HARDWARE CONFIGURATION-INDEPENDENT CHARACTERIZATION OF MULTI-CORE APPLICATIONS

By

Mohammed Sultan Ahmed Mohammed

Supervisor

Dr. Gheith Ali Abandah

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

Faculty of Graduate Studies

The University of Jordan

April, 2015

COMMITTEE DECISION

This Thesis/Dissertation (Hardware Configuration-Independent Characterization of Multi-Core Applications) was Successfully Defended and Approved on ------------

Examination Committee	**Signature**
Dr. Gheith A. Abandah. (Supervisor) Assoc. Prof. of Computer Engineering	-----------------------
Dr. Andraws A. Swidan (Member) Prof. of Computer Engineering	-----------------------
Dr. Basel A. Mahafzah (Member) Assoc. Prof. of Computer Science	-----------------------
Dr. Jehad A. Al-Sadi (Member) Assoc. Prof. of Computer Science (Arab Open University)	-----------------------

ACKNOWLEDGEMENTS

First and foremost, I would like to begin by thanking Almighty Allah, for providing me with the health, strength, patience, and for guiding me through all the difficulties to carry out this work.

I would like to acknowledge the great support of my supervisor Dr. Gheith Abandah. I specially thank him for his advice, guidance, patience, and encouragement throughout my thesis. He spent a lot of time in editing this thesis, helping in setting the research direction, and offering great ideas when I was confused. Without his help, this thesis would not be possible.

I would like to thank the University of Hodeidah and German Academic Exchange Service (DAAD) for their generous financial support. I also would like to thank the Computer Engineering Department of the University of Jordan for providing a multi-core computer.

I would specially like to thank my parents. They were always supporting and encouraging with their best wishes. Finally, to my lovely wife, Ahlam. She was always there cheering me up and stood by me through the good and bad times.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

API	Application Program Interface
ATOM	Analysis Tools with OM
BIRD	Binary Interpretation using Runtime Disassembly
CCAT	Communication Contention Analysis Tool
CDAT	Configuration Dependent Analysis Tool
CIAT	Configuration Independent Analysis Tool
CISC	Complex Instruction Set Computer
CPSlib	Compiler Parallel Support Library
EEL	Executable Editing Library
HJM	Heath Jarrow Morton
HPC	High-Performance Computing
JIT	Just in Time
LLC	Last Level Cache
MC	Monte Carlo
OM	Object Module
PA-RISC	Precision Architecture Reduced Instruction Set Computer
PARSEC	Princeton Application Repository for Shared-Memory Computers
PEBIL	PMaC’s Efficient Binary Instrumentation Toolkit for Linux
PMaC	Performance Modeling and Characterization
PSMAIT	Pin Shared-Memory Application Instrumentation Tool

POSIX	Portable Operating System Interface
RAR	Read after Read
RAW	Read after Write
RISC	Reduced Instruction Set Computer
RMS	Recognition, Mining, and Synthesis
SA	Simulated Annealing
SMAIT	Shared-Memory Application Instrumentation Tool
SPH	Smoothed Particle Hydrodynamics
SPLASH	Stanford ParalleL Applications for SHared memory
TBB	Threading Building Blocks
TDAT	Time Distribution Analysis Tool
TLB	Translation Lookaside Buffer
WAR	Write after Read
WAW	Write after Write

HARDWARE CONFIGURATION-INDEPENDENT CHARACTERIZATION OF MULTI-CORE APPLICATIONS

By
Mohammed S. Mohammed

Supervisor
Dr. Gheith A. Abandah

ABSTRACT

Multi-core processor architectures have been gaining increasing popularity in recent years. However, many available applications cannot take full advantage of increasing number of processing cores because these applications either are serial applications or have intensive communication patterns and high parallelization overhead. Therefore, it is important to characterize such applications on multi-core platforms to help the programmers in tuning them and developing future parallel applications, and to help the designers in developing multi-core architectures that efficiently run parallel applications.

This thesis presents a unique approach for characterizing the parallel applications on multi-core platforms. This approach is a configuration independent characterization. The configuration independent characterization is characterizing the inherent application's characteristics by tracking only the accesses on each memory location and it does not depend on any specific configuration. Therefore, this approach is faster than the traditional configuration dependent characterization. An analysis trace is piped on-the-fly to the configuration independent analysis tool (CIAT). On-the-fly analysis enables analyzing large problems without needing huge storage medium.

In this research, first 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 (SPLASH-2). Second, we developed an instrumentation tool, which is called Pin Shared-Memory Application Instrumentation Tool (PSMAIT), for instrumenting the selected applications. Third, we ported CIAT, which was developed for RISC multiprocessor platforms, to work on commodity multi-core platforms. Finally, we conducted many experiments with various numbers of threads for two problem sizes of each of the selected applications.

CIAT characterizes four aspects of the studied applications' characteristics: memory access instructions, communication patterns, communication slack, and communication locality. The obtained results show that two of the eight studied applications have high parallelization overhead, which are Cholesky and Fluidanimate. Cholesky has 80% of parallelization overhead when running 16 threads because it has a large communication to computation ratio. Fluidanimate has 60% of parallelization overhead when running 16 threads because of the large communication at the cell borders. The high parallelization overhead may limit the speedup of these applications. The most common communication patterns in the studied applications are read after write and write after read. However, there are two application that have large portions of write after write (WAW), which are Radix and Swaptions. In Radix, about 36% of the total communication is WAW when running 16 threads due to the permutation operations. In Swaption, almost 100% of the total communication patterns are WAW when running 16 threads due to reuse of some variables. The large portion of WAW may lead to spending a lot of time in handling store misses. The communication slack results show that all the studied applications can make use of prefetching. The communication locality results show that the initial thread communicates with the other threads in all the studied applications. Therefore, it is advisable to assign the initial thread into central core to reduce the communication cost.

CHAPTER 1: INTRUDUCTION

1.1 Background

Most of processors nowadays are multi-core processors, i.e., there are multiple processors on a single chip. Manufacturers of processors tend to increase the number of processor cores to increase the performance (Devadas, 2013). However, unfortunately, most of the available applications cannot take full advantage of this increasing number of processing cores. Multi-core architectures are relatively very complex and require close cooperation between the hardware and software developers.

Hardware designers must develop new techniques to overcome the limitations in the current multi-core designs such as the used cache coherence protocols. Particularly, the large number of available cores requires designing scalable cache coherence protocols (Shriraman *et al.*, 2013). Also the networks interconnecting these cores, as the number of cores increases, are having increasing latencies leading to reduced overall performance (Krishna *et al.*, 2013), and increased power consumption (Schuchhardt *et al.*, 2013).

Software developers are using available parallel development tools to develop parallel-multithreaded applications such as pthreads (Buttlar *et al.*, 1996). Such tools depend on the programmer to identify parallelism and to create, manage, and synchronize threads to exploit the available data and task parallelism. Another more productive tool is the OpenMP (Dagum *et al.*, 1998). OpenMP depends on the programmer to identify parallelism, but depends on the compiler to generate the code required to exploit this parallelism. The recent threading library Threading Building Blocks (TBB) (Reinders 2007) 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.

1.2 Motivation and Problem Statement

Multi-core architecture is the current and the foreseeable-future approach that manufacturers of processors use to build high performance and low power processors (Borkar, 2007). This trend motivates studies to find new techniques that help to improve the overall performance of multi-core systems. However, multi-core architectures are relatively very complex, so there are many aspects that need to be tackled to improve multi-core performance.

Some studies try to improve the multi-core performance by proposing techniques that map applications to cores in order to reduce the interference between these applications in multi-core systems. Das, *et al.* (2013) proposed a technique that maps applications to cores to reduce inter-application interference in multi-core system. This technique clusters the cores into subnetworks then assigns the interference-sensitive applications to specific clusters, and finally maps the application's threads to cores in the cluster. Pusukuri, *et al.* (2013) map threads to cores by using a supervised learning technique that monitors application resource usage characteristics to detect the effects of the interference between multi-threaded applications. Jaleel, *et al.* (2012) assign applications to cores by using knowledge of the last level cache (LLC) replacement behavior and the application cache utility information.

Other studies try to improve the multi-core performance by proposing coherence techniques that minimize the impact of coherence protocols on the multi-core performance. Schuchhardt, *et al.* (2013) proposed a dynamic directories technique to put coherence directories close to cores of the consistent cache blocks to reduce the interconnection cost between the cores. Shriraman, *et al.* (2013) proposed an application-tailored technique, which is application independent technique; to reduce communication overhead of the coherence protocols and make them scalable with large number of cores.

As reviewed above, there are several techniques that have been proposed to improve the overall multi-core system performance. However, to exploit the potential of multi-core architecture, the applications must be parallelized and distributed on the multi-core processors efficiently. This task is very difficult without knowledge of the applications' behavior and characteristics on multiple cores with shared memory.

Therefore, it is important to understand how efficiently current applications utilize multi-core architecture by characterizing such applications on multi-core platforms. This will lead to developing future parallel applications that efficiently utilize multi-core architecture and developing multi-core architectures that run the current and the future applications with higher performance.

1.3 Thesis Contributions

In this thesis, we have the following contributions:

- Investigating the available benchmarks or applications that are representative of multi-core applications and selecting a representative set of benchmarks/applications for further study. We chose eight applications from two benchmark suites.
- Investigating available application instrumentation techniques to select recent techniques suitable for instrumenting the selected multi-core applications. We developed an instrumentation tool based on binary dynamic instrumentation.
- Porting configuration independent analysis tool to commodity multi-core architectures where it was developed for RISC multiprocessor systems.
- Conducting many experiments of the selected representative applications with various numbers of threads on a multi-core system. The characterization results

are documented for the benefits of future application and architecture development studies.

1.4 Thesis Organization

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

Chapter two presents a survey of some related work. It includes both trace collection techniques and analysis and characterization techniques.

Chapter three summarizes our methodology for monitoring and characterizing multi-core applications and describes the tools developed to characterize these applications.

Chapter four presents the results of the configuration independent characterization approach we used for characterizing the multi-core applications.

Chapter five presents the main conclusions regarding the thesis's methodology, the developed tools, and characterization results. Additionally, it presents some proposed future work.

CHAPTER 2: LITRATURE REVIEW

2.1 Introduction

This chapter presents a survey of some related work. It includes benchmarks, trace collection techniques, and analysis and characterization techniques.

2.2 Benchmarks

There are many general proposed benchmark suits such as; 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. SPLASH-2 suite (Woo, et al., 1996) is a collection of multithreaded applications, which is representative of scientific, engineering, and graphic applications. These applications are widely used in the high-performance computing (HPC) domain. PARSEC suite (Bienia, et al., 2008) is a collection of multithreaded commercial and new applications in recognition, 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. Phoenix suite (Ranger, et al., 2007) is a collection of data-insensitive applications that implement the MapReduce programming model (Dean and Ghemawat, 2004).

Despite that SPLASH-2 was released at the beginning of the 1990s for the HPC domain, it is widely used besides PARSEC in the recent multi-core research (Shi and Khan 2013), (Shriraman *et al.*, 2013), and (Krishna *et al.*, 2013). Bienia, *et al.* (2008b) show that SPLASH-2 and PARSEC complement each other in term of diversity of working set size, cache miss rate, and distribution of instructions.

2.3 Trace Collection

Characterizing multi-core applications includes trace collection and trace analysis. Trace collection means collecting information traces of application behavior on multiple cores that have shared memory by using suitable instrumentation techniques. There are three main techniques of instrumentation: source code instrumentation, static binary instrumentation, and dynamic binary instrumentation.

2.3.1 Source Code Instrumentation

The source code instrumentation allows collecting various information traces by inserting instrumentation code to the source code files of an application. The source code instrumentation requires that the source code is available. Therefore, this technique cannot instrument an object code for third party applications and system libraries. The following paragraph presents some tools that use this technique of instrumentation.

Abandah (1996) developed an instrumentation tool called Shared-Memory Application Instrumentation Tool (SMAIT) to collect traces from multi-threaded shared memory parallel applications. This tool can pipe the traces at the run-time execution and it has three levels of instruction instrumentation: procedure call instructions, load and store instructions, and branch instructions. Abandah (1997) used SMAIT to collect memory traces from NAS shared memory applications. Nguyen, *et al.* (1996) proposed Augmint that is an instrumentation toolkit that supports execute-driven simulation for Intel x86 architectures where the previous execute-driven simulation approaches usually support RISC architectures.

2.3.2 Static Binary Instrumentation

The static binary instrumentation allows collecting various information traces by inserting instrumentation code to the executable file of an application. The static binary instrumentation does not require that the source code is available. Therefore, it can instrument the user's code of an application and all third party applications and system libraries that are linked with that application. However, this technique requires stopping an application execution to make any modification. The following paragraphs present some tools that use this technique of instrumentation.

One of the earliest instrumentation tools that uses this technique is Analysis Tools with Object Module (ATOM) that was developed by Srivastava and Eustace (1994). ATOM is developed to work on the Alpha platform. It uses OM (Srivastava and Wall, 1992) link-time optimizer to make an application and user's analysis routines in the same address space. Larus and Schnarr (1995) developed an instrumentation tool called Executable Editing Library (EEL). EEL is developed to work on the SPARC platform. It is hardware- and software-independent editing tool for editing the executable files.

Nanda, *et al.* (2006) developed an instrumentation tool called Binary Interpretation using Runtime Disassembly (BIRD). BIRD is developed to work on Windows/x86 platform. It uses redirecting approach, i.e., it inserts the instrumentation code to an unused memory region and uses a branch instruction at an instrumentation point that redirects the control to the instrumentation code.

Laurenzano, *et al.* (2010) at performance modeling and characterization (PMaC) laboratory developed an instrumentation tool called PMaC's Efficient Binary Instrumentation Toolkit for Linux (PEBIL). PEBIL is developed to work on the Linux platforms. It relocates each function to provide enough space for the branch instruction at any instrumentation point. Additionally, it uses instrumentation functions and hand-coded assembly to accomplish instrumentation operations.

2.3.3 Dynamic Binary Instrumentation

The dynamic binary instrumentation allows collecting various information traces by inserting instrumentation code into the executable file of an application during the execution. Similar to the static instrumentation, the source code is not required. The advantage of this technique is that it allows adding or removing instrumentation code while the application is running. There are two approaches for the dynamic instrumentation, which are probe-based and just-in-time JIT-based instrumentation (Luk, *et al.*, 2005). Some of the instrumentation tools that use the dynamic instrumentation technique are presented in the following paragraphs.

Buck and Hollingsworth (2000) proposed a dynamic instrumentation tool called Dyninst that provides C++ library for instrumenting applications. They also developed an application program interface (API) that allows inserting code and modifying the application while it is running. Cantrill, *et al.* (2004) proposed a dynamic instrumentation tool called DTrace, which instruments user level and kernel applications. Both Dyninst and DTrace are a probe-based dynamic instrumentation tools.

Luk, *et al.* (2005) developed a dynamic binary instrumentation tool for Linux and Windows called Pin, which is JIT-based dynamic instrumentation tool. It instruments single and multiple threaded applications and it supports different types of processors including Intel's IA-32 (x86 32-bit), IA-32E (x86 64-bit), and Itanium processors. Pin is wildly used for instrumenting application that are running on multi-core systems (Jaleel, *et al.*, 2008), (Bertels, *et al.*, 2011), and (Wang, *et al.*, 2013).

2.4 Application Analysis and Characterization

Several characterization techniques are used to characterize and analyze applications. The following subsections present some of these techniques.

2.4.1 Hardware-Assisted Characterization

Many characterization researches have used hardware performance counters, which are registers on the processor that keep track of hardware events to characterize various aspects of applications. Dongarra, *et al.* (2004) used hardware performance counters to characterize data cache and translation lookaside Buffer (TLB) behavior of their microbenchmarks.

Bhadauria, *et al.* (2009) characterized PARSEC benchmark on multiple aspects: cache performance, sensitivity to DRAM speed and bandwidth, multi-thread scalability, and micro-architecture design choices on a variety of real multi-core systems.

Ferdman, *et al.* (2012) used hardware performance counters to study micro-architectural behavior of their CloudSuite benchmarks. They concluded that existing processor micro-architectures are inefficient for running their benchmarks.

Jia, *et al.* (2013) used hardware performance counters to characterize eleven of data analysis workloads of a data center to determine their micro-architectural characteristics on systems equipped with modern superscalar out-of-order processors. They also developed a benchmark suite called DCBench for typical data center workloads.

2.4.2 Message-Passing Characterization

Message passing characterization technique is used for characterizing parallel scientific applications in distributed systems. The following paragraphs present some research that use this technique.

Alam, *et al.* (2006) characterized scaling behavior of a set of micro-benchmarks, kernels, and scientific workloads on HPC systems. They used AMD Opteron multi-core processors. They concluded that the current cache coherence protocol of the Opteron processor is insufficient to exploit the full bandwidth capability of the memory interface.

Chai, *et al.* (2007) characterized micro-benchmarks and application level benchmarks on Intel dual-core cluster. They observed that resource contention leads to reduced overall performance. Consequently, the multi-core cluster architecture does not scale as a single core cluster architecture for shared memory applications.

2.4.3 Configuration Dependent Analysis

The configuration dependent characterization technique characterizes the application depending on a specific configuration of some system components. This technique is widely used in characterizing applications. The following paragraphs present some research that use this technique.

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.

Jaleel, *et al.* (2006) characterized last-level cache memory behavior of parallel bioinformatics data-mining workloads on multi-core processors. They concluded that shared last-level cache memory is better than private last-level cache memory for high-performance systems. Jaleel, *et al.* (2008) also used a dynamic binary instrumentation tool as an alternative for trace-driven and execute-driven approach. They proposed a memory system simulator to characterize memory performance of x86 workloads on multi-core processors.

Bienia, *et al.* (2008) characterized PARSEC benchmarks to show that their benchmark suit has various types of multi-threaded behaviors. Bhattacharjee and Martonosi (2009) characterized TLB behavior of the PARSEC benchmark applications.

Contreras and Martonosi (2008) characterized a subset of PARSEC benchmark applications that were compiled with Intel TBB on AMD dual-core processors to determine the sources of overhead within the TBB.

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 OMP, and SPLASH-2 to understand 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.

2.4.4 Configuration Independent Analysis

The configuration independent characterization technique is a unique technique for characterizing the inherent characteristics that do not change with changing system configuration. Abandah and Davidson (1998) proposed 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. Their characterization results are useful for developing and tuning shared memory applications for multiprocessor systems.

Our work is porting CIAT to characterize the hardware-configuration independent characteristics of parallel applications on multi-core platforms. We characterize the communication among cores and shared memory behavior of parallel applications irrespective of the hardware-configuration or cache coherence protocol. This approach of characterization is fast because it gets the parallel application characteristics directly from an execution trace and tracks only the changing of accesses on each memory location and does not use a specific cache coherence or any specific system configuration parameters (Abandah, 1998).

CHAPTER 3: METHODOLOGY AND TOOLS

3.1 Introduction

This chapter summarizes our methodology for monitoring and characterizing multi-core applications and describes the tools developed to characterize these applications.

3.2 Overview

The methodology relies on choosing a set of benchmarks, which are representative of multi-core applications based on the current related studies, and collecting detailed traces and sending these traces to analysis tool as shown in Figure 3.1. The PSMAIT is a tool based on Pin that developed to instrument the multi-core applications. The ported CIAT is used to analyze traces.

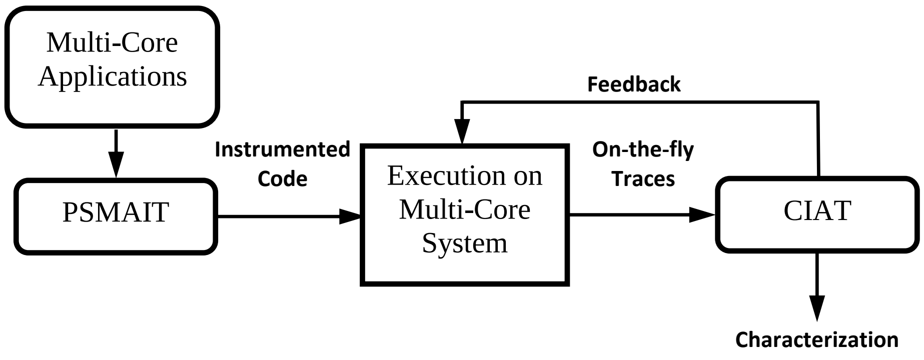


Figure 3.1. Our methodology of characterizing multi-core applications.

As shown in Figure 3.1, PSMAIT supports on-the-fly analysis; the instrumented code pipes the traces directly to the analysis tool when executed on a multi-core system. The instrumented process accepts feedback from CIAT to control the execution timing of the process on the multi-core system according to CIAT's analysis model. On-the-fly analysis enables analyzing large problems without needing huge storage medium and uses feedback from the analysis tool to avoid the non-deterministic conduct of the instrumented applications.

3.3 Instrumentation Tool (PSMAIT)

PSMAIT is based on Pin (Luk, *et al*., 2005), a dynamic binary instrumentation tool for Linux and Windows. Pin is a JIT-based dynamic instrumentation tool. It uses dynamic compilation techniques to instrument applications while they are running. Pin instruments single and multi-threaded applications and it supports Intel® IA-32, Intel® IA-64 and Intel® many integrated core architectures (Intel, 2015). It has a rich set of API's that can be used to instrument applications without the need to master the underlying instruction set.

PSMAIT is a tool written in C++. It consists of a set of instrumentation and analysis routines as shown in Figure 3.2, where the instrumentation routine determines where instrumentation is inserted and the analysis routine determines what to do when instrumentation is activated. PSMAIT is designed to collect traces of multi-threaded parallel applications and send these traces directly, on-the-fly, to CIAT. PSMAIT is a run-time binary instrumentation, which means that it does not need the source code of the parallel application. It instruments both the parallel applications' user code and all the libraries that are used during executing these applications.

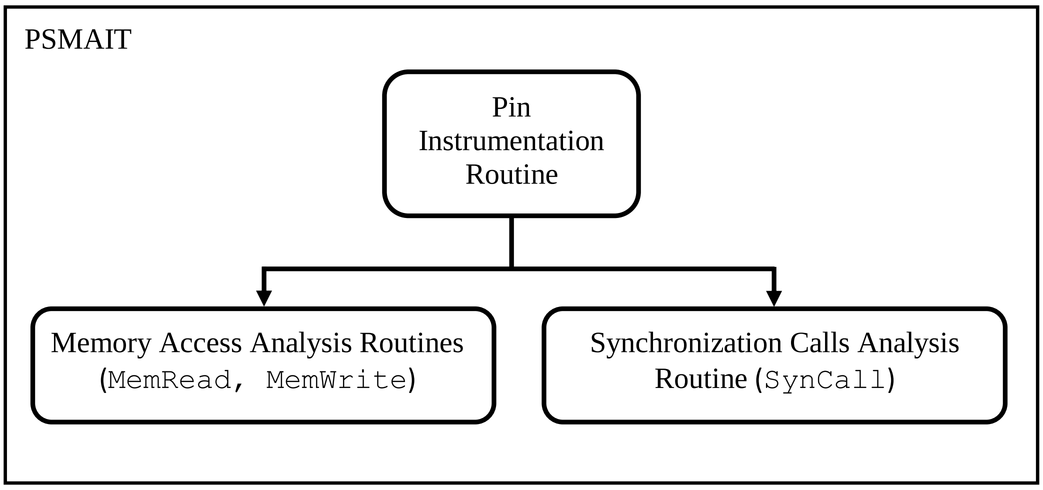

PSMAIT
Pin Instrumentation Routine
Memory Access Analysis Routines (MemRead, MemWrite)
Synchronization Calls Analysis Routine (SynCall)

Figure 3.2 PSMAIT implementation overview.

PSMAIT depends on Pin instrumentation routine to capture every memory access instruction, count number of operands in each instruction, and check type of memory access. We added code to the instrumentation routine to determine whether the captured instruction is integer or floating-point instruction. For each operand, if the memory access is load, the instrumentation routine calls `MemRead` analysis routine and if the memory access is store, the instrumentation routine calls `MemWrite` analysis routine. `MemRead` and `MemWrite` are analysis routines that we built to receive the instruction point type (fixed or floating), the size of the memory transfer, the type of the transfer (load or store), and the starting virtual address of the memory location referenced from the instrumentation routine. Subsequently, they generate simple two-field records. The simple record has two fields each field is four bytes long. The first field specifies the instruction point type (fixed or floating), the size of the memory transfer, and the type of the transfer (load or store), as shown in Figure 3.3. The second field contains the starting virtual address of the memory location referenced.

Additionally, before every synchronization and thread management call such as lock, barrier, and condition call, the instrumentation routine calls `Syncall` analysis routine. `Syncall` is analysis routine that we built to receive the call type and thread number from the instrumentation routine. Subsequently, it sends special records for the synchronization and thread management calls to CIAT. These records help CIAT to control the parallel application execution. PSMAIT creates two pipes, one pipe for sending traces and synchronization calls to CIAT, and another pipe for receiving feedback from CIAT. When PSMAIT sends a special record of some thread, it blocks that thread until it gets a feedback from CIAT to resume the execution of the stopped thread.

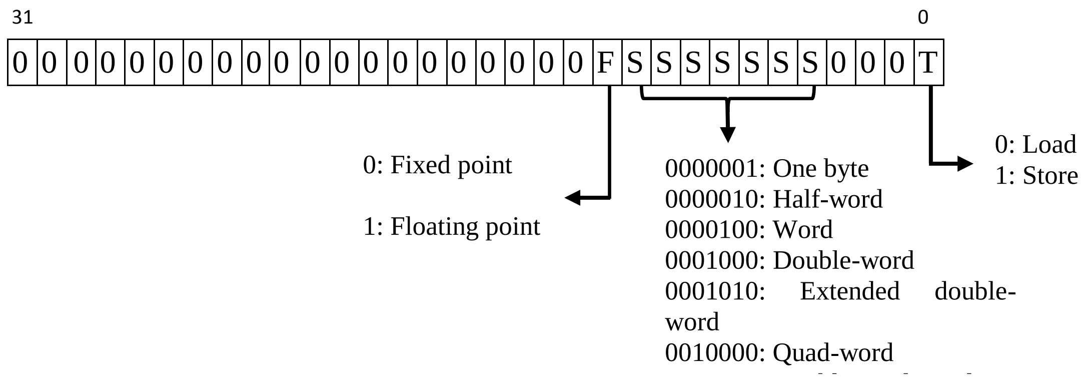


Figure 3.3. First field format of memory-access record.

3.4 Analysis Tool (CIAT)

Our analysis tool is based on CIAT for the multiprocessor environment developed by Abandah, (1998). CIAT aims to characterize inherent application characteristics such as memory access instructions, concurrency, communication patterns, and sharing behavior of parallel applications that are independent from one multiprocessor configuration to another. A multiprocessor configuration includes the hierarchy of processor, the interconnection topology, the coherence protocol, the cache configuration, and the sizes and speeds of the multiprocessor system components.

CIAT uses many variables to count the various events by tracking the memory load/store operations. It accepts traces from PSMAIT, which generates *n* trace pipes for the *n* executing threads. CIAT supports various *execution phases;* it assumes that the traces come from a parallel application either in a *serial* or in a *parallel* phase. In a serial phase, there is only one thread active, while the other threads are idle. In a parallel phase, more than one thread can be active. CIAT uses the special records of the thread spawn and thread join calls to identify switches between serial and parallel phases. At the end of each phase, CIAT generates statistics and save them in a report file. At the end of the last phase, CIAT reports the aggregate statistics in the report file.

CIAT assumes that *n* processors in multiprocessors can execute *n* instructions at the same time and each instruction takes a fixed time. Therefore, a *pseudo clock* in instruction units is used to keep track of the execution time. However, CIAT currently only sees the memory accesses and advances the clock by one for each thread whenever it receives a memory access record. And this is an approximation of the instruction stream. CIAT interleaves the analysis of multiple thread traces on the processors according to the thread spawn and join calls, and follows the constraints of the lock, conditional wait, and barrier synchronization calls. The following subsections summarize the configuration independent characteristics of the parallel applications that are measured and reported by CIAT.

3.4.1 Memory Access Instructions

CIAT counts the occurrences of the different types of memory access instructions and reports the following memory access statistics in number and percentage:

- Load and store instructions.
- Instructions in each size of accessed data, which include byte, half-word, word, double-word, quad-word, double quad-word, single-precision floating-point, double-precision floating-point, and extended-precision floating-point for both load and store instructions.
- Recurrence of load and store sequences according to the length of each sequence. For example, a load sequence of length one means that there is a sequence of one load from one thread preceded and followed by stores. A store sequence of length two means that there is two a sequence of two stores from one thread not interrupted by any load.

3.4.2 Communication Patterns

The communication among processors in multiprocessor occurs when those processors access same shared memory locations. For each memory location, CIAT keeps track of the type of accesses and the processors that perform these accesses. Consequently, CIAT reports the amount of the following four type of communication patterns and sharing and invalidation degrees:

- Read after write accesses (RAW): A RAW communication pattern occurs when one processor writes to a memory location and other processors read from that location.
- Sharing degree for RAW (*S*): This is a vector of sharing degrees, where *S*[*p*] is the number of times that *p* processors read from a memory location after being previously written.
- Write after read accesses (WAR): A WAR communication pattern occurs when a processor writes to a memory location that was read by other processors.
- Invalidation degree for WAR (*I*)*:* This is a vector of invalidation degrees, where *I*[*p*] is the number of times that a memory location was written into after being previously read by *p* processors.
- Write after write accesses (WAW): A WAW communication pattern occurs when a processor writes to a memory location that was written by another processor.
- Read after read accesses (RAR): A RAR communication pattern occurs when a processor reads from a memory location that was read by another processor and the first visible access to this location is a read.

3.4.3 Communication Slack

The communication slack is the time between writing a new value to a memory location and referencing it by other processors using RAW or WAR access. CIAT measures the communication slack as the time in instruction unit, i.e., it counts the number of instructions from writing the value to the memory location until referencing it.

3.4.4 Communication Locality

The communication locality is a measure how the processors communicate with each other. CIAT characterizes the communication locality of shared-memory parallel applications by counting the number of communication events for each processor pair. CIAT generates $n \times n$ matrix where n is the number of processors. The rows in this matrix represent the producer processors and the columns represent the consumer processors. For example, the value in the row i and column j is the number of communication events from processor i to processor j. This value is incremented by one in the following cases:

- Each time processor j loads a memory location that was stored into by processor i (RAW).
- Each time processor j stores into a memory location that was previously stored into by processor i (WAW).
- Each time processor i updates a memory location that was previously stored into by processor i and loaded by processor j (WAR).

3.5 Modifications to Support Intel Multi-Core Processors

This section presents the modifications on CIAT to characterize all the inherent application's characteristics that are mentioned in section 3.4 on multi-core platform. To ensure CIAT supports multi-core applications characterization, it is modified to support a new processor architecture and new parallelism techniques.

3.5.1 Processor Architecture

CAIT was designed to characterize shared-memory applications on HP precision architecture reduced instruction set computer (PA-RISC) multiprocessors (HP, 1994). It is modified to work on complex instruction set computer (CISC) multi-core processors (Hennessy and Patterson, 2012).

As it is known, RISC microprocessors allow only load and store instructions to access memory with only one memory operand for each instruction. In contrast, in CISC microprocessors, many instructions can access the memory and possibly with more than one operand. However, PSMIAT is developed to capture every access to the memory, not just load/store instructions, and send a simple memory record to CIAT for each access. It may send multiple memory access records to CIAT for one instruction. Each simple record contains the type and size of memory access and the virtual address of the accessed location. Additionally, CIAT is modified to support large access sizes such as extended floating, double-word, quad-word, and even double quad-word. This modification is done on the source code of CIAT.

3.5.2 Parallelism Technique

CIAT was built to work on HP-UX systems that use the compiler parallel support library (CPSlib) for thread management and synchronization (HP, 1997). CIAT is modified to support systems that use the portable operating system interface (POSIX) threads also called Pthreads for thread management and synchronization.

CIAT is modified to receive special call records from PSMAIT for tracing thread management and synchronization calls such as `pthread_spawn`, `pthread_mutex_lock`, `pthread_mutex_unlock`, `pthread_barrier_wait`, `pthread_cond_wait`, `pthread_join`, `pthread_cond_broadcast`, and `pthread_mutex_trylock`. By these records, CIAT controls the execution of the application. For example, if PSMAIT sends a record for a `pthread_mutex_lock` call from a thread to CIAT, PSMAIT blocks that thread until it gets feedback. CIAT checks the simulated mutex lock and assigns it to the called thread if it is free and sends a feedback to PSMAIT to unblock the stopped thread. Otherwise, CIAT blocks receiving traces from the called thread and puts it in a waiting list until the simulated mutex lock is freed. Therefore, PSMAIT continues blocking the calling thread until it gets the feedback from CIAT. When the simulated mutex lock is freed, CIAT unblocks one thread from the waiting list. It uses FIFO and sends a feedback to PSMAIT.

The old version of CIAT assumes that all threads' spawns occur at same time in the parallel phase, thus it expects traces from all threads in a parallel region. However, this does not work for some new applications that do not spawn all threads at the beginning of the parallel phase. Therefore, CIAT is modified to activate receiving the traces from only the spawned threads. Any time a new thread is spawned, CIAT activates receiving traces from it.

Additionally, CIAT is modified to deal with the `pthread_cond_wait` calls used in barriers. Because some modern applications use the `pthread_cond_wait` call as a barrier wait.

3.6 Validation of the Characterization Tools

We have verified of the correctness of our characterization tools by running these tools on simple synthetic applications that have known analysis results. In addition, we have compared our analysis results of the applications from SPLASH-2 suite (presented in Chapter 4) against the configuration independent analysis presented in (Abandah, 1998). The results are agree in general, but there are some differences due to working on two different hardware architectures and instrumentation tools. We work on CISC architecture that has a few number of registers compared with RISC architecture that has a large number of registers. Therefore, the number of memory accesses in our work is larger. Moreover, PSMAIT instruments the application’s user code and all the libraries that are used during executing the parallel applications. Whereas, SMIAT that is used in (Abandah, 1998) instruments only the application’s user code.

We have verified that our characterization tools are hardware configuration-independent by running them on two machines that have different types of multi-core processor. One of them has dual core processor and the other has quad core processor. The characterization results on the two machines are the same. Therefore, we can make sure that our characterization tools do not depend on the hardware configuration.

CHAPTER 4: RESULTS DISCUSSION AND APPLICATIONS

CHARACTERIZATION

4.1 Introduction

This chapter presents the results of the configuration independent characterization approach we used for characterizing the multi-core applications. We use our tools to characterize eight parallel benchmark applications: four applications from Stanford SPLASH-2 (Woo, *et al.*, 1996) and four applications from Princeton PARSEC application suite that run on multi-core systems (Bienia, *et al.*, 2008). These applications were selected because they represent a wide range of applications and often used in multi-core research as mentioned in Chapter 2.

The next section describes the eight benchmarks that are used in this study. Section 4.3 presents the characterization results found.

4.2 Applications

We used eight benchmark applications in our study. Four of these applications are from SPLASH-2 suite, which are Radix, FFT, LU, and Cholesky. The other four are from PARSEC suite, which are Canneal, Blackscholes, Fluidanimate, and Swaptions. The following paragraphs present short descriptions of these applications.

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.

FFT is a one-dimensional kernel of the radix-6 steps fast Fourier transform algorithm that is optimized to minimize interprocessor communication. The data set 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 interprocessor communication occurs in three matrix transpose steps. Each processor transposes a neighboring submatrix 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.

LU is a kernel benchmark that decomposes a dense square matrix into the product of a lower triangular and an upper triangular matrix. The $n\times n$ matrix is divided into an 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, a 2-D scatter decomposition is used to assign blocks to processors.

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.

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 (Kirkpatrick *et al.*, 1983 and Carny, 1987). Canneal simulates putting elements on a chip with minimum routing cost.

Blackscholes is an Intel RMS application. It calculates the prices for a portfolio of European options analytically by using the Black-Scholes partial differential equation solution (Black and Scholes, 1973). It partitions the portfolio work among the threads and processes them simultaneously.

Fluidanimate is an Intel RMS application that uses an extension of the smoothed particle hydrodynamics (SPH) 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.

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 (MC) simulation to compute the prices.

To study the impact of application problem size on the communication behavior, we work on two problem sizes of each application; Size I and Size II. Where Size I is smaller than Size II. Table 4.1 shows the problem sizes and the abbreviations that are used for naming these applications.

Suite	Application	Abbreviation	Size I	Size II
SPLASH-2	Radix	Radix	256K integers	2M integers
	FFT	FFT	64K points	1M points
	LU	LU	256×256	512×512
	Cholesky	Chole	`tk15.O` file	`tk29.O` file
PARSEC	Canneal	Cann	100,000 elements, 32 temperature steps	200,000 elements, 32 temperature steps
	Blackscholes	Black	4K options	16K options
	Fluidanimate	Fluid	5 frames, 35K particles	5 frames, 100K particles
	Swaptions	Swap	16 swaptions, 10,000 simulations	32 swaptions, 20,000 simulations

Table 4.1. The applications problem sizes.

4.3 Characterization Results

This section presents the results of the characterization of the multi-core applications that are measured and reported by CIAT. The following four subsections present the inherent characteristics of the studied applications. These characteristics are memory access instructions, communication patterns, communication slack, and communication locality.

4.3.1 Memory Access Instructions

This subsection presents and analyzes the first independent characteristic of multi-core applications. As mentioned in Chapter 3, the developed tools can capture every operation on the memory and report the number of load and store operations as shown in Table 4.2.

Table 4.2 shows the number of memory accesses in billions for the eight studied applications when using one thread for the two problem seizes. These numbers represent the number of operations on memory not the memory access instructions where some memory access instructions may do more than one operation on the memory.

In all applications, loads are more frequent. The load operations ratio is about twice the store operations in the most of the studied applications. Some applications such as Cholesky, Fluidanimate, and Sawptions have even larger percentages of the load operations. Cholesky has about four times more of load operations than store operations because it operates on sparse matrices and needs to find the indices of the non-zero elements in these matrices. Fluidanimate has about five times more load operations than store operations. Sawptions has about three times more load operations than store operations. The load and store operations in Canneal are relatively equifrequent.

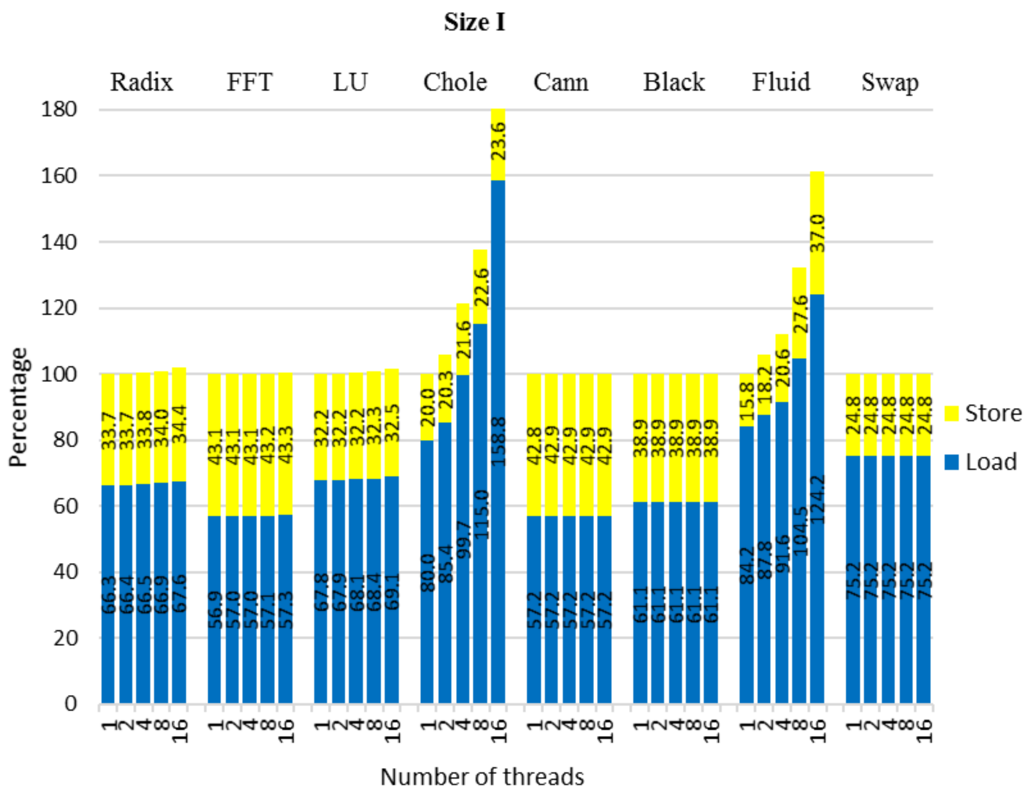
Table 4.2. The counts and percentages of load and store operations for one thread.

Size I								
	Radix	FFT	LU	Chole	Cann	Black	Fluid	Swap
No. of Loads (in 10^9)	0.034 (66.3%)	0.011 (56.9%)	0.015 (67.8%)	0.150 (80%)	1.586 (57.1%)	0.053 (61.1%)	0.500 (84.2%)	0.562 (75.2%)
No. of Stores (in 10^9)	0.017 (33.7%)	0.008 (43.1%)	0.007 (32.2%)	0.038 (20 %)	1.189 (42.9%)	0.034 (38.9%)	0.094 (15.8%)	0.186 (24.8%)

Size II								
No. of Loads (in 10^9)	0.285 (66.7%)	0.190 (57.1%)	0.109 (68.2%)	0.381 (77.6%)	3.577 (57.7%)	0.213 (61.1%)	1.146 (81.9%)	2.246 (75.2%)
No. of Stores (in 10^9)	0.143 (33.3%)	0.144 (42.9%)	0.051 (31.8%)	0.110 (22.4%)	2.625 (42.3%)	0.136 (38.9%)	0.253 (18.1%)	0.742 (24.8%)

Figure 4.1 shows the percentage of memory accesses for running the eight applications with various numbers of threads for the two problem sizes. The percentages are normalized to the number of memory accesses when running the respective applications with single thread. Thus, we can notice the parallelization overhead. As obvious, the parallelization overhead is negligible in most of the studied applications. However, there are two of the eight studied applications have a high percentage of parallelization overhead, which are Cholesky from SPLASH-2 and Fluidanimate from PARSEC. Cholesky has about 80% and 50% of memory accesses as overhead when running 16 threads for sizes I and II, respectively. This overhead is due to Cholesky's work on sparse matrices which have a larger communication to computation ratio. Fluidanimate has about 60% and 33% of memory accesses as overhead when running 16 threads for problem sizes I and II, respectively. This overhead is due to Fluidanimate's partitioning of the work among the threads and each thread handles its portion also interacts with other threads to handle the shared data.

Figures 4.2 and 4.3 show the number of synchronization calls per 10^6 memory accesses for problem sizes I and II. The synchronization calls include locks, barriers, and conditions calls. Fluidanimate has the highest number of synchronization calls, which is 3303 synchronization calls per 10^6 memory accesses when running 16 threads. This high rate is due to the high number of locks that are used at borders to avoid race conditions. All studied applications from PARSEC either have few numbers of synchronization calls such as Canneal, which has about 2 synchronization calls per 10^7 memory accesses when running 16 threads, or almost do not have synchronization calls such as Blackscholes and Swaptions because there is no sharing of data among threads. Choleskey and LU have large numbers of synchronization calls compared to the rest of the studied SPLASH-2 applications, where Cholesky has 82 synchronization calls per 10^6 memory accesses and



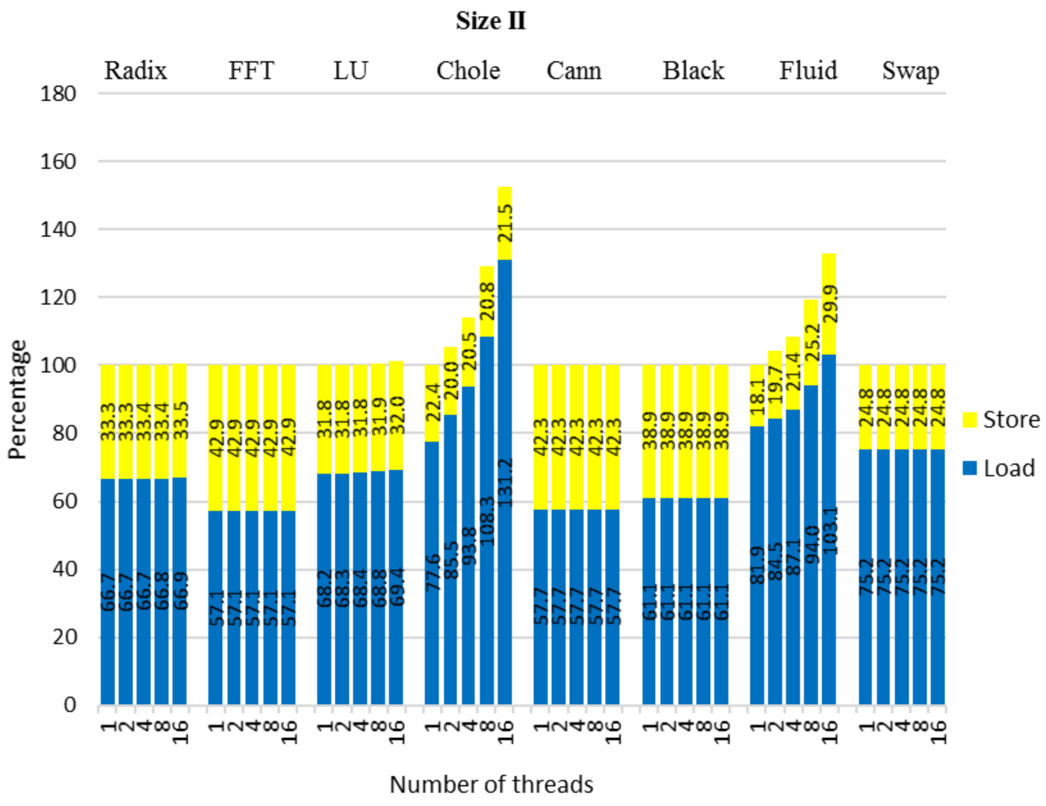


Figure 4.1. Percentage of memory accesses for 1-16 threads normalized to the memory accesses of one thread.

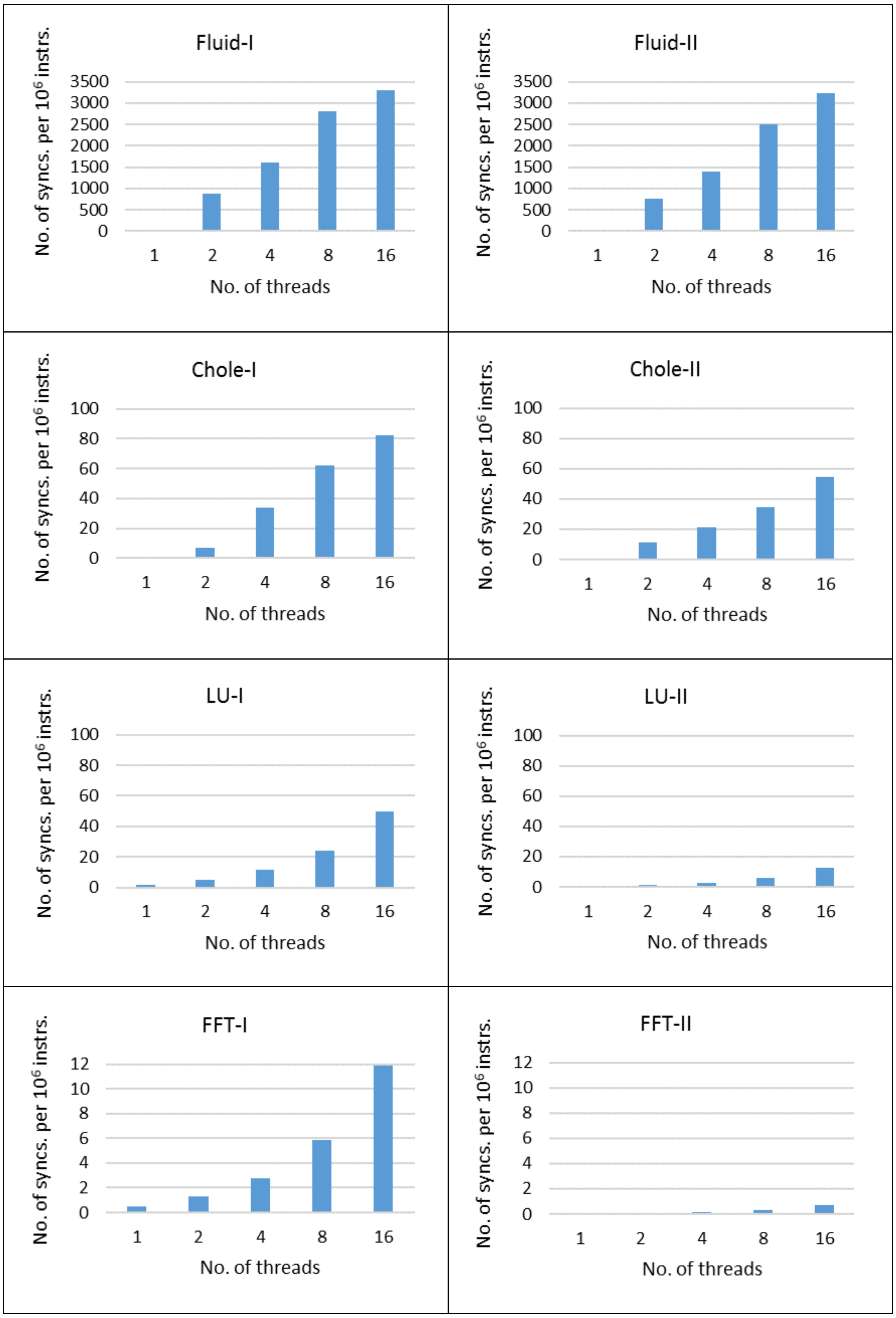


Figure 4.2. Number of synchronization calls per 10^6 memory accesses for Fluidanimate, Cholesky, LU, and FFT for Size I (left) and Size II (right).

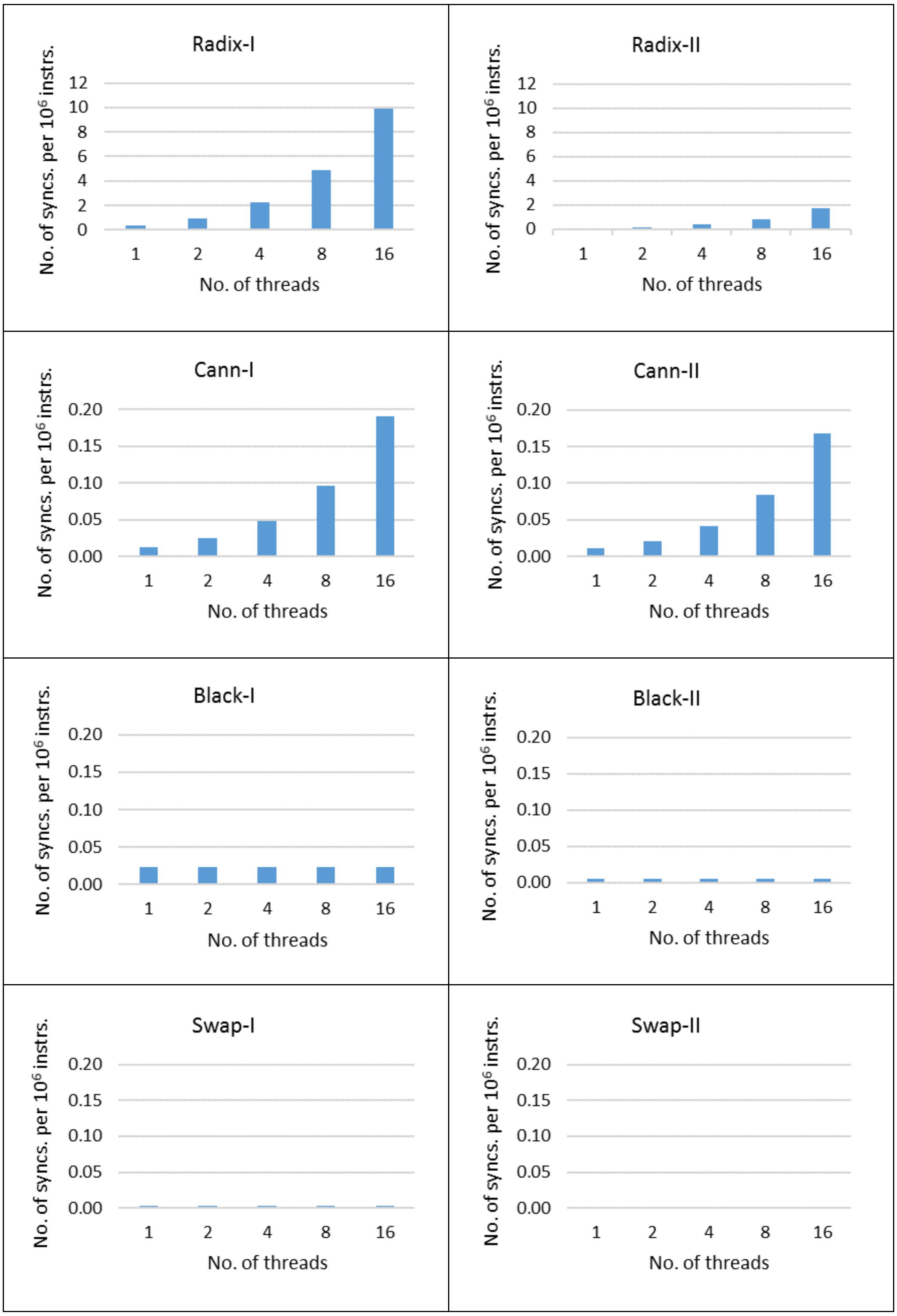


Figure 4.3. Number of synchronization calls per 10^6 memory accesses for Radix, Canneal, Blackscholes, and Swaptions for Size I (left) and Size II (right).

LU has 49 synchronization calls per 10^6 memory accesses when running 16 threads. The number of synchronization calls generally increase as the number of threads increases due to contention on shared data.

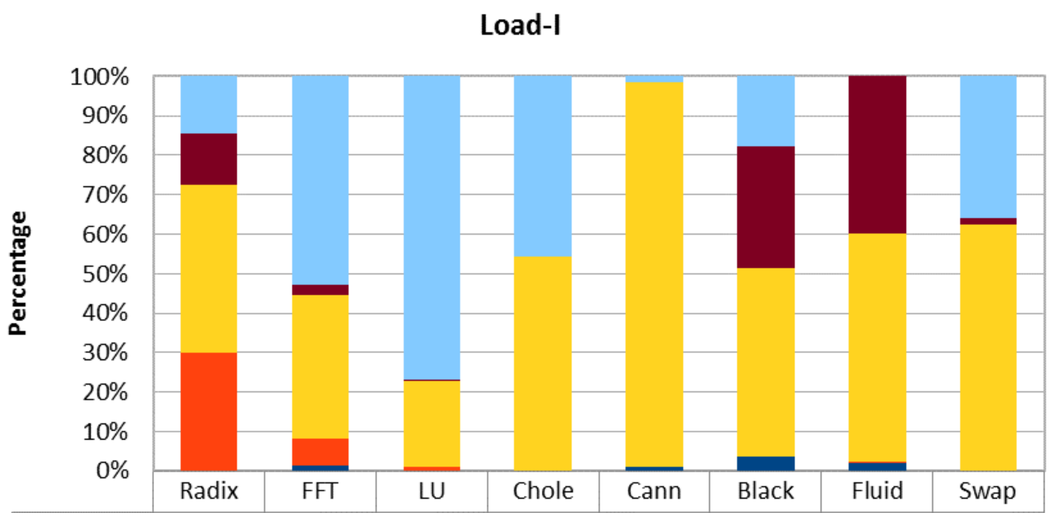
When the problem size is scaled up, the number of synchronization calls per 10^6 memory accesses generally decreases by different percentages in all the studied applications. This decrease is because these synchronization calls either are at fixed points of the code and they do not increase as the problem size increases such as in FFT, or they do not increase as much as processing data increases such as in other applications.

Figure 4.4 and 4.5 show the percentage of the byte, half-word (2 bytes), word (4 bytes), double-word (8 bytes), float (single-precision floating-point), and double-float (double-precision floating-point) load and store operations when running 16 threads for the two problem sizes. All the studied applications do not have any quad-word (16 bytes) or extended-float (extended-precision floating-point) memory accesses. All the studied applications are scientific benchmarks, which have a large percentage of floating-point operations except Radix and Canneal, which are integer kernel applications.

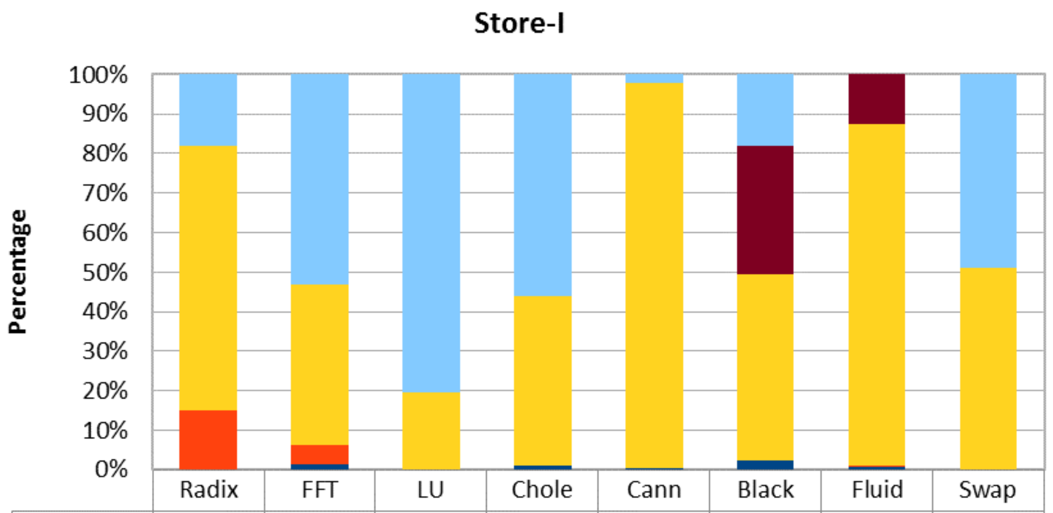
The percentages of byte and half-word accessed data is insignificant in almost all the studied applications except Radix and FFT that have 25% and 6% half-word load and store operations, respectively. These relatively large percentages are because they have large portion of integer computation.

4.3.2 Communication Patterns

As mentioned in Chapter 3, the communication occurs when multiple threads access a shared location. CIAT reports four types of the communication patterns, which are RAR, RAW, WAR, and WAW. Figure 4.6 shows the percentages of these four types of the communication patterns to the total number of memory accesses.

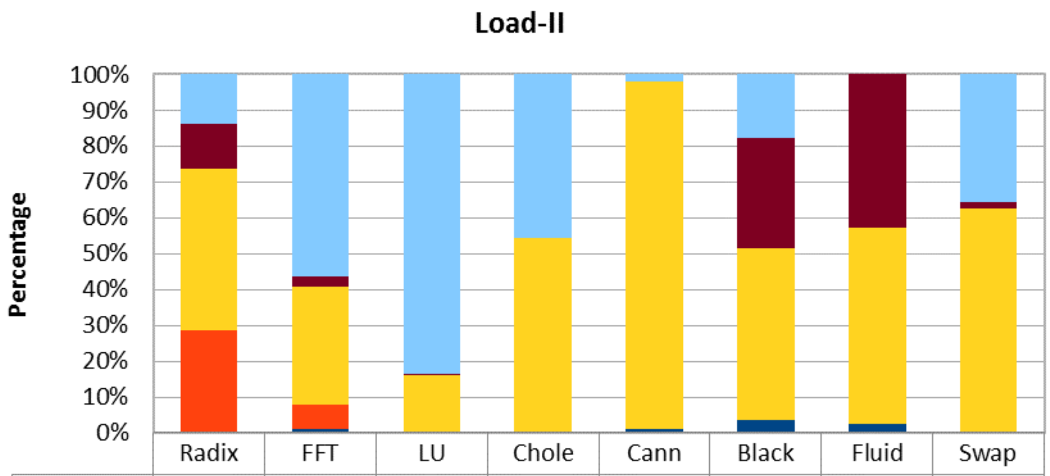


	Radix	FFT	LU	Chole	Cann	Black	Fluid	Swap
🟦 ld_dfloat	9.6	30.1	52.2	39.2	0.9	10.8	0.0	26.9
🟥 ld_float	8.6	1.5	0.3	0.0	0.0	18.8	30.8	1.3
🟩 ld_dword	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
🟨 ld_word	28.3	20.6	14.9	46.6	55.7	29.3	44.5	47.0
🟧 ld_half	19.8	4.0	0.6	0.0	0.0	0.0	0.2	0.0
🟦 ld_byte	0.0	0.7	0.0	0.1	0.6	2.2	1.6	0.0

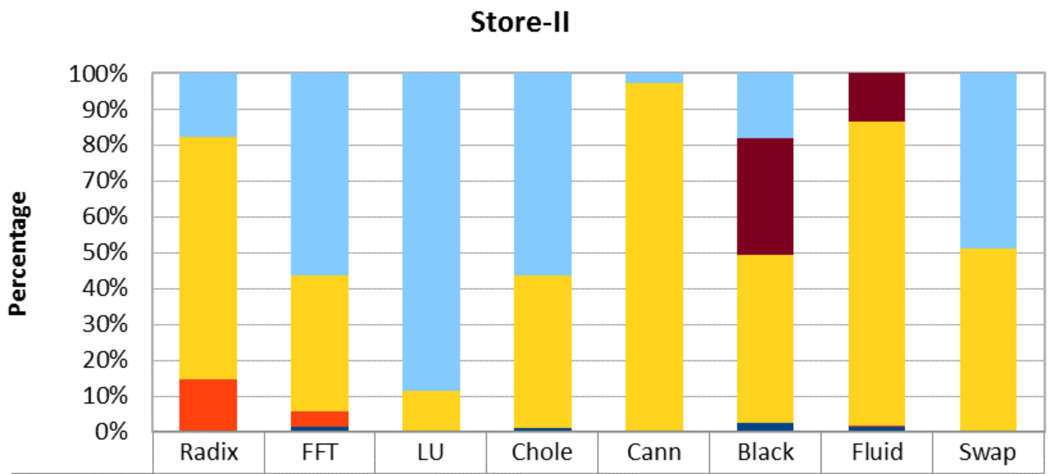


	Radix	FFT	LU	Chole	Cann	Black	Fluid	Swap
🟦 st_dfloat	6.1	22.9	25.7	7.9	0.9	7.1	0.0	12.2
🟥 st_float	0.0	0.0	0.0	0.0	0.0	12.6	2.9	0.0
🟩 st_dword	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
🟨 st_word	22.5	17.5	6.3	6.0	41.8	18.3	19.8	12.7
🟧 st_half	5.1	2.0	0.0	0.0	0.0	0.0	0.1	0.0
🟦 st_byte	0.0	0.7	0.0	0.1	0.2	0.9	0.2	0.0

Figure 4.4. Percentages of the load and store operations according to the type and size of accessed data for problem Size I.



	Radix	FFT	LU	Chole	Cann	Black	Fluid	Swap
🟦 ld_dfloat	9.3	32.3	57.2	39.2	1.2	10.8	0.0	26.9
🟥 ld_float	8.3	1.5	0.2	0.0	0.0	18.8	33.2	1.3
🟩 ld_dword	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
🟨 ld_word	30.0	19.0	10.8	46.6	55.9	29.2	42.3	47.0
🟧 ld_half	19.1	3.8	0.3	0.0	0.0	0.0	0.2	0.0
■ ld_byte	0.0	0.6	0.0	0.1	0.5	2.2	1.8	0.0



	Radix	FFT	LU	Chole	Cann	Black	Fluid	Swap
🟦 st_dfloat	5.9	24.2	27.9	7.9	1.2	7.1	0.0	12.2
🟥 st_float	0.0	0.0	0.0	0.0	0.0	12.6	3.1	0.0
🟩 st_dword	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
🟨 st_word	22.6	16.2	3.6	6.0	40.9	18.3	19.0	12.7
🟧 st_half	4.9	1.9	0.0	0.0	0.0	0.0	0.1	0.0
■ st_byte	0.0	0.6	0.0	0.1	0.2	0.9	0.3	0.0

Figure 4.5. Percentages of the load and store operations according to the type and size of accessed data for problem Size II.

Except for Radix that has a large portion of WAW pattern, which is about 36% of total communication patterns when running 16 threads due to the permutation operations, and Swaptions that has only WAW pattern when running 16 threads due to reuse of some variables. The most communication in the remaining applications are RAW and WAR. In general, PARSEC applications have less communication percentages, which are less than 0.5%, because the total numbers of memory accesses in PARSEC applications are very large compared to the numbers of communication events. For Blacksholes the numbers of communication events are small compared to the numbers of memory accesses. Therefore, the communication percentages are very small. Almost all the communication events in Blacksholes are RAW patterns because the memory locations are not updated by any WAR patterns. Only Canneal has a small portion of RAR communication patterns due to reading data not initialized by the application user code.

The communication percentage generally increases as the number of threads increases because of the increase in the data sharing. For Fluidanimate in problem size I, the communication percentage decreases when the number of threads increases from 8 to 16 because the percentage of increase in the total accesses is larger than the percentage of increase in the communication events.

When the problem is scaled up, the communication patterns increase in all applications, but the percentage of these communication patterns depends on the total number of accesses. Therefore, this percentage increases in some applications like Radix, Canneal, and Fluidanimate because the increase in communication patterns is more than the increase in total accesses. However, it decreases in some other applications like FFT, LU, Cholesky, and Swaptions because the increase in communication pattern is less than the increase in total accesses. In Blackscholes, it does not change because the increase in communication patterns and the increase in total accesses are equal.

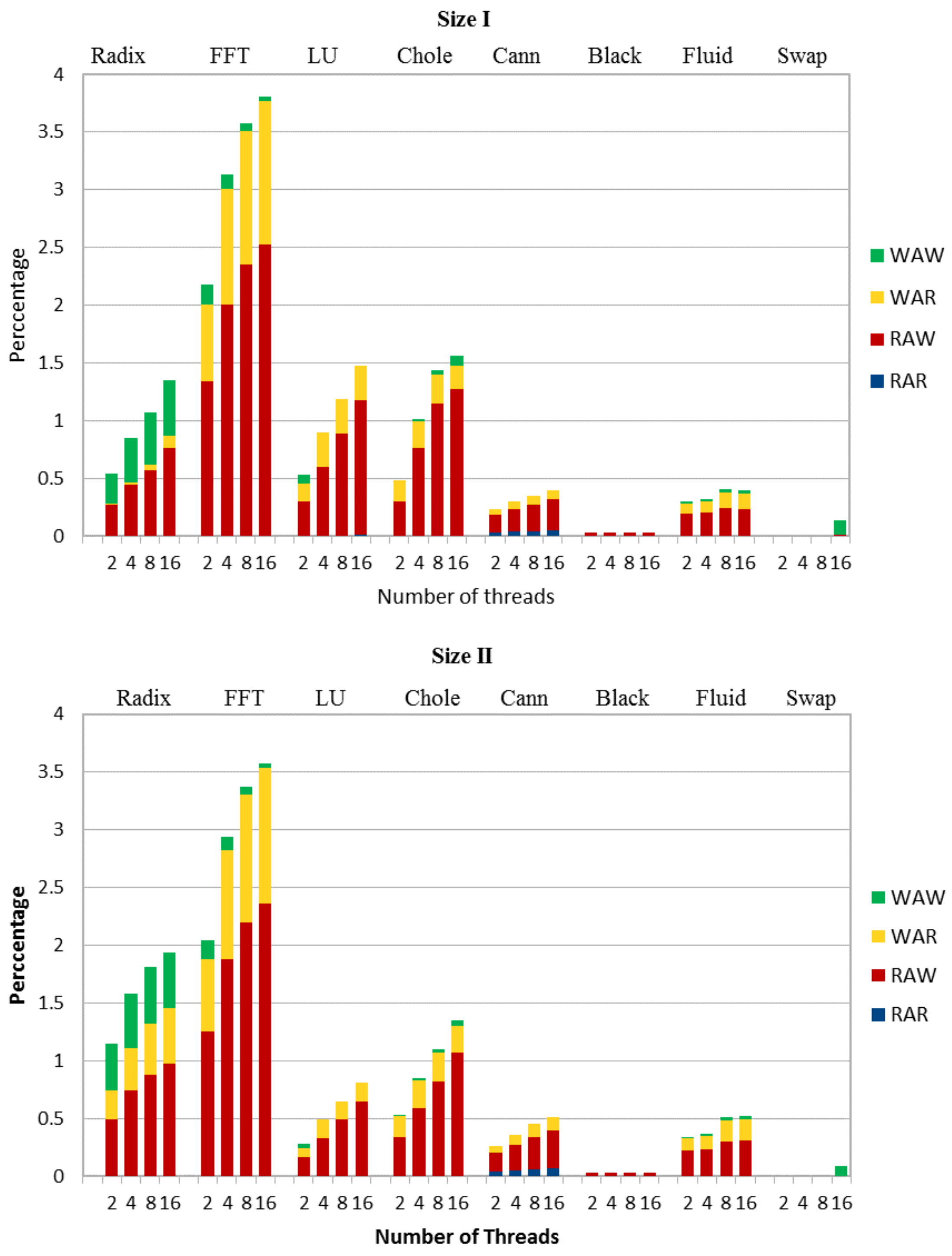


Figure 4.6. Percentages of the four types of communication patterns as a function of the number of threads.

Figure 4.7 shows the distributions of sharing degrees for RAW accesses in problem size I. It presents the percentages of sharing degrees when running 16 threads. These percentages are calculated by using the following formula:

$$S[p] \times 100 \Big/ \sum_{i=1}^{16} S[i]; \; p = 1,\dots,16$$

Where *S*[*p*] is the number of times that *p* threads read from a memory location after being previously written. Radix has a small sharing degree where 90% of shared data are shared with only one thread, 5% are shared with two threads, 3% are shared with four threads, and 2% are shared with three threads. FFT and Blackscholes also have a small sharing degree where almost all shared data are shared with one thread. Fluidanimate and Swaptions have two sharing degrees. In Fluidanimate, 75% of shared data are shared with one threads and 25% of shared data are shared with two threads. In Swaptions, 66% of shared data are shared with one threads and 34% are shared with two threads. LU, Cholesky, and Canneal have more than two sharing degrees. In LU, 92% of shared data are shared with four threads and the remaining percentages of shared data are shared with one, two, and three threads. In Cholesky, about 50% of shared data are shared with two and more threads and 50% of shared data are shared with one thread. In Canneal, 32% of shared data are shared with two and more threads and 68% of shared data are shared with one thread.

Figures 4.9 and 4.10 show the distributions of invalidation degrees of the WAR access for problem sizes I and II, respectively. They present the percentages of invalidation degrees when running 16 threads. These percentages are calculated by using the following formula:

$$I[p] \times 100 \Big/ \sum_{j=1}^{16} I[j]; \; p = 1,\dots,16$$

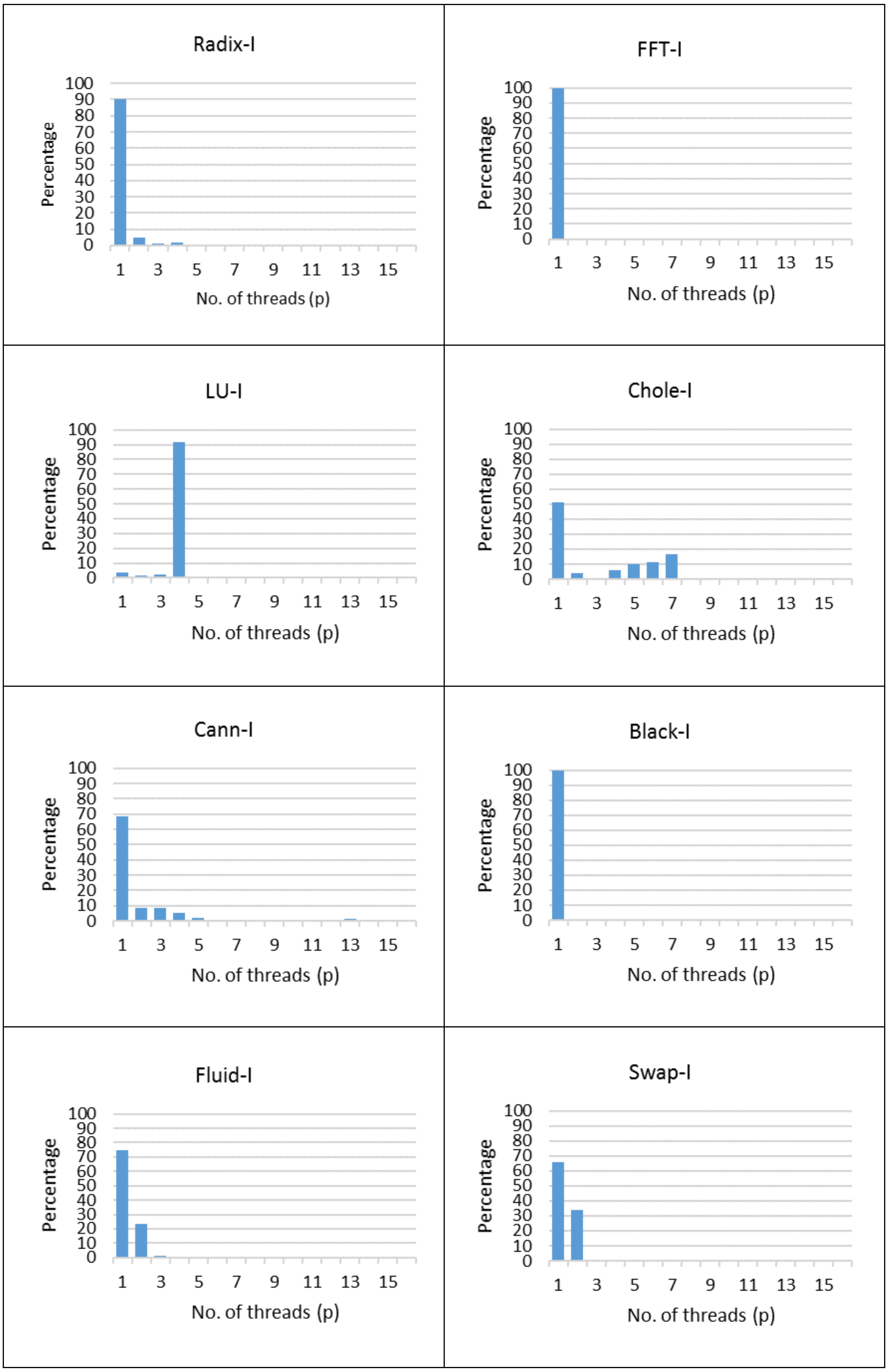


Figure 4.7. RAW sharing degree for 16 threads Size I.

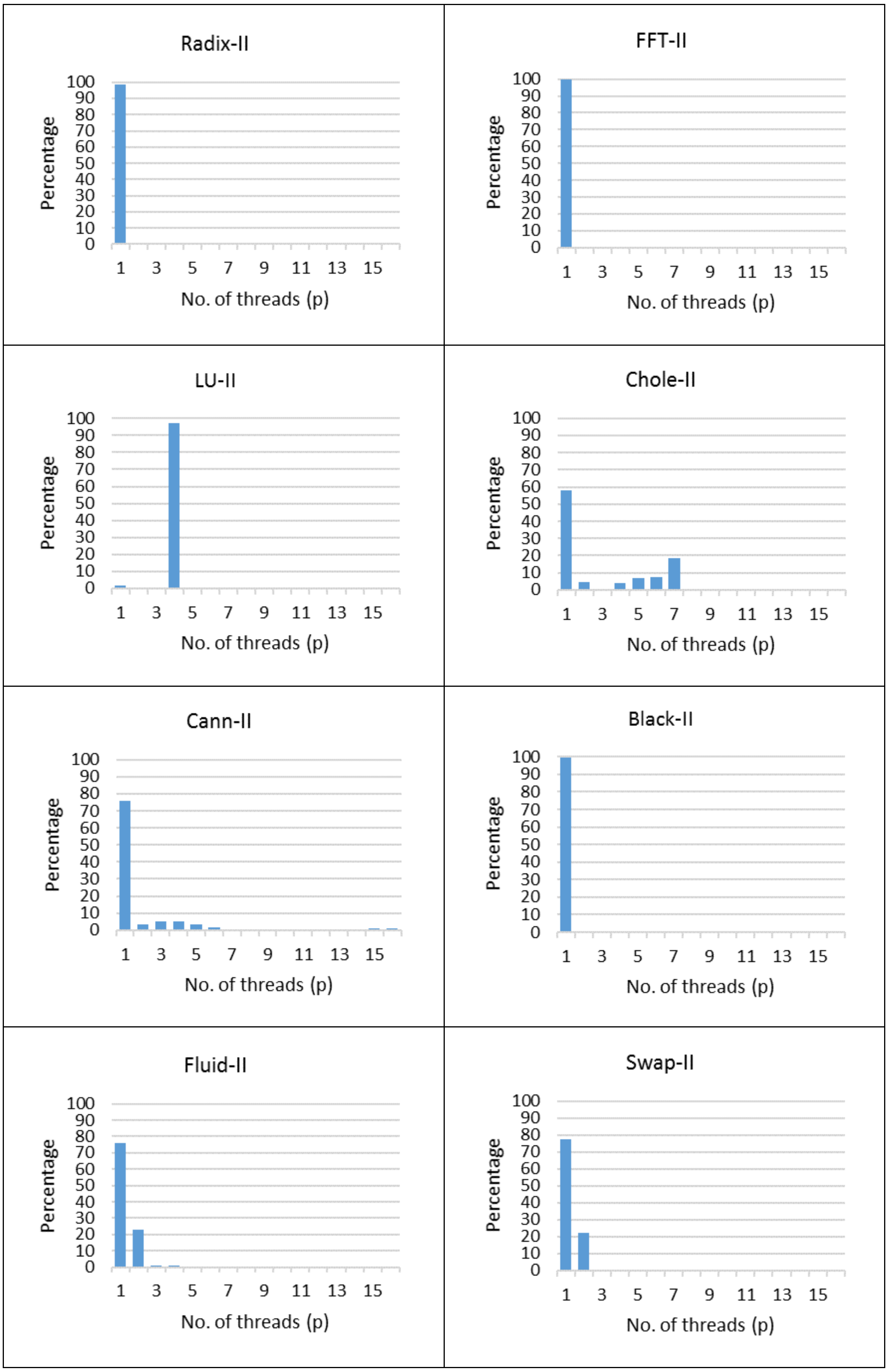


Figure 4.8. RAW sharing degree for 16 threads Size II.

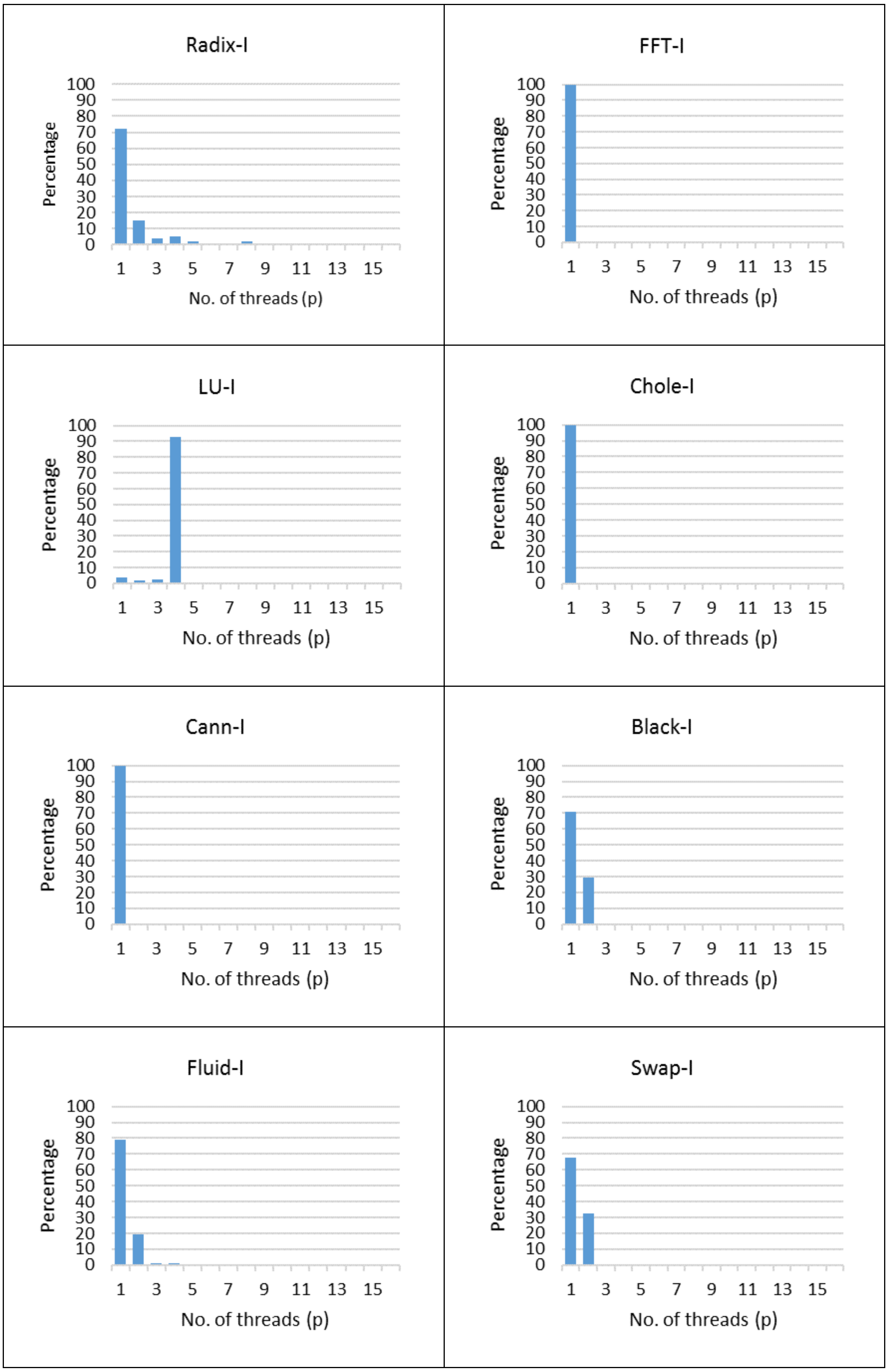


Figure 4.9. WAR invalidation degree for 16 threads Size I.

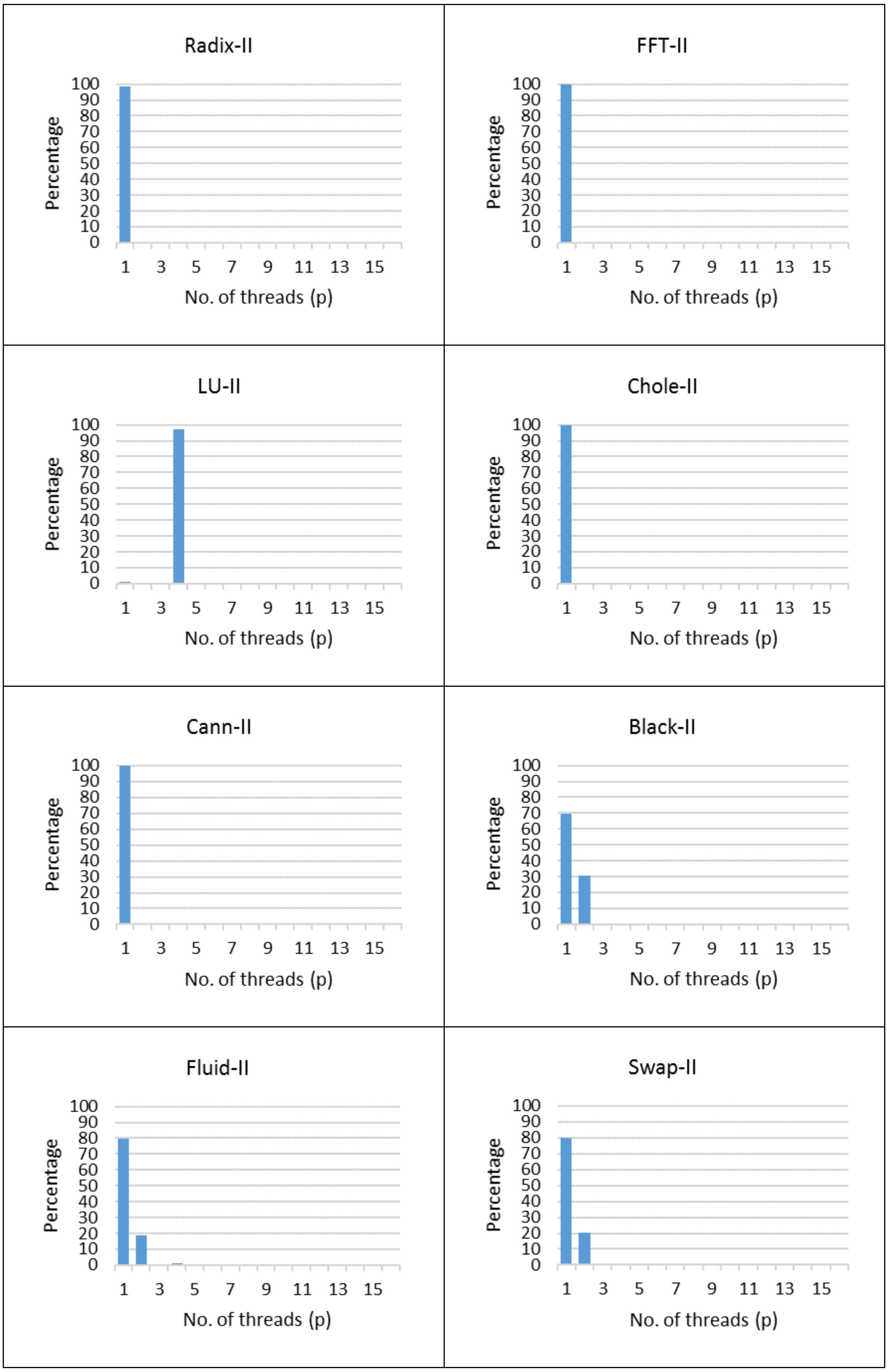


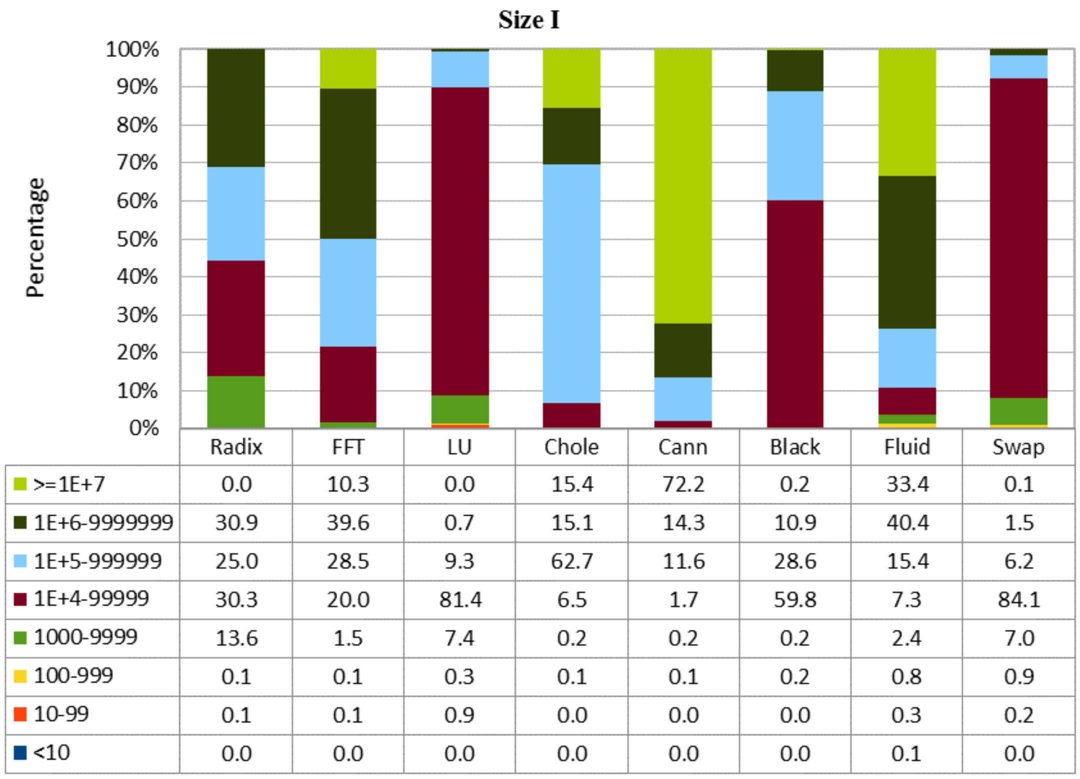
Figure 4.10. WAR invalidation degree for 16 threads Size II.

Where $I[p]$ is the number of times that a memory location was updated after being previously read by p threads. The invalidation degree in Radix, FFT, LU, Fluid, and Swaptions is almost similar to their sharing degree because the memory location that is accessed by one or more RAW patterns is updated by a WAR pattern. Choleskey and Canneal's invalidation degrees drop to one because some shared data is never updated by the WAR pattern. Blackscholes's invalidation degree is two, but in fact, only a small number of memory locations are updated. Only less than 120 memory locations are updated and this number is almost equal in the two problem sizes.

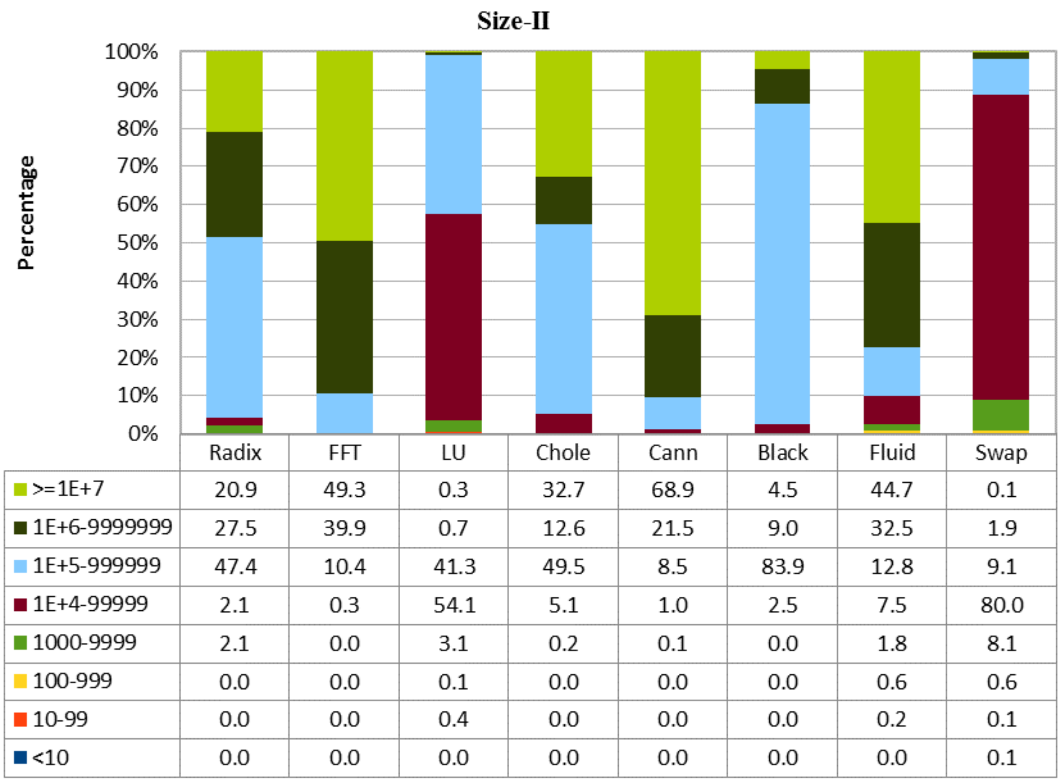
4.3.3 Communication Slack

The communication slack is a measure to know how much time is present between writing a value to a memory location and referencing it by either read or write operation. CIAT measures this time by counting the number of instructions from writing the value until referencing it. The communication slack is distributed into eight ranges from less than ten instructions to more than ten million instructions. Figure 4.11 shows the percentages of the communication slack distributions using 16 threads for problem sizes I and II. These percentages are the number of instructions in each range over the total number of memory accesses.

Radix's communication has a large percentage of slack in the ranges of tens of thousands of instructions and more, where it has 86.2% of slack in these ranges for problem size I and 97% of slack in these ranges for problem size II. FFT's communication has a large percentage of slack in the ranges of millions of instructions and more, where it has 78.4% of slack in these ranges for problem size I and 99.6% of slack in these ranges for problem II. LU's communication has a large percentage of slack in the ranges of tens of thousands of instructions and more, where it has 91.4% of slack in these ranges for problem size I and 96.4% of slack in these ranges for problem size II.



	Radix	FFT	LU	Chole	Cann	Black	Fluid	Swap
🟩 >=1E+7	0.0	10.3	0.0	15.4	72.2	0.2	33.4	0.1
⬛ 1E+6-9999999	30.9	39.6	0.7	15.1	14.3	10.9	40.4	1.5
🟦 1E+5-999999	25.0	28.5	9.3	62.7	11.6	28.6	15.4	6.2
🟥 1E+4-99999	30.3	20.0	81.4	6.5	1.7	59.8	7.3	84.1
🟩 1000-9999	13.6	1.5	7.4	0.2	0.2	0.2	2.4	7.0
🟨 100-999	0.1	0.1	0.3	0.1	0.1	0.2	0.8	0.9
🟧 10-99	0.1	0.1	0.9	0.0	0.0	0.0	0.3	0.2
🟦 <10	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.0



	Radix	FFT	LU	Chole	Cann	Black	Fluid	Swap
🟩 >=1E+7	20.9	49.3	0.3	32.7	68.9	4.5	44.7	0.1
⬛ 1E+6-9999999	27.5	39.9	0.7	12.6	21.5	9.0	32.5	1.9
🟦 1E+5-999999	47.4	10.4	41.3	49.5	8.5	83.9	12.8	9.1
🟥 1E+4-99999	2.1	0.3	54.1	5.1	1.0	2.5	7.5	80.0
🟩 1000-9999	2.1	0.0	3.1	0.2	0.1	0.0	1.8	8.1
🟨 100-999	0.0	0.0	0.1	0.0	0.0	0.0	0.6	0.6
🟧 10-99	0.0	0.0	0.4	0.0	0.0	0.0	0.2	0.1
🟦 <10	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1

Figure 4.11. Communication slack distributions for 16 threads.

Cholesky's communication has a large percentage of slack in the ranges of hundreds of thousands of instructions and more, where it has 93.2% of slack in these ranges for problem size I and 94.8% of slack in these ranges for problem size II. Canneal's communication has a large percentage of slack in the ranges of tens of millions of instructions and more, where it has 86.5% of slack in these ranges for problem size I and 90.4% of slack in these ranges for problem size II. Blackscholes's communication has a large percentage of slack in the ranges of tens of thousands of instructions and more, where it has 99.5% of slack in these ranges for problem size I and 99.9% of slack in these ranges for problem size II. Fluidanimate's communication has a large percentage of slack in the ranges of millions of instructions and more, where it has 89.2% of slack in these ranges for problem size I and 90% of slack in these ranges for problem size II. Swaptions's communication has a large percentage of slack in the ranges of tens of thousands of instructions and more, where it has 91.9% of slack in these ranges for problem size I and 91.1% of slack in these ranges for problem size II. In general, the percentage of slack tend to higher ranges as the problem size increases because the increase of processing data makes referencing the produced data takes longer time.

In all the studied applications, the communication has most of the slack in range tens thousands of instructions and more. These ranges are enough to make use of prefetching.

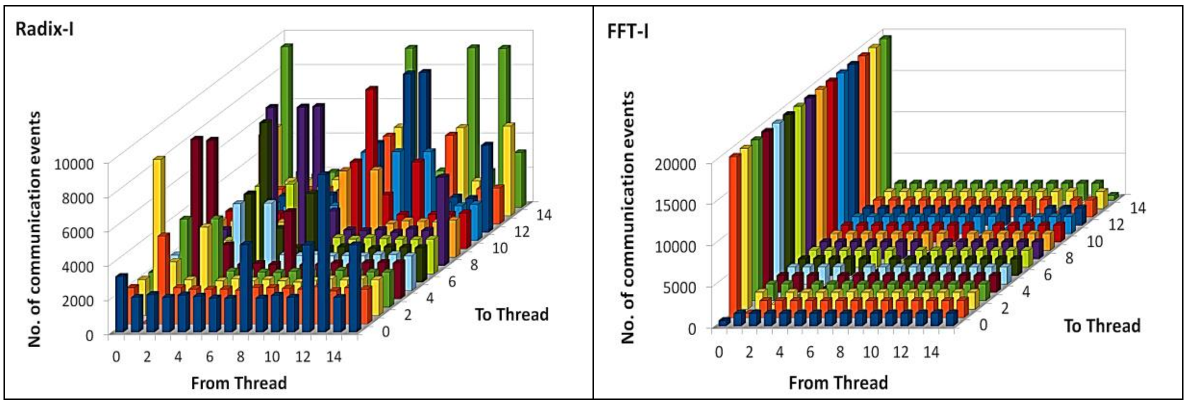
4.3.4 Communication Locality

As mentioned in Chapter 3, the communication locality is a measure how the cores communicate with each other. CIAT generates 2D matrix of communication events where the rows represent the data producer threads and the columns represent the data consumer threads.

Figures 4.12 and 4.13 present the communication matrices for the studied applications using 16 threads for problem sizes I and II, respectively. In Radix, each thread communicates with all other threads. Also, there are some additional communications with the neighbors where some odd threads communicates with only the next thread and some even threads may communicate with more than one threads.

In FFT, the communication is uniform communication. Additionally, there are a large amount of the communications from initial thread to every other thread. In LU, the communication is clustered within groups of $g = n/4$ threads where n is the number of threads that are used to run the application. For example, when running LU using 16 threads, $g = 16/4 = 4$ threads. Additionally, each thread communicates to and from the thread that is located after multiple of g threads from it. For example, if $g = 4$, Thread 1 communicates with Threads 5, 9, and 13. Moreover, initial thread communicates to all other threads and from the last *g* threads.

In Cholesky, the communication is non-uniform 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 communicates with all other threads. In Canneal, the communication is uniform communication. Additionally, each thread communicates with itself and initial thread communicates with all other threads. In Blackscholes, the communication is only with initial thread because there is no data sharing among threads. In Fluidanimate, the communication is non-uniform communication. Each thread communicates with itself and initial thread communicates with all other threads. In Swaptions, there are low communications. Each thread communicates with itself and there is some additional communication due to WAW accesses.



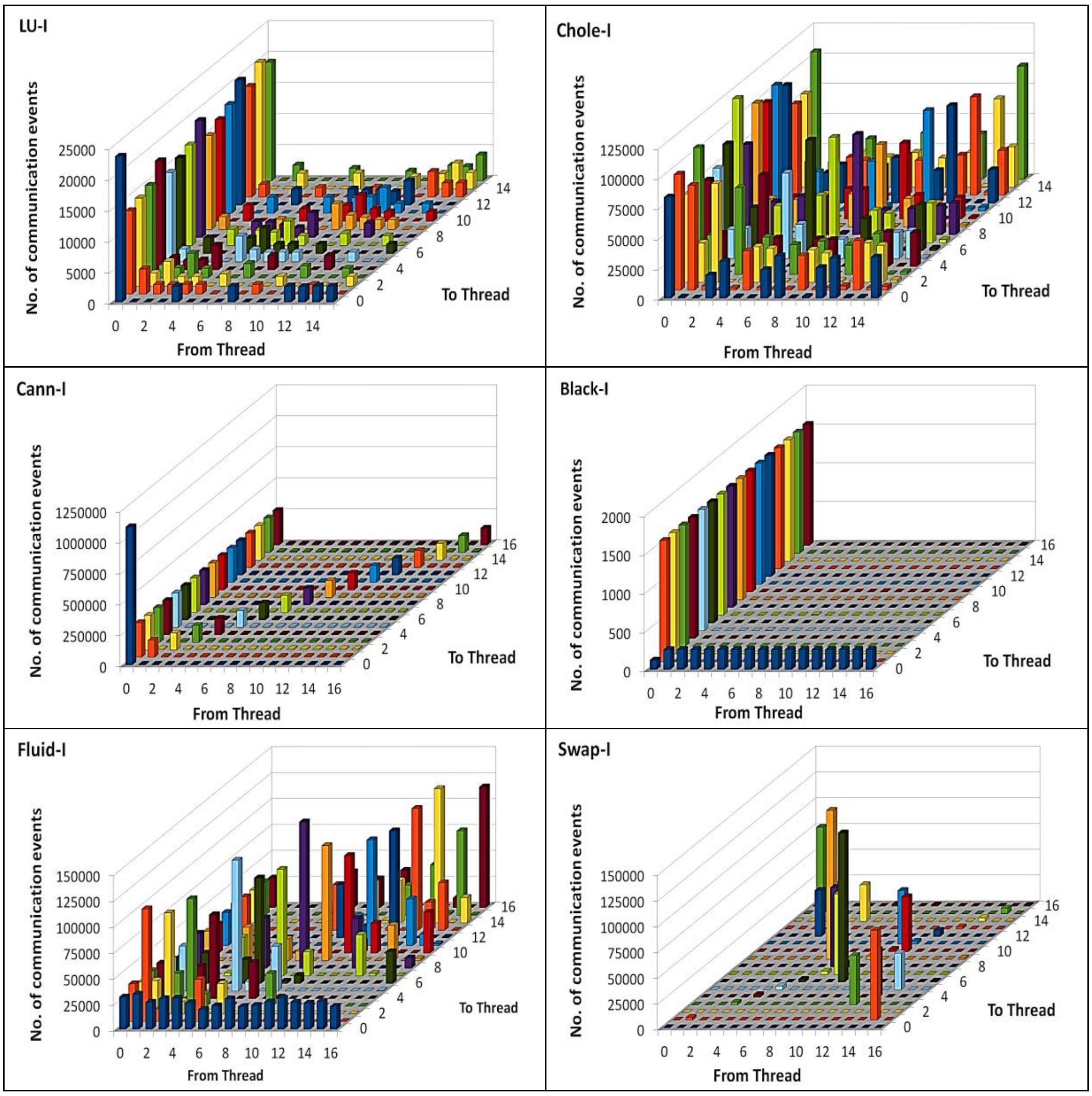
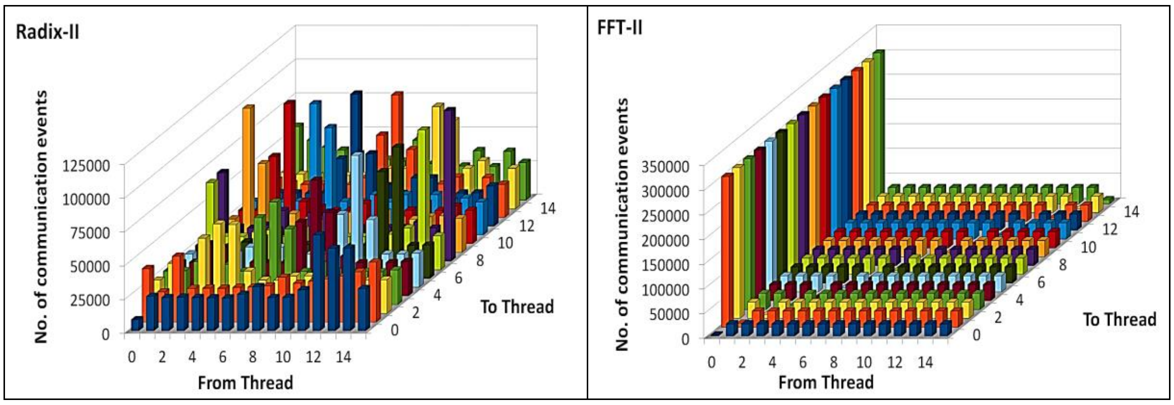


Figure 4.12. Number of communication events per thread pair for 16 threads Size I.



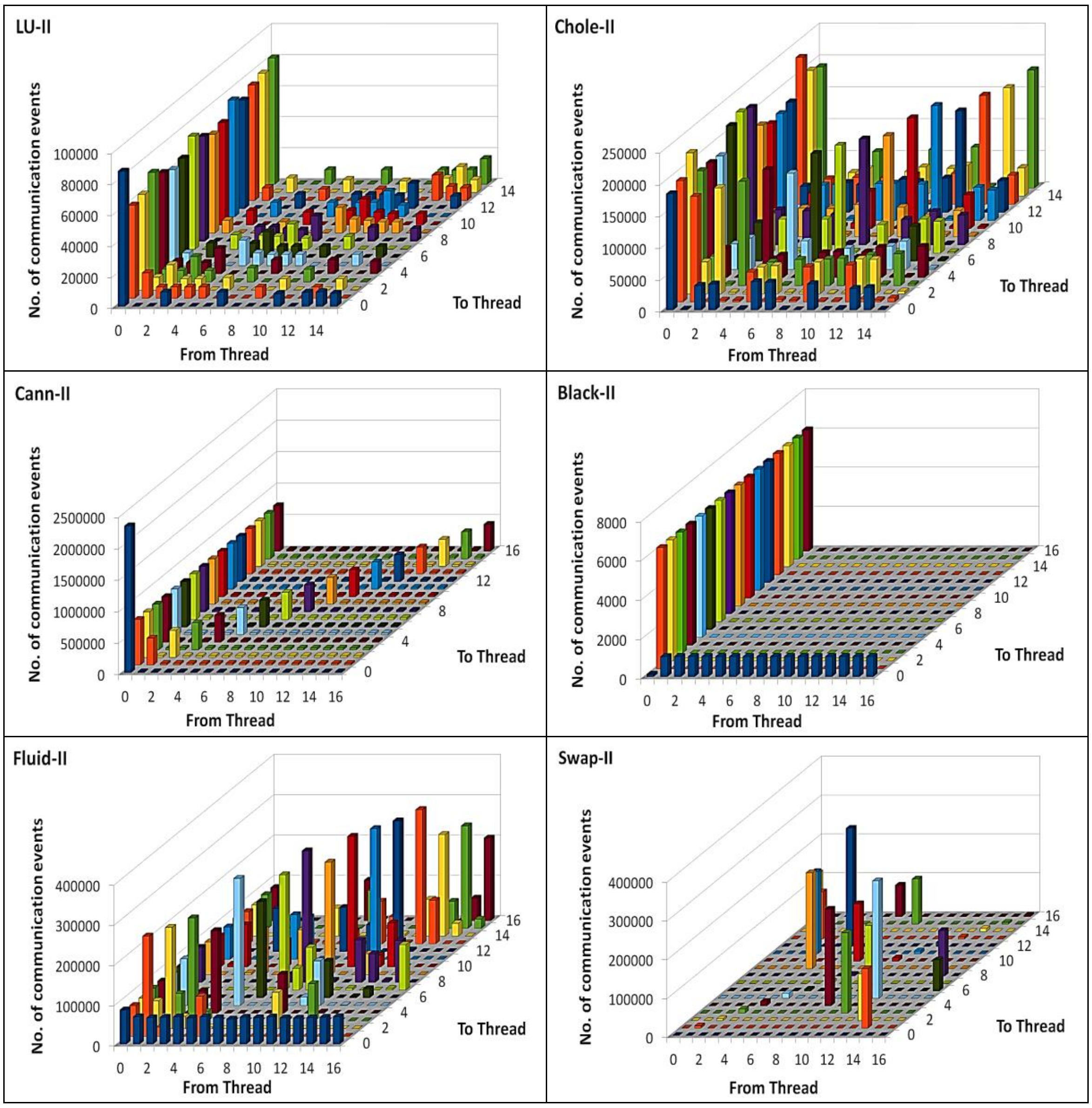


Figure 4.13. Number of communication events per thread pair for 16 threads Size II.

CHAPTER 5: CONCLUSIONS AND FUTURE WORK

This chapter presents the conclusions regarding the thesis's methodology, the developed tools, and characterization results. Also, it presents some proposed future work.

5.1 Conclusions

The purpose of this study was to characterize the hardware-configuration independent characteristics of representative parallel applications on multi-core platforms. To achieve this goal, first, 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-2 suite, which are Radix, FFT, LU, and Cholesky. The other four are from PARSEC suite, which are Canneal, Blackscholes, Fluidanimate, and Swaptions. These particular 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: Size I and Size II, where Size I < Size II.

Second, Intel's Pin instrumentation tool is used to instrument the studied applications. We chose Pin because it has a rich set of API's that can be used to instrument applications without the need to master the underlying instruction set and it is widely used in multi-core research. We developed a tool based on Pin, which is called PSMAIT, that captures every memory access then it sends a simple trace record to CIAT. This record contains the type of the access, its size, and the starting virtual address of the memory location referenced. PSMAIT sends memory access records to CIAT on-the-fly by using pipes and it receives confirmation feedback from CIAT. Additionally, PSMAIT captures the synchronization calls such as mutex lock, barrier, and conditional wait then it sends special trace records to CIAT. By these records and the response feedback from CIAT, PSMAIT and CIAT interact with each other to control the parallel application execution.

Finally, we ported CIAT, which was developed for RISC multiprocessor systems, to work on CISC multi-core systems. We modified CIAT to handle the memory accesses that have larger operand size such as extended floating-point, quad-word, and even double quad-word. In addition, we modified CIAT to support Pthreads for thread management and synchronization and support the applications that use conditional wait calls as barrier. Moreover, we modified CIAT to activate receiving the traces from only the spawned threads at the beginning of the parallel phase. Any time a new thread is spawned, CIAT activates receiving traces from it.

After the PSMAIT and CIAT are ready, we conducted many experiments of the studied applications with various numbers of threads for the two problem sizes on a multi-core system. We characterized several aspects of the studied applications' characteristics including:

- *Memory access instructions* include the number of memory accesses, the number of synchronization calls, and percentage of memory accesses by type of access and access data size. This characterization is useful to know the amount of parallelization overhead.
- *Communication patterns* include the amount of communication for each type of the four-commination patterns, which are RAW, WAR, WAW, and RAR. Moreover, we characterized the sharing degree of RAW access and the invalidation degree of WAR access. Characterizing the communication patterns is important to know which of the communication patterns are common. Thus, facilitating the design of system that support these patterns efficiently in the

current applications and facilitating tuning applications to have less expensive patterns.

- *Communication slack* is the amount of time between writing a value to a memory location and referencing it by other cores. The communication slack characterization is useful to know whether the target applications can make use of prefetching or not.
- *Communication locality* is a measure how the cores communicate with each other. Characterizing the communication locality helps both software developers in assigning threads to the cores and hardware designers in selecting suitable system topology.

In general, the number of memory accesses, in the studied applications, does not change significantly as the number of threads increases. However, there are some applications such as Cholesky and Fluidanimate that show high parallelization overhead and intensive synchronization calls. This overhead may limit the speedup of these applications. The largest percentages of memory accesses in the scientific applications are of floating point accesses.

The most common communication patterns are RAW and WAR. Swaptions shows a large percentage of WAW when running more than eight threads due to reuse of some variables. In general, the communication percentages in PARSEC applications are smaller than the communication percentages in SPLASH-2 applications. Radix, FFT, and Blackscholes have a small sharing degree. Fluidanimate and Swaptions have a medium sharing degree compared with the other applications. LU, Cholesky, and Canneal have a large sharing degree compared with the other applications.

In all the studied applications, the communication has most of the slack in the range of tens of thousands of instructions and more. These ranges are enough to make use of prefetching. There is considerable diversity in communication locality of the studied applications. Some applications show uniform communication such as FFT, Canneal, and Blackscholes. Others show non-uniform communication and almost in all applications, initial thread communicates with other threads. The important characteristics of each applications are summarized in the following paragraphs.

Radix is an integer kernel application, so it has a small percentage (24%) of floating-point accesses. It has a large portion (36%) of the WAW communication pattern due to the permutation operations. Radix has small sharing and invalidation degrees, where it has more than 90% of data sharing and invalidation with only one thread. Each thread communicates with all other threads.

FFT has a large percentage of communication percentage (3.81%) due to matrix transposition operations. Like Radix, FFT has a small sharing and invalidation degrees, where it has 100% of the data sharing and invalidation with only one thread. FFT has uniform communication and there are large amounts (85-90%) of communications from initial thread.

LU has large percentage of load and store floating-point accesses (85%) because it works on double floating-point dense matrices. It has large sharing and invalidation degrees, where it has more than 92% of the data sharing and invalidation with four threads or more because the communication is clustered within groups of threads.

Cholesky has large parallelization overhead (80%) and high percentage of synchronization calls (82 calls per 10^6 memory accesses) because it has a large communication to computation ratio. It divides nonzero elements into blocks among threads and it requires synchronization to update the values in each block. Cholesky has large sharing degree, where it has 42% of the data sharing with two threads or more, but the invalidation degree is one because some shared data is never updated by the WAR pattern. The communication in Cholesky is non-uniform communication.

Canneal is another integer kernel application, so it has a small number of floating-point load and store accesses (2.4%). It has a small portion (14%) of RAR communication pattern due to reading data not initialized by the application's user code. Canneal has large sharing degree, where it has 32% of the data sharing with two threads or more, but the invalidation degree is one because some shared data is never updated by the WAR pattern. It has uniform communication.

Blackscholes does not have synchronization calls because threads process the work independently from each other and there is no communication among threads. Almost all the communication patterns in Blacksholes are RAW accesses. Both the sharing and invalidation degrees are one, where it has 100% of sharing data and invalidation with only one thread. The communication is only with the initial thread.

Fluidanimate has high percentage (60%) of parallelization overhead and large number of synchronization calls (3300 calls per 10^6 memory accesses) because Fluidanimate divides the work into cells among threads where the cells that are located on the borders must be locked before being modified. The number of these cells increases as the number of threads increases. The sharing and invalidation degrees are equal in Fluidanimate. It has non-uniform communication.

Swaptions does not have synchronization calls because threads process the work independently from each other and there is no communication among threads. Almost 100% of the total communication patterns in Swaptions are WAW patterns when running 16 threads due to reuse of some variables.

5.2 Future Work

In our work, CIAT reports only the memory access instructions on multi-core systems and this is an approximation of the instruction stream. Therefore, the future work is to develop CIAT to capture the entire instruction stream on a multi-core system. In addition, we need to develop CIAT to handle new parallelization techniques where there are some modern applications that use different parallelization techniques such as the pipeline parallelization model that is used in three applications of PARSEC suite, which are Dedup, Ferret, and X264.

Abandah, (1998) developed three tools for characterizing applications on distributed shared memory systems, which are CIAT, CDAT, and Communication Contention Analysis Tool (CCAT). In addition, he developed a tool to characterize event time distributions, which is called Time Distribution Analysis Tool (TDAT). In this thesis, we ported one of these tools, which is CIAT to work on multi-core systems. We plan to port the remaining tools to work on the modern multi-core systems. As mentioned in this thesis, CIAT characterizes the inherent characteristics of applications that do not depend on system configuration. Other characteristics depend on configuration parameters such as cache misses and false sharing need to be characterized. Therefore, it is important to port CDAT and CCAT for characterizing these characteristics. Also, TDAT is important to characterize the communication events over time.

APPENDIX A: USAGE INSTRUCTIONS

All developed tools and the studied applications are put in one compressed file, which is called `new-ciat.tar.gz`. Extract the compressed file in the home directory. The `new-ciat` directory contains three directories, which are `ciat`, `pin`, and `parsec-splash`, which contains both PARSEC and SPLASH-2 suites.

A.1 Applications Installation

First, go to `parsec-splash` directory and add the needed environment variables by running the following commands:

```
cd ~/new-ciat/parsec-splash
source env.sh
```

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`
expat	`sudo apt-get install libexpat1-dev`
xt	`sudo apt-get install libxt-dev`
xext	`sudo apt-get install libxext-dev`
xmu	`sudo apt-get install libxmu-dev`
xi	`sudo apt-get install libxi-dev`
m4	`sudo apt-get install m4`
perl5	1- Download perl, perl-base, and perl-module of version 5.20.1-1 from https://launchpad.net/ubuntu/+source/perl 2- Force install by "`sudo dpkg --force-all -i perl*`", where you must run it from the same downloaded files directory.

In addition, there are other libraries that are included in `parsec-splash` directory. Install them by running the following command:

```
parsecmgmt -a  build –p libs
```

Second, to build one of the applications, run the following command:

```
 parsecmgmt -a build -p [suite].[app]
```

Where `suite` is the benchmark suite, either `parsec` or `splash2` and `app` is the name of application that you want to build. The names for `suite.app` are in Table A.2. For example to build Radix, run the following command:

```
parsecmgmt -a build –p splash2.radix
```

Table A.2. The names of `suite.app` for the studied applications.

Application	Suite.app
Radix	`splash2.radix`
FFT	`splash2.fft`
LU	`splash2.lu_cb`
Chole	`splash2.cholesky`
Cann	`parsec.canneal`
Black	`parsec.blackscholes`
Fluid	`parsec.fluidanimate`
Swap	`parsec.swaptions`

Note: The source files of some PARSEC's and SPLASH-2's applications have been modified according to the instructions in the site:

https://lists.cs.princeton.edu/pipermail/parsec-users/2012-September/001421.html

A.2 CIAT Installation

After building the applications, go to `ciat` directory and build it by running the following commands:

```
cd ../ciat

make -f ciat.mak
```

Copy the executable file of CIAT to `psmait` directory by running the following command: `cp ./ciat  ../pin/source/tools/psmait`

A.3 Pin Installation

After installing the applications and CIAT, it is time to install Pin, but before that, you need to add the following environment variables to **`/etc/environment`**:

```
PATH=”/home/[user-name]/new-ciat/pin”
LD_LIBRARY_PATH="/usr/local/lib"
```

Where `user-name` is your user name. You can edit **`/etc/environment`** by running the following commands:

```
sudo gedit /etc/environment
```

After adding the environment variables, go to `psmait` directory within the `pin` directory and build PSMAIT by running the following commands:

```
cd ~/new-ciat/pin/source/tools/psmait
make obj-ia32/psmait.so
```

Where `obj-32` is a directory that will be created in `psmait` directory when building PSMAIT.

A.4 Running

To run the tools, run the following commands as root:

```
sudo su
cd ~/new-ciat/pin/source/tools/psmait
pin -t obj-ia32/psmait.so -n [number of threads] -f [CIAT report
file name] -- [application’s path/app] [application’s
parameters]
```

Table A.3 shows the full path and the parameters for each application. For example to characterize FFT on four threads and using problem Size I, run the following command:

```
pin –t obj-ia32/psmait.so –n 4 –f fft -- ~/new-ciat/parsec-
splash/ext/splash2/kernels/fft/inst/i686-linux.gcc/bin/
fft -p4 -m16
```

Table A.3. The full path and the parameters for the eight applications.

Application	Application’s path /app	Parameters	
		Size I	Size II

Radix	~/new-ciat/parsec-splash/ext/splash2/kernels/radix/inst/i686-linux.gcc/bin/radix	-p*n* -r1024 -n262144 -m524288	-p*n* -r1024 -n2097152 -m4194304
FFT	~/new-ciat/parsec-splash/ext/splash2/kernels/fft/inst/i686-linux.gcc/bin/fft	-p*n* -m16	-p*n* -m20
LU	~/new-ciat/parsec-splash/ext/splash2/kernels/lu_cb/inst/i686-linux.gcc/bin/lu_cb	-p*n* -n256 -b16	-p*n* -n512 -b16
Chole	~/new-ciat/parsec-splash/ext/splash2/kernels/cholesky/inst/ i686-linux.gcc/bin/cholesky	-p*n* < tk15.O	-p*n* < tk29.O
Cann	~/new-ciat/parsec-splash/pkgs/kernels/canneal/inst/i686-linux.gcc/bin/canneal	*n* 10000 2000 100000.nets 32	*n* 15000 2000 200000.nets 64
Black	~/new-ciat/parsec-splash/pkgs/apps/blackscholes/inst/i686-inux.gcc/bin/blackscholes	*n* in_4K.txt prices.txt	*n* in_16K.txt prices.txt
Fluid	~/new-ciat/parsec-splash/pkgs/apps/fluidanimate/inst/i686-linux.gcc/bin/fluidanimate	*n* 5 in_35K.fluid out.fluid	*n* 5 in_100K.fluid out.fluid
Swap	~/new-ciat/parsec-splash/pkgs/apps/swaptions/inst/i686-linux.gcc/bin/swaptions	-ns 16 -sm 10000 -nt *n*	-ns 32 -sm 20000 -nt *n*

Notes: *n* = number of threads.

All input files must be in the same directory of the application.

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.

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.

Bertels, K., Ostadzadeh, S. A., and Meeuws, R. J. (2011), Advanced profiling of applications for heterogeneous multi-core platforms. In Proceedings of the **2011 International Conference on Engineering of Reconfigurable Systems & Algorithms (ERSA**), Las Vegas, Nevada, USA, 18-21 July 2011, 171-183.

Bhadauria, M., Weaver, V. M., and McKee, S. A. (2009), Understanding PARSEC Performance on Contemporary CMPs. In Proceedings of the **2009 IEEE International Symposium on Workload Characterization (IISWC),** Austin, TX, USA, 4-6 October 2009, 98-107.

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.

Bienia, C., Kumar, S., Singh, J. P., and Li, K. (2008), The PARSEC Benchmark Suite: Characterization and Architectural Implications. In Proceedings of the **17th International Conference on Parallel Architectures and Compilation Techniques (PACT)**, Toronto, ON, Canada, 25-29 October 2008, 72-81.

Bienia, C., Kumar, S., and Li, K. (2008b). PARSEC vs. SPLASH-2: 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.

Borkar, S. (2007), Thousand Core Chips: A Technology Perspective. In Proceedings of the **44th Annual Design Automation Conference (DAC),** San Diego, CA, USA, 4-8 June 2007, 746-749.

Buck, B. and Hollingsworth, J. K. (2000), An API for Runtime Code Patching. **International Journal of High Performance Computing Applications**, 14(4), 317-329.

Buttlar, D. and Farrell, J. (1996), **PThreads Programming: A POSIX Standard for Better Multiprocessing**, (1st Ed.). California: O'Reilly Media, Inc.

Cantrill, B., Shapiro, M. W., and Leventhal, A. H. (2004), Dynamic Instrumentation of Production Systems. In Proceedings of the **2004 USENIX Annual Technical Conference, General Track**, Boston, MA, USA, 27 June -2 July 2004, 15-28.

Chai, L., Gao, Q., and Panda, D. K. (2007), Understanding the Impact of Multi-Core Architecture in Cluster Computing: A Case Study with Intel Dual-Core System. In Proceedings of the **7th Symposium on Cluster Computing and the Grid (CCGrid)**, Rio de Janeiro, Brazil, 14-17 May 2007, 471-478.

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.

Dagum, L. and Menon, R. (1998), OpenMP: An Industry Standard API for Shared-Memory Programming. **IEEE Computational Science and Engineering**, 5(1), 46-55.

Das, R., Ausavarungnirun, R., Mutlu, O., Kumar, A., and Azimi, M. (2013), Application-to-Core Mapping Policies to Reduce Memory System Interference in Multi-Core Systems. In Proceedings of the **19th International Symposium on High Performance Computer Architecture (HPCA),** Shenzhen, China, 23-27 February 2013, 107-118.

Dean, J. and Ghemawat, S. (2004), MapReduce: Simplified Data Processing on Large Clusters. In Proceedings of the **6th Symposium on Operating System Design and Implementation (OSDI),** San Francisco, CA, USA, 6-8 December 2004, 137-150.

Devadas, S. (2013), Toward a Coherent Multicore Memory Model. **IEEE Computer**, 46(10), 30-31.

Dey, T., Wang, W., Davidson, J. W., and Soffa, M. L. (2011), Characterizing Multi-Threaded Applications Based on Shared-Resource Contention. In Proceedings of the **2011 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)**, Austin, TX, USA, 10-12 April 2011, 76-86.

Dongarra, J., Moore, S., Mucci, P., Seymour, K., & You, H. (2004). Accurate cache and TLB characterization using hardware counters. In Proceedings of the **International Conference on Computational Science** (**ICCS**), Kraków, Poland, 6-9 June 2004, 432-439. Springer Berlin Heidelberg.

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

Ferdman, M., Adileh, A., Kocberber, O., Volos, S., Alisafaee, M., Jevdjic, D., ansu Kaynak, C., Popescu, A., Ailamaki, A., and Falsafi, B. (2012). Clearing the clouds: a study of emerging scale-out workloads on modern hardware. **ACM SIGARCH Computer Architecture News**, 40(1), 37-48.

Hennessy, J. L., and Patterson, D. A. (2012). **Computer architecture: a quantitative approach,** (5th Ed.). Massachusetts: Elsevier.

HP (1997), **Exemplar Programming Guide for HP-UX Systems,** (1st ed.), Hewlett-Packard.

HP (1994), **PA-RISC 1.1 Architecture and Instruction Set**, (3rd ed.), Hewlett-Packard.

Jaleel, A., Mattina, M., and Jacob, B. (2006), Last Level Cache (LLC) Performance of Data Mining Workloads on a CMP- A Case Study of Parallel Bioinformatics Workloads. In Proceedings of the **12th International Symposium on High-Performance Computer Architecture (HPCA)**, Austin, TX, USA, 11-15 February 2006, 88-98.

Jaleel, A., Cohn, R. S., Luk, C. K., and Jacob, B. (2008), CMP$im: A Pin-Based On-The-Fly Multi-Core Cache Simulator. In Proceedings of the **4th Annual Workshop on Modeling, Benchmarking and Simulation (MoBS)**, Beijing, China, 22 June 2008, 28-36.

Jaleel, A., Najaf-Abadi, H. H., Subramaniam, S., Steely, S. C., and Emer, J. (2012), CRUISE: Cache Replacement and Utility-Aware Scheduling. **ACM SIGARCH Computer Architecture News**, 40(1), 249-260.

Jia, Z., Wang, L., Zhan, J., Zhang, L., and Luo, C. (2013). Characterizing Data Analysis Workloads in Data Centers. **2013 IEEE International Symposium on Workload Characterization (IISWC)**, Portland, OR, USA, 22-24 September 2013, 66-76.

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.

Pusukuri, K., Kishore, Gupta, R., and Bhuyan, L. N. (2013), ADAPT: A Framework for Coscheduling Multithreaded Programs. **ACM Transactions on Architecture and Code Optimization (TACO)**, 9(4), 45.

Larus, J. R., and Schnarr, E. (1995), EEL: Machine-independent executable editing. **ACM Sigplan Notices**, 30(6), 291-300.

Laurenzano, M., Tikir, M., Carrington, L., and Snavely, A. (2010), PEBIL: Efficient static binary instrumentation for linux. In Performance the **2010 IEEE International Symposium on Analysis of Systems & Software** (**ISPASS**), White Plains, NY, USA, 28-30 March 2010, 175-183. IEEE Computer Society.

Luk, C. K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. (2005), Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. **ACM SIGPLAN Notices**, 40(6), 190-200.
Nanda, S., Li, W., Lam, L. C., and Chiueh, T. C. (2006), BIRD: Binary interpretation using runtime disassembly. In Proceedings of the **International Symposium on Code Generation and Optimization** (CGO), New York, New York, USA, 26-29 March 2006, 358-370. IEEE Computer Society.

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.

Nguyen, A. T., Michael, M., Sharma, A., and Torrellas, J. (1996), The Augmint Multiprocessor Simulation Toolkit for Intel x86 Architectures. In Proceedings of the **1996 International Conference on Computer Design (ICCD), VLSI in Computers and Processors**, Austin, TX, USA, 7-9 October 1996, 486-490.

Pin -A Dynamic Binary Instrumentation Tool, Intel, Retrieved March 22, 2015, from **https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.**

Ranger, C., Raghuraman, R., Penmetsa, A., Bradski, G., and Kozyrakis, C. (2007), Evaluating MapReduce for Multi-Core and Multiprocessor Systems. In Proceedings of the **13th International Symposium on High Performance Computer Architecture (HPCA)**, Phoenix, AZ, USA, 10-14 February 2007, 13-24.

Reinders, J. (2007), **Intel Threading Building Blocks: outfitting C++ for multi-core processor parallelism**. California: O'Reilly Media, Inc.

Schuchhardt, M., Memik, G., Choudhary, A., Das, A., and Hardavellas, N. (2013), The Impact of Dynamic Directories on Multicore Interconnects. **IEEE Computer**, 46(10), 32-39.

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

Srivastava, A., and Eustace, A. (1994). ATOM: A system for building customized program analysis tools. **ACM SIGPLAN Notices**, 29(6), 196-205.

Srivastava, A., and Wall, D. W. (1992) A practical system for intermodule code optimization at link-time. **Journal of Programming Languages**, 1(1), 1–18.

Standard Performance Evaluation Corporation (SPEC), SPEC CPU2006, Retrieved March 25, 2014, from **http://www.spec.org/cpu2006/**.

Wang, R., Gao, Y., and Zhang, G. (2013), Real Time Cache Performance Analyzing for Multi-core Parallel Programs. In Proceedings of the **2013 International Conference on Cloud and Service Computing** (**CSC**), Beijing, China, 4-6 November 2013, 16-23. IEEE Computer Society.

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

توصيف برمجيات الحواسيب متعددة النوى بدون الاعتماد على مواصفات تركيب الحاسوب

إعداد
محمد سلطان أحمد محمد

المشرف
الدكتور غيث علي عبندة

الملخص

اكتسبت معماريات المعالجات المتعددة النوى شعبية متزايدة في السنوات الأخيرة. لكن عدد من التطبيقات المتوفرة حالياً لا تستفيد بشكل كامل من الزيادة في عدد نوى المعالجة وذلك بسبب أنّ هذه التطبيقات هي إما تطبيقات متسلسلة أو أنها تملك كمية كبيرة من أنواع الاتصال وأعباء إضافية نتيجة الموازاة (parallelization overhead). لذلك من المهم توصيف هذه التطبيقات على منصات متعددة النوى لمساعدة المبرمجين في ضبط هذه التطبيقات وتطوير تطبيقات متوازية في المستقبل، وكذلك لمساعدة المصممين في تصميم معماريات متعددة النوى تقوم بتشغيل التطبيقات المتوازية بكفاءة.

هذه الاطروحة تعرض طريقة فريدة في توصيف التطبيقات المتوازية على المنصات متعددة النوى. هذه الطريقة هي التوصيف غير المعتمد على المواصفات. التوصيف غير المعتمد على المواصفات هو توصيف الخصائص المتأصلة للتطبيقات وذلك بتتبع عمليات القراءة والكتابة لكل موقع في الذاكرة، وهي لا تعتمد على أي مواصفات محددة. وهذه الطريقة في التوصيف أسرع من الطريقة التقليدية التي تعتمد على المواصفات. يتم ارسال بيانات المراقبة الى أداة التحليل بدون الاعتماد على المواصفات مباشرة دون التخزين على وحدات التخزين. عملية إرسال بيانات التحليل مباشرة تسمح بتوصيف تطبيقات ذات مدخلات كبيرة بدون الاحتياج الى مساحات تخزين هائلة.

في هذا البحث، قمنا أولاً باختيار ثمانية تطبيقات من مجموعتين من التطبيقات القياسية (PARSEC وSPLASH-2). ثم قمنا بتطوير أداة مراقبة (PSMAIT) لمراقبة التطبيقات المختارة. وقمنا بتعديل أداة CIAT التي كانت تعمل على منصات متعددة المعالجات لكي تعمل على منصات متعددة النوى. كما قمنا بتنفيذ عدد من التجارب بأعداد مختلفة من المسارات (threads) لحجمين من المدخلات لكل التطبيقات المختارة.

أداة CIAT تصف أربعة جوانب من خصائص التطبيقات المدروسة وهي تعليمات القراءة والكتابة من الذاكرة وأنواع الاتصال (communication patterns) وفواصل الاتصال (communication slack) وأنماط الاتصال (communication locality). نتائج هذه التجارب أظهرت أنّ اثنان من التطبيقات المدروسة لديها نسبة عالية من الأعباء الإضافية نتيجة الموازاة وهما Cholesky وFluidanimate. Cholsky لدية 80% من الأعباء الإضافية نتيجة الموازاة عند تشغيل 16-مسار بسبب انه يملك نسبه عالية من الاتصال مقارنة مع نسبة العمليات الحسابية. Fluidanimate لدية 60% من الأعباء الإضافية نتيجة الموازاة عند تشغيل 16-مسار بسبب الاتصال الكبير على الحدود بين الخلايا. الأعباء الإضافية نتيجة الموازاة قد تؤدي الى الحد من زيادة سرعة هذه التطبيقات. معظم أنوع الاتصال الشائعة في التطبيقات المدروسة هي القراءة بعد الكتابة والكتابة بعد القراءة. ولكن هناك تطبيقان يمتلكان جزء كبير من الكتابة بعد الكتابة (WAW) وهما Radix وSwaptions. في Radix حوالي 36% من أجمالي أنواع الاتصال هي WAW عند تشغيل 16-مسار بسبب عمليات التبديل. في Swaptions تقريباً 100% من أجمالي أنواع الاتصال هي WAW عند تشغيل 16-مسار بسبب إعادة استخدام بعض المتغيرات. الجزء الكبير من WAW قد يؤدي الى قضاء وقت كبير في معالجة مفقودات التخزين (store misses). تظهر نتائج فاصل الاتصال أنّ جميع التطبيقات المدروسة يمكنها أن تستفيد من الجلب المسبق لمعلومات. نتائج أنماط الاتصال تظهر أنّ المسار الأول يتصل مع بقية المسارات الأخرى في كل التطبيقات المدروسة. لذلك ينصح بجعل هذا المسار على النوى الذي في المنتصف لتقليل كلفة الاتصال.