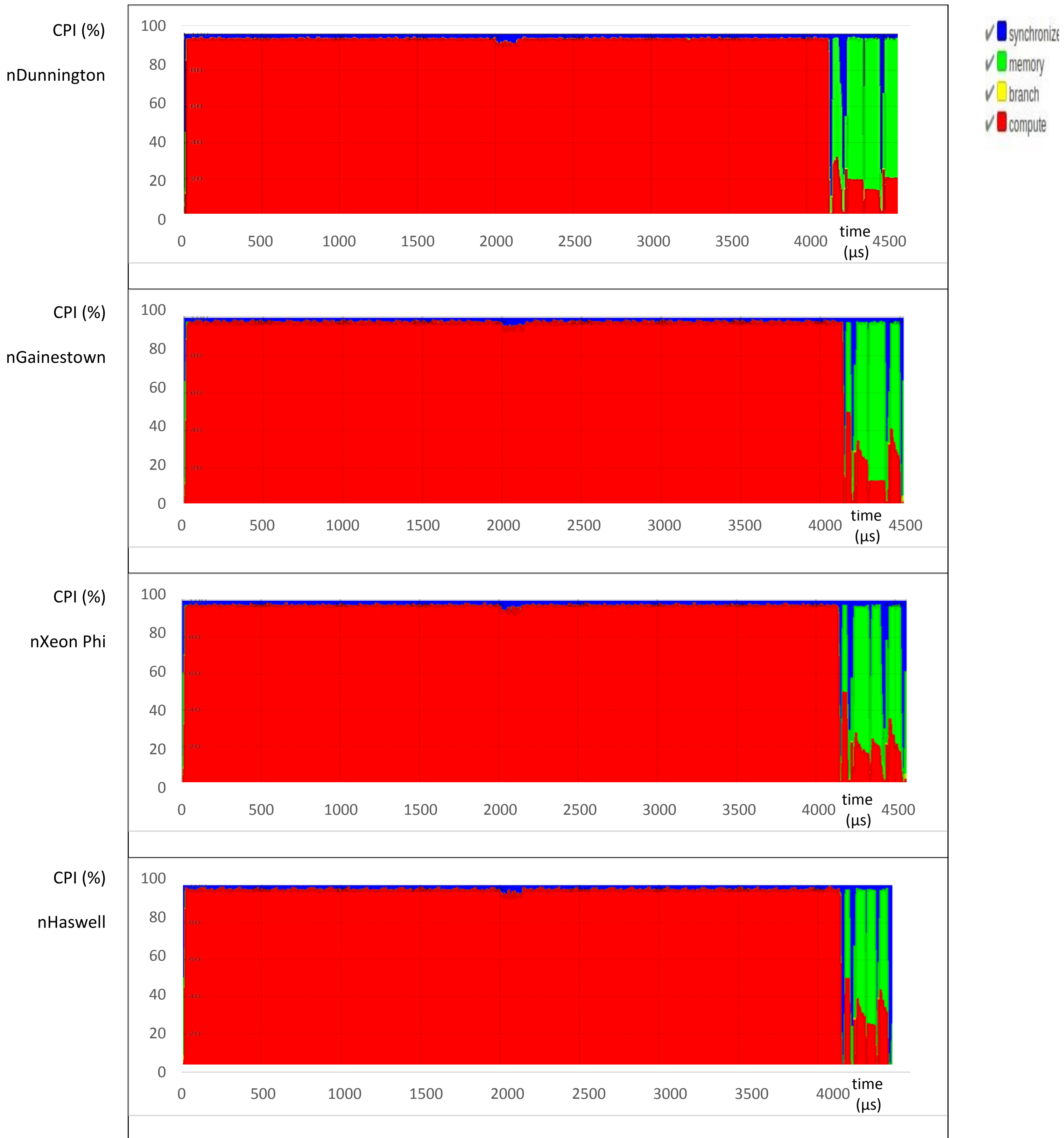


Figure 4.23 CPI stack over time for running SPLASH2-Radix benchmark application with small input size over the four normalized multicore design alternatives.

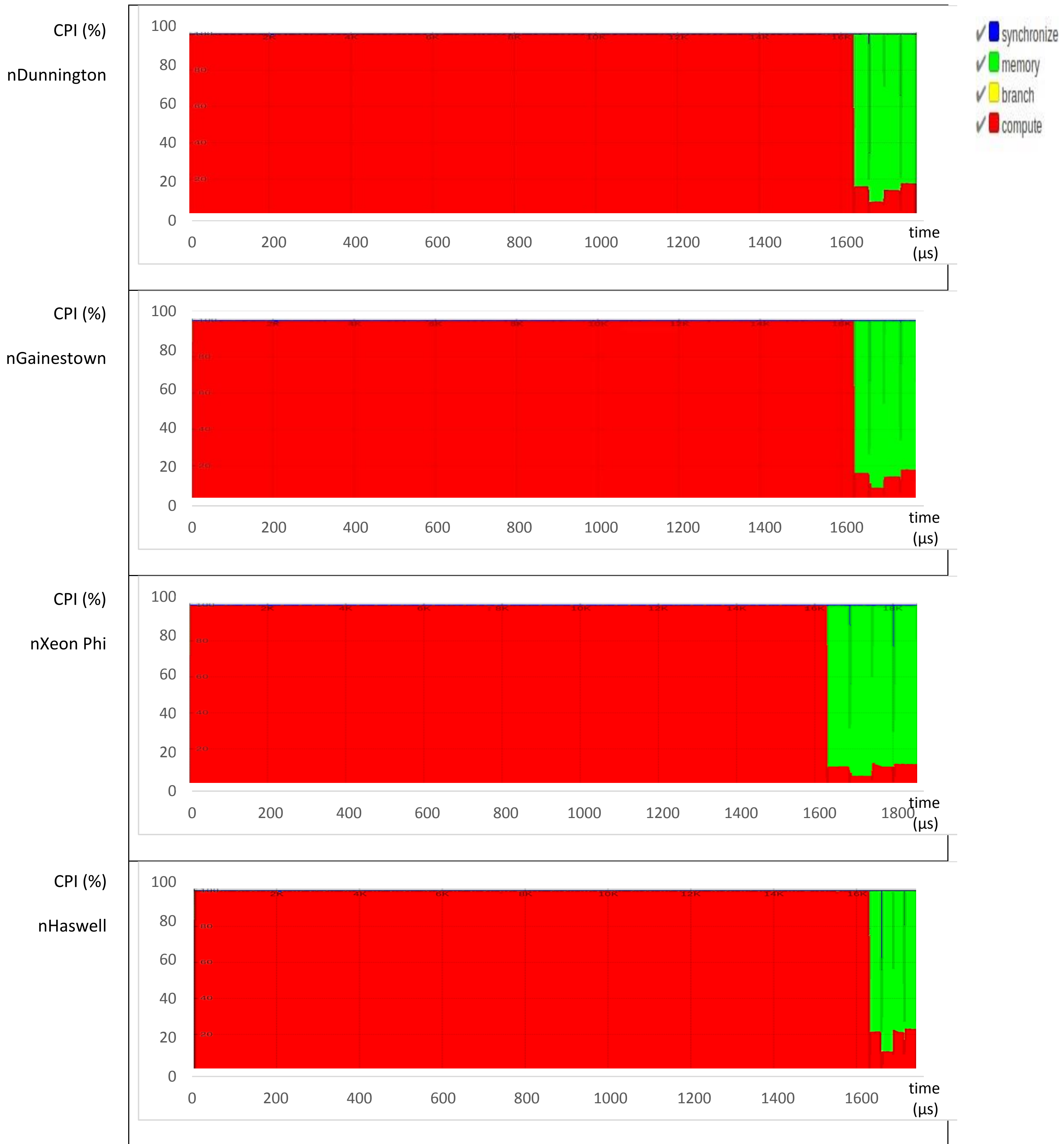


Figure 4.24 CPI stack over time for running SPLASH2-Radix benchmark application with large input size over the four normalized multicore design alternatives.

Lu.cont benchmark CPI stacks are shown in Figures 4.25 and 4.26 , it is a compute intensive multithreaded application, that have 85 % floating-point and need to synchronize data between cores every time period. So, the synchronization become design bottleneck in performance evaluation as we scaled up for larger and larger data sets. Lu.cont needs design that minimizes the core-to-core communication overhead. Mohammad and Abandah (2015) presented that Lu.cont has more than 92% of the data sharing among each four threads. Multicore that able to minimize memory contribution on CPI stack. In small problem size, nHaswell has the hieghest performance or minimum execution time (7.32 ms). Due to its efficient memory design that minimize time spent for the memory components, and for it is high bidirectional speed ring which achieves good tolerance with core-to-core communication overhead (less than 0.10 % on average). nXeon, on the other hand, exhibits similar synchronization contribution, but the design bottleneck was the memory components.

Bus based systems behave differently. nDunnington suffers from large synchronization contribution comparable to nGainestown. nGainestown benefits from QPI to speed up core-to-core communication. The other advantage of nGainestown is its private level 2 cache which minimize L2 cache misses leading to minimizing memory contribution on CPI stack.

Figure 4.26 shows Lu.cont benchmark CPI stacks in the case of large problem size. We concluded that the four multicore design alternatives behave in the same manner of small input size. The differences are increasing memory contribution and decreasing synchronization contribution on CPI stacks. nHaswell has best performance, then nGainestown, nDunnington, and finally nXeon, with 52.36 ms, 54.40 ms, 54.36 and 58.00 ms, respectively.

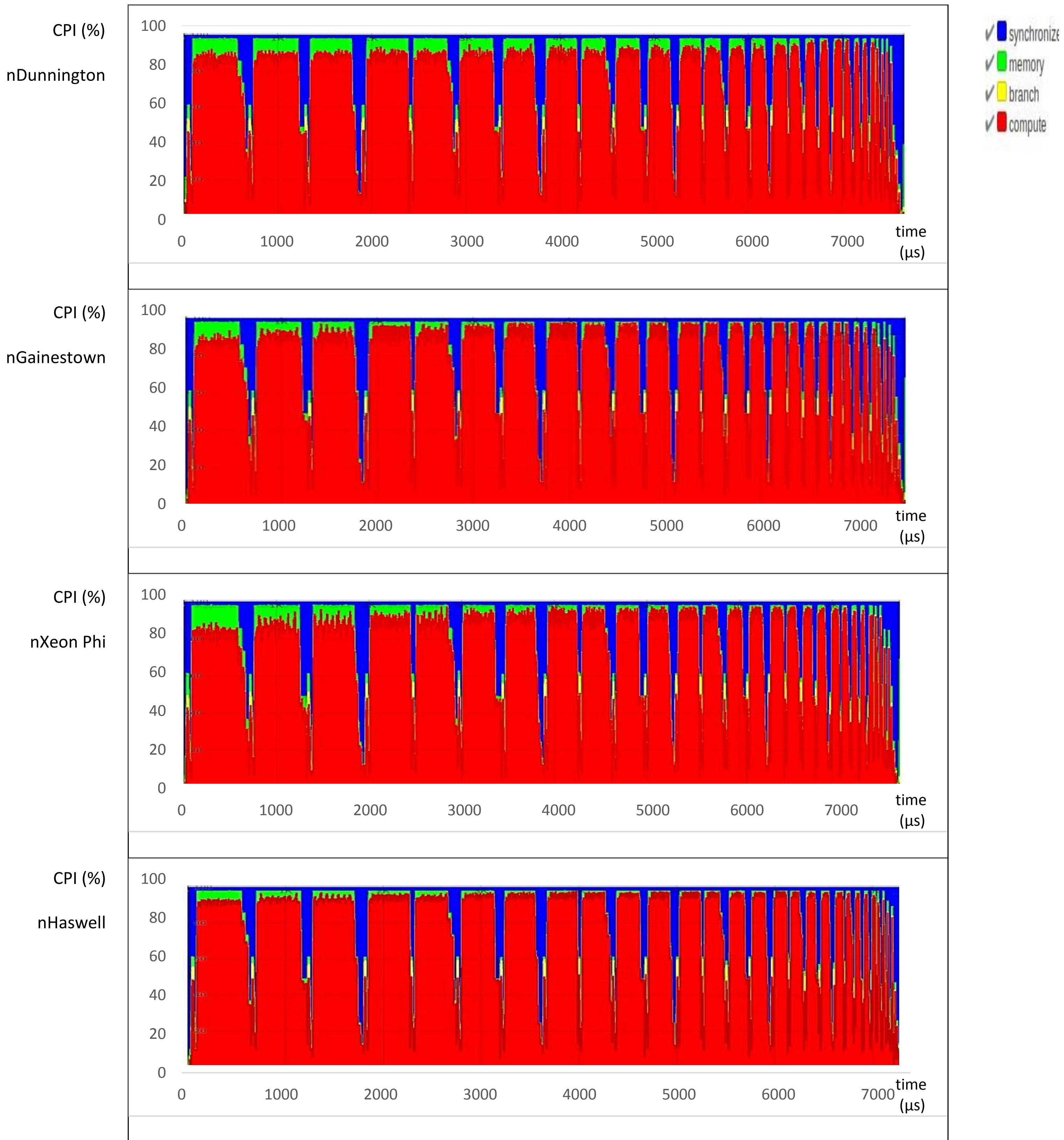


Figure 4.25 CPI stack over time for running SPLASH2-Lu.cont benchmark application with small input size over the four normalized multicore design alternatives.

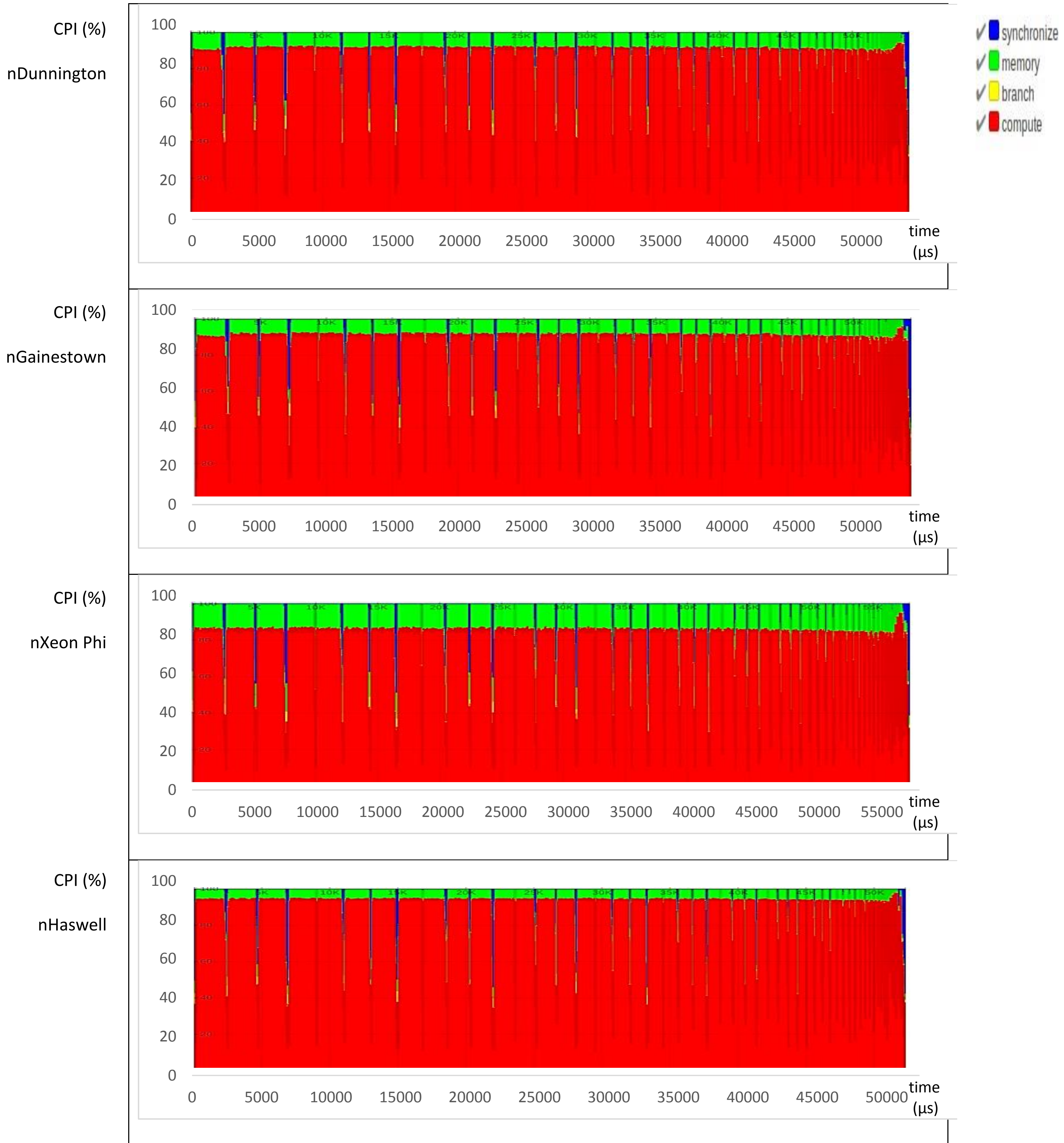


Figure 4.26 CPI stack over time for running SPLASH2-Lu.cont benchmark application with large input size over the four normalized multicore design alternatives.

Cholesky benchmark application is shown in Figures 4.27 and 4.28, for the small and large problem sizes, respectively. It is classified as compute and memory intensive application. Mohammad and Abandah (2015), concluded that it has minimum core-to-core communication, each thread communicates with itself, i.e. each thread reads from or writes to memory locations that it previously wrote to them and shared them with other threads. Initial thread sometimes communicates with all other threads. We get same conclusion. All figures present minimal synchronization CPI contribution (less than 0.01 % in most execution time). nDunnington has larger memory contribution than nGainestown. The worst case refers to the weaknesses of memory hierarchy of nXeon (max of 52% CPI to 17% at the end of simulation at large problem size). The lack for level 3 cache is the main reason. On the other hand, nHaswell exhibit highest performance because it has better memory efficiency (max of 38% decreasing to 23% at the end of the simulation at large problem size). We get same behavior of design alternatives in small input size. Notice that memory contribution for large problem size increases their contribution. An interesting point that Cholesky, after period of execution, benefits from its communication slack in decreasing far memory accesses and therefore increasing IPC.

The branch prediction contribution on CPI starts to be one of the design bottlenecks in PARSEC benchmark applications (**Blackscholes, Swaptions and Fluidanimate**), they have larger contribution (relatively to SPLASH2 applications) valuable percent of synchronization overhead near 25 % of overall system CPI. Branch prediction contribution differences on all normalized systems will be negligible because we fixed their branch prediction features.

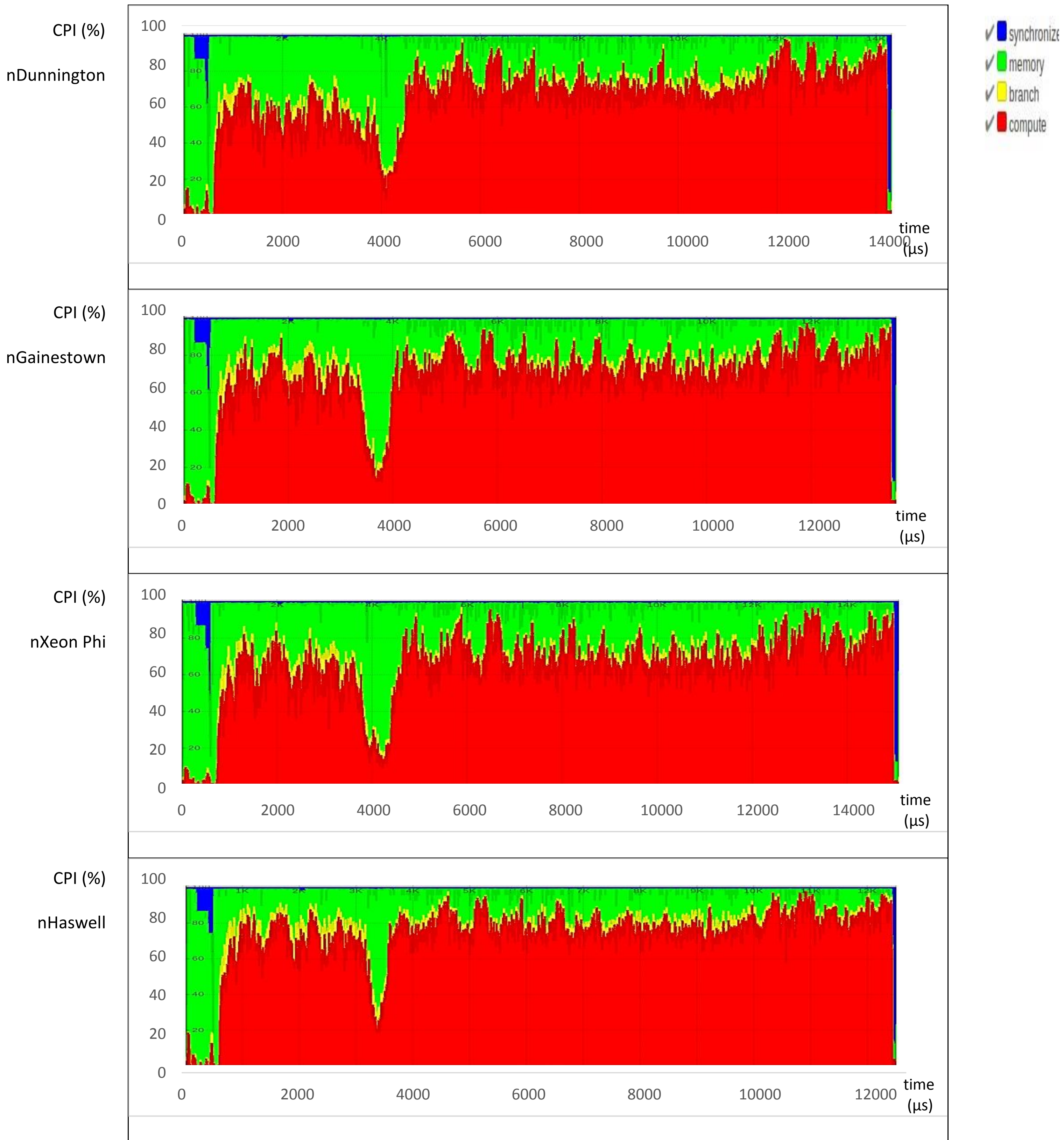


Figure 4.27 CPI stack over time for running SPLASH2-Cholesky benchmark application with small input size over the four normalized multicore design alternatives.

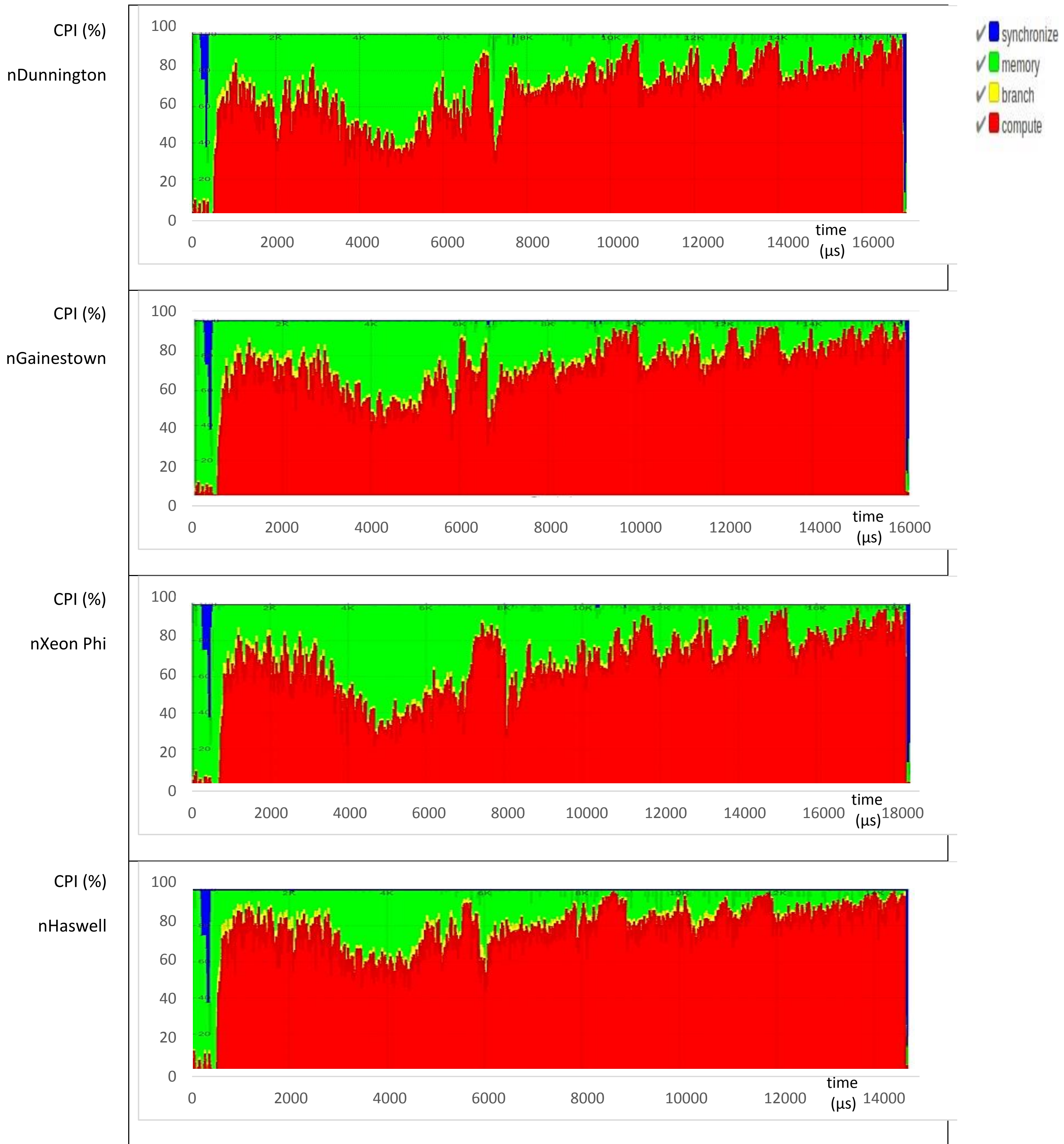


Figure 4.28 CPI stack over time for running SPLASH2-Cholesky benchmark application with large input size over the four normalized multicore design alternatives.

Blackscholes application CPI stacks changes over time which are shown in Figures 4.29 and 4.30, Blackscholes have small sharing degree between threads, minimum core-to-core communication, like FFT benchmark (Mohammed and Abandah, 2015). We get the same application characterization. Figure 4.29 show that synchronization contribution on CPI of all multithreaded workload does not exceed 2% of their CPI stacks.

Blackscholes is a compute intensive application more than 80 % of CPI in most of design alternatives, the floating-point units in Blackscholes are design bottlenecks. But, as we normalized all in-core specifications and without core-to-core communications. There are negligible differences on alternatives performance due to computation contribution. Furthermore, nDunnington exhibit more memory contribution than nGainestown. nXeon the largest (relatively) memory contribution. nHaswell the best design that behaves will for memory design bottleneck.

Mohammad and Abandah (2015), presented that Blackscholes communication has a large percent of slack in the ranges of 99.5% for small input size, and 99.9% of slack for large problem size. That interprets why memory contribution decreases as we go to end of execution time.

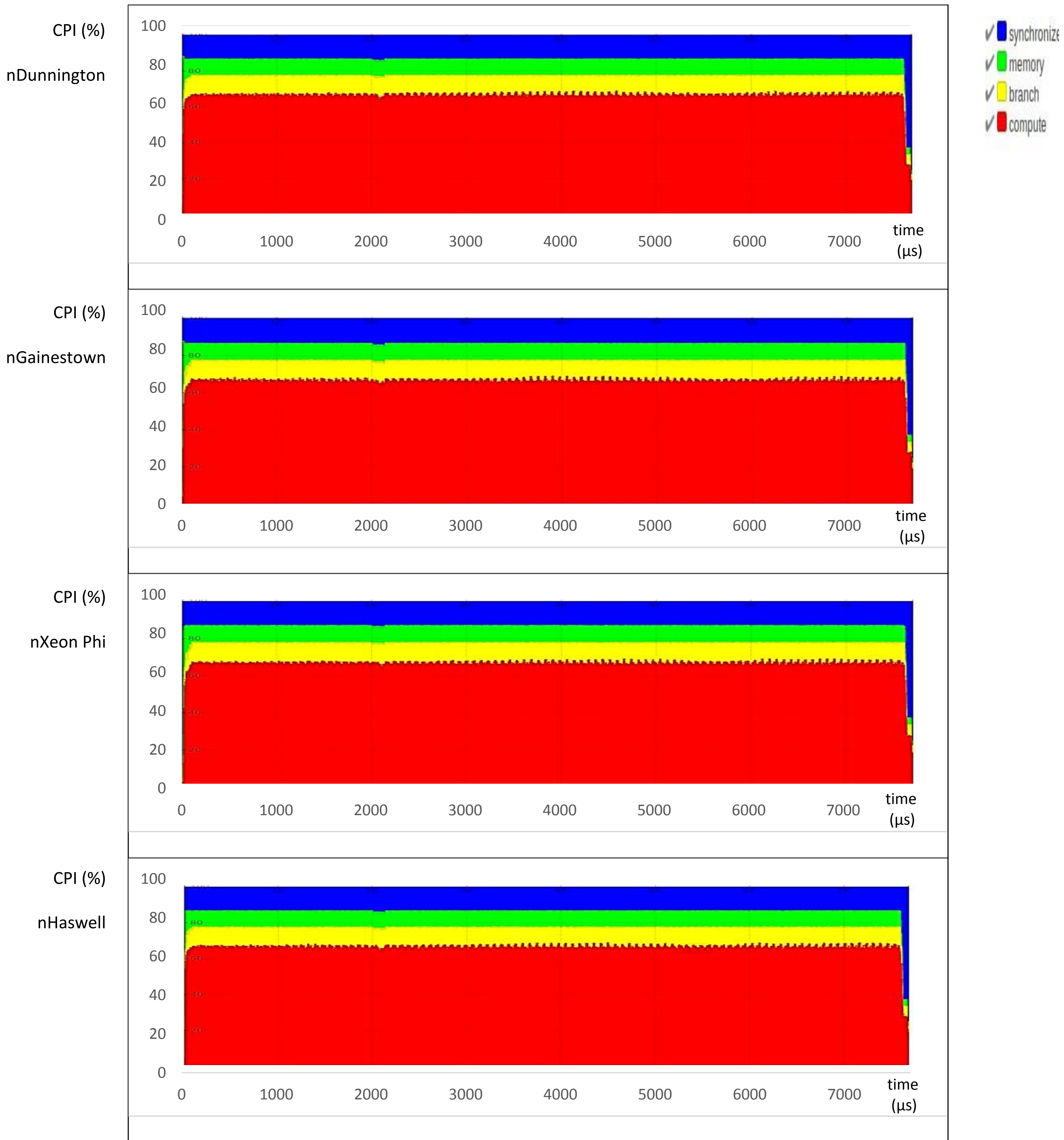


Figure 4.29 CPI stack over time for running PARSEC-Blackscholes benchmark application with small input size over the four normalized multicore design alternatives.

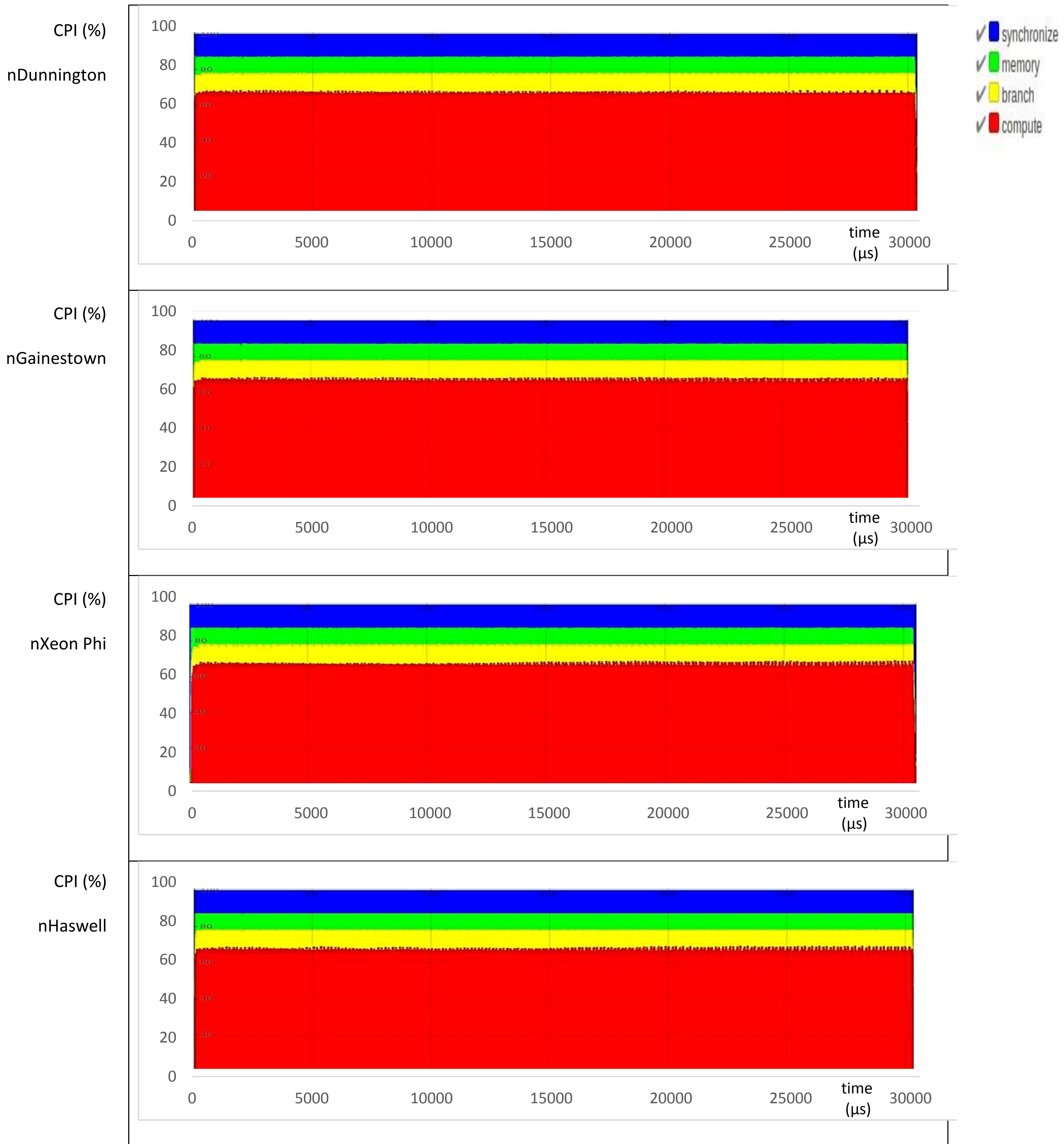


Figure 4.30 CPI stack over time for running PARSEC-Blackscholes benchmark application with large input size over the four normalized multicore design alternatives.

Canneal is a memory intensive application because it has around 70% of memory contribution on CPI stack for small problem size and around 72% for large problem size, as shown in Figures 4.31 and 4.32, respectively. For large input size, nHaswell has the largest IPC (0.595) and smallest execution time (57.96ms). The mem-DRAM contribution is the reason behind the large memory access latency, nXeon Phi clearly shows larger memory contribution on CPI stack rather than other designs (a round 65% of processor CPI). On the other hand, nHaswell drops its mem-DRAM contribution from 8% at the start of simulation to 0.06 % at the end of simulation. However, Mohammad and Abandah (2015) concluded that Canneal's communication has a large percent of slack in the ranges of tens of millions of instructions and more, where it has 86.5% of slack in these ranges for small problem size and 90.4% of slack in these ranges for large problem size. This behavior indicates that cache hierarchy of L1, L2, and NUCA L3 slices in nHaswell satisfies the benchmarks data requirements. More cache hits due to communication slack. Hence, minimize DRAM accesses. This type of applications needs efficient memory hierarchies. Larger cache sizes and more than two levels of caches. Also, ring NoC serviced synchronization overhead better than others (0.23 to max of 0.25 CPI over time). nXeon with 2d mesh has synchronization CPI of (0.4 to 0.43 CPI over time). The bus-based two systems exhibit medium performance values (IPC and ex. times) from the other nHaswell and nXeon alternatives, but with small differences between them. nDunnington presents more memory contribution on CPI stack (1.54- 1.55 CPI) rather than nGainestown memory contribution (1.51 -1.52 CPI) over simulation time. The reason is the shared L2 per 2 cores in ndunnington versus the same size private L2. Higher mem-l2 contribution is for nDunnington which starts from (0.51% to minimum 0.40) at the end of simulation time (benifiting from communication slack). And (0.46 % to 0.40) for nGainestown. Because there are more available L2 memory banks for

each core. All multicores have the same integer compute contribution on CPI stack which equals (0.28 CPI).

Fluidanimate application CPI stacks change over time are shown in Figures 4.33 and 4.34. Fluidanimate has large percent of synchronization overhead (close to 50% of CPI stack) for all multicore design alternatives. This overhead is due to Fluidanimate's partitioning of the work among the threads and each thread handles its portion, also they interact with other threads to handle the shared data (Mohammed and Abandah, 2015). The nHaswell design presents better performance over other multicore design alternatives. There execution times are 193.5ms, 196.4ms, 196.9ms and 203.1ms, for nHaswell, nGainestown, nDunnington and nXeon, respectively. The higher memory contribution in nXeon is responsible for higher CPI. On the other hand, nHaswell LLCs minimize the time loss due to memory components. nGainestown's private L2 cache exihipit smaller contribution than nDunnington shared L2 cache. **Swaptions** application CPI Stacks in Figures 4.35 and 4.36. It shows different behavior versus other applications. The performance CPI statistics, for large problem size, behave more like some constant lines over two partitions of execution time. In the first half of application execution time, the CPI contribution is mainly on the compute components (arround 52.5 % of CPI stack) and (12.5 % of CPI satck) for synchronization and (20% of CPI) for branch prediction, finally, (15 % CPI) for memory components. Then, in the second half of execution time, the main contribution component transfers to be the synchronization (87% of CPI stack), due to the high percent of core-to-core communication overheads. Therefore, that explain the huge decreases in IPC in the second half of of application execution. We saw that Swaptions performance statistics have negligible differencess over the four normalized systems. In small input size, the memory contribution becomes less slightly than it in large problem size.

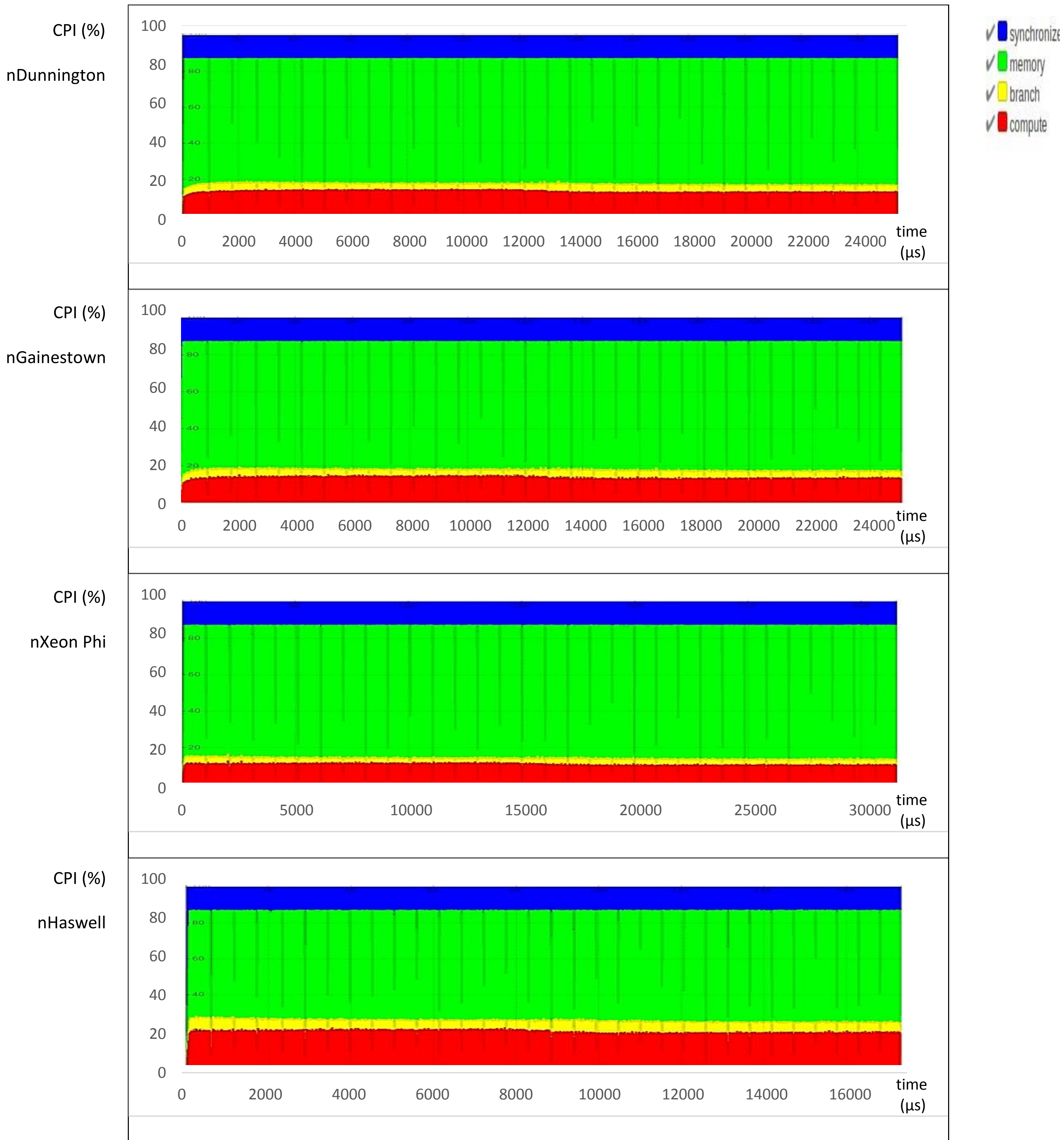


Figure 4.31 CPI stack over time for running PARSEC-Canneal benchmark application with small input size over the four normalized multicore design alternatives.

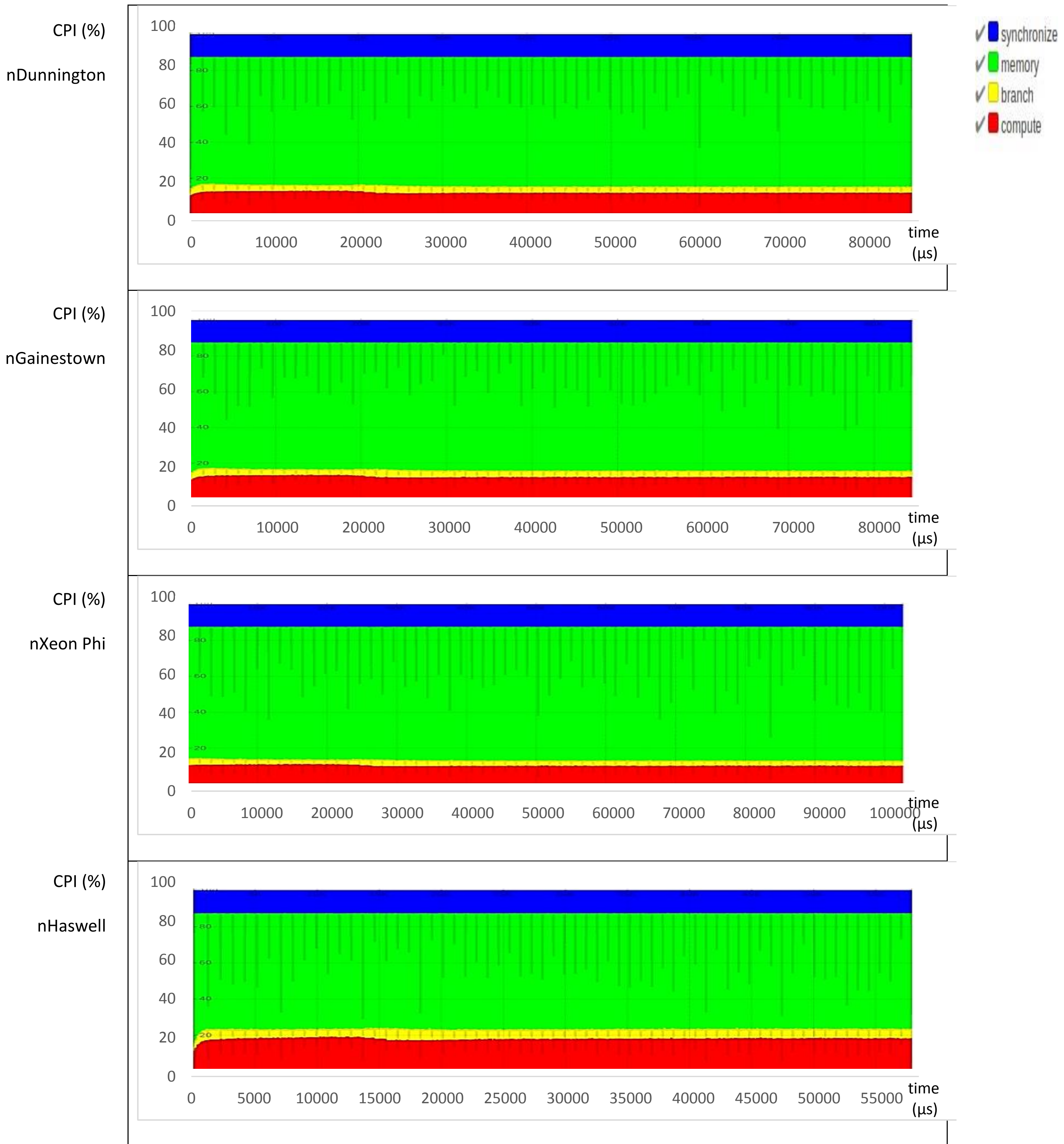


Figure 4.32 CPI stack over time for running PARSEC-Canneal benchmark application with large input size over the four normalized multicore design alternatives.

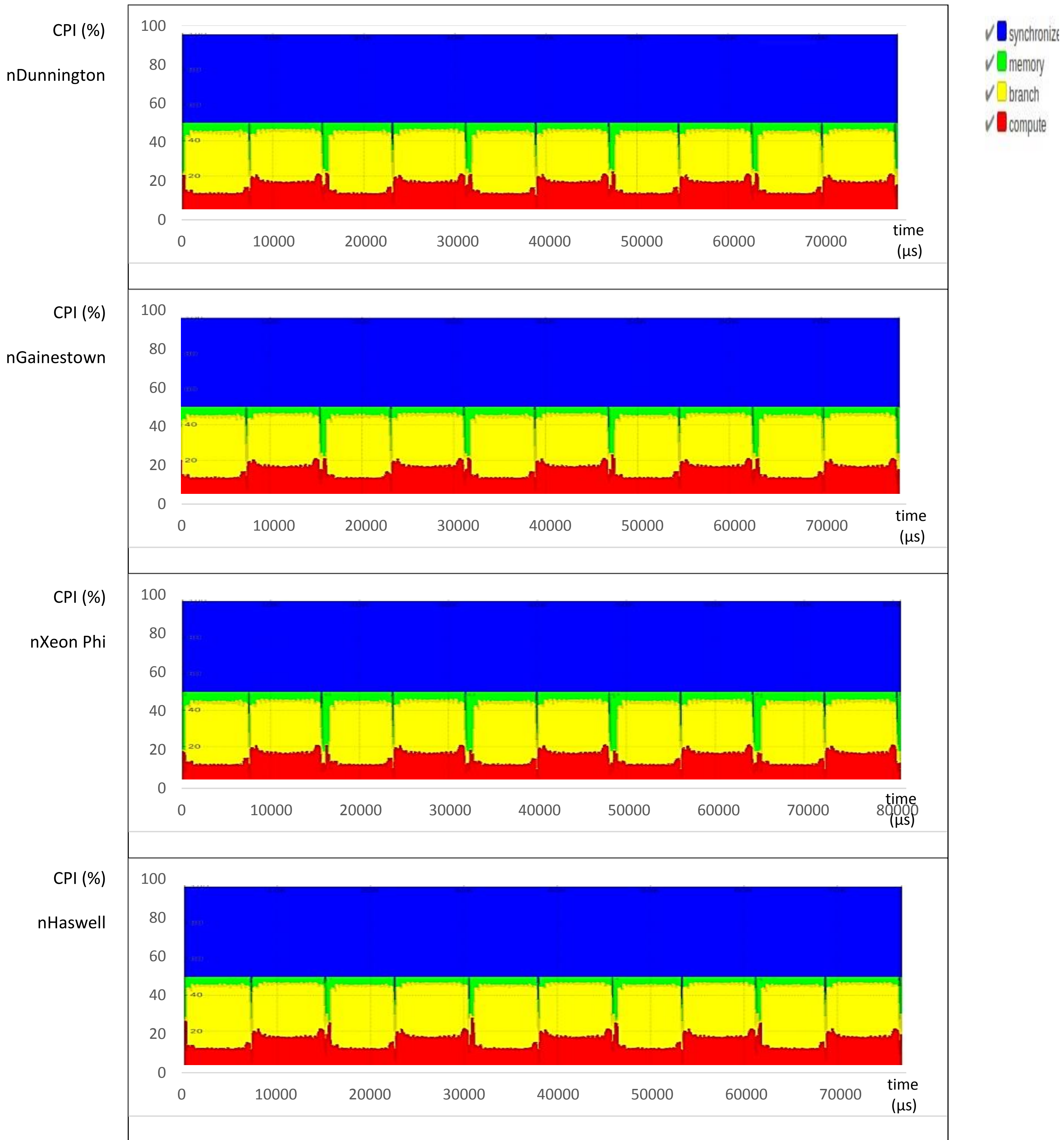


Figure 4.33 CPI stack over time for running PARSEC-Fluidanimate benchmark application with small input size over the four normalized multicore design alternatives.

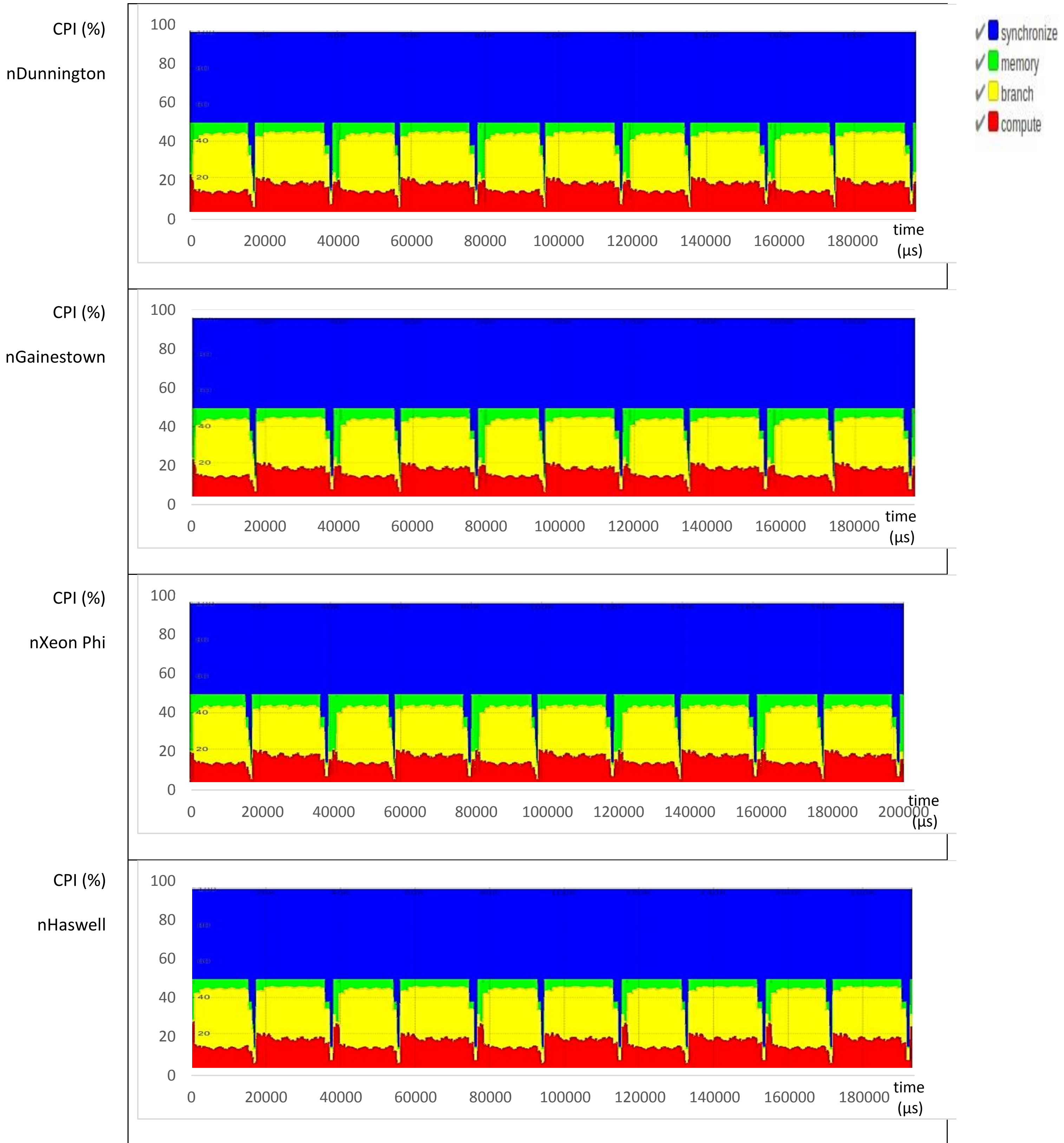


Figure 4.34 CPI stack over time for running PARSEC-Fluidanimate benchmark application with large input size over the four normalized multicore design alternatives.

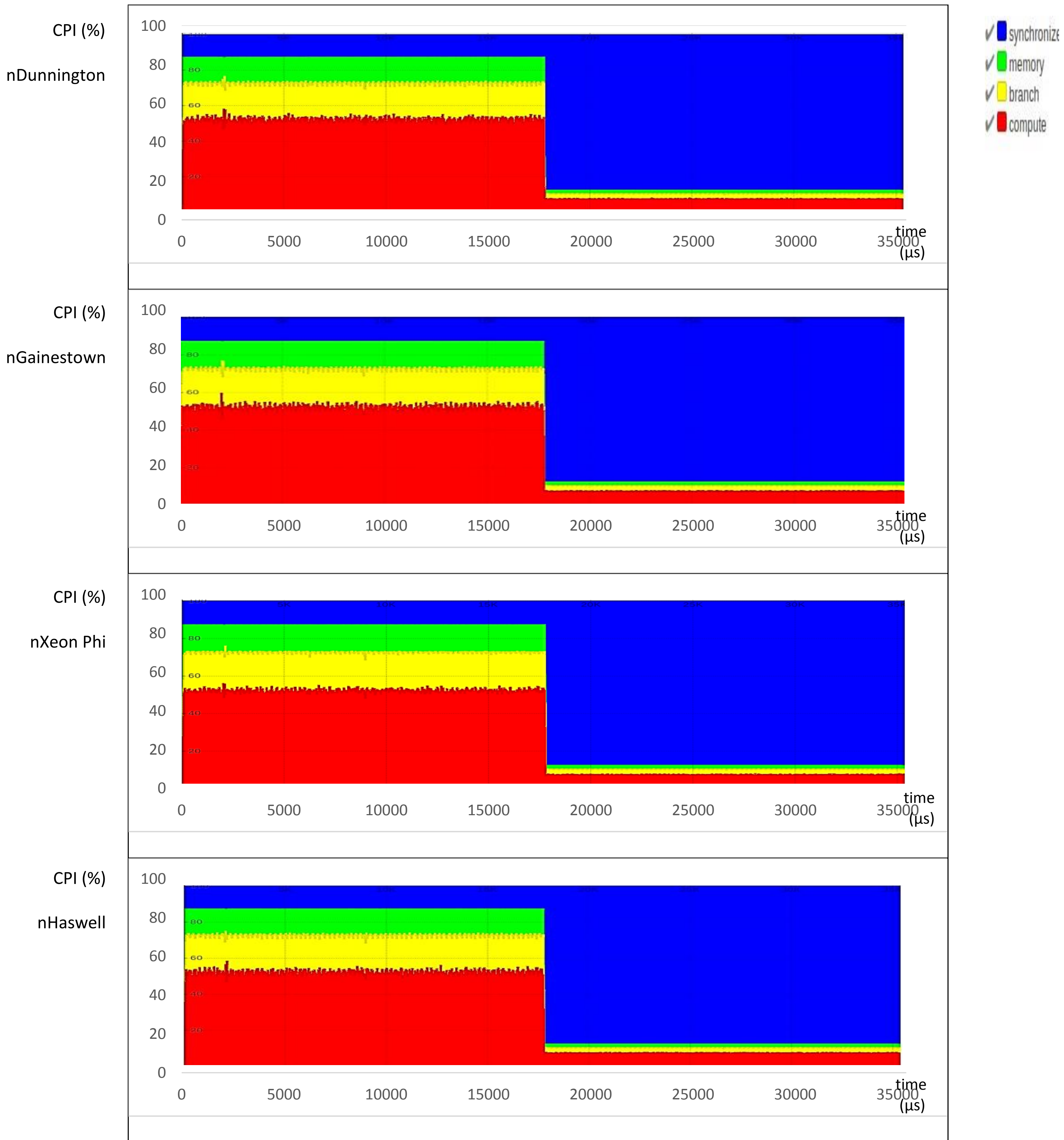


Figure 4.35 CPI stack over time for running PARSEC-Swaptions benchmark application with small input size over the four normalized multicore design alternatives.

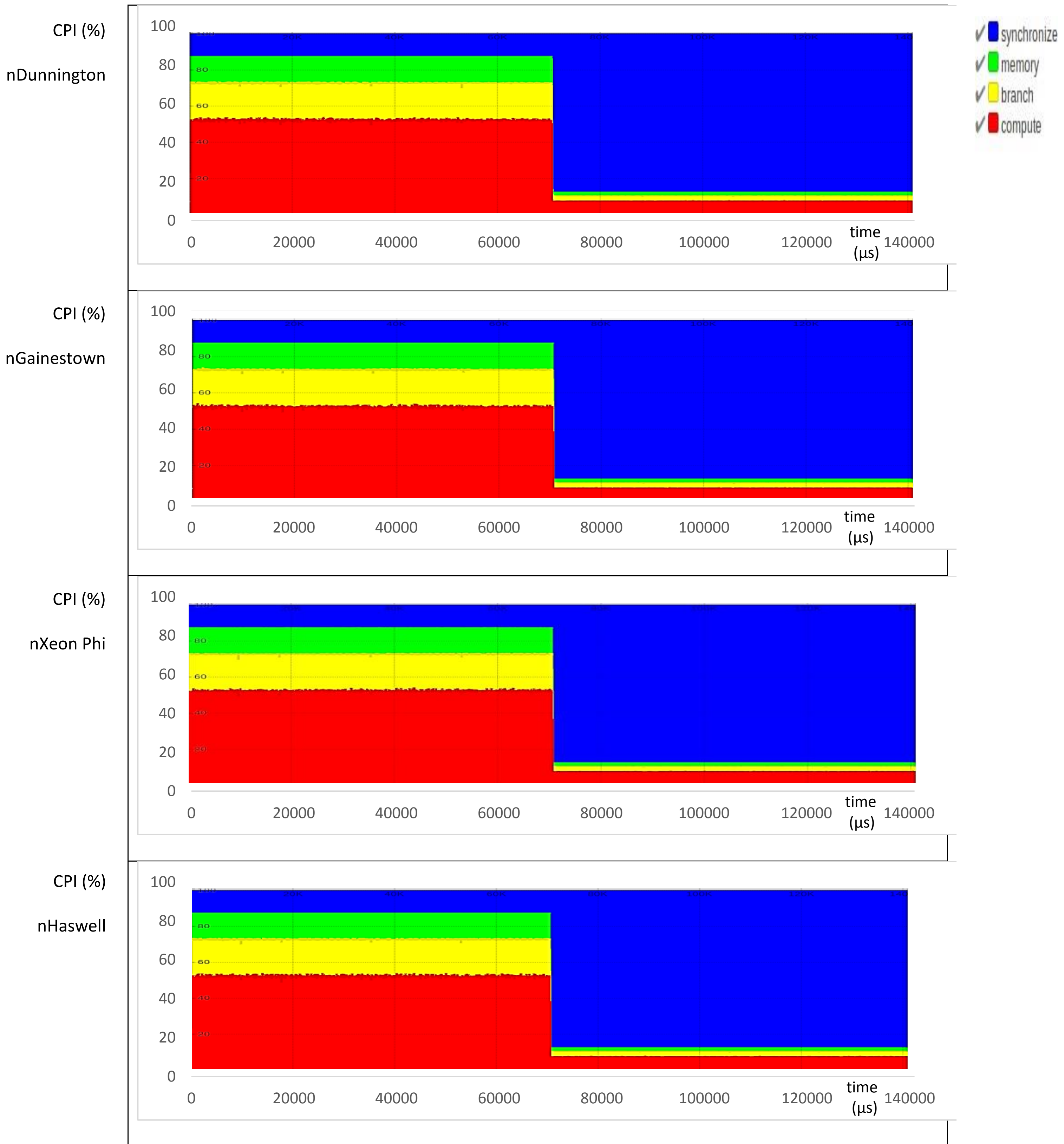


Figure 4.36 CPI stack over time for running PARSEC-Swaptions benchmark application with large input size over the four normalized multicore design alternatives.

As conclusion, high speed synchronization over ring NoC in nHaswell and 2d mesh in nXeon Phi play a main role in minimizing the core-to-core communication overheads. But, the synchronization design bottleneck due to core-to-core communication overhead appears on the two bus-based microarchitectures. Moreover, nDunnington suffers more from shared L2 losses. Also, nXeon Phi suffers from large memory contribution on CPI losses especially in large data set. In compute-intensive or communication-intensive applications the memory weakness design has a negligible affect on the overall system performance. The different behaviour with FFT and Canneal, nHaswell consumes higher execution times because it has minimum interprocessor communication and not benefits from high speed ring network.

4.3.4 Average Core Utilization

The elapsed idle time is the inverse of system utilized time. The processor idle time relates to the core time spent without usefull work. The core is called idle when it is waiting for threads synchronization futex or core-to-core communication. In addition, when it suffers from system stall cycles waiting for data from low level cache hierarchies. Hence, leading to larger delay and larger execution time. The sytem waiting futex simply occures when we are tracing the original parent thread, and it is doing nothing but waiting for some other threads to finish. We conclude that the high utilization multicore system is the one which can hide losses and minimizes synchronization and memory bottlenecks.

For the tilization metric, we use average core utilization percent (%). The following equations are used to calculate this metric:

Utilization % = per thread utilization time * 100 % .... (1)

Per thread utilization time = total execution time – thread idle time .... (2)

Figures 4.37 and 4.38 show the average core utilization for running the eight benchmark applications with the small and large input sizes over the four normalized multicore design alternatives.

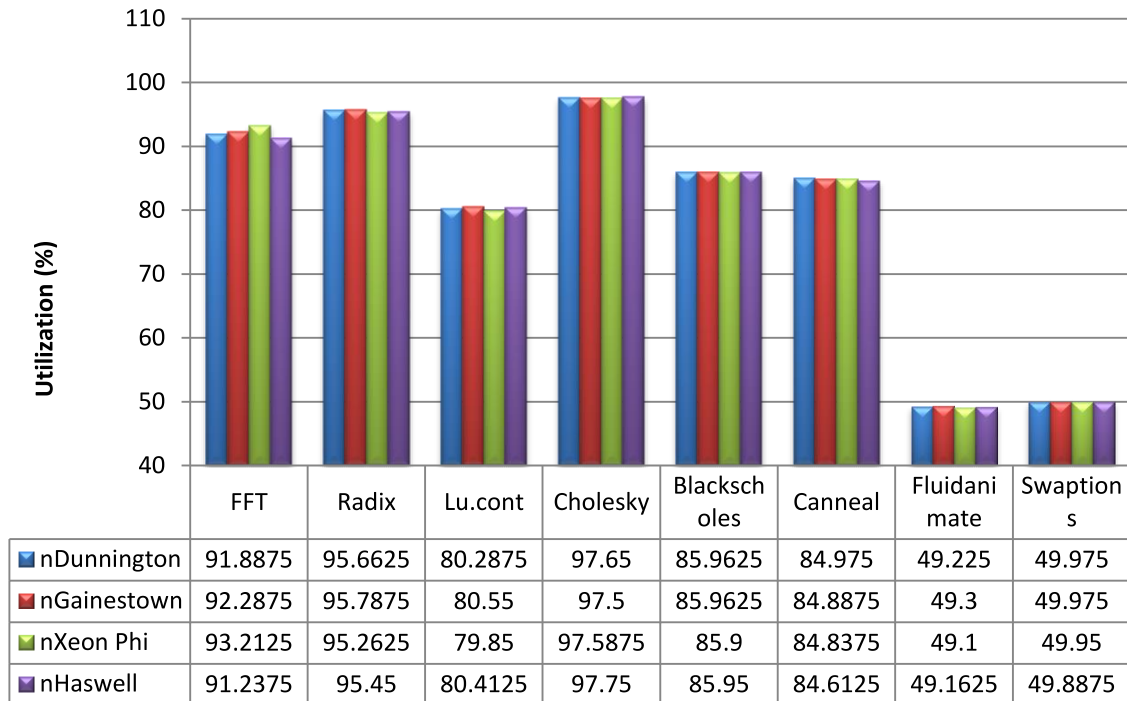
Low time utilization percent and high performance are two opposite goals. However, sometimes the higher utilization is not required, like with power aware systems. However, power consumption increases in case of higher utilization percent. Utilization and power consumption are a budjet factors in designing multicore processors. They are often used in trade-off performance evaluations.

Although the design utilization differencies look small but they show some indications. nXeon exhibits minimum utilization relative to the other designs. It has more idle time due the memory contribution in Radix, Lu.cont, Blackscholes, Fluidanimate, and Swaptions. nHaswell exhipits larger utilization percent in many benchmarks such as Radix, Lu.cont, Cholesky, Blackscholes. It minimizes the cycle stalls and idle time leading to minimum execution times. An interesting conclusion from these graphs is that a linear relation between minimum execution times and higher utilization is not always agiven. The reason for the exception may refer to applicatiom load imbalance.

It is worth mentioning that not all benchmarks are good scaling application. However, **Fluidanimate** only uses five cores in all simulations, the rest of three cores are idle all the time. Thus, Fluidanimate is classified as poor scaling application. Fluidanimate averaged cache L2 miss rates that appeare in Figures 4.10 and 4.11 are averaged and calculated over five cores. All other seven workloads make use of all eight cores.

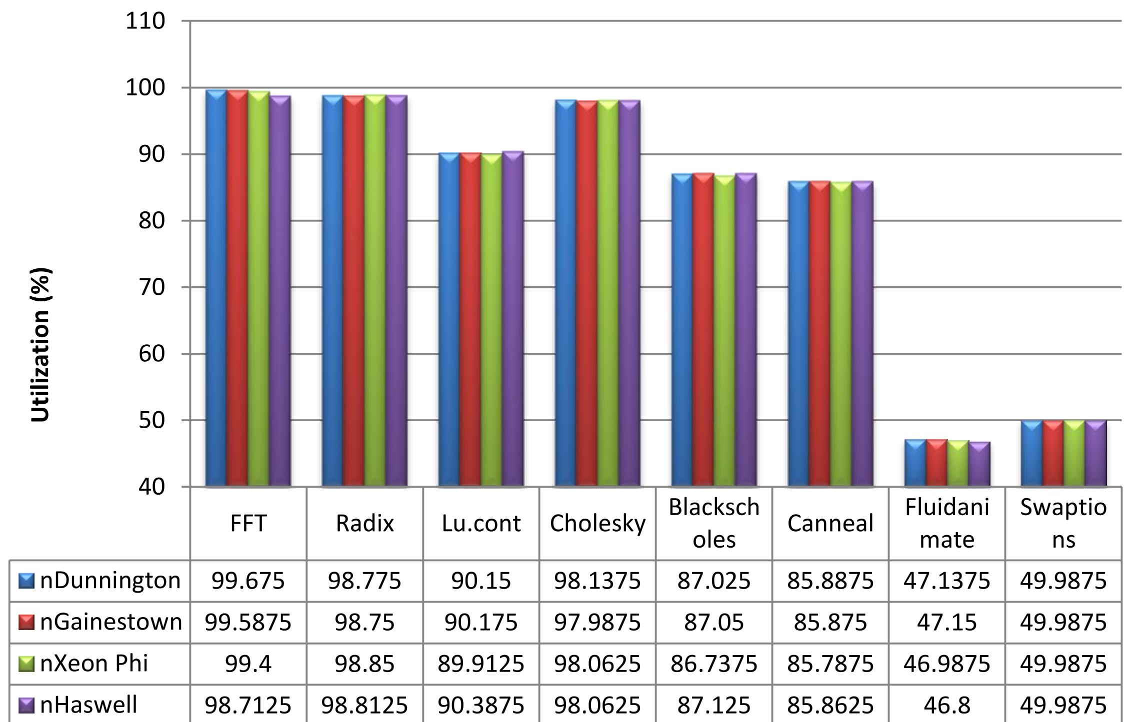
The second interesting behavior that we can concluded is that some multithreaded benchmarks have poor load balance. **Blackscholes, Canneal**, **Swaptions** and **Fluidanimate** have idle time percent larger than or equal to 99.5 % for a working core and its cache miss rate is large relatively to the other cores in the other design alternatives. This core is responsiple for application initialization and communicates with other threads in the processor. **Cholesky, FFT, Lu.cont,** and **Radix** on the other hand, have a good load balance, all threads in the benchmarks have the same percent of instructions executed and the same percent of idle time.

Swaptions and **Fluidanimate** have low utilization compared with the other benchmarks, because they have high communication overhead. The cores stay idle more than 50% of the time waiting for data from each other. The synchronization contribution for **Fluidanimate** is around 50% of the CPI stack (Figures 4.33 and 4.34). **Swaptions** has synchronization contribution around 25% in the first half of execution then transfer to be 87% in the second half of the execution time.



	FFT	Radix	Lu.cont	Cholesky	Blacksch oles	Canneal	Fluidani mate	Swaption s
nDunnington	91.8875	95.6625	80.2875	97.65	85.9625	84.975	49.225	49.975
nGainestown	92.2875	95.7875	80.55	97.5	85.9625	84.8875	49.3	49.975
nXeon Phi	93.2125	95.2625	79.85	97.5875	85.9	84.8375	49.1	49.95
nHaswell	91.2375	95.45	80.4125	97.75	85.95	84.6125	49.1625	49.8875

Figure 4.37 Average core utilization for running the eight benchmark applications with the small input sizes over the four normalized multicore design alternatives.



	FFT	Radix	Lu.cont	Cholesky	Blacksch oles	Canneal	Fluidani mate	Swaptio ns
nDunnington	99.675	98.775	90.15	98.1375	87.025	85.8875	47.1375	49.9875
nGainestown	99.5875	98.75	90.175	97.9875	87.05	85.875	47.15	49.9875
nXeon Phi	99.4	98.85	89.9125	98.0625	86.7375	85.7875	46.9875	49.9875
nHaswell	98.7125	98.8125	90.3875	98.0625	87.125	85.8625	46.8	49.9875

Figure 4.38 Average core utilization for running the eight benchmark applications with the large input sizes over the four-normalized multicore

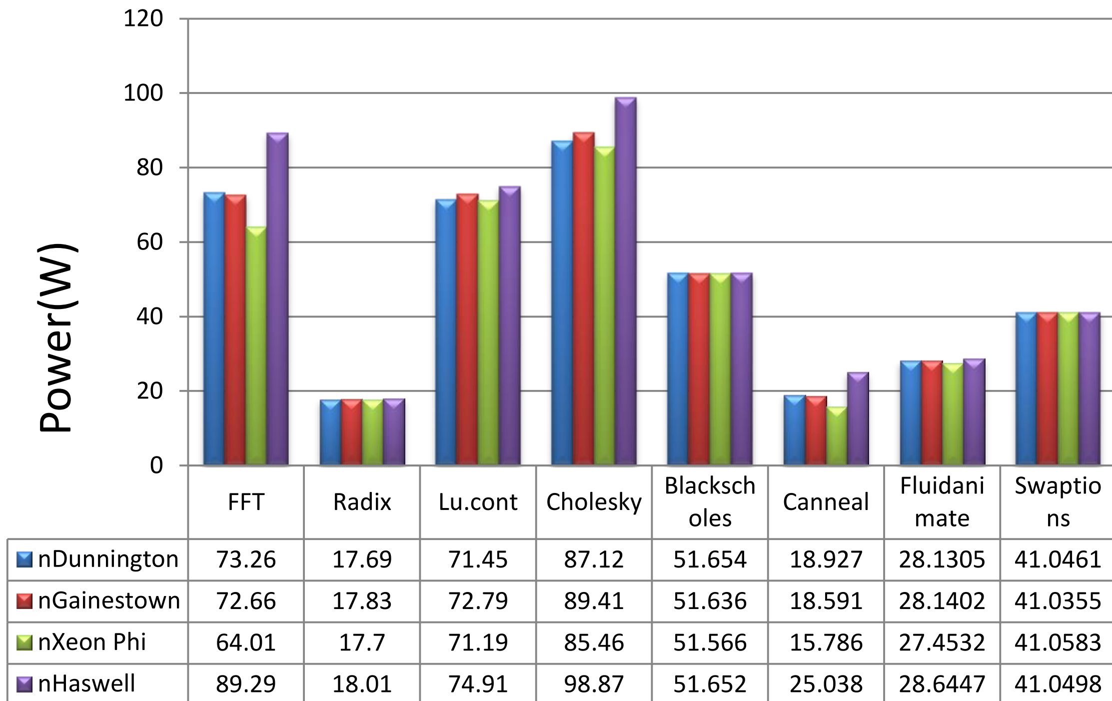
4.3.5 Power Consumption

Figures 4.39 and 4.40 show the average runtime dynamic power for running the eight benchmark applications with the small and large input sizes over the four normalized multicore design alternatives.

The higher power dissipation for nHaswell in most benchmarks is due to high utilization time percent and minimum idle time. Also, due to the FLow control unITs (FLITs) size of ring topology, a FLIT is a unit or amount of data when the message is transmitting in any network link. However, the message or packet size is the dominant deciding factor among many others in deciding the flit width. Based on the message size, there are two design choices, if we want to keep the size of each packet small, then the number of packets must increase, hence, increasing the traffic. The alternative option is to keep the size of the packet large and make lesser transactions. So, based on the size of the packets, the width of the physical link between two routers have to be increased. Meaning, larger link width leads to more area and higher power dissipation.

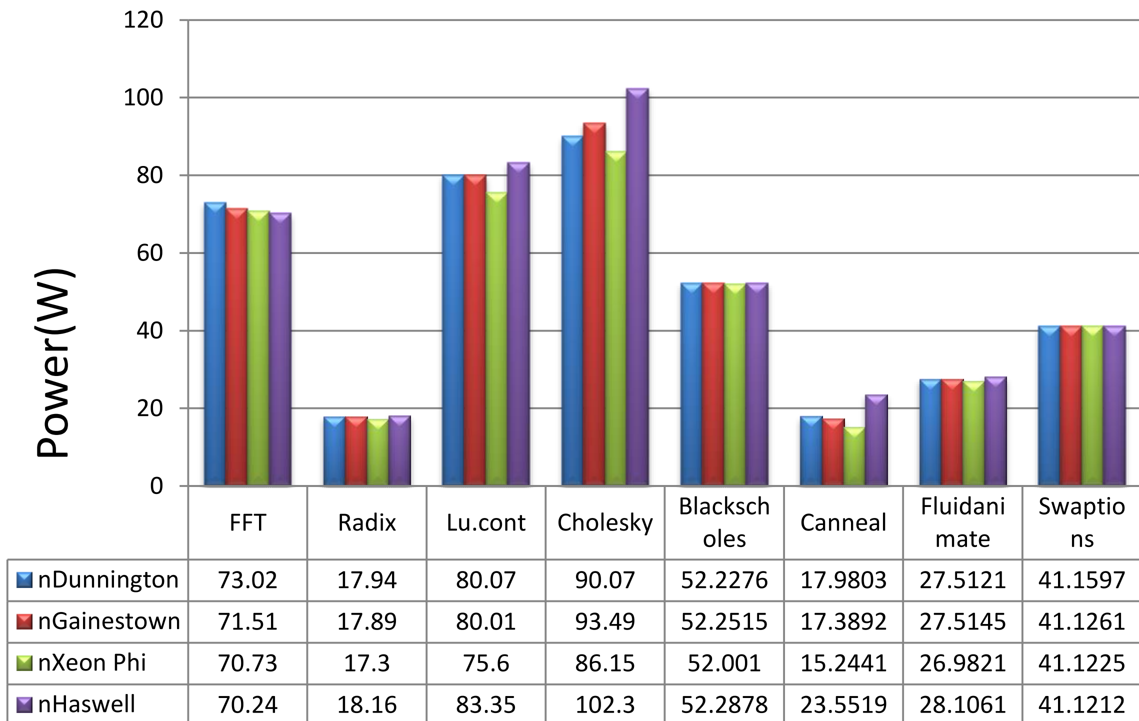
The small power dissipation in 2D mesh relates to the large (relatively) core idle time. Because the cores spend more time in waiting due to the memory stalls. Also, the lower size of link transfer unit becomes the second reason for lower power consumption. However, 2D mesh network has small FLITs relative to the ring topology. In addition, the relative smaller area of Xeon Phi leads to small power consumption because of the lack of L3 cache.

The power consumption for bus-based systems is less than the power of Haswell architecture and larger than Xeon Phi processors. Bus based systems have more time spent for synchronization issues, then minimum utilization.



	FFT	Radix	Lu.cont	Cholesky	Blackscholes	Canneal	Fluidanimate	Swaptions
nDunnington	73.26	17.69	71.45	87.12	51.654	18.927	28.1305	41.0461
nGainestown	72.66	17.83	72.79	89.41	51.636	18.591	28.1402	41.0355
nXeon Phi	64.01	17.7	71.19	85.46	51.566	15.786	27.4532	41.0583
nHaswell	89.29	18.01	74.91	98.87	51.652	25.038	28.6447	41.0498

Figure 4.39 Average runtime dynamic power for running the eight benchmark applications with the small input sizes over the eight normalized multicore design alternatives.



	FFT	Radix	Lu.cont	Cholesky	Blackscholes	Canneal	Fluidanimate	Swaptions
nDunnington	73.02	17.94	80.07	90.07	52.2276	17.9803	27.5121	41.1597
nGainestown	71.51	17.89	80.01	93.49	52.2515	17.3892	27.5145	41.1261
nXeon Phi	70.73	17.3	75.6	86.15	52.001	15.2441	26.9821	41.1225
nHaswell	70.24	18.16	83.35	102.3	52.2878	23.5519	28.1061	41.1212

Figure 4.40 Average runtime dynamic power for running the eight benchmark applications with the large input sizes over the four normalized multicore design alternatives.

The L3 cache in bus based designs plays a key role in increasing power consumption rather than nXeon Phi architecrure. More area a multicore has, more power it consumes. In fact, nGainestown consumes more power than nDunnington, due to the higher utilization of processors because that QPI links decreases the time of synchronization cycles. FFT and Canneal benchmark applications have different behaviour with large problem size. However, nHaswell consumes minimum power because it has minimum core-to-core communication and minimum synchronization waiting cycles.

An exception case appears with Swaptions and Fluidanimate. The rule that higher utilization gives higher power consumption is not present here. They have the lowest utilization, but they consume larger power than two of the other benchmarks. This behavior can be interpreted by their high core to core communication as mentioned in the previous section (they have more than 50% synchronization contribution of the CPI stack). The cores stay idle waiting each other but the whole processor works on synchronization between them. We perform Kiviat chart for the four-metrics used in performance evaluation. It summarizes all the above explanations.

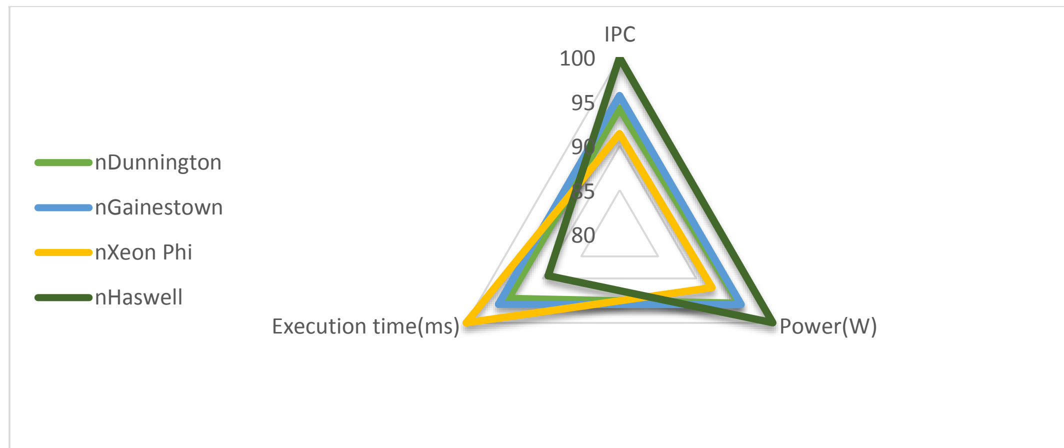


Figure 4.41. Kiviat chart for performance evaluation for the four multicore design alternatives.

4.3.6 Cache Coherence Protocol

In the MESI protocol, when a processor requests a cache line that is stored in multiple locations, every location might reply with the data. However, the requesting processor just needs a single copy of the data, so the system is wasting link bandwidth for sending extra data. The added F state changes the role of the Shared (S) state. In the MESIF protocol, only a single copy of a cache line may be in the F state and that instance is the only one that may be duplicated the cache line in the F state is used to respond to any read requests, while the S state cache lines are now silent and do not respond. By designating a single cache line to respond to requests, coherency traffic is substantially reduced. Figure 4.42 demonstrates the advantages of MESIF versus the old MESI protocol, reducing two data responses to a single response (acknowledgements are not shown). Note that a peer node is simply any node in the system that contains a cache line. Normalized Haswell does not benefit from MESIF protocol because all cores share L3 NUCA cache which has the inclusive property. Hence, any cache request will found in L3 or in DRAM not in other core caches. Swaptions and Fluidanimate have performance drop. MESIF sometimes adds relative overhead to executing some benchmarks; like Fluidanimate and Swaptions due to their low sharing degree. Figures 4.43 and 4.44 show how the MESIF cache coherence protocol enhances the system performance.

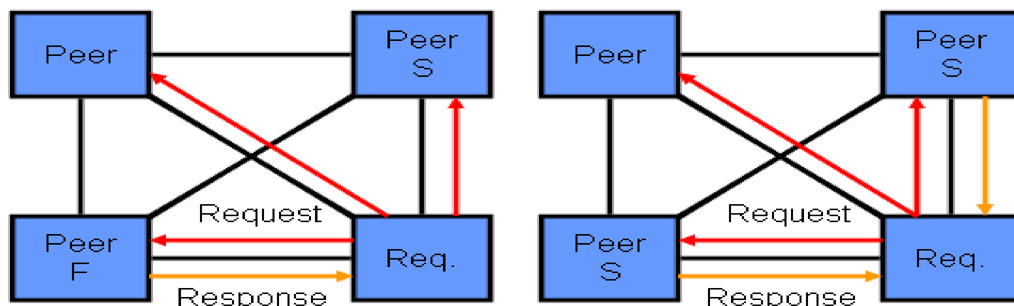


Figure 4.42. MESI versus MESIF Protocol[1].

1. http://www.realworldtech.com/common-system-interface/5/

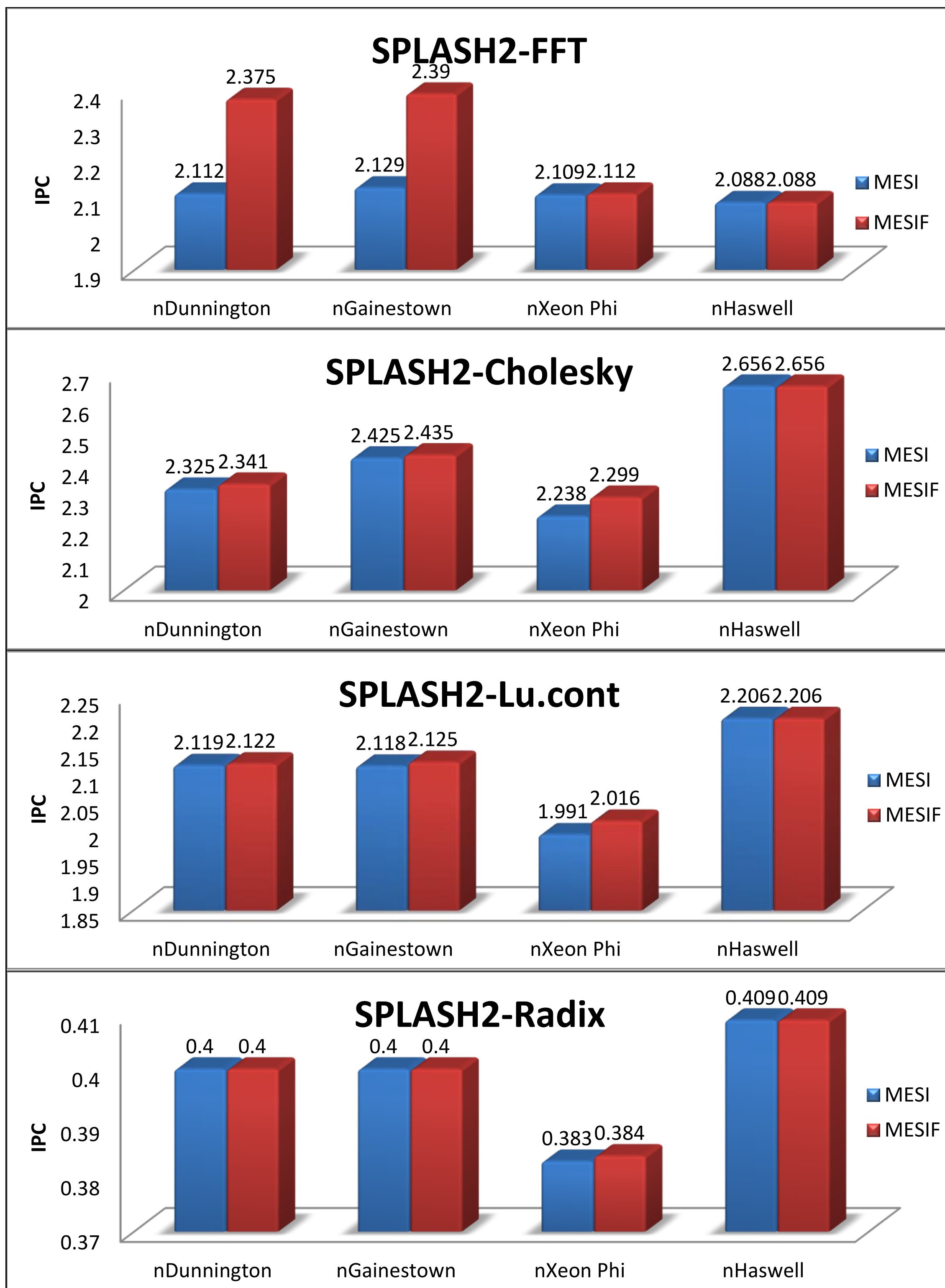


Figure 4.43 Average core IPC for running four SPLASH2 benchmark applications with large input size over the four studied multicore design alternatives with MESI/MESIF cache coherence protocols.

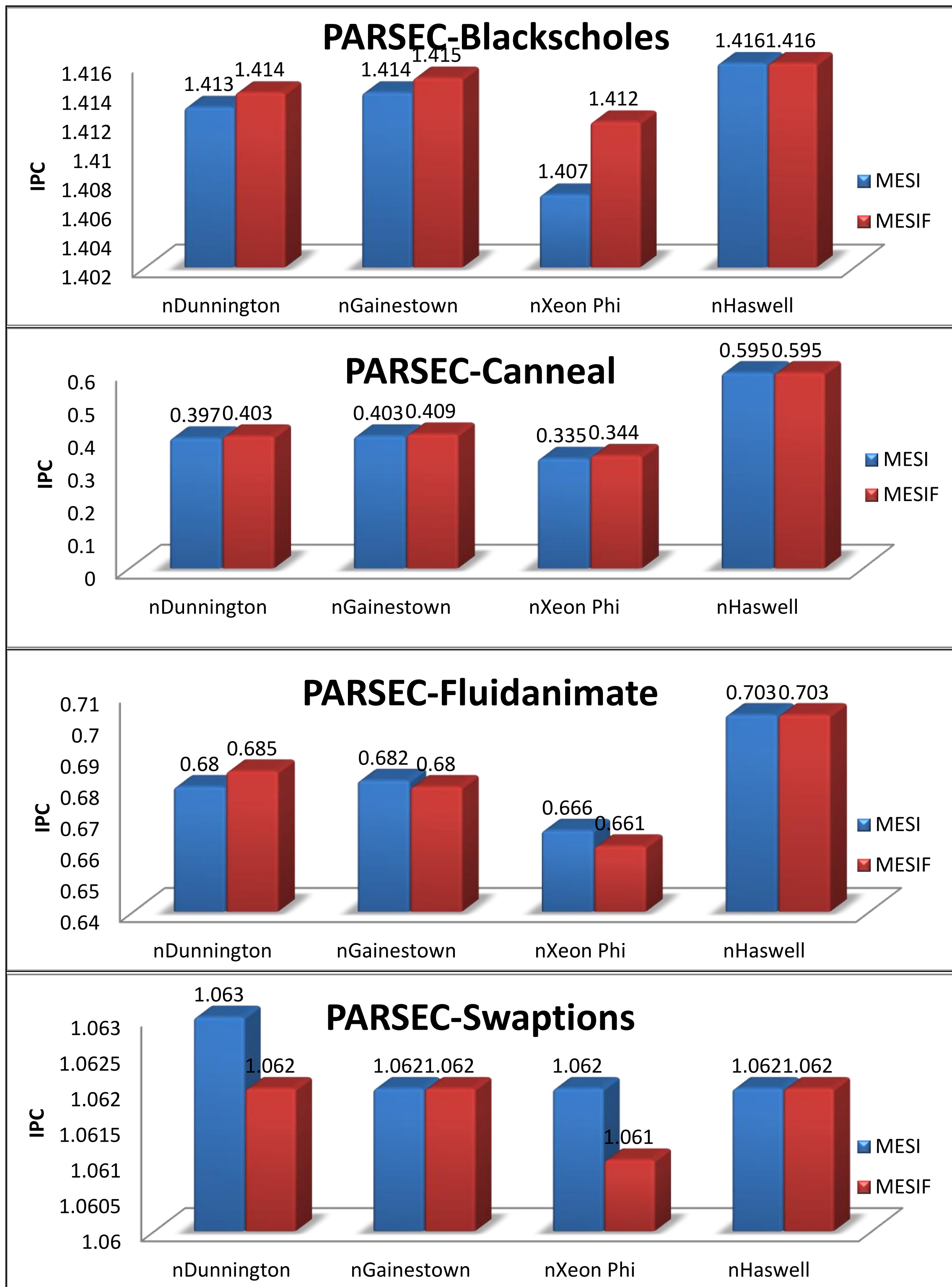


Figure 4.44 Average core IPC for running four PARSEC benchmark applications with large input size over the four studied multicore design alternatives with MESI/MESIF cache coherence protocols.

CHAPTER 5: THESIS CONCLUSION AND FUTURE WORK

5.1 Introduction

This chapter presents the conclusions regarding the thesis's methodology, raw and normalized multicores comparisons, and results analyses. Also, it presents some proposed future work.

5.2 Conclusions

The purpose of this thesis is to evaluate the performance of commodity multicore processors and find the strengths and weaknesses of their design features to help designers to develop multicore applications and design new processors. To achieve this goal, first, we chose four commercial multicore processors from Intel's server list (Xeon brand), two of them are bus based multi-socket architectures (Core2 and Nehalem based multicores). And two are NoC based architectures (2d mesh and bidirectional Ring interconnection network) where they are Haswell and Xeon Phi based processors. They were chosen because they cover wide range of recent multicore design options.

Second, we chose a set of parallel applications that are representative of multi-core applications and are widely used in recent multi-core research. This set consists of eight applications from two benchmark suits. Four of these applications are from SPLASH suite, where they are Radix, FFT, LU, and Cholesky. The other four are from PARSEC suite, which are Canneal, Blackscholes, Fluidanimate, and Swaptions. These applications were selected because they represent a wide range of applications and are often used in multi-core research. To study the impact of application problem size on the communication behavior, we worked on two problem sizes of each application: small and large sizes (Table 3.2).

Third, Sniper multicore simulator is used to evaluate performance of the selected multicore design alternatives. We chose Sniper simulator because it is relatively fast and is an accurate execution driven simulator and is validated under real hardware (Carlson, et al. 2014a).

Multithreaded benchmark applications behave differently due to scaling input sets. When using a small input, most of the working set of multithreaded workloads fits in the last-level cache LLC, and the time that is spent on the compute units contributes (relatively) more to the total run time. On the other hand, with a large input, most benchmarks stress the memory hierarchy which results in a significant fraction of time spent on cache misses and off-chip DRAM accesses.

By characterizing benchmarks performance, we show how different these benchmarks are with respect to each other, some of them have steady cmponents contribution like Blackscholes and Canneal. However, the contribution of CPI component seems to be constant over all the time of simulation. Swaptions applications is computation and memory intensive workload in the first half of the execution time, then transfers to become large synchronization and communication intensive workload. This parallel application needs multicore processor design that have high level of core compute units beside high speed communication and efficient memory designs. Memory contribution on CPI stack of some benchmarks gradually decreases over time due to its high percent of communication slack; like Cholesky and Lu.cont benchmarks.

Most benchmarks have good scaling feature and make use of all processor cores except Fluidanimate benchmarks. However, it uses only five cores out of eight cores in all multicore alternatives. Another load imbalance appers in using initial core for benchmarks. Blackscholes, Fluidanimate, Canneal, and Swaptions benchmarks have this initial thread with nearly 99.9% idle time. Cholesky, FFT, Lu.cont, and Radix have good load balancing.

Using cycle stacks provides excellent indication on how design bottlenecks change as multicore configurations changes. As we change sytem design features, the contributions of individual cycle components vary significantly. From nHaswell and nXeon Phi to bus-based microarchitectures (nDunnington and nGainestown) an interesting system strengths and weaknesses are concluded.

We found that normalized Haswell exhibits better performance in forms of execution time (69 ms) and system throughput (1.39 IPC) averaged over the eight multithreaded benchmarks for the large data sets. This relatively high performance is probably due to the following architectural features: private level 2 cache, large level 3 shared non-uniform cache access (NUCA). And the high-speed core-to-core communication through the bidirectional ring NoC. On the other hand, it relatively consumes large power (52.3 watts).

We concluded that the bus-based microarchitectures are no longer able to meet the requirements of new HPC workloads due to the obvious weakness in their handling of the synchronization and communication overheads, which for sure will increase in future many-core architectures. The normalized Gainestown and normalized Dunnington have average execution times of 74 ms and 73 ms, respectively and have average throughput of 1.33 IPC and 1.31 IPC, respectively. Also, they consume power equal to 50.14 watts and 49.99 watts on average, respectively.

When analyzing nXeon Phi architecture, we concluded that although it shows lower power consumptions (48.14 watts). Designers should do further research in developing its memory components. Xeon Phi suffers relatively from larger CPI loss in memory intensive applications (1.27 IPC) leading to larger execution times (77.25 ms on average). Most of the performance bottleneck concentrates on the off-chip DRAM access and smaller in core units.

Finally, we have shown that the Modified Exclusive Shared Invalid Forward (MESIF) cache coherence protocol enhances the multicore performance, compared with the older MESI protocol. Normalized Dunnington has speedup of 1.028x. normalized Gainestown has 1.027x of speed up. Normalized Xeon Phi has 1.01x speed up. But normalized haswell does not benefit from MESIF protocol because it has only one socket and all cores share NUCA cache. MESIF sometimes adds relative overhead to executing some benchmarks; like Fluidanimate and Swaptions, because of their low sharing degree.

5.3 Future work

In our work, we did microarchitectural simulations for common Intel multicore processors. Therefore, the future work is to do performance evaluations for other multicore processors from other vendors like AMD, ARM, and Nvidia, etc.

In addition, we plan to develop Sniper multicore simulator to support MOESI cache coherence protocol which it is the recent cache coherence protocol for AMD multicore processors. The workload which is used in this research is multithreaded benchmark applications. So, one of the future works is to use multi-program workload in simulations (Multi multithreaded workload).

The studied multicore processors are homogenuous processors, that mean all cores have the same specifications, but there is a new open research studies about hetrogenuous multicore designs, which is a good future work. The hyper multithreading SMT property and Intel over clocking are implemented in recent multicore processors. Hence, adding these features to the simulations will be a future work.

Finally, Sniper supports simulating many core processors from 10 to 100 cores. We plan to study our design alternatives with more than eight cores and evaluate their performance. By so, we can examine their scalability bottlenecks.

APPENDIX A: USAGE INSTRUCTIONS

All simulator files and the studied applications are put in one compressed file, which is called `Sniper.tar.gz`. Extract the compressed file in the home directory. The `Sniper` directory contains three directories, which are `Sniper files`, `pin_kit`, `Boost`, and `Benchmarks`, which contains both PARSEC and SPLASH suites.

A.1 Environment Setup

First, the environment should be prepared to run Sniper successfully. Make sure the required libraries shown in Table A.1 are installed. This table specifies the required libraries and how to install them.

Table A.1. The required libraries.

Library	Method of installation
g++	`sudo apt-get install g++`
x11	`sudo apt-get install libx11-dev`
zlib1g	`sudo apt-get install zlib1g-dev`
libbz2	`sudo apt-get install libbz2-dev`
libsqlite3	`sudo apt-get install libsqlite3-dev`
Libboost	`sudo apt-get install libboost-dev`
xsltproc	`sudo apt-get install xsltproc`
Libxmu	`sudo apt-get install libxmu-dev`
gfortran	`sudo apt-get install gfortran`
Expat	`sudo apt-get install libexpat1-dev`
Xt	`sudo apt-get install libxt-dev`

<table>
<tr><td>Xext</td><td><code>sudo apt-get install libxext-dev</code></td></tr>
<tr><td>Xmu</td><td><code>sudo apt-get install libxmu-dev</code></td></tr>
<tr><td>Xi</td><td><code>sudo apt-get install libxi-dev</code></td></tr>
<tr><td>m4</td><td><code>sudo apt-get install m4</code></td></tr>
<tr><td>perl5</td><td>1- Download perl, perl-base, and perl-module of version 5.14.2-21 from
https://launchpad.net/ubuntu/raring/amd64/perl/5.14.2-21
2- Force install by "<code>sudo dpkg --force-all -i perl*</code>",
Where you must run it from the same downloaded files directory.
Note: If system deny downgrade perl package, type in terminal:
<code>sudo rm /var/lib/dpkg/lock</code> then downgrade perl.</td></tr>
<tr><td>Boost</td><td>1- Download Boost version 1_59_0 from the link below
http://sourceforge.net/projects/boost/files/boost/1.59.0/
2- Extract the downloaded package in the Sniper directory.</td></tr>
<tr><td>gnuplot</td><td>1- Download gnuplot-5.0.2 from the link below
https://sourceforge.net/projects/gnuplot/files/
2- Extract and Install it by the print following commands:
<code>$ tar xzf gnuplot-5.0.1.tar.gz</code>
<code>$ cd gnuplot-5.0.1</code>
<code>$ ./configure</code>
<code>$ make</code>
<code>$ sudo make install</code></td></tr>
</table>

A.2 Downloading Sniper

Sniper simulator is an open source, so we request the the download link for the latest version `Sniper 6.1` from their website `http://www.Snipersim.org/w/Download`

Then extract it in the home directory by running the following commands.

```
Cd Downloads
```

```
Wget < download link you got by mail>
```

A.3 PIN Installation

First, download pin, binary instrumentation tool, version pin-2.14-71313-gcc.4.4.7-linux, from this link https://software.intel.com/en-us/articles/pin-a-binary-instrumentation-tool-downloads

Extract the downloaded pin tool in a folder `pin_kit` then copy to a Sniper directory, for note; we now have 2 folders in you Sniper directory; `pin` and `pin_kit`.

A.4 Compiling Sniper

Now, the environment is ready for installing Sniper, install it by running the following commands:

```
cd Sniper
```

`~/Sniper$ Make -j 2` (to take advantage of parallel simulation) here our host machine has 2 cores.

Next, you can verify your installation by running a small test Application.

```
~/Sniper$ cd test/fft
```

```
~/Sniper/test/fft$ make run
```

A.5 Benchmarks Downloading, Installation and Building

Sniper is compatible with **SPLASH2** and **PARSEC**, the selected multithreaded applications, to downloading and building the benchmarks, run the following command:

```
cd Sniper
```

```
~/Sniper$ wget http://Snipersim.org/packages/Sniper-benchmarks.tbz
```

Extract the benchmarks compressed file:

```
~/Sniper$ tar xjf Sniper-benchmarks.tbz
```

Enter the banchmarks folder and set the roots:

```
~/Sniper$ cd benchmarks
```

```
~/Sniper/benchmarks$ export GRAPHITE_ROOT=/path/to/Sniper
```

Here, you should write your Sniper folder path, an example of my Sniper path is:

`/home/aiesha/Sniper` instead of `/path/to/Sniper`

```
~/Sniper/benchmarks$ export BENCHMARKS_ROOT=$(pwd
~/Sniper/benchmarks$ make
```

SPLASH2 has four input sets: tiny, small, test (this is the defualt input size for all benchmarks applications), and large (this is the best to show the performance of multicores) on the other hand, PARSEC has simsmall, test, simdev, simlarge, simmedium sizes. Table A.2 show the studied eight multithreaded applications from the two selected representative benchmarks SPLASH2 and PARSEC, and also, there input sets.

Table A.2. The names of the studied benchmarks applications and input sets.

Suite	Benchmarks Applications	Input sets	
SPLASH2	Radix	Small	large
	Fft	Small	large
	lu.cont	Small	large
	Cholesky	Small	large
PARSEC	Canneal	Simsmall	simmeduim
	Blackscholes	Simsmall	simmeduim
	Fluidanimate	Simsmall	simmeduim
	Swaptions	Simsmall	simmeduim

A.6 Running Simulations

Sniper supports python user configuration files which passing configuration options to the simulations. We use four configuration files original and normalized design alternatives, these files in Appendix B., For example, to characterize FFT on eight threads and using problem Size "small" on Haswell microarchitecture, providing the power and visualization option, run the following command.

```
cd Sniper/benchmarks

~/Sniper/benchmarks$./run-Sniper -p SPLASH2-fft -i small -n 8 –viz --
power -c haswell
```

APPENDIX B: Sniper configuration files

Core2/Dunnington Microarchitecture	Nehalem core/Gainestown Microarchitecture	Haswell Microarchitecture	Xeon Phi Microarchitecture
[bbv] sampling = 0 [caching_protocol] type = "parametric_DRAM_directory_msi" variant = "mesi" [clock_skew_minimization] report = "false" scheme = "barrier" [clock_skew_minimization/barrier] quantum = 100 [core] spin_loop_detection = "false" [core/cheetah] enabled = "false" max_size_bits_global = 36 max_size_bits_local = 30 min_size_bits = 10 [core/hook_periodic_ins] ins_global = 1000000 ins_per_core = 10000 [core/light_cache] num = 0 [dvfs] transition_latency = 10000 type = "simple" [dvfs/simple] cores_per_socket = 4 [fault_injection] injector = "none" type = "none" [general] enable_icache_modeling = "true" enable_pinplay = "false" enable_signals = "false" enable_smc_support = "false" enable_syscall_emulation = "true" inst_mode_end = "fast_forward" inst_mode_init = "cache_only" inst_mode_output = "true" inst_mode_roi = "detailed" issue_memops_at_functional = "false" magic = "true"	[bbv] sampling = 0 [caching_protocol] type = "parametric_DRAM_directory_msi" variant = "mesi" [clock_skew_minimization] report = "false" scheme = "barrier" [clock_skew_minimization/barrier] quantum = 100 [core] spin_loop_detection = "false" [core/cheetah] enabled = "false" max_size_bits_global = 36 max_size_bits_local = 30 min_size_bits = 10 [core/hook_periodic_ins] ins_global = 1000000 ins_per_core = 10000 [core/light_cache] num = 0 [dvfs] transition_latency = 2000 type = "simple" [dvfs/simple] cores_per_socket = 1 [fault_injection] injector = "none" type = "none" [general] enable_icache_modeling = "true" enable_pinplay = "false" enable_signals = "false" enable_smc_support = "false" enable_syscall_emulation = "true" inst_mode_end = "fast_forward" inst_mode_init = "cache_only" inst_mode_output = "true" inst_mode_roi = "detailed" issue_memops_at_functional = "false"	[bbv] sampling = 0 [caching_protocol] type = "parametric_DRAM_directory_msi" variant = "mesif" [clock_skew_minimization] report = "false" scheme = "barrier" [clock_skew_minimization/barrier] quantum = 100 [core] spin_loop_detection = "false" [core/cheetah] enabled = "false" max_size_bits_global = 36 max_size_bits_local = 30 min_size_bits = 10 [core/hook_periodic_ins] ins_global = 1000000 ins_per_core = 10000 [core/light_cache] num = 0 [dvfs] transition_latency = 2000 type = "simple" [dvfs/simple] cores_per_socket = 1 [fault_injection] injector = "none" type = "none" [general] enable_icache_modeling = "true" enable_pinplay = "false" enable_signals = "false" enable_smc_support = "false" enable_syscall_emulation = "true" inst_mode_end = "fast_forward" inst_mode_init = "cache_only" inst_mode_output = "true" inst_mode_roi = "detailed" issue_memops_at_functional = "false" magic = "true"	[bbv] sampling = 0 [caching_protocol] type = "parametric_DRAM_directory_msi" variant = "mesi" [clock_skew_minimization] report = "false" scheme = "barrier" [clock_skew_minimization/barrier] quantum = 100 [core] spin_loop_detection = "false" [core/cheetah] enabled = "false" max_size_bits_global = 36 max_size_bits_local = 30 min_size_bits = 10 [core/hook_periodic_ins] ins_global = 1000000 ins_per_core = 10000 [core/light_cache] num = 0 [dvfs] transition_latency = 2000 type = "simple" [dvfs/simple] cores_per_socket = 1 [fault_injection] injector = "none" type = "none" [general] enable_icache_modeling = "true" enable_pinplay = "false" enable_signals = "false" enable_smc_support = "false" enable_syscall_emulation = "true" inst_mode_end = "fast_forward" inst_mode_init = "cache_only" inst_mode_output = "true" inst_mode_roi = "detailed" issue_memops_at_functional = "false" magic = "true"

<table>
<tr>
<td>num_host_cores = 0

roi_script = "false"
suppress_stderr = "false"
suppress_stdout = "false"
syntax = "intel"
total_cores = 8

[hooks]

[instruction_tracer]
type = "none"

[log]
circular_log = "false"
disabled_modules = ""
enabled = "false"
enabled_modules = ""
mutex_trace = "false"
pin_codecache_trace = "false"
stack_trace = "false"

[loop_tracer]
iter_count = 36
iter_start = 0

[network]
collect_traffic_matrix = "false"
memory_model_1 = "bus"
memory_model_2 = "bus"
system_model = "magic"

[network/bus]
bandwidth = 8
ignore_local_traffic = "false"

[network/bus/queue_model]
type = "contention"

[network/emesh_hop_by_hop]
concentration = 1
dimensions = 2
hop_latency = 2
link_bandwidth = 64
size = ""
wrap_around = "false"

[network/emesh_hop_by_hop/broadcast_tree]
enabled = "false"

[network/emesh_hop_by_hop/queue_model]
enabled = "true"
type = "history_list"

[network/emesh_hop_counter]
hop_latency = 2
link_bandwidth = 64

[osemu]</td>
<td>magic = "true"
num_host_cores = 0

roi_script = "false"
suppress_stderr = "false"
suppress_stdout = "false"
syntax = "intel"
total_cores = 8

[hooks]

[instruction_tracer]
type = "none"

[log]
circular_log = "false"
disabled_modules = ""
enabled = "false"
enabled_modules = ""
mutex_trace = "false"
pin_codecache_trace = "false"
stack_trace = "false"

[loop_tracer]
iter_count = 36
iter_start = 0

[network]
collect_traffic_matrix = "false"
memory_model_1 = "bus"
memory_model_2 = "bus"
system_model = "magic"

[network/bus]
bandwidth = 25.6
ignore_local_traffic = "true"

[network/bus/queue_model]
type = "contention"

[network/emesh_hop_by_hop]
concentration = 1
dimensions = 2
hop_latency = 2
link_bandwidth = 64
size = ""
wrap_around = "false"

[network/emesh_hop_by_hop/broadcast_tree]
enabled = "false"

[network/emesh_hop_by_hop/queue_model]
enabled = "true"
type = "history_list"

[network/emesh_hop_counter]
hop_latency = 2
link_bandwidth = 64</td>
<td>num_host_cores = 0

roi_script = "false"
suppress_stderr = "false"
suppress_stdout = "false"
syntax = "intel"
total_cores = 8

[hooks]

[instruction_tracer]
type = "none"

[log]
circular_log = "false"
disabled_modules = ""
enabled = "false"
enabled_modules = ""
mutex_trace = "false"
pin_codecache_trace = "false"
stack_trace = "false"

[loop_tracer]
iter_count = 36
iter_start = 0

[network]
collect_traffic_matrix = "false"
memory_model_1 = "emesh_hop_by_hop"
system_model = "magic"

[network/bus]
ignore_local_traffic = "true"

[network/bus/queue_model]
type = "contention"

[network/emesh_hop_by_hop]
concentration = 1
dimensions = 1
hop_latency = 2
link_bandwidth = 80
size = ""
wrap_around = "true"

[network/emesh_hop_by_hop/broadcast_tree]
enabled = "false"

[network/emesh_hop_by_hop/queue_model]
enabled = "true"
type = "windowed_mg1"

[network/emesh_hop_counter]
hop_latency = 2
link_bandwidth = 64

[osemu]
clock_replace = "true"</td>
<td>num_host_cores = 0

roi_script = "false"
suppress_stderr = "false"
suppress_stdout = "false"
syntax = "intel"
total_cores = 8

[hooks]

[instruction_tracer]
type = "none"

[log]
circular_log = "false"
disabled_modules = ""
enabled = "false"
enabled_modules = ""
mutex_trace = "false"
pin_codecache_trace = "false"
stack_trace = "false"

[loop_tracer]
iter_count = 36
iter_start = 0

[network]
collect_traffic_matrix = "false"
memory_model_1 = "emesh_hop_by_hop"
system_model = "magic"

[network/bus]
ignore_local_traffic = "true"

[network/bus/queue_model]
type = "contention"

[network/emesh_hop_by_hop]
concentration = 1
dimensions = 2
hop_latency = 4
link_bandwidth = 256
size = "4:2"
wrap_around = "false"

[network/emesh_hop_by_hop/broadcast_tree]
enabled = "false"

[network/emesh_hop_by_hop/queue_model]
enabled = "true"
type = "windowed_mg1"

[network/emesh_hop_counter]
hop_latency = 2
link_bandwidth = 64

[osemu]
clock_replace = "true"</td>
</tr>
</table>

```
clock_replace = "true"
nprocs = 0
pthread_replace = "false"
time_start = 1337000000

[perf_model]

[perf_model/branch_predictor]
mispredict_penalty = 15
size = 1024
type = "pentium_m"

[perf_model/cache]
levels = 3

[perf_model/core]
core_model = "nehalem"
frequency = 2.666
logical_cpus = 1
type = "interval"

[perf_model/core/interval_timer]
dispatch_width = 4
issue_contention = "true"
issue_memops_at_dispatch = "false"
lll_cutoff = 30
lll_dependency_granularity = 64
memory_dependency_granularity = 8
num_outstanding_loadstores = 8
window_size = 96

[perf_model/core/static_instruction_c
osts]
add = 1
branch = 1
delay = 0
div = 18
dynamic_misc = 1
fadd = 3
fdiv = 6
fmul = 5
fsub = 3
generic = 1
jmp = 1
mem_access = 0
mul = 3
recv = 1
spawn = 0
string = 1
sub = 1
sync = 0
tlb_miss = 0
unknown = 0

[perf_model/DRAM]
controller_positions = ""
controllers_interleaving = 4
direct_access = "false"
latency = 173
num_controllers = -1
per_controller_bandwidth = 2.5
type = "constant"
```

```
[osemu]
clock_replace = "true"
nprocs = 0
pthread_replace = "false"
time_start = 1337000000

[perf_model]

[perf_model/branch_predictor]
mispredict_penalty = 8
size = 1024
type = "pentium_m"

[perf_model/cache]
levels = 3

[perf_model/core]
core_model = "nehalem"
frequency = 2.66
logical_cpus = 1
type = "rob"

[perf_model/core/interval_timer]
dispatch_width = 4
issue_contention = "true"
issue_memops_at_dispatch = "false"
lll_cutoff = 30
lll_dependency_granularity = 64
memory_dependency_granularity = 8
num_outstanding_loadstores = 10
window_size = 128

[perf_model/core/rob_timer]
address_disambiguation = "true"
commit_width 4
in_order = "false"
issue_contention = "true"
issue_memops_at_issue = "true"
mlp_histogram = "false"
outstanding_loads = 48
outstanding_stores = 32
rob_repartition = "true"
rs_entries = 36
simultaneous_issue = "true"
store_to_load_forwarding = "true"

[perf_model/core/static_instruction_c
osts]
add = 1
branch = 1
delay = 0
div = 18
dynamic_misc = 1
fadd = 3
fdiv = 6
fmul = 5
fsub = 3
generic = 1
jmp = 1
mem_access = 0
mul = 3
recv = 1
```

```
nprocs = 0
pthread_replace = "false"
time_start = 1337000000

[perf_model]

[perf_model/branch_predictor]
mispredict_penalty = 14
size = 4096
type = "pentium_m"

[perf_model/cache]
levels = 2

[perf_model/core]
core_model = "nehalem"
frequency = 3
logical_cpus = 1
type = "rob"

[perf_model/core/interval_timer]
dispatch_width = 4
issue_contention = "true"
issue_memops_at_dispatch = "false"
lll_cutoff = 30
lll_dependency_granularity = 64
memory_dependency_granularity = 8
num_outstanding_loadstores = 72
window_size = 192

[perf_model/core/rob_timer]
address_disambiguation = "true"
commit_width = 4
in_order = "false"
issue_contention = "true"
issue_memops_at_issue = "true"
mlp_histogram = "false"
outstanding_loads = 72
outstanding_stores = 42
rob_repartition = "true"
rs_entries = 60
simultaneous_issue = "true"
store_to_load_forwarding = "true"

[perf_model/core/static_instruction_c
osts]
add = 1
branch = 1
delay = 0
div = 10
dynamic_misc = 1
fadd = 5
fdiv = 10
fmul = 10
fsub = 5
generic = 1
jmp = 1
mem_access = 0
mul = 10
recv = 1
spawn = 0
string = 1
```

```
nprocs = 0
pthread_replace = "false"
time_start = 1337000000

[perf_model]

[perf_model/branch_predictor]
mispredict_penalty = 5
size = 1024
type = "pentium_m"

[perf_model/cache]
levels = 2

[perf_model/core]
core_model = "nehalem"
frequency = 1.0
logical_cpus = 1
type = "interval"

[perf_model/core/interval_timer]
dispatch_width = 2
issue_contention = "true"
issue_memops_at_dispatch = "false"
lll_cutoff = 30
lll_dependency_granularity = 64
memory_dependency_granularity = 8
num_outstanding_loadstores = 8
window_size = 64

[perf_model/core/static_instruction_c
osts]
add = 1
branch = 1
delay = 0
div = 18
dynamic_misc = 1
fadd = 3
fdiv = 6
fmul = 5
fsub = 3
generic = 1
jmp = 1
mem_access = 0
mul = 3
recv = 1
spawn = 0
string = 1
sub = 1
sync = 0
tlb_miss = 0
unknown = 0

[perf_model/DRAM]
controller_positions = ""
controllers_interleaving = 0
direct_access = "false"
latency = 80
num_controllers = 1
per_controller_bandwidth = 32
type = "constant"
```

```
[perf_model/DRAM/cache]
enabled = "false"

[perf_model/DRAM/normal]
standard_deviation = 0

[perf_model/DRAM/queue_model]
enabled = "true"
type = "history_list"

[perf_model/DRAM_directory]
associativity = 16
directory_cache_access_time = 10
directory_type = "full_map"
home_lookup_param = 6
interleaving = 1
locations = "DRAM"
max_hw_sharers = 64
total_entries = 1048576

[perf_model/DRAM_directory/limitless]
software_trap_penalty = 200

[perf_model/dtlb]
associativity = 1
size = 0

[perf_model/fast_forward]
model = "oneipc"

[perf_model/fast_forward/oneipc]
include_branch_misprediction = "false"
include_memory_latency = "false"
interval = 100000

[perf_model/itlb]
associativity = 1
size = 0

[perf_model/l1_dcache]
address_hash = "mask"
associativity = 8
cache_block_size = 64
cache_size = 32
data_access_time = 3
dvfs_domain = "core"
next_level_read_bandwidth = 0
outstanding_misses = 0
passthrough = "false"
perf_model_type = "parallel"
perfect = "false"
prefetcher = "none"
replacement_policy = "lru"
shared_cores = 1
tags_access_time = 1
writeback_time = 150
writethrough = 0

[perf_model/l1_dcache/atd]
```

```
spawn = 0
string = 1
sub = 1
sync = 0
tlb_miss = 0
unknown = 0

[perf_model/DRAM]
chips_per_dimm = 8
controller_positions = ""
controllers_interleaving = 4
dimms_per_controller = 4
direct_access = "false"
latency = 45
num_controllers = -1
per_controller_bandwidth = 7.6
type = "constant"

[perf_model/DRAM/cache]
enabled = "false"

[perf_model/DRAM/normal]
standard_deviation = 0

[perf_model/DRAM/queue_model]
enabled = "true"
type = "history_list"

[perf_model/DRAM_directory]
associativity = 16
directory_cache_access_time = 10
directory_type = "full_map"
home_lookup_param = 6
interleaving = 1
locations = "DRAM"
max_hw_sharers = 64
total_entries = 1048576

[perf_model/DRAM_directory/limitless]
software_trap_penalty = 200

[perf_model/dtlb]
associativity = 4
size = 64

[perf_model/fast_forward]
model = "oneipc"

[perf_model/fast_forward/oneipc]
include_branch_misprediction = "false"
include_memory_latency = "false"
interval = 100000

[perf_model/itlb]
associativity = 4
size = 128

[perf_model/l1_dcache]
address_hash = "mask"
associativity = 8
```

```
sub = 1
sync = 0
tlb_miss = 0
unknown = 0

[perf_model/DRAM]
chips_per_dimm = 1
controller_positions = ""
controllers_interleaving = 8
dimms_per_controller = 4
direct_access = "false"
latency = 45
num_controllers = -1
per_controller_bandwidth = 68
type = "constant"

[perf_model/DRAM/cache]
enabled = "false"

[perf_model/DRAM/normal]
standard_deviation = 0

[perf_model/DRAM/queue_model]
enabled = "true"
type = "history_list"

[perf_model/DRAM_directory]
associativity = 16
directory_cache_access_time = 10
directory_type = "full_map"
home_lookup_param = 6
interleaving = 1
locations = "llc"
max_hw_sharers = 64
total_entries = 1048576

[perf_model/DRAM_directory/limitless]
software_trap_penalty = 200

[perf_model/dtlb]
associativity = 4
size = 64

[perf_model/fast_forward]
model = "none"

[perf_model/fast_forward/oneipc]
include_branch_misprediction = "false"
include_memory_latency = "false"
interval = 100000

[perf_model/itlb]
associativity = 4
size = 128

[perf_model/l1_dcache]
address_hash = "mask"
associativity = 8
cache_block_size = 64
cache_size = 32
```

```
[perf_model/DRAM/cache]
enabled = "false"

[perf_model/DRAM/normal]
standard_deviation = 0

[perf_model/DRAM/queue_model]
enabled = "true"
type = "windowed_mg1"

[perf_model/DRAM_directory]
associativity = 64
directory_cache_access_time = 10
directory_type = "full_map"
home_lookup_param = 6
interleaving = 1
locations = "llc"
max_hw_sharers = 64
total_entries = 1048576

[perf_model/DRAM_directory/limitless]
software_trap_penalty = 200

[perf_model/dtlb]
associativity = 1
size = 0

[perf_model/fast_forward]
model = "oneipc"

[perf_model/fast_forward/oneipc]
include_branch_misprediction = "false"
include_memory_latency = "false"
interval = 100000

[perf_model/itlb]
associativity = 1
size = 0

[perf_model/l1_dcache]
address_hash = "mask"
associativity = 8
cache_block_size = 64
cache_size = 32
data_access_time = 3
dvfs_domain = "core"
next_level_read_bandwidth = 0
outstanding_misses = 0
passthrough = "false"
perf_model_type = "parallel"
perfect = "false"
prefetcher = "none"
replacement_policy = "lru"
shared_cores = 1
tags_access_time = 1
writeback_time = 0
writethrough = 0

[perf_model/l1_dcache/atd]
```

```
[perf_model/l1_icache]
address_hash = "mask"
associativity = 8
cache_block_size = 64
cache_size = 32
coherent = "true"
data_access_time = 3
dvfs_domain = "core"
next_level_read_bandwidth = 0
passthrough = "false"
perf_model_type = "parallel"
perfect = "false"
prefetcher = "none"
replacement_policy = "lru"
shared_cores = 1
tags_access_time = 1
writeback_time = 0
writethrough = 0

[perf_model/l1_icache/atd]

[perf_model/l2_cache]
address_hash = "mod"
associativity = 12
cache_block_size = 64
cache_size = 3072
data_access_time = 14
dvfs_domain = "core"
next_level_read_bandwidth = 0
passthrough = "false"
perf_model_type = "parallel"
perfect = "false"
prefetcher = "none"
replacement_policy = "lru"
shared_cores = 2
tags_access_time = 3
writeback_time = 60
writethrough = 0

[perf_model/l2_cache/atd]

[perf_model/l3_cache]
address_hash = "mask"
associativity = 16
cache_block_size = 64
cache_size = 16384
data_access_time = 96
dvfs_domain = "global"
passthrough = "false"
perf_model_type = "parallel"
perfect = "false"
prefetcher = "none"
replacement_policy = "lru"
shared_cores = 4
tags_access_time = 10
writeback_time = 390
writethrough = 0

[perf_model/l3_cache/atd]

[perf_model/l4_cache]
```

```
cache_block_size = 64
cache_size = 32
data_access_time = 4
dvfs_domain = "core"
next_level_read_bandwidth = 0
outstanding_misses = 10
passthrough = "false"
perf_model_type = "parallel"
perfect = "false"
prefetcher = "none"
replacement_policy = "lru"
shared_cores = 1
tags_access_time = 1
writeback_time = 0
writethrough = 0

[perf_model/l1_dcache/atd]

[perf_model/l1_icache]
address_hash = "mask"
associativity = 4
cache_block_size = 64
cache_size = 32
coherent = "true"
data_access_time = 4
dvfs_domain = "core"
next_level_read_bandwidth = 0
passthrough = "false"
perf_model_type = "parallel"
perfect = "false"
prefetcher = "none"
replacement_policy = "lru"
shared_cores = 1
tags_access_time = 1
writeback_time = 0
writethrough = 0

[perf_model/l1_icache/atd]

[perf_model/l2_cache]
address_hash = "mask"
associativity = 8
cache_block_size = 64
cache_size = 256
data_access_time = 8
dvfs_domain = "core"
next_level_read_bandwidth = 0
passthrough = "false"
perf_model_type = "parallel"
perfect = "false"
prefetcher = "none"
replacement_policy = "lru"
shared_cores = 1
tags_access_time = 3
writeback_time = 50
writethrough = 0

[perf_model/l2_cache/atd]

[perf_model/l3_cache]
address_hash = "mask"
associativity = 16
```

```
data_access_time = 3
dvfs_domain = "core"
next_level_read_bandwidth = 0
outstanding_misses = 10
passthrough = "false"
perf_model_type = "parallel"
perfect = "false"
prefetcher = "none"
replacement_policy = "lru"
shared_cores = 1
tags_access_time = 1
writeback_time = 0
writethrough = 0

[perf_model/l1_dcache/atd]

[perf_model/l1_icache]
address_hash = "mask"
associativity = 8
cache_block_size = 64
cache_size = 32
coherent = "true"
data_access_time = 3
dvfs_domain = "core"
next_level_read_bandwidth = 0
passthrough = "false"
perf_model_type = "parallel"
perfect = "false"
prefetcher = "none"
replacement_policy = "lru"
shared_cores = 1
tags_access_time = 1
writeback_time = 0
writethrough = 0

[perf_model/l1_icache/atd]

[perf_model/l2_cache]
address_hash = "mask"
associativity = 8
cache_block_size = 64
cache_size = 256
data_access_time = 8
dvfs_domain = "core"
next_level_read_bandwidth = 0
passthrough = "false"
perf_model_type = "parallel"
perfect = "false"
prefetcher = "none"
replacement_policy = "lru"
shared_cores = 1
tags_access_time = 3
writeback_time = 50
writethrough = 0

[perf_model/l2_cache/atd]

[perf_model/l3_cache]
address_hash = "mask"
associativity = 16
cache_block_size = 64
cache_size = 8192
```

```
[perf_model/l1_icache]
address_hash = "mask"
associativity = 4
cache_block_size = 64
cache_size = 32
coherent = "true"
data_access_time = 3
dvfs_domain = "core"
next_level_read_bandwidth = 0
passthrough = "false"
perf_model_type = "parallel"
perfect = "false"
prefetcher = "none"
replacement_policy = "lru"
shared_cores = 1
tags_access_time = 1
writeback_time = 0
writethrough = 0

[perf_model/l1_icache/atd]

[perf_model/l2_cache]
address_hash = "mask"
associativity = 8
cache_block_size = 64
cache_size = 512
data_access_time = 22
dvfs_domain = "core"
next_level_read_bandwidth = 0
passthrough = "false"
perf_model_type = "parallel"
perfect = "false"
prefetcher = "none"
replacement_policy = "lru"
shared_cores = 1
tags_access_time = 5
writeback_time = 1
writethrough = 0

[perf_model/l2_cache/atd]

[perf_model/l3_cache]
passthrough = "false"
perfect = "false"

[perf_model/l4_cache]
passthrough = "false"
perfect = "false"

[perf_model/llc]
evict_buffers = 8

[perf_model/nuca]
enabled = "false"

[perf_model/stlb]
associativity = 1
size = 0

[perf_model/sync]
reschedule_cost = 1000
```

```
passthrough = "false"
perfect = "false"

[perf_model/llc]
evict_buffers = 8

[perf_model/nuca]
enabled = "false"

[perf_model/stlb]
associativity = 1
size = 0

[perf_model/sync]
reschedule_cost = 1000

[perf_model/tlb]
penalty = 0
penalty_parallel = "true"

[power]
technology_node = 45
vdd = 1.6

[progress_trace]
enabled = "false"
filename = ""
interval = 5000

[queue_model]

[queue_model/basic]
moving_avg_enabled = "true"
moving_avg_type =
"arithmetic_mean"
moving_avg_window_size = 1024

[queue_model/history_list]
analytical_model_enabled = "true"
max_list_size = 100

[queue_model/windowed_mg1]
window_size = 1000

[routine_tracer]
type = "none"

[sampling]
enabled = "false"

[scheduler]
type = "pinned"

[scheduler/big_small]
debug = "false"
quantum = 1000000

[scheduler/pinned]
core_mask = 1
interleaving = 1
quantum = 1000000
cache_block_size = 64
cache_size = 8192
data_access_time = 30
dvfs_domain = "global"
passthrough = "false"
perf_model_type = "parallel"
perfect = "false"
prefetcher = "none"
replacement_policy = "lru"
shared_cores = 4
tags_access_time = 10
writeback_time = 0
writethrough = 0

[perf_model/l3_cache/atd]

[perf_model/l4_cache]
passthrough = "false"
perfect = "false"

[perf_model/llc]
evict_buffers = 8

[perf_model/nuca]
enabled = "false"

[perf_model/stlb]
associativity = 4
size = 512

[perf_model/sync]
reschedule_cost = 1000

[perf_model/tlb]
penalty = 30
penalty_parallel = "true"

[power]
technology_node = 45
vdd = 1.2

[progress_trace]
enabled = "false"
filename = ""
interval = 5000

[queue_model]

[queue_model/basic]
moving_avg_enabled = "true"
moving_avg_type =
"arithmetic_mean"
moving_avg_window_size = 1024

[queue_model/history_list]
analytical_model_enabled = "true"
max_list_size = 100

[queue_model/windowed_mg1]
window_size = 1000

[routine_tracer]
data_access_time = 30
dvfs_domain = "global"
passthrough = "false"
perf_model_type = "parallel"
perfect = "false"
prefetcher = "none"
replacement_policy = "lru"
shared_cores = 8
tags_access_time = 10
writeback_time = 0
writethrough = 0

[perf_model/l4_cache]
passthrough = "false"
perfect = "false"

[perf_model/llc]
evict_buffers = 8

[perf_model/nuca]
address_hash = "mask"
associativity = 16
bandwidth = 64
cache_size = 8192
data_access_time = 30
enabled = "true"
replacement_policy = "lru"
tags_access_time = 10

[perf_model/nuca/queue_model]
enabled = "true"
type = "history_list"

[perf_model/stlb]
associativity = 4
size = 1024

[perf_model/sync]
reschedule_cost = 1000

[perf_model/tlb]
penalty = 30
penalty_parallel = "true"

[power]
technology_node = 22
vdd = 1.2

[progress_trace]
enabled = "false"
filename = ""
interval = 5000

[queue_model]

[queue_model/basic]
moving_avg_enabled = "true"
moving_avg_type =
"arithmetic_mean"
moving_avg_window_size = 1024

[queue_model/history_list]
[perf_model/tlb]
penalty = 0
penalty_parallel = "true"

[power]
technology_node = 22
vdd = 1.05

[progress_trace]
enabled = "false"
filename = ""
interval = 5000

[queue_model]

[queue_model/basic]
moving_avg_enabled = "true"
moving_avg_type =
"arithmetic_mean"
moving_avg_window_size = 1024

[queue_model/history_list]
analytical_model_enabled = "true"
max_list_size = 100

[queue_model/windowed_mg1]
window_size = 1000

[routine_tracer]
type = "none"

[sampling]
enabled = "false"

[scheduler]
type = "pinned"

[scheduler/big_small]
debug = "false"
quantum = 1000000

[scheduler/pinned]
core_mask = 1
interleaving = 1
quantum = 100

[scheduler/roaming]
core_mask = 1
quantum = 1000000

[scheduler/static]
core_mask = 1

[tags]

[traceinput]
address_randomization = "false"
enabled = "false"
mirror_output = "false"
num_runs = 1
restart_apps = "false"
stop_with_first_app = "true"
```

<table>
<tr>
<td>[scheduler/roaming]
core_mask = 1
quantum = 1000000

[scheduler/static]
core_mask = 1

[tags]

[traceinput]
address_randomization = "false"
enabled = "false"
mirror_output = "false"
num_runs = 1
restart_apps = "false"
stop_with_first_app = "true"
trace_prefix = ""</td>
<td>type = "none"

[sampling]
enabled = "false"

[scheduler]
type = "pinned"

[scheduler/big_small]
debug = "false"
quantum = 1000000

[scheduler/pinned]
core_mask = 1
interleaving = 1
quantum = 1000000

[scheduler/roaming]
core_mask = 1
quantum = 1000000

[scheduler/static]
core_mask = 1

[tags]

[traceinput]
address_randomization = "false"
enabled = "false"
mirror_output = "false"
num_runs = 1
restart_apps = "false"
stop_with_first_app = "true"
trace_prefix = ""</td>
<td>analytical_model_enabled = "true"
max_list_size = 100

[queue_model/windowed_mg1]
window_size = 1000

[routine_tracer]
type = "none"

[sampling]
enabled = "false"

[scheduler]
type = "pinned"

[scheduler/big_small]
debug = "false"
quantum = 100

[scheduler/pinned]
core_mask = 1
interleaving = 1
quantum = 100

[scheduler/roaming]
core_mask = 1
quantum = 100

[scheduler/static]
core_mask = 1

[tags]

[traceinput]
address_randomization = "false"
enabled = "false"
mirror_output = "false"
num_runs = 1
restart_apps = "false"
stop_with_first_app = "true"
trace_prefix = ""</td>
<td>trace_prefix = ""</td>
</tr>
<tr>
<td>nDunington

[bbv]
sampling = 0

[caching_protocol]
type =
"parametric_DRAM_directory_msi"
variant = "mesi"

[clock_skew_minimization]
report = "false"
scheme = "barrier"

[clock_skew_minimization/barrier]
quantum = 100

[core]
spin_loop_detection = "false"</td>
<td>nGainestown

[bbv]
sampling = 0

[caching_protocol]
type =
"parametric_DRAM_directory_msi"
variant = "mesi"

[clock_skew_minimization]
report = "false"
scheme = "barrier"

[clock_skew_minimization/barrier]
quantum = 100

[core]
spin_loop_detection = "false"</td>
<td>nHaswell

[bbv]
sampling = 0

[caching_protocol]
type =
"parametric_DRAM_directory_msi"
variant = "mesif"

[clock_skew_minimization]
report = "false"
scheme = "barrier"

[clock_skew_minimization/barrier]
quantum = 100

[core]
spin_loop_detection = "false"</td>
<td>nXeon Phi

[bbv]
sampling = 0

[caching_protocol]
type =
"parametric_DRAM_directory_msi"
variant = "mesi"

[clock_skew_minimization]
report = "false"
scheme = "barrier"

[clock_skew_minimization/barrier]
quantum = 100

[core]
spin_loop_detection = "false"</td>
</tr>
</table>

<table>
<tr>
<td>[core/cheetah]
enabled = "false"
max_size_bits_global = 36
max_size_bits_local = 30
min_size_bits = 10

[core/hook_periodic_ins]
ins_global = 1000000
ins_per_core = 10000

[core/light_cache]
num = 0

[dvfs]
transition_latency = 2000
type = "simple"

[dvfs/simple]
cores_per_socket = 1

[fault_injection]
injector = "none"
type = "none"

[general]
enable_icache_modeling = "true"
enable_pinplay = "false"
enable_signals = "false"
enable_smc_support = "false"
enable_syscall_emulation = "true"
inst_mode_end = "fast_forward"
inst_mode_init = "cache_only"
inst_mode_output = "true"
inst_mode_roi = "detailed"
issue_memops_at_functional = "false"
magic = "true"
num_host_cores = 0

roi_script = "false"
suppress_stderr = "false"
suppress_stdout = "false"
syntax = "intel"
total_cores = 8

[hooks]

[instruction_tracer]
type = "none"

[log]
circular_log = "false"
disabled_modules = ""
enabled = "false"
enabled_modules = ""
mutex_trace = "false"
pin_codecache_trace = "false"
stack_trace = "false"

[loop_tracer]
iter_count = 36
iter_start = 0</td>
<td>[core/cheetah]
enabled = "false"
max_size_bits_global = 36
max_size_bits_local = 30
min_size_bits = 10

[core/hook_periodic_ins]
ins_global = 1000000
ins_per_core = 10000

[core/light_cache]
num = 0

[dvfs]
transition_latency = 2000
type = "simple"

[dvfs/simple]
cores_per_socket = 1

[fault_injection]
injector = "none"
type = "none"

[general]
enable_icache_modeling = "true"
enable_pinplay = "false"
enable_signals = "false"
enable_smc_support = "false"
enable_syscall_emulation = "true"
inst_mode_end = "fast_forward"
inst_mode_init = "cache_only"
inst_mode_output = "true"
inst_mode_roi = "detailed"
issue_memops_at_functional =
"false"
magic = "true"
num_host_cores = 0

roi_script = "false"
suppress_stderr = "false"
suppress_stdout = "false"
syntax = "intel"
total_cores = 8

[hooks]

[instruction_tracer]
type = "none"

[log]
circular_log = "false"
disabled_modules = ""
enabled = "false"
enabled_modules = ""
mutex_trace = "false"
pin_codecache_trace = "false"
stack_trace = "false"

[loop_tracer]
iter_count = 36
iter_start = 0</td>
<td>[core/cheetah]
enabled = "false"
max_size_bits_global = 36
max_size_bits_local = 30
min_size_bits = 10

[core/hook_periodic_ins]
ins_global = 1000000
ins_per_core = 10000

[core/light_cache]
num = 0

[dvfs]
transition_latency = 2000
type = "simple"

[dvfs/simple]
cores_per_socket = 1

[fault_injection]
injector = "none"
type = "none"

[general]
enable_icache_modeling = "true"
enable_pinplay = "false"
enable_signals = "false"
enable_smc_support = "false"
enable_syscall_emulation = "true"
inst_mode_end = "fast_forward"
inst_mode_init = "cache_only"
inst_mode_output = "true"
inst_mode_roi = "detailed"
issue_memops_at_functional =
"false"
magic = "true"
num_host_cores = 0

roi_script = "false"
suppress_stderr = "false"
suppress_stdout = "false"
syntax = "intel"
total_cores = 8

[hooks]

[instruction_tracer]
type = "none"

[log]
circular_log = "false"
disabled_modules = ""
enabled = "false"
enabled_modules = ""
mutex_trace = "false"
pin_codecache_trace = "false"
stack_trace = "false"

[loop_tracer]
iter_count = 36</td>
<td>[core/cheetah]
enabled = "false"
max_size_bits_global = 36
max_size_bits_local = 30
min_size_bits = 10

[core/hook_periodic_ins]
ins_global = 1000000
ins_per_core = 10000

[core/light_cache]
num = 0

[dvfs]
transition_latency = 2000
type = "simple"

[dvfs/simple]
cores_per_socket = 1

[fault_injection]
injector = "none"
type = "none"

[general]
enable_icache_modeling = "true"
enable_pinplay = "false"
enable_signals = "false"
enable_smc_support = "false"
enable_syscall_emulation = "true"
inst_mode_end = "fast_forward"
inst_mode_init = "cache_only"
inst_mode_output = "true"
inst_mode_roi = "detailed"
issue_memops_at_functional =
"false"
magic = "true"
num_host_cores = 0

roi_script = "false"
suppress_stderr = "false"
suppress_stdout = "false"
syntax = "intel"
total_cores = 8

[hooks]

[instruction_tracer]
type = "none"

[log]
circular_log = "false"
disabled_modules = ""
enabled = "false"
enabled_modules = ""
mutex_trace = "false"
pin_codecache_trace = "false"
stack_trace = "false"

[loop_tracer]
iter_count = 36</td>
</tr>
</table>

[network] collect_traffic_matrix = "false" memory_model_1 = "bus" memory_model_2 = "bus" system_model = "magic" [network/bus] bandwidth = 256 ignore_local_traffic = "false" [network/bus/queue_model] type = "contention" [network/emesh_hop_by_hop] concentration = 1 dimensions = 2 hop_latency = 2 link_bandwidth = 64 size = "" wrap_around = "false" [network/emesh_hop_by_hop/broadcast_tree] enabled = "false" [network/emesh_hop_by_hop/queue_model] enabled = "true" type = "windowed_mg1" [network/emesh_hop_counter] hop_latency = 2 link_bandwidth = 64 [osemu] clock_replace = "true" nprocs = 0 pthread_replace = "false" time_start = 1337000000 [perf_model] [perf_model/branch_predictor] mispredict_penalty = 14 size = 4096 type = "pentium_m" [perf_model/cache] levels = 3 [perf_model/core] core_model = "nehalem" frequency = 3.2 logical_cpus = 1 type = "rob" [perf_model/core/interval_timer] dispatch_width = 4 issue_contention = "true" issue_memops_at_dispatch = "false" lll_cutoff = 30	[network] collect_traffic_matrix = "false" memory_model_1 = "bus" memory_model_2 = "bus" system_model = "magic" [network/bus] bandwidth = 256 ignore_local_traffic = "true" [network/bus/queue_model] type = "contention" [network/emesh_hop_by_hop] concentration = 1 dimensions = 2 hop_latency = 2 link_bandwidth = 64 size = "" wrap_around = "false" [network/emesh_hop_by_hop/broadcast_tree] enabled = "false" [network/emesh_hop_by_hop/queue_model] enabled = "true" type = "windowed_mg1" [network/emesh_hop_counter] hop_latency = 2 link_bandwidth = 64 [osemu] clock_replace = "true" nprocs = 0 pthread_replace = "false" time_start = 1337000000 [perf_model] [perf_model/branch_predictor] mispredict_penalty = 14 size = 4096 type = "pentium_m" [perf_model/cache] levels = 3 [perf_model/core] core_model = "nehalem" frequency = 3.2 logical_cpus = 1 type = "rob" [perf_model/core/interval_timer] dispatch_width = 4 issue_contention = "true" issue_memops_at_dispatch = "false"	iter_start = 0 [network] collect_traffic_matrix = "false" memory_model_1 = "emesh_hop_by_hop" system_model = "magic" [network/bus] ignore_local_traffic = "true" [network/bus/queue_model] type = "contention" [network/emesh_hop_by_hop] concentration = 1 dimensions = 1 hop_latency = 2 link_bandwidth = 256 size = "" wrap_around = "true" [network/emesh_hop_by_hop/broadcast_tree] enabled = "false" [network/emesh_hop_by_hop/queue_model] enabled = "true" type = "windowed_mg1" [network/emesh_hop_counter] hop_latency = 2 link_bandwidth = 64 [osemu] clock_replace = "true" nprocs = 0 pthread_replace = "false" time_start = 1337000000 [perf_model] [perf_model/branch_predictor] mispredict_penalty = 14 size = 4096 type = "pentium_m" [perf_model/cache] levels = 2 [perf_model/core] core_model = "nehalem" frequency = 3.2 logical_cpus = 1 type = "rob" [perf_model/core/interval_timer] dispatch_width = 4 issue_contention = "true" issue_memops_at_dispatch = "false"	iter_start = 0 [network] collect_traffic_matrix = "false" memory_model_1 = "emesh_hop_by_hop" system_model = "magic" [network/bus] ignore_local_traffic = "true" [network/bus/queue_model] type = "contention" [network/emesh_hop_by_hop] concentration = 1 dimensions = 2 hop_latency = 2 link_bandwidth = 256 size = "4:2" wrap_around = "false" [network/emesh_hop_by_hop/broadcast_tree] enabled = "false" [network/emesh_hop_by_hop/queue_model] enabled = "true" type = "windowed_mg1" [network/emesh_hop_counter] hop_latency = 2 link_bandwidth = 64 [osemu] clock_replace = "true" nprocs = 0 pthread_replace = "false" time_start = 1337000000 [perf_model] [perf_model/branch_predictor] mispredict_penalty = 14 size = 4096 type = "pentium_m" [perf_model/cache] levels = 2 [perf_model/core] core_model = "nehalem" frequency = 3.2 logical_cpus = 1 type = "rob" [perf_model/core/interval_timer] dispatch_width = 4 issue_contention = "true" issue_memops_at_dispatch = "false" lll_cutoff = 30

<table>
<tr>
<td>lll_dependency_granularity = 64
memory_dependency_granularity = 8
num_outstanding_loadstores = 72
window_size = 192

[perf_model/core/rob_timer]
address_disambiguation = "true"
commit_width = 4
in_order = "false"
issue_contention = "true"
issue_memops_at_issue = "true"
mlp_histogram = "false"
outstanding_loads = 72
outstanding_stores = 42
rob_repartition = "true"
rs_entries = 60
simultaneous_issue = "true"
store_to_load_forwarding = "true"

[perf_model/core/static_instruction_costs]
add = 1
branch = 1
delay = 0
div = 10
dynamic_misc = 1
fadd = 5
fdiv = 10
fmul = 10
fsub = 5
generic = 1
jmp = 1
mem_access = 0
mul = 10
recv = 1
spawn = 0
string = 1
sub = 1
sync = 0
tlb_miss = 0
unknown = 0

[perf_model/DRAM]
chips_per_dimm = 1
controller_positions = ""
controllers_interleaving = 8
dimms_per_controller = 4
direct_access = "false"
latency = 45
num_controllers = -1
per_controller_bandwidth = 68
type = "constant"

[perf_model/DRAM/cache]
enabled = "false"

[perf_model/DRAM/normal]
standard_deviation = 0

[perf_model/DRAM/queue_model]
enabled = "true"</td>
<td>lll_cutoff = 30
lll_dependency_granularity = 64
memory_dependency_granularity = 8
num_outstanding_loadstores = 72
window_size = 192
[perf_model/core/rob_timer]
address_disambiguation = "true"
commit_width = 4
in_order = "false"
issue_contention = "true"
issue_memops_at_issue = "true"
mlp_histogram = "false"
outstanding_loads = 72
outstanding_stores = 42
rob_repartition = "true"
rs_entries = 60
simultaneous_issue = "true"
store_to_load_forwarding = "true"

[perf_model/core/static_instruction_costs]
add = 1
branch = 1
delay = 0
div = 10
dynamic_misc = 1
fadd = 5
fdiv = 10
fmul = 10
fsub = 5
generic = 1
jmp = 1
mem_access = 0
mul = 10
recv = 1
spawn = 0
string = 1
sub = 1
sync = 0
tlb_miss = 0
unknown = 0

[perf_model/DRAM]
chips_per_dimm = 1
controller_positions = ""
controllers_interleaving = 8
dimms_per_controller = 4
direct_access = "false"
latency = 45
num_controllers = -1
per_controller_bandwidth = 68
type = "constant"

[perf_model/DRAM/cache]
enabled = "false"

[perf_model/DRAM/normal]
standard_deviation = 0

[perf_model/DRAM/queue_model]
enabled = "true"
type = "history_list"</td>
<td>lll_cutoff = 30
lll_dependency_granularity = 64
memory_dependency_granularity = 8
num_outstanding_loadstores = 72
window_size = 192
[perf_model/core/rob_timer]
address_disambiguation = "true"
commit_width = 4
in_order = "false"
issue_contention = "true"
issue_memops_at_issue = "true"
mlp_histogram = "false"
outstanding_loads = 72
outstanding_stores = 42
rob_repartition = "true"
rs_entries = 60
simultaneous_issue = "true"
store_to_load_forwarding = "true"

[perf_model/core/static_instruction_costs]
add = 1
branch = 1
delay = 0
div = 10
dynamic_misc = 1
fadd = 5
fdiv = 10
fmul = 10
fsub = 5
generic = 1
jmp = 1
mem_access = 0
mul = 10
recv = 1
spawn = 0
string = 1
sub = 1
sync = 0
tlb_miss = 0
unknown = 0

[perf_model/DRAM]
chips_per_dimm = 1
controller_positions = ""
controllers_interleaving = 8
dimms_per_controller = 4
direct_access = "false"
latency = 45
num_controllers = -1
per_controller_bandwidth = 68
type = "constant"

[perf_model/DRAM/cache]
enabled = "false"

[perf_model/DRAM/normal]
standard_deviation = 0

[perf_model/DRAM/queue_model]
enabled = "true"
type = "history_list"</td>
<td>lll_dependency_granularity = 64
memory_dependency_granularity = 8
num_outstanding_loadstores = 72
window_size = 192

[perf_model/core/rob_timer]
address_disambiguation = "true"
commit_width = 4
in_order = "false"
issue_contention = "true"
issue_memops_at_issue = "true"
mlp_histogram = "false"
outstanding_loads = 72
outstanding_stores = 42
rob_repartition = "true"
rs_entries = 60
simultaneous_issue = "true"
store_to_load_forwarding = "true"

[perf_model/core/static_instruction_costs]
add = 1
branch = 1
delay = 0
div = 10
dynamic_misc = 1
fadd = 5
fdiv = 10
fmul = 10
fsub = 5
generic = 1
jmp = 1
mem_access = 0
mul = 10
recv = 1
spawn = 0
string = 1
sub = 1
sync = 0
tlb_miss = 0
unknown = 0

[perf_model/DRAM]
chips_per_dimm = 1
controller_positions = ""
controllers_interleaving = 8
dimms_per_controller = 4
direct_access = "false"
latency = 45
num_controllers = -1
per_controller_bandwidth = 68
type = "constant"

[perf_model/DRAM/cache]
enabled = "false"

[perf_model/DRAM/normal]
standard_deviation = 0

[perf_model/DRAM/queue_model]
enabled = "true"</td>
</tr>
</table>

type = "history_list" [perf_model/DRAM_directory] associativity = 16 directory_cache_access_time = 10 directory_type = "full_map" home_lookup_param = 6 interleaving = 1 locations = "llc" max_hw_sharers = 64 total_entries = 1048576 [perf_model/DRAM_directory/limitless] software_trap_penalty = 200 [perf_model/dtlb] associativity = 4 size = 64 [perf_model/fast_forward] model = "oneipc" [perf_model/fast_forward/oneipc] include_branch_misprediction = "false" include_memory_latency = "false" interval = 100000 [perf_model/itlb] associativity = 4 size = 128 [perf_model/l1_dcache] address_hash = "mask" associativity = 8 cache_block_size = 64 cache_size = 32 data_access_time = 3 dvfs_domain = "core" next_level_read_bandwidth = 0 outstanding_misses = 10 passthrough = "false" perf_model_type = "parallel" perfect = "false" prefetcher = "none" replacement_policy = "lru" shared_cores = 1 tags_access_time = 1 writeback_time = 0 writethrough = 0 [perf_model/l1_dcache/atd] [perf_model/l1_icache] address_hash = "mask" associativity = 8 cache_block_size = 64 cache_size = 32 coherent = "true" data_access_time = 3 dvfs_domain = "core"	[perf_model/DRAM_directory] associativity = 16 directory_cache_access_time = 10 directory_type = "full_map" home_lookup_param = 6 interleaving = 1 locations = "llc" max_hw_sharers = 64 total_entries = 1048576 [perf_model/DRAM_directory/limitless] software_trap_penalty = 200 [perf_model/dtlb] associativity = 4 size = 64 [perf_model/fast_forward] model = "oneipc" [perf_model/fast_forward/oneipc] include_branch_misprediction = "false" include_memory_latency = "false" interval = 100000 [perf_model/itlb] associativity = 4 size = 128 [perf_model/l1_dcache] address_hash = "mask" associativity = 8 cache_block_size = 64 cache_size = 32 data_access_time = 3 dvfs_domain = "core" next_level_read_bandwidth = 0 outstanding_misses = 10 passthrough = "false" perf_model_type = "parallel" perfect = "false" prefetcher = "none" replacement_policy = "lru" shared_cores = 1 tags_access_time = 1 writeback_time = 0 writethrough = 0 [perf_model/l1_dcache/atd] [perf_model/l1_icache] address_hash = "mask" associativity = 8 cache_block_size = 64 cache_size = 32 coherent = "true" data_access_time = 3 dvfs_domain = "core"	[perf_model/DRAM_directory] associativity = 16 directory_cache_access_time = 10 directory_type = "full_map" home_lookup_param = 6 interleaving = 1 locations = "llc" max_hw_sharers = 64 total_entries = 1048576 [perf_model/DRAM_directory/limitless] software_trap_penalty = 200 [perf_model/dtlb] associativity = 4 size = 64 [perf_model/fast_forward] model = "oneipc" [perf_model/fast_forward/oneipc] include_branch_misprediction = "false" include_memory_latency = "false" interval = 100000 [perf_model/itlb] associativity = 4 size = 128 [perf_model/l1_dcache] address_hash = "mask" associativity = 8 cache_block_size = 64 cache_size = 32 data_access_time = 3 dvfs_domain = "core" next_level_read_bandwidth = 0 outstanding_misses = 10 passthrough = "false" perf_model_type = "parallel" perfect = "false" prefetcher = "none" replacement_policy = "lru" shared_cores = 1 tags_access_time = 1 writeback_time = 0 writethrough = 0 [perf_model/l1_dcache/atd] [perf_model/l1_icache] address_hash = "mask" associativity = 8 cache_block_size = 64 cache_size = 32 coherent = "true" data_access_time = 3 dvfs_domain = "core"	type = "history_list" [perf_model/DRAM_directory] associativity = 16 directory_cache_access_time = 10 directory_type = "full_map" home_lookup_param = 6 interleaving = 1 locations = "llc" max_hw_sharers = 64 total_entries = 1048576 [perf_model/DRAM_directory/limitless] software_trap_penalty = 200 [perf_model/dtlb] associativity = 4 size = 64 [perf_model/fast_forward] model ="oneipc" [perf_model/fast_forward/oneipc] include_branch_misprediction = "false" include_memory_latency = "false" interval = 100000 [perf_model/itlb] associativity = 4 size = 128 [perf_model/l1_dcache] address_hash = "mask" associativity = 8 cache_block_size = 64 cache_size = 32 data_access_time = 3 dvfs_domain = "core" next_level_read_bandwidth = 0 outstanding_misses = 10 passthrough = "false" perf_model_type = "parallel" perfect = "false" prefetcher = "none" replacement_policy = "lru" shared_cores = 1 tags_access_time = 1 writeback_time = 0 writethrough = 0 [perf_model/l1_dcache/atd] [perf_model/l1_icache] address_hash = "mask" associativity = cache_block_size = 64 cache_size = 32 coherent = "true" data_access_time = 3 dvfs_domain = "core"

next_level_read_bandwidth = 0 passthrough = "false" perf_model_type = "parallel" perfect = "false" prefetcher = "none" replacement_policy = "lru" shared_cores = 1 tags_access_time = 1 writeback_time = 0 writethrough = 0 [perf_model/l1_icache/atd] [perf_model/l2_cache] address_hash = "mask" associativity = 8 cache_block_size = 64 cache_size = 256 data_access_time = 8 dvfs_domain = "core" next_level_read_bandwidth = 0 passthrough = "false" perf_model_type = "parallel" perfect = "false" prefetcher = "none" replacement_policy = "lru" shared_cores = 2 tags_access_time = 3 writeback_time = 50 writethrough = 0 [perf_model/l2_cache/atd] [perf_model/l3_cache] address_hash = "mask" associativity = 16 cache_block_size = 64 cache_size = 8192 data_access_time = 96 dvfs_domain = "global" passthrough = "false" perf_model_type = "parallel" perfect = "false" prefetcher = "none" replacement_policy = "lru" shared_cores = 8 tags_access_time = 10 writeback_time = 0 writethrough = 0 [perf_model/l3_cache/atd] [perf_model/l4_cache] passthrough = "false" perfect = "false" [perf_model/llc] evict_buffers = 8 [perf_model/nuca] enabled = "false"	next_level_read_bandwidth = 0 passthrough = "false" perf_model_type = "parallel" perfect = "false" prefetcher = "none" replacement_policy = "lru" shared_cores = 1 tags_access_time = 1 writeback_time = 0 writethrough = 0 [perf_model/l1_icache/atd] [perf_model/l2_cache] address_hash = "mask" associativity = 8 cache_block_size = 64 cache_size = 256 data_access_time = 8 dvfs_domain = "core" next_level_read_bandwidth = 0 passthrough = "false" perf_model_type = "parallel" perfect = "false" prefetcher = "none" replacement_policy = "lru" shared_cores = 1 tags_access_time = 3 writeback_time = 50 writethrough = 0 [perf_model/l2_cache/atd] [perf_model/l3_cache] address_hash = "mask" associativity = 16 cache_block_size = 64 cache_size = 8192 data_access_time = 96 dvfs_domain = "global" passthrough = "false" perf_model_type = "parallel" perfect = "false" prefetcher = "none" replacement_policy = "lru" shared_cores = 8 tags_access_time = 10 writeback_time = 0 writethrough = 0 [perf_model/l3_cache/atd] [perf_model/l4_cache] passthrough = "false" perfect = "false" [perf_model/llc] evict_buffers = 8 [perf_model/nuca] enabled = "false"	next_level_read_bandwidth = 0 passthrough = "false" perf_model_type = "parallel" perfect = "false" prefetcher = "none" replacement_policy = "lru" shared_cores = 1 tags_access_time = 1 writeback_time = 0 writethrough = 0 [perf_model/l1_icache/atd] [perf_model/l2_cache] address_hash = "mask" associativity = 8 cache_block_size = 64 cache_size = 256 data_access_time = 8 dvfs_domain = "core" next_level_read_bandwidth = 0 passthrough = "false" perf_model_type = "parallel" perfect = "false" prefetcher = "none" replacement_policy = "lru" shared_cores = 1 tags_access_time = 3 writeback_time = 50 writethrough = 0 [perf_model/l2_cache/atd] [perf_model/l3_cache] address_hash = "mask" associativity = 16 cache_block_size = 64 cache_size = 8192 data_access_time = 30 dvfs_domain = "global" passthrough = "false" perf_model_type = "parallel" perfect = "false" prefetcher = "none" replacement_policy = "lru" shared_cores = 8 tags_access_time = 10 writeback_time = 0 writethrough = 0 [perf_model/l4_cache] passthrough = "false" perfect = "false" [perf_model/llc] evict_buffers = 8 [perf_model/nuca] address_hash = "mask" associativity = 16 bandwidth = 64 cache_size = 8192	next_level_read_bandwidth = 0 passthrough = "false" perf_model_type = "parallel" perfect = "false" prefetcher = "none" replacement_policy = "lru" shared_cores = 1 tags_access_time = 1 writeback_time = 0 writethrough = 0 [perf_model/l1_icache/atd] [perf_model/l2_cache] address_hash = "mask" associativity = 8 cache_block_size = 64 cache_size = 256 data_access_time = 8 dvfs_domain = "core" next_level_read_bandwidth = 0 passthrough = "false" perf_model_type = "parallel" perfect = "false" prefetcher = "none" replacement_policy = "lru" shared_cores = 1 tags_access_time = 3 writeback_time = 50 writethrough = 0 [perf_model/l2_cache/atd] [perf_model/l3_cache] passthrough = "false" perfect = "false" [perf_model/l4_cache] passthrough = "false" perfect = "false" [perf_model/llc] evict_buffers = 8 [perf_model/nuca] enabled = "false" [perf_model/stlb] associativity = 4 size = 1024 [perf_model/sync] reschedule_cost = 1000 [perf_model/tlb] penalty = 30 penalty_parallel = "true" [power] technology_node = 22 vdd = 1.2

```
[perf_model/stlb]
associativity = 4
size = 1024

[perf_model/sync]
reschedule_cost = 1000

[perf_model/tlb]
penalty = 30
penalty_parallel = "true"

[power]
technology_node = 22
vdd = 1.2

[progress_trace]
enabled = "false"
filename = ""
interval = 5000

[queue_model]

[queue_model/basic]
moving_avg_enabled = "true"
moving_avg_type =
"arithmetic_mean"
moving_avg_window_size = 1024

[queue_model/history_list]
analytical_model_enabled = "true"
max_list_size = 100

[queue_model/windowed_mg1]
window_size = 1000

[routine_tracer]
type = "none"

[sampling]
enabled = "false"

[scheduler]
type = "pinned"

[scheduler/big_small]
debug = "false"
quantum = 100

[scheduler/pinned]
core_mask = 1
interleaving = 1
quantum = 100

[scheduler/roaming]
core_mask = 1
quantum = 100

[scheduler/static]
core_mask = 1

[tags]
```

```
[perf_model/stlb]
associativity = 4
size = 1024

[perf_model/sync]
reschedule_cost = 1000

[perf_model/tlb]
penalty = 30
penalty_parallel = "true"

[power]
technology_node = 22
vdd = 1.2

[progress_trace]
enabled = "false"
filename = ""
interval = 5000

[queue_model]

[queue_model/basic]
moving_avg_enabled = "true"
moving_avg_type =
"arithmetic_mean"
moving_avg_window_size = 1024

[queue_model/history_list]
analytical_model_enabled = "true"
max_list_size = 100

[queue_model/windowed_mg1]
window_size = 1000

[routine_tracer]
type = "none"

[sampling]
enabled = "false"

[scheduler]
type = "pinned"

[scheduler/big_small]
debug = "false"
quantum = 100

[scheduler/pinned]
core_mask = 1
interleaving = 1
quantum = 100

[scheduler/roaming]
core_mask = 1
quantum = 100

[scheduler/static]
core_mask = 1

[tags]
```

```
data_access_time = 30
enabled = "true"
replacement_policy = "lru"
tags_access_time = 10

[perf_model/nuca/queue_model]
enabled = "true"
type = "history_list"

[perf_model/stlb]
associativity = 4
size = 1024

[perf_model/sync]
reschedule_cost = 1000

[perf_model/tlb]
penalty = 30
penalty_parallel = "true"

[power]
technology_node = 22
vdd = 1.2

[progress_trace]
enabled = "false"
filename = ""
interval = 5000

[queue_model]

[queue_model/basic]
moving_avg_enabled = "true"
moving_avg_type =
"arithmetic_mean"
moving_avg_window_size = 1024

[queue_model/history_list]
analytical_model_enabled = "true"
max_list_size = 100

[queue_model/windowed_mg1]
window_size = 1000

[routine_tracer]
type = "none"

[sampling]
enabled = "false"

[scheduler]
type = "pinned"

[scheduler/big_small]
debug = "false"
quantum = 100

[scheduler/pinned]
core_mask = 1
interleaving = 1
quantum = 100
```

```
[progress_trace]
enabled = "false"
filename = ""
interval = 5000

[queue_model]

[queue_model/basic]
moving_avg_enabled = "true"
moving_avg_type =
"arithmetic_mean"
moving_avg_window_size = 1024

[queue_model/history_list]
analytical_model_enabled = "true"
max_list_size = 100

[queue_model/windowed_mg1]
window_size = 1000

[routine_tracer]
type = "none"

[sampling]
enabled = "false"

[scheduler]
type = "pinned"

[scheduler/big_small]
debug = "false"
quantum = 100

[scheduler/pinned]
core_mask = 1
interleaving = 1
quantum = 100

[scheduler/roaming]
core_mask = 1
quantum = 100

[scheduler/static]
core_mask = 1

[tags]

[traceinput]
address_randomization = "false"
enabled = "false"
mirror_output = "false"
num_runs = 1
restart_apps = "false"
stop_with_first_app = "true"
trace_prefix = ""
```

[traceinput] address_randomization = "false" enabled = "false" mirror_output = "false" num_runs = 1 restart_apps = "false" stop_with_first_app = "true" trace_prefix = ""	[traceinput] address_randomization = "false" enabled = "false" mirror_output = "false" num_runs = 1 restart_apps = "false" stop_with_first_app = "true" trace_prefix = ""	[scheduler/roaming] core_mask = 1 quantum = 100 [scheduler/static] core_mask = 1 [tags] [traceinput] address_randomization = "false" enabled = "false" mirror_output = "false" num_runs = 1 restart_apps = "false" stop_with_first_app = "true" trace_prefix = ""	

REFERENCES

Abandah, G. A. (1996), Tools for Characterizing Distributed Shared Memory Applications. **Technical Report**, HPL-96-157, Hewlett-Packard Labs.

Abandah, G. A. (1997). Characterizing Shared-memory Applications: A Case Study of NAS Parallel Benchmarks. **Technical Report**, HPL-97-24, Hewlett-Packard Labs.

Abandah, G. A. (1998), Reducing Communication Cost in Scalable Shared Memory Systems. **Doctoral Dissertation**, University of Michigan, Ann Arbor, MI, USA.

Abandah, G. A. and Davidson, E. S. (1998), Configuration Independent Analysis for Characterizing Shared-Memory Applications. In Proceedings of the **12th International Parallel Processing Symposium (IPPS)**, Orlando, FL, USA, 30 March - 3 April 1998, 485-491.

Abandah, Gheith A., and Edward S. Davidson. (1998), "A comparative study of cache-coherent nonuniform memory access systems." **High Performance Computing Systems and Applications.Springer** US, 1998.

Alam, S. R., Barrett, R. F., Kuehn, J. A., Roth, P. C., and Vetter, J. S. (2006), Characterization of Scientific Workloads on Systems with Multi-Core Processors. In Proceedings of the **2006 IEEE International Symposium on Workload Characterization (IISWC)**, San Jose, CA, USA, 25-27 October 2006, 225-236.

Agarwal, Virat, et al. (2010)."Scalable graph exploration on multicoreprocessors."Proceedings of the **2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis**. IEEE Computer Society.

Akhter, S., & Roberts, J. (2006). **Multi-core programming** (Vol. 33). Hillsboro**:** Intel press.

Al-Manasia, M., & Chaczko, Z. (2015). An Overview of Chip Multi-Processors Simulators Technology. In **Progress in Systems Engineering** (pp. 877-884). Springer International Publishing.

Ardestani, E. K., & Renau, J. (2013, February). ESESC: A fast multicore simulator using time-based sampling. In High Performance Computer Architecture (HPCA2013), **2013 IEEE 19th International Symposium on (pp. 448-459)**. IEEE.

Ardestani, E. K., Southern, G., Doung, J., Ebrahimi, E., & Renau, J. (2013). ESESC: A fast performance, power, and temperature multicore simulator**. Power (W), 8, 9.**

Bhattacharjee, A. and Martonosi, M. (2009), Characterizing the TLB Behavior of Emerging Parallel Workloads on Chip Multiprocessors. In Proceedings of the **18th International** Conference on **Parallel Architectures and Compilation Techniques (PACT),** Raleigh, NC, USA, 12-16 September 2009, 29-40.

Bhople, S. S., & Gaikwad, M. A. (2013). Comparative study of different topologies for network-on-chip architecture. **International Journal of Computer Applications**, 1-3.

Barbic, J. (2007). Multi-core architectures. Lecture Notes. [Online]. Available: **http://www. co-array. org/cafvsmpi. htm**.

Binkert, N., Beckmann, B., Black, G., Reinhardt, S. K., Saidi, A., Basu, A., & Wood, D. A. (2011). The gem5 simulator. **ACM SIGARCH Computer Architecture News**, 39(2), 1-7.

Bienia, C., Kumar, S., and Li, K. (2008). PARSEC vs. SPLASH: A quantitative comparison of two multithreaded benchmark suites on chip-multiprocessors. In Proceedings of the **2008 IEEE International Symposium on Workload Characterization (IISWC)**, Seattle, WA, USA, 14-16 September 2008, 47-56.

Blake, G., Dreslinski, R. G., Mudge, T., & Flautner, K. (2010, June). Evolution of thread-level parallelism in desktop applications. **In ACM SIGARCH Computer Architecture News** (Vol. 38, No. 3, pp. 302-313). ACM.

Blake, Geoffrey, Ronald G. Dreslinski, and Trevor Mudge. "A survey of multi-core processors." **Signal Processing Magazine**, IEEE 26.6 (2009): 26-37.

Bononi, L., Concer, N., Grammatikakis, M., Coppola, M., & Locatelli, R. (2007, August). NoC topologies exploration based on mapping and simulation models. In **Digital System Design Architectures, Methods and Tools**, 2007. DSD 2007. **10th Euromicro Conference** on (pp. 543-546). IEEE.

Carlson, T. E., Heirman, W., Eyerman, S., Hur, I., & Eeckhout, L. (2014). An evaluation of high-level mechanistic core models. **ACM Transactions on Architecture and Code Optimization (TACO)**, 11(3), 28.

Carlson, T. E., Heirman, W., Patil, H., & Eeckhout, L. (2014). Efficient, accurate and reproducible simulation of multi-threaded workloads. In REPRODUCE: **Workshop on Reproducible Research Methodologies**. IEEE.

Carlson, T. E., Heirman, W., Van Craeynest, K., & Eeckhout, L. (2014). Node performance and energy analysis with the Sniper multi-core simulator. **In Tools for High Performance Computing 2013 (pp. 79-89). Springer International Publishing**.

Carlson, T. E., Heirmant, W., & Eeckhout, L. (2011, November). Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation. **In High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for** (pp. 1-12). IEEE.

Chaturvedi, N., & Gurunarayanan, S. (2013). Study of various factors affecting performance of multi-core processors**. International Journal of Distributed and Parallel Systems**, 4(4), 37.

Cruz, Eduardo HM, et al. (2014) "Dynamic thread mapping of shared memory applications by exploiting cache coherence protocols." **Journal of Parallel and Distributed Computing** 74.3 (2014): 2215-2228.

Contreras, G. and Martonosi, M. (2008), Characterizing and Improving the Performance of Intel Threading Building Blocks. In Proceedings of the **2008 IEEE International Symposium on Workload Characterization (IISWC)**, Seattle, WA, USA, 14-16 September 2008, 57-66.

DeMassas, Pierre Guironnet, and FrédéricPétrot. (2008) "Comparison of memory write policies for NoC based multicore cache coherent systems." **Design, Automation and Test in Europe IEEE**, 2008.DATE'08.

Dey, T., Wang, W., Davidson, J. W., & Soffa, M. L. (2011, April). Characterizing multi-threaded applications based on shared-resource contention. In **Performance Analysis of Systems and Software** (ISPASS), **2011 IEEE International Symposium** on (pp. 76-86). IEEE.

Ding, J. H., Chang, P. C., Hsu, W. C., & Chung, Y. C. (2011, December). PQEMU: A parallel system emulator based on QEMU. **In Parallel and Distributed Systems (ICPADS)**, **2011 IEEE 17th International Conference** on (pp. 276-283). IEEE.

Duarte, Filipa, and Stephan Wong.(2010)"Cache-based memory copy hardware accelerator for multicore systems." **Computers, IEEE Transactions** on 59.11 (2010): 1494-1507.

Dubey, P. (2005), Recognition, Mining and Synthesis Moves Computers to the Era of Tera. **Technology@Intel Magazine**, 1-10.

Eeckhout, L. (2010). Computer architecture performance evaluation methods. **Synthesis Lectures on Computer Architecture**, *5*(1), 1-145.

Fasiku, A. I., Oyinloye, O. E., Falaki, S. O., & Adewale, O. S. (2014). Performance Evaluation of Multicore Processors. **International Journal of Engineering and Technology**, 4(1).

Florea, A., Buduleci, C., Chis, R., Gellert, A., & Vintan, L. (2014, October). Enhancing the Sniper simulator with thermal measurement. In **System Theory, Control and Computing (ICSTCC)**, 2014 18th **International Conference** (pp. 31-36). IEEE.

Furber, S. (2000). **ARM System-on-Chip Architecture**.

Heinrich, F., Carpen-Amarie, A., Degomme, A., Hunold, S., Legrand, A., Orgerie, A. C., & Quinson, M. (2017). Predicting **the Performance and the Power Consumption of MPI Applications With SimGrid**.

Heirman, W., Carlson, T. E., Che, S., Skadron, K., & Eeckhout, L. (2011, November). Using cycle stacks to understand scaling bottlenecks in multi-threaded workloads. In **Workload Characterization (IISWC), 2011 IEEE International Symposium** on (pp. 38-49). IEEE.

Heirman, W., Carlson, T., & Eeckhout, L. (2012). Sniper: scalable and accurate parallel multi-core simulation. In **8th International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems** (ACACES-2012) (pp. 91-94). **High-Performance and Embedded Architecture and Compilation Network of Excellence** (HiPEAC).

Heirman, W., Sarkar, S., Carlson, T. E., Hur, I., & Eeckhout, L. (2012, September). Power-aware multi-core simulation for early design stage hardware/software co-optimization. In Proceedings of the **21st international conference on Parallel architectures and compilation techniques** (pp. 3-12). ACM.

Ingle, V. V., Mahendra, A., & Gaikwad, C. Z. (2013). Review of mesh topology of NoC architecture using source routing algorithms. **International Journal of Computer Applications**, 30-34.

Jaleel, A., Cohn, R. S., Luk, C. K., & Jacob, B. (2008, June). CMP $ im: A Pin-based on-the-fly multi-core cache simulator. In Proceedings of **the Fourth Annual Workshop on Modeling, Benchmarking and Simulation (MoBS), co-located with ISCA** (pp. 28-36).

Jarus, M., Varrette, S., Oleksiak, A., & Bouvry, P. (2013, April). Performance evaluation and energy efficiency of high-density HPC platforms based on Intel, AMD and ARM processors. In **European Conference on Energy Efficiency in Large Scale Distributed Systems** (pp. 182-200). Springer Berlin Heidelberg.

Jha, S. S., Heirman, W., Falcón, A., Tubella, J., González, A., & Eeckhout, L. (2017). Shared resource aware scheduling on power-constrained tiled many-core processors. **Journal of Parallel and Distributed Computing**, 100, 30-41.

Johnsson, Lennart. "**Multiple Caches–Shared Memory**." (2013).

Kakoulli, Elena, VassosSoteriou, and Theocharis Theocharides. (2012) "Intelligent hotspot prediction for network-on-chip-based multicore systems." **Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions** on 31.3 (2012): 418-431.

Khan, Samira M., et al. (2013) "Improving multicore performance using mixed-cell cache architecture." **High Performance Computer Architecture** (HPCA2013), 2013 IEEE the **19th International Symposium** on. IEEE.2013.

Krishna, T., Kwon, W. C., Subramanian, S., Chen, C. H. O., Park, S., Chandrakasan, A. P., and Peh, L. S. (2013), Single-Cycle Multihop Asynchronous Repeated Traversal: A SMART Future for Reconfigurable On-Chip Networks. **IEEE Computer**, 46(10), 48-55.

Lecler, J. J., & Baillieu, G. (2011). Application driven network-on-chip architecture exploration & refinement for a complex SoC. **Design Automation for Embedded Systems**, *15*(2), 133-158.

Lee, Chia Che, Weichun Xu, and Yutian Gui. "**Memory Hierarchies-Effectiveness and implementations.**"

Li, S., Ahn, J. H., Strong, R. D., Brockman, J. B., Tullsen, D. M., & Jouppi, N. P. (2009, December). McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In **Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium** on (pp. 469-480). IEEE.

Li, S., Ahn, J. H., Strong, R. D., Brockman, J. B., Tullsen, D. M., & Jouppi, N. P. (2013). The McPAT framework for multicore and manycore architectures: Simultaneously modeling power, area, and timing. **ACM Transactions on Architecture and Code Optimization (TACO)**, 10(1), 5.

Lis, M., Ren, P., Cho, M. H., Shim, K. S., Fletcher, C. W., Khan, O., & Devadas, S. (2011, April). Scalable, accurate multicore simulation in the 1000-core era. In **Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium** on (pp. 175-185). IEEE.

Luk, C. K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., ... & Hazelwood, K. (2005, June). Pin: building customized program analysis tools with dynamic instrumentation. In **Acm sigplan notices** (Vol. 40, No. 6, pp. 190-200). ACM.

Lustig, D., Bhattacharjee, A., & Martonosi, M. (2013). TLB improvements for chip multiprocessors: Inter-core cooperative prefetchers and shared last-level TLBs. **ACM Transactions on Architecture and Code Optimization** (TACO), 10(1), 2.

Malhotra, G., Aggarwal, P., Sagar, A., & Sarangi, S. R. (2014, March). ParTejas: A parallel simulator for multicore processors. In **Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on** (pp. 130-131). IEEE.

Martin, Milo MK, Mark D. Hill, and Daniel J. Sorin. "Why on-chip cache coherence is here to stay." **Communications of the ACM** 55.7 (2012): 78-89.

Marty, M. R. (2008). Cache coherence techniques for multicore processors (**Doctoral dissertation,** University of Wisconsin--Madison).

Miller, J. E., Kasture, H., Kurian, G., Gruenwald, C., Beckmann, N., Celio, C., ... & Agarwal, A. (2010, January). Graphite: A distributed parallel simulator for multicores. In **High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium** on (pp. 1-12). IEEE.

Mittal, S. (2016). A Survey of Techniques for Architecting TLBs. **Concurrency and Computation: Practice and Experience**, 1-35.

Mohammed, M. S., & Abandah, G. A. (2015, November). Communication characteristics of parallel shared-memory multicore applications. In **Applied Electrical Engineering and Computing Technologies (AEECT), 2015 IEEE Jordan Conference** on (pp. 1-6). IEEE.

Mohammed, M. S., & Abandah, G. A. (2016). Characterization of Shared-Memory Multi-Core Applications. **Jordanian Journal of Computers and Information Technology**, 2(1), 37-54

Mohanty, Ram Prasad, Ashok Kumar Turuk, and BibhudattaSahoo. (2013) "Performance evaluation of multicore processors with varied interconnect networks." **Advanced Computing, Networking and Security (ADCONS), 2013 2nd International Conference on.IEEE,** 2013.

Molka, Daniel, et al.(2009) "Memory performance and cache coherency effects on an intel nehalem multiprocessor system." **Parallel Architectures and Compilation Techniques, 2009.PACT'09.18th International Conference on.IEEE**, 2009.

Molka, D., Hackenberg, D., Schöne, R., & Nagel, W. E. (2015, September). Cache coherence protocol and memory performance of the intel haswell-ep architecture. In **Parallel Processing (ICPP), 2015 44th International Conference** on (pp. 739-748). IEEE.

Natarajan, R., and Chaudhuri, M. (2013). Characterizing Multi-Threaded Applications for Designing Sharing-Aware Last-Level Cache Replacement Policies. In Proceedings of the **2013 IEEE International Symposium on Workload Characterization (IISWC)**, Portland, OR, USA, 22-24 September 2013, 1-10.

Olukotun, Kunle, et al.(1996) "The case for a single-chip multiprocessor." **ACM Sigplan Notices** 31.9 (1996): 2-11.

Pan, Xiaoyue, and Bengt Jonsson.(2014) "Modeling cache coherence misses on multicores." **Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on. IEEE,** 2014.

Panourgias, Iakovos. "NUMA effects on multicore, multi socket systems." **The University of Edinburgh** (2011).

Patel, A., Afram, F., & Ghose, K. (2011, March). Marss-x86: A qemu-based micro-architectural and systems simulator for x86 multicore processors. In **1st International Qemu Users' Forum** (pp. 29-30).

Patel, A., Afram, F., Chen, S., & Ghose, K. (2011, June). MARSS: a full system simulator for multicore x86 CPUs. In **Proceedings of the 48th Design Automation Conference** (pp. 1050-1055). ACM.

Pin -A Dynamic Binary Instrumentation Tool, Intel, Retrieved March 22, 2017, from **https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.**

Priya, B. K., Joshi, A. D., & Ramasubramanian, N. (2016, August). A Survey on Performance of On-Chip Cache for Multi-core Architectures. In **Proceedings of the International Conference on Informatics and Analytics** (p. 35). ACM.

Rahman, R. (2013). Intel® Xeon Phi™ Coprocessor Architecture and Tools: The Guide for Application Developers. **Apress**.

Ramasubramanian, N., and N. Ammasai Gounden. "Performance of Cache Memory Subsystems for Multi-core Architectures." **arXiv preprint arXiv**:1111.3056 (2011).

Ren, P., Lis, M., Cho, M. H., Shim, K. S., Fletcher, C. W., Khan, O., ... & Devadas, S. (2012). Hornet: A cycle-level multicore simulator. **Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions** on, 31(6), 890-903.

Ribeiro, Christiane Pousa. (2011) " Contributions on memory affinity management for hierarchical shared memory multicore platforms".**Diss. University of Grenoble**.

Rico Carro, A. (2013). Raising the level of abstraction: simulation of large chip multiprocessors running multithreaded applications.

Rolf, Trent. (2009) "Cache organization and memory management of the Intel Nehalem computer architecture**." University of Utah Computer Engineering**.

Rusu, Stefan, et al. (2007) "A 65-nm dual-core multithreaded Xeon® processor with 16-MB L3 cache." **Solid-State Circuits, IEEE Journal** of 42.1 (2007): 17-25.

Sanchez, D., & Kozyrakis, C. (2013, June). ZSim: fast and accurate microarchitectural simulation of thousand-core systems. In **ACM SIGARCH Computer Architecture News** (Vol. 41, No. 3, pp. 475-486). ACM.

Shukla, Surendra Kumar, C. N. S. Murthy, and P. K. Chande. (2015) "A Survey of Approaches used in Parallel Architectures and Multicore Processors, For Performance Improvement." Progress in **Systems Engineering.Springer International Publishing**, 2015.537-545.

Shukla, Surendra Kumar, C. N. S. Murthy, and P. K. Chande. (2015) "Parameter Trade-off and Performance Analysis of Multicore Architecture." Progress in **Systems Engineering.Springer International Publishing**, 2015.403-409.

Shriraman, A., Zhao, H., and Dwarkadas, S. (2013), An Application-Tailored Approach to Hardware Cache Coherence. **IEEE Computer**, 46(10), 40-47.

Southern, G. (2016). Effective Performance Analysis of Modern CPUs.
Standard Performance Evaluation Corporation (SPEC), SPEC CPU2006, Retrieved March 25, 2017, from **http://www.spec.org/cpu2006/.**

Tendler, J. M., Dodson, J. S., Fields, J. S., Le, H., & Sinharoy, B. (2002). POWER4 system microarchitecture. **IBM Journal of Research and Development**, 46(1), 5-25.

Tiwari, Anoop. (2014) "Performance comparison of cache coherence protocol on multicore architecture. " **Diss**. 2014.

Ubal, Rafael, et al. (2007) "Multi2sim: A simulation framework to evaluate multicore-multithreaded processors." **Computer Architecture and High Performance Computing**, 2007.SBAC-PAD 2007.19th International Symposium on. 2007.

Vajda, A. (2011). Multi-core and many-core processor architectures. In **Programming Many-Core Chips** (pp. 9-43). Springer US.

Villanueva, J. C., Flich, J., Duato, J., Eberle, H., Gura, N., & Olesinski, W. (2009, December). A performance evaluation of 2D-mesh, ring, and crossbar interconnects for chip multi-processors. In **Network on Chip Architectures, 2009. NoCArc 2009. 2nd International Workshop** on (pp. 51-56). IEEE.

Wang, J. (2011). Manifold: A parallel simulation framework for multicore systems (**Doctoral dissertation**, GEORGIA INSTITUTE OF TECHNOLOGY).

Wang, J., Beu, J., Bheda, R., Conte, T., Dong, Z., Kersey, C., ... & Xu, P. (2014, March). Manifold: A parallel simulation framework for multicore systems. In **Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium** on (pp. 106-115). IEEE.

Woo, S. C., Ohara, M., Torrie, E., Singh, J. P., and Gupta, A. (1995), The SPLASH Programs: Characterization and Methodological Considerations. **ACM SIGARCH Computer Architecture News**, 23(2), 24-36.

X. Zhou, W. Chen, W. Zheng, (2009) "Cache Sharing Management for Performance Fairness in Chip Multiprocessors", in: **International Conferenceon Parallel Architectures and Compilation Techniques** (PACT), 20.

تقييم بدائل تصاميم مشهورة للحواسيب متعددة النوى باستعمال التحليل المعتمد على التركيب

إعداد

عائشة فالح المسلم بني صخر

المشرف

الأستاذ الدكتور غيث عبندة

الملخص

لقد اكتسبت معماريات المعالج متعدد النواة شعبية متزايدة في السنوات الأخيرة في مجال الأداء العالي. وهناك العديد من التصاميم للعديد من المعالجات متعددة النواة التجارية المشهورة. ولذلك، من المهم القيام بتقييم أداء البدائل المتوفرة للتصميم على أساس مؤشرات توصيف التطبيقات المتعددة على هذه المنصات لمساعدة المبرمجين في ضبط وتطوير التطبيقات المتوازية في المستقبل. ومساعدة المصممين في تطوير تصاميم متعددة النوى تعمل بكفاءة مع التطبيقات المتوازية. إن الغرض من هذه الدراسة هو تقييم التصاميم المتعددة النوى والعديد من التصاميم الجوهرية، وتحديد نقاط القوة والضعف في المعالجات الحالية، وتحديد جوانب التصميم ذات الأثر الإيجابي على تلك المعالجات والمجالات التي تحتاج إلى المزيد من التحقيق والتحسين في مسببات الاختناقات في الأنظمة.

إن تصميم المعالجات متعددة النوى قد تطور للغاية من خلال عملية المحاكاة. وبالتالي، فإن هذه الأطروحة تقدم محاكاة معمارية باستخدام برمجية محاكاة (Sniper) وهو نظام محاكاة للمعالجات المتعددة النواة. قمنا باستخدام هذا المحاكي لتقييم أربع معالجات مشهورة من معالجات خوادم إنتل متعددة النواة و هي :Xeon Phi, Dnnington, Gainestown, Haswell . لقد تم اختيارهم لأنهم يغطوا مجموعة واسعة من خيارات التصميم متعددة النوى. في هذا البحث، اخترنا ثمانية تطبيقات متوازية تمثيلية من مجموعتين قياسيتين: مستودع تطبيق برينستون لأجهزة الكمبيوتر المشتركة الذاكرة (PARSEC) وتطبيقات ستانفورد المتوازية للذاكرة المشتركة (SPLASH2). ولقد أجرينا العديد من التجارب مع مختلف التصاميم لحجمين مختلفين من البيانات المدخلة في كل من التطبيقات المختارة.

استخدمنا مجموعة شاملة من المقاييس لتقييم الأداء للحفاظ على الواقعية وتحقيق المفاضلة بين أكثر من مقياس وهي: وقت التنفيذ، متوسط عدد الأوامر في الدورة، متوسط الاستخدام الأساسي، واستهلاك الطاقة. كما قمنا بتحليل التغير في عدد الدورات المستهلك لكل أمر مع الزمن.

ولعمل مقارنة عادلة تم وضع بدائل تصميم متعدد النوى على نفس المستوى التكنولوجي مع مكونات مماثلة في الحجم والسرعة. وهذه المقارنة أفضل لانها تعرض اختلافات الأداء بسبب الميزات المعمارية الرئيسية مثل تنظيم التسلسل الهرمي للذاكرة، وشبكة طوبولوجيا الربط والبروتوكولات المستخدمة، بدلا من التكنولوجيا الحالية وأحجام وسرع المكونات.

ولقد وجدنا أن أفضل تصميم يتصرف بشكل أفضل مع الحوسبة المتوازية من حيث سرعة التنفيذ هو Haswell (69 مللي ثانية) ومعدل إنتاج النظام (IPC 1.39) . ويرجع ذلك بسبب اشتماله على ذاكرة سريعة خاصة L2 ووجود ذاكرة L3 مشتركة، وبسبب سرعة التواصل بين النوى العائد لسرعة الشبكة الحلقية. من ناحية أخرى، فإنه يستهلك طاقة كبيرة نسبيا (52.3 واط).

وخلصنا أيضا إلى أن التصاميم القائمة على المسارب مثل Dunnington و Gainestown لم تعد قادرة على تلبية متطلبات أعباء العمل الجديدة بسبب الضعف الواضح في التعامل مع الضغط الناجم عن التواصل والتزامن في تنفيذ التطبيقات المتوازية الحديثة. ولقد حصل nGainestown وnDunnington على متوسط زمن تنفيذ يساوي 74 مللي ثانية و 73 مللي ثانية، على التوالي. وعلى متوسط الإنتاجية يساوي IPC 1.33 و IPC 1.31، على التوالي. كما أنهما يستهلكان طاقة تساوي 50.14 واط و 49.99 واط في المتوسط، على التوالي.

وأخيرا، عرضنا كيف يعزز بروتوكولMESIF أداء بدائل التصميم متعددة النوى بالمقارنة مع قيم بروتوكولات MESI القديمة. nDunnington حصل على تسريع بمقدار 1.028x. أما nGainestown فقد حصل على تسريع يساوي 1.027x . nXeon Phi حصل على تسريع بمقدار 1.01x. أما nHaswell فلم يستفد من بروتوكول MESIF وذلك لأنه يتكون من وحدة واحدة فيها ذاكرة مشتركة L3 لكل النوى.