Communication Driven Mapping of Applications on Multicore Platforms

Communication Driven Mapping of Applications on Multicore Platforms

Proefschrift

ter verkrijging van de graad van doctor

aan de Technische Universiteit Delft,

op gezag van de Rector Magnificus prof. ir. K. C. A. M. Luyben,

voorzitter van het College voor Promoties,

in het openbaar te verdedigen op donderdag 28 april 2016 om 10:00 uur

door

Imran ASHRAF

Master of Science in Embedded Systemen

geboren te Mansehra, Pakistan.

This dissertation has been approved by the promotor:

Prof. dr. K. L. M. Bertels

Composition of the doctoral committee:

Rector Magnificus,	voorzitter
Prof. dr. K. L. M. Bertels,	Delft University of Technology, promotor

Independent members:

Prof. dr. E. Charbon,	Delft University of Technology
Prof. dr. A. Mendelson,	Technion University, Israel
Prof. Dr.-Ing. M. Hübner,	Ruhr-Universität Bochum, Duitsland
Dr. F. Silla,	Technical University of Valencia, Spain
Prof. dr. ir. P. F. A. Van Mieghem,	Delft University of Technology, reserve member

Other members:

Dr. J. C. Le Lann,	ENSTA-Bretagne, Brest, France
Dr. Z. Al-Ars,	Delft University of Technology

Keywords: Data-communication profiling, heterogeneous computing, binary instrumentation, code parallelization, shadow memory

ISBN 978-94-6186-633-2

An electronic version of this dissertation is available at
http://repository.tudelft.nl.

To the children;
Usman, Simra, Moiz and Fatima,
for bringing joy to our family.

Acknowledgements

Alhamdolillah, it is a matter of great pleasure and satisfaction for me to write this last part of my dissertation. Although I had my moments of ups and downs during this arduous journey, I consider myself exceedingly fortunate to have some wonderful people to keep my morale high.

First and foremost, I am grateful to Koen for providing me the opportunity to pursue my PhD research under his guidance. His experience, perseverance, bigheartedness, and consistent supervision enabled me to develop as an independent researcher. I am thankful to him for admitting me in various European projects; helping me to excel in collaboration in both academia and industry. At CE Lab, I relished working as lab designer and teaching assistant, occasional lecturer and supervisor of graduate and undergraduate students. Koen had always been able to provide ample time despite his tight and busy schedules. But above all, CE Lab should be thankful to Koen for creating a friendly environment; encouraging various social events, promoting open-door policy for all, emphasizing on coffee-breaks and brainstorming sessions. Overall, all these measures helped in increasing connectivity and friendliness among CE faculty and students.

My special gratitude goes to the members of my PhD committee for accepting their role, reading my dissertation, and providing useful feedback. I am also grateful to the graduate school for channelizing the PhD process by introducing various courses and workshops. I was the first test-case of the graduate school at CE Lab, which confused me a bit in the beginning, due to the uncertainty of the process, but eventually it ameliorated my research and personality. I also appreciate the administrative and technical support provided by Lidwina, Eef and Erik.

I am overwhelmed with gratitude to all my office mates. In initial days, Roel Meeuws and Arash Ostadzadeh assisted me with their insightful technical discussions. Later on, I joined Hamid, Nauman, and Nader, who always engaged me with exciting discussions and debates. Moreover, I am thankful to Vlad for his useful and encouraging contribution throughout my PhD. Vlad has a remarkable experience in tool development which really helped me. In particular, I want to mention Mota for being a great friend in all thick and thin. He is a wonderful person who is always ready to help, even when help is not needed. :)

I am also grateful to Mota and Faisal for the time taken to proof-read my papers and parts of thesis by providing useful comments, feedback, and suggestions. I am also thankful to the rest of CE fellows for being wonderful colleagues. One cannot

forget Marius and Mihai for organizing the weekly CE football event.

I would also like to thank all my Pakistani friends in Delft, especially the senior PhD fellows — Hasham, Husnul Amin, Mehfooz, Laiq, and Hamayun — for supporting me like their younger brother. I am also grateful to my friends Faisal, Seyab, Fakhar, Umer Altaf, Muhammad Nadeem, Sajjad Sandilo, Di Cao, Haroon Sadiq, Shakeel and their families for all the gatherings, dinners, card games and appreciation of my cooking prowess. I never felt away from home. Thanks to the members of Delft Cricket Club, Shah G, Patel the great, Fahim, Usama the saddar, Umer Ijaz, Tabish, Atif, Ragu, Ram, Rahul, JK and many more for enjoyable cricket matches.

I must also appreciate my UET friends Saad, Moeen and Shoaib for always being there for me in low times. I am especially thankful to Sadi, who has been my best buddy since times immemorial. I am also thankful to MaanJi for her affection and prayers throughout my studies.

On this important juncture of my life, I am sure that nobody in the world will be as happy as my parents; my Ammi and Abbu g. You have sacrificed all your comforts to see me successful. I can never thank you enough. I am also thankful to my brothers and sisters for taking care of my parents while I was away from home for such a long time.

Last but not least, my special thanks and admiration go to Sonia, Usman and Fatima. Sonia! I do not have words to express my appreciation towards you. Without you, I would not have had the peace of mind to work. I thank you for your understanding and cooperation especially during the busy times of my PhD process. Usman and Fatima, thank you so much for bringing joy and happiness to my life. Your hugs and smiles imparted the right energy to maintain momentum in my research.

Imran Ashraf
March 23, 2016
Delft, The Netherlands

Contents

List of Acronyms

API Application Programming Interface 54
BRAM Block RAM, a local block of RAM on FPGA 70
CPU Central Processing Unit 7
DSP Digital Signal Processor or Digital Signal Processing 4
ELF Executable and Linkable Format, formerly known as Extensible Linking Format 36
FPGA Field Programmable Gate Array 4
GPP General-Purpose Processor 70
GPU Graphical Processing Unit 3
GPGPU General Purpose Computing on GPUs 4
PE Processing Element 77
FLOPS Floating Point Operations Per Second 10
PCIe Peripheral Component Interconnect Express 11
NoC Network on Chip 74

1

Introduction

The growing demand for processing is being satisfied by the growing number of homogeneous and heterogeneous processing cores in a computing platform. However, this trend goes hand-in-hand with issues pertaining to program parallelization, application partitioning, deep memory hierarchy, limited communication bandwidth, power consumption, etc. Some of these problems could be alleviated by utilizing tools to unleash the performance of these emerging computing systems. In this chapter, we present current computing trends and describe the basic concepts required to understand the questions raised in the dissertation. Subsequently, we discuss some opportunities offered by these multicore platforms and highlight the main challenges in the efficient utilization of these platforms. Thereafter, we briefly describe the research directions of this dissertation followed by the main contributions. Finally, we provide an outline to the remainder of the dissertation.

1.1. Background

Although the number of transistors per chip is growing due to technology scaling [2], increasing the clock rate of processors is becoming economically less viable due to fabrication cost and power consumption [3]. These limitations shifted the trend towards the integration of a growing number of homogeneous or heterogeneous processing cores [4] in general-purpose [5], embedded [6, 7] and high-performance computing platforms [8] as depicted in Figure 1.1. However, these multicore architectures pose specific challenges regarding their programmability, as the effective utilization of these platforms in an architecture agnostic way is not possible. Hardware constraints, such as memory bandwidth, local scratch-pad memory etc. need

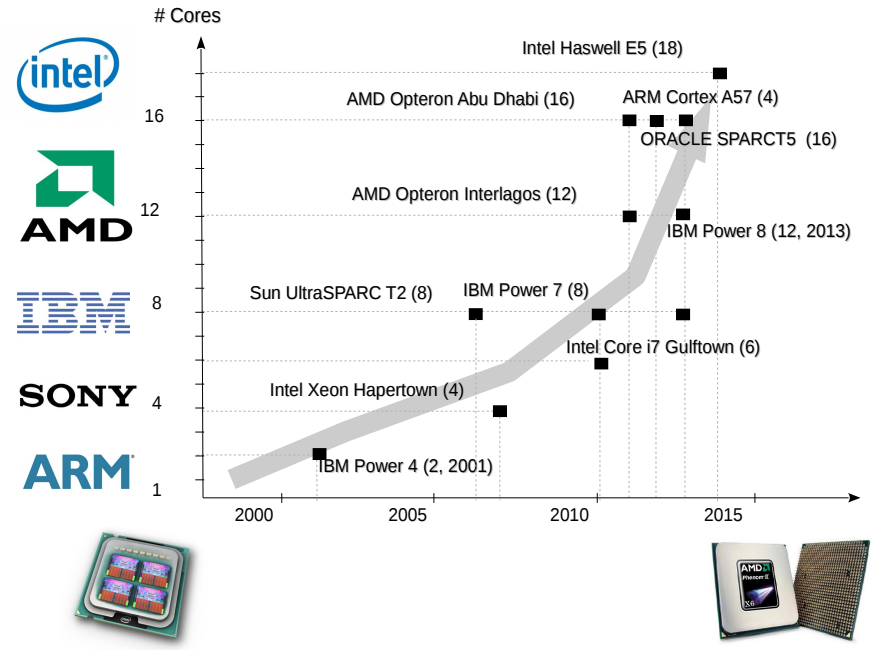

Figure 1.1: Multicore processor proliferation [1].

to be explicitly taken into account.

Because of the growing gap between processor and memory speeds [9], it is becoming more and more important to optimize memory-access behavior of applications. Secondly, with the growing number of cores, the degradation of performance improvement exacerbates, as communication is typically more time-consuming than computation. Hence, this communication is considered as the major design challenge in multicore architectures [10]. In addition, it is a major source of energy consumption [11].

1.1.1. Application Partitioning

There is a huge code base of legacy, sequential applications which need to be ported to emerging multicore architectures, and thus need to be parallelized. To port an existing sequential application to multicore architecture, applications must be divided into smaller parts which are mapped to the available cores in the architecture as shown in Figure 1.2. This is known as application partitioning and it is a critical task, as an improper partitioning and mapping may result in performance degradation. Main identifiable reasons are irregular memory-access patterns and the communication among cores which may reduce the anticipated performance improvement.

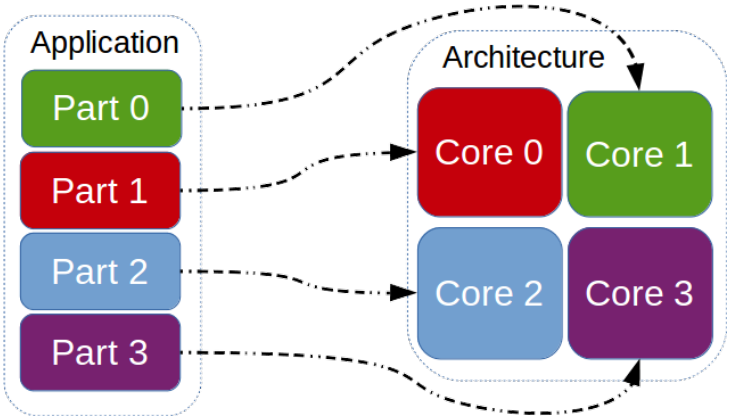

Figure 1.2: Partitioning of an application into four parts to be mapped on the available four cores in the architecture.

1.1.2. Classes of Parallelism

Parallel computing is the form of computing in which various computations can be carried out simultaneously. This is possible because of the availability of parallelism in applications. Parallelism exists in applications in two forms.

- Data-Level Parallelism refers to the class of parallelism in which there are many data items that can be processed in parallel. For instance, addition of two matrices where addition of elements can happen in parallel.
- Task-Level Parallelism refers to the class of parallelism in which various tasks in application can be processed in parallel. For instance, if in an application we have to add, subtract and multiply two matrices, then these three tasks can be executed in parallel.

1.1.3. Classes of Parallel Architectures

Computing systems exploit the parallelism available in application to gain performance. This can be done in four major ways.

- Instruction-Level Parallelism exploits data-level parallelism available in the application by pipelining and speculative execution.
- Data-Level Parallelism is exploited by vector architectures and Graphical Processing Unit (GPU) by applying a single instruction to multiple data items in parallel.

- Thread-Level Parallelism exploits data-level or task-level parallelism. This is achieved in a tightly coupled hardware model where parallel threads can interact among each other.
- Request-Level Parallelism exploits task level parallelism among largely decoupled tasks specified by operating system or programmer.

1.1.4. Heterogeneous Computing

Heterogeneous computing utilizes heterogeneity in the architecture to perform efficient processing. Heterogeneity refers to the availability of more than one kind of cores in the architecture. Some cores have good single thread performance, whereas, the other cores have high throughput. Heterogeneous systems gain high performance and energy efficiency due to these dissimilar cores which are specialized for specific type of processing. This means that each application or part of an application is matched to the core, based on its performance demand.

A well known form of heterogeneous computing utilizes accelerators to gain performance. Here, compute intensive parts of an application are off-loaded to the accelerator. An accelerator is a computing unit comprised of many simple processing units specifically designed to run computationally intensive part of the application very fast.

GPU is a well known example of an accelerator. GPU was originally designed to efficiently process images, but their ability to perform floating point operations at extremely high speed has given rise to the form of computing coined as General Purpose Computing on GPUs (GPGPU). Apart form GPU, Xeon-Phi [12], Digital Signal Processor (DSP) [7] and Field Programmable Gate Array (FPGA) [6, 8] are also used as accelerators in various application domains.

1.1.5. Profilers

Profilers are program analysis tools which provide information about various aspects of programs, for instance, number and types of instructions, frequency of function calls [13], time consumed per function[14, 15], call-graph [13–16] etc. In [17], the need for such tools has been well formulated as:

> *"Program analysis tools are extremely important for understanding program behavior. Computer architects need such tools to evaluate how well programs will perform on new architectures. Software writers need tools to analyze their programs and identify critical sections of code. Compiler writers often use such tools to find out how well their instruction scheduling or branch prediction algorithm is performing."*

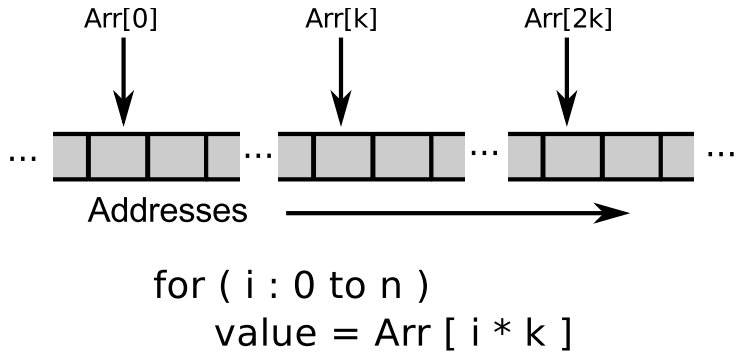

Figure 1.3: Memory-access with a stride of k.

1.1.6. Memory Profilers

When a function loads/stores data from/to an address in memory, it is known as read/write memory-access. Memory profilers [14, 15, 18–20] monitor these memory reads and writes to provide information about the memory-access patterns of functions.

Cache memories try to make predictions mainly based on the following types of access patterns:

1. **Temporal:** recently accessed data will be needed again in the near future.
2. **Spatial:** data adjacent to the currently referenced data, will be accessed.
3. **Strided:** memory is accessed in some predictable pattern. An example of strided memory-access with a stride of k is shown in Figure 1.3. The code snippet performing this memory-access is also shown, where different values of k will result in different amounts of strides.
   - When k=1, $Array$ elements are accessed as $0, 1, 2, 3, \cdots$. This pattern is known as sequential memory-access.
   - When k=2, $Array$ elements are accessed as $0, 2, 4, 6, \cdots$.
4. **Random:** data is accessed randomly and is un-predictable.

Due to the growing gap between processor and Memory speeds [9], it is becoming increasingly important to characterize the memory-access patterns of applications for performance improvement. For instance, the information about access patterns is utilized by caches, which are small memories utilized to hide the main memory-access latency by predicting the next memory-access [21, 22]. Cache designers need the information about the memory-access patterns to design efficient cache controllers. Programmers need this information to match the access patterns of an application at hand with the cache architecture.

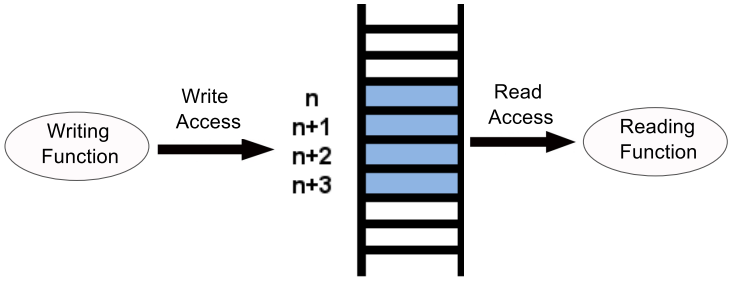

Figure 1.4: A memory write access followed by one or more memory read accesses form a data-communication relationship.

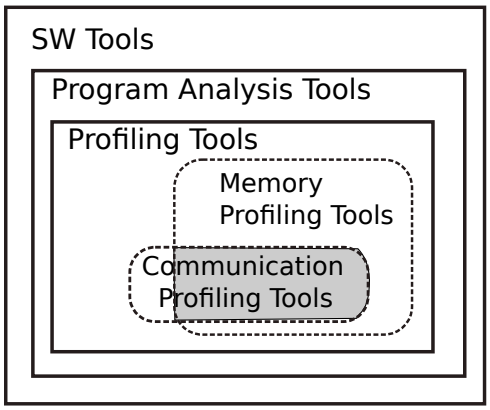

Figure 1.5: Data-communication profilers are generally special class of memory profiling tools. Some data-communication profilers report communication without tracing memory accesses.

1.1.7. Data-Communication Profilers

Data-communication in an application occurs when certain part of a program writes data into memory and later the same data is read by another (or same) part of the application. Therefore to track data-communication, both memory read and write accesses need to be tracked as shown in Figure 1.4.

At the fine-granularity level, this data-communication can be reported at the instruction or the basic block level. At the coarser-granularity, data-communication can be reported between functions in a sequential application or between threads in parallel applications. Profilers that provide this inter/intra-function data-communication information are termed as data-communication profilers. Figure 1.5 shows the clear relationship between memory profilers and data-communication profilers, where it can be seen that data-communication profilers are a sub class of memory profilers. Furthermore, it can also be seen that some data-communication profilers do not utilize memory access information to report data-communication. These profilers track, for instance, network traffic to report communication.

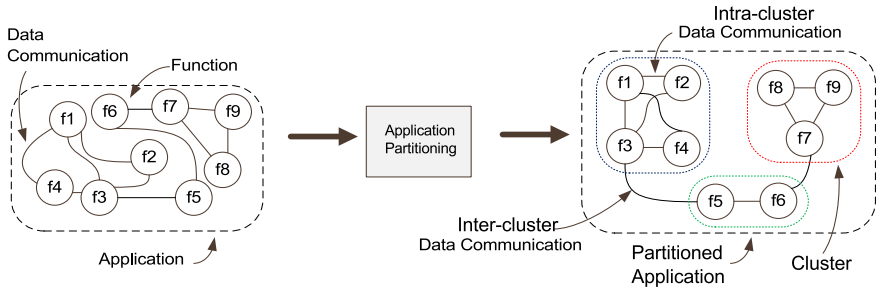

Figure 1.6: An application is partitioned into three clusters depicting intra-cluster and inter-cluster data-communication.

Data-communication can occur in various patterns [23]. We briefly describe here the common data-communication patterns.

1. **Read-only:** In this communication pattern, data is written to a memory location once but read multiple times. This pattern involves reading the same data over and over again like reading constants, which are initialized once but read multiple times throughout the execution of application.

2. **Migratory:** This communication pattern occurs when a data structure is repeatedly read and then written by a number of threads in the atomic regions during program execution. An example can be the processing of an image by various threads repeatedly in a pipeline fashion.

3. **Producer-consumer:** Producer of a data structure is the thread which writes data to a data structure. The thread that reads this data structure is called the consumer. Hence, the process occurs in the form of a producer-consumer communication pattern.

In order to provide a motivational example for data-communication profiling, consider the inter-function data-communication at the application level depicted on the left side of Figure 1.6. To map this application on a multicore architecture, functions can be clustered together as depicted on the right-hand side after application partitioning. Each cluster can be mapped onto a core in multicore architecture, converting inter-cluster communication into the inter-core communication. Consequently, it is very important to perform partitioning, utilizing the data-communication information in an intelligent manner in order to avoid expensive external communication.

For homogeneous architectures the data-communication information can be utilized by setting the thread affinity to bind highly communicating threads to the same core to reduce inter-core communication. For heterogeneous architectures, which utilize Central Processing Unit (CPU)/FPGA/DSP/Xeon-Phi as an accelerator, this information can be utilized to perform the mapping such that the expensive

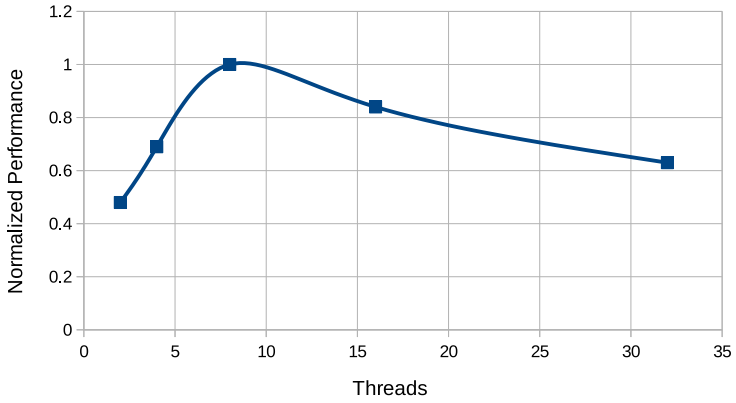

Figure 1.7: Performance of SP benchmark degrades with growing number of cores [26].

data-communication between CPU and the accelerator is reduced as much as possible.

1.2. Problem Overview

Due to technology scaling [2], the computing performance increased orders of magnitude during the last few decades. However, as the transistor size is approaching atomic scale, it is becoming increasingly difficult to improve microchip performance without making a considerable sacrifice on power and cost [4, 24, 25]. Due to this disproportional increase in power and cost, the industry can no longer rely on performance doubling every 18 months, though computational demands continue to increase sharply.

In order to satisfy the growing processing demands, the trend shifted towards growing number of cores [4, 27] to gain performance. Here, an application is analyzed to extract the parallel parts of the application which are executed as threads in parallel on the available cores in the architecture. Various challenges make it hard to parallelize applications to gain performance from multicore architectures [28–30]. First of all, not all applications are parallel in nature. Secondly, extracting the available parallelism from existing applications is not trivial because of the way they are written. For instance dependence analysis becomes difficult when features like indirect addressing, pointers, recursion and indirect functions are used. Secondly, determining the the loop boundaries and coordinating the accesses to global resources are also tedious and error prone.

Even if the applications are parallelized to take benefit of the available cores in the architecture, application performance does not necessarily scale with an increas-

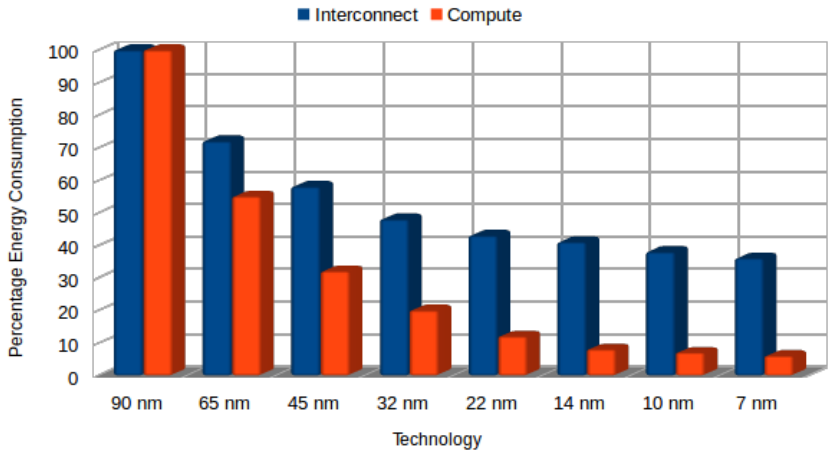

Figure 1.8: Chip-level energy trends [36]

Table 1.1: Estimated energy consumption for arithmetic operations.

	Integer		Floating Point	
Operation	Size	Energy	Size	Energy
Add	8-bit	0.03 pJ	16-bit	0.4 pJ
	32-bit	0.1 pJ	32-bit	0.9 pJ
Mult	8-bit	0.2 pJ	16-bit	1 pJ
	32-bit	3.1 pJ	32-bit	4 pJ

ing number of cores[31]. The performance is limited mainly due to the inter-core communication and contention while accessing shared memory [26, 32–34]. Figure 1.7 shows the performance of SP benchmark from NAS parallel benchmark suite [35] for various number of cores. It can be seen that the optimal number of cores is 8 and performance degrades for core count greater than 8. Similar results are reported in literature[31] for other real applications.

Realizing an exascale-level performance by the end of this decade imposes a major challenge on energy and power consumption [11, 36, 38]. The power consumption

Table 1.2: Estimated energy consumption for memory access.

Memory Type	Size	Energy
Cache	8 KB	10 pJ
	32 KB	20 pJ
	1 MB	100 pJ
DRAM		1.3 - 2.6 nJ

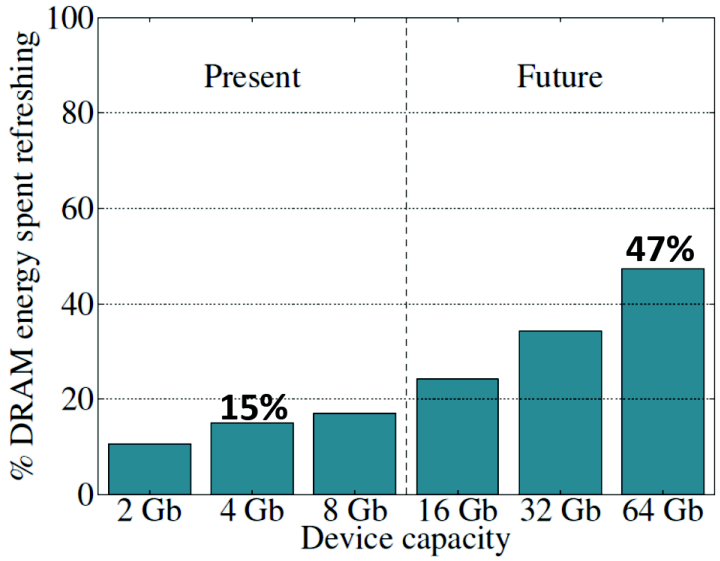

Figure 1.9: Energy spent of DRAM refresh [37]

is increasingly dominated by data-transfer and memory accesses [36, 39, 40] in 65nm and smaller technologies as depicted in Figure 1.8. The research[38] presents the estimated energy cost for arithmetic operations and memory access (cache and DRAM) as given in Table 1.1 and Table 1.2, respectively. It can be seen that *add* and *multiply* operations of various sizes cost 0.03pJ-4pJ, whereas, cache memory access costs 10pJ-100pJ. The situation worsens if DRAM access is involved (1.3-2.6nJ of energy is consumed for a DRAM access [40]). DRAM consumes energy even when it is not being used due to cell refreshing. The energy consumption increases with the memory size [37], as depicted in Figure 1.9. The off-chip memory energy consumption is even worse as $40-50\%$ of energy is spent in off-chip memory [38].

Heterogeneous multicores are envisioned to be a promising design paradigm to combat today's challenges in reducing power consumption and alleviate the impact of the memory wall [11, 27, 41]. This can be observed from the growing number of accelerator based systems in TOP500 list [42]. Currently the world's most powerful supercomputer, Tianhe-2 [43], utilizes accelerators. In general, the TOP500 list shows a growing momentum of accelerator based systems as currently more than 100 supercomputers use accelerators. These systems account for 143 petaflops of processing power which is over one-third of the list's total Floating Point Operations Per Second (FLOPS).

In these accelerator-based systems, the compute intensive part of the application is offloaded from CPU to the accelerator. The problem which arises from this offload model of computation is that data needs to be moved from CPU to the accelerator as this data-communication generally occurs on a slow bus, such as Peripheral

Component Interconnect Express (PCIe). This data-communication is considered as the primary bottleneck [44–47]. Recent work [48] shows that about 70% of the stalls are comprised of data-dependence and memory stalls. This percentage increases significantly when the peak bandwidth is reduced clearly indicating that limited off-chip memory bandwidth is a critical performance bottleneck for these applications. Therefore, partitioning an application to map on these architectures is a critical task [49, 50] as it requires structuring algorithms such that the expensive data-transfers are completely avoided or reduced as much as possible.

Another trend in computing, especially with accelerator-based computing, is that these architectures have deeper memory hierarchy. These systems normally have distinct on- and off-chip address spaces and require software to move data in between them. Example of such architectures exist in general purpose [5], [51], embedded [6, 7, 52], and high performance computing platforms [8, 53–55]. Explicit memory management is necessary for program correctness as well as to boost performance [25, 56–58] for these architectures. This implies that placing the data in the memory hierarchy closer to the processor is critical. This placement of data can be on-die cache or scratch-pad, local DRAM, or remote memory accessed over high-speed interconnect.

Figure 1.10 shows the memory hierarchy of the Nvidia GeForce GT-640 GPU architecture. Accessing the shared memory is an order of magnitude faster than accessing the global memory [60]. However, the trade off is that shared memory is limited and requires management by a programmer. The problem with these exposed memory hierarchy architectures is that it is not trivial to program these architectures. This is illustrated by Figure 1.11 where it can be seen that the computational capacity of multicore architectures is underutilized due to the programming bottleneck.

In a nutshell, the key to scaling computing performance is to reduce the data movement as much as possible. Accomplishing this requires a deep understanding of the memory access behavior of an application and careful look at the data flow in the application. Manual analysis of applications is tedious and error prone. Therefore, tools are required to characterize the data-communication in an application and highlight the communication hot spots. These tools can help programmers to perform communication-aware partitioning and mapping decisions based on detailed quantitative application profile. These tools are also helpful to system architects to design future interconnects considering the communication behavior of the target application domain.

1.3. Research Questions

To efficiently map a sequential application onto a multicore architecture, various aspects need to be taken into account. These aspects are detection of computational

1

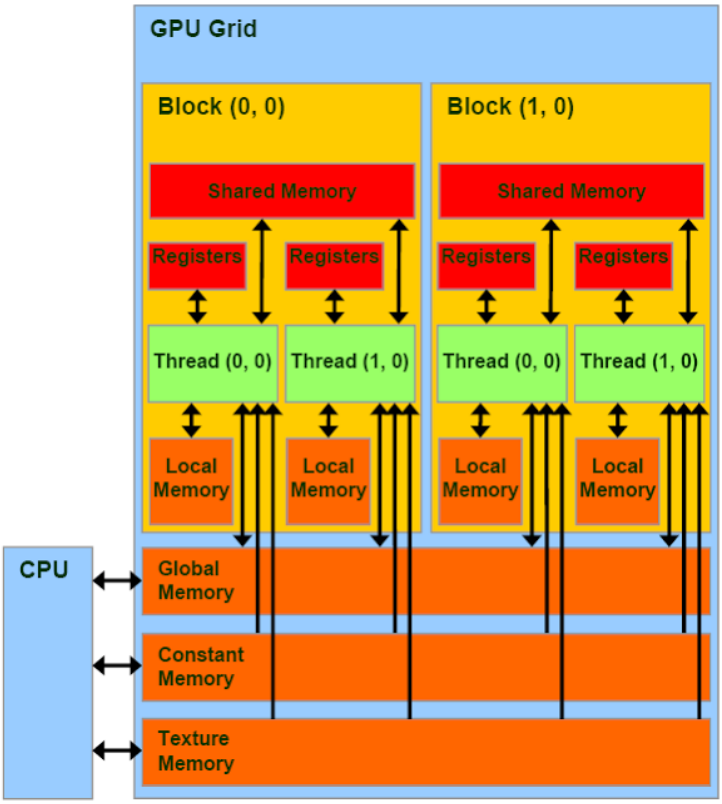

Figure 1.10: GPU memory hierarchy [54]

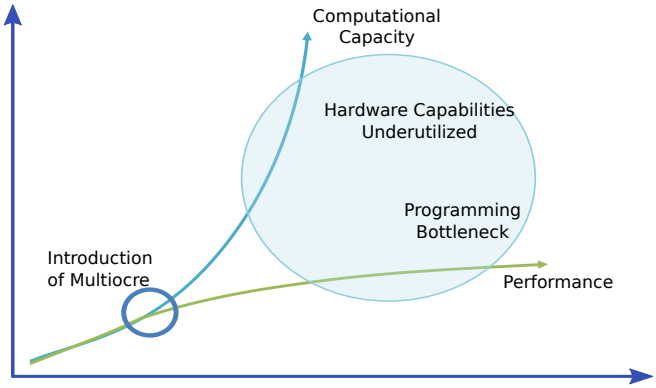

Figure 1.11: Productivity gap [59]

and memory hot-spots, extraction of parallelism, application partitioning, design space exploration, memory assignment, data-movement among cores etc. In this work we focus on some of the research questions as listed below.

Question 1 - How can we efficiently report data-communication among parts of an application at various granularity levels?

We discussed earlier that to report data-communication information, all the write and read memory accesses need to be related together to form data-communication relationship. The challenge lies in the fact that these reads and writes can happen anywhere in the 128 TB user address space, so keeping track of the writer of a memory location efficiently, is not trivial. Hence, the design of the tool has a great impact on the execution-time and memory-usage overheads of a profiler.

Question 2 - How far can we automate the process of application parallelization based on the detailed static and dynamic application profile?

Ideally we want to perform the application parallelization automatically by parallelizing compilers. But this approach has not been successful as compilers do not have enough information such as dependence at compile time. Therefore, the challenge is to utilize the detailed run-time information in conjunction with compile time information to automate the process as much as possible.

Question 3 - How can we utilize the data-communication patterns to assign the data-structures in an application to the available memory-hierarchy in the architecture?

Data-structures are allocated in memory and accessed by various parts of an application. Recent systems have various memory spaces available in the architecture. These spaces have different access-time and sizes. Proper utilization of these memory spaces based on the access pattern is critical to gain performance. The challenge in this regard is to report the access patterns in application at the data-structure level so that it can be utilized to perform proper memory assignment.

Question 4 - How can we utilize the memory-access and data-communication patterns in an application to design efficient hybrid interconnects for embedded and high performance computing platforms utilizing FPGA as an accelerator?

Using FPGA as accelerator has the advantage that its internal structure can be customized based on the computation in the application. In order to reduce the performance loss due to data-communication, can the interconnect be customized to match the application data-communication patterns?

Question 5 - How can we evaluate the quality of solutions generated by different partitioning algorithms in terms of computation and communication time of an application?

Due to a large variety of architectures and the lack of proper benchmarks, it is hard to reproduce experimental results, for fair comparison, on the target platforms. Furthermore, the complexity furthers in case of reconfigurable architectures as the application development process involves building and synthesizing hardware blocks. Considering these challenges, can we have a proper methodology to quickly evaluate the quality of the partitioning algorithms?

1.4. Dissertation Contributions

In order to answer the research questions posed in the previous section, this thesis makes the following contributions.

Contribution 1 - Developed an efficient, open-source profiler, *MCProf*. This profiler has at least an order of magnitude less overhead as compared to the state-of-the-art data-communication profilers for a variety of benchmarks. Furthermore, *MCProf* generates a detailed memory-access and data-communication profile of the application at various granularity levels. To generated information is related to the source-code making it easy for programmers or tools to utilize it.

Contribution 2 - Developed a framework which automates the process of application parallelization. Source-level information is combined with the run-time information generated by *MCProf* to automatically generate parallel representation of an input sequential application.

Contribution 3 - Validated the tool by utilizing the memory-access information generated by *MCProf* for the memory-intensive objects in the application to assign data-structures to the available memory spaces available in the accelerator, such as GPU and FPGA.

Contribution 4 - Developed a Partition Evaluation Tool (PET) which can compare various partitioning algorithms based on the quality of the solutions found by these algorithms.

1.5. Dissertation Organization

Remainder of this dissertation is organized as follows.

Chapter 2 surveys the memory-access optimization profilers and presents propose a classification of these profilers. Focus is on data-communication profilers which is a sub-class of memory-access optimization profilers. A detailed comparison of data-communication profilers is provided to highlight their strong and weak

aspects. Finally, recommendations for improving existing data-communication profilers and/or designing the future ones are thoroughly discussed.

Chapter 3 presents the design of *MCProf*, an efficient memory-access and data-communication profiler. In contrast to prior work, *MCProf* reports detailed profile at various granularity levels with manageable overheads for realistic workloads. Experimental results show that on the average, the proposed profiler has at least an order of magnitude less overhead as compared to the state-of-the-art data-communication profilers for a variety of benchmarks. A case-study is presented which shows the utilization of *MCProf* to efficiently map a sequential application on a platform using GPU as an accelerator.

Chapter 4 first presents a semi-automatic parallelization methodology based on *MCProf* to help programmers extract and express parallelization. Later on, a tool-chain is presented which automates the whole process of application parallelization to generate parallel representation of the input sequential application. We compare our approach with automatic parallelization and other semi-automatic parallelization and outline the advantages and limitations of each approach.

Chapter 5 addresses the challenges of efficiently utilizing accelerator based architectures by utilizing the detailed memory access and data-communication profile of an application. Both software and hardware based optimizations for platforms utilizing FPGA as accelerator are presented. Experimental results are provided for real applications to show the effectiveness of the optimizations.

Chapter 6 concludes this dissertation and presents future research areas.

2

Background and Related Work

With the advent of technology, multi-core architectures are prevalent in embedded, general-purpose as well as high-performance computing. Efficient utilization of these platforms in an architecture agnostic way is an extremely challenging task. Hence, profiling tools are essential for programmers to optimize the applications for these architectures and understand the bottlenecks. Typical bottlenecks are irregular memory-access patterns and data-communication among cores which may reduce anticipated performance improvement.

In this chapter, we survey the memory-access optimization profilers and propose a classification of these profilers. Focus is on data-communication profilers which is a sub-class of memory-access optimization profilers. A detailed comparison of data-communication profilers is provided to highlight their strong and weak aspects. Finally, recommendations for improving existing data-communication profilers and/or designing the future ones are thoroughly discussed.

2.1. Introduction

Profilers are program analysis tools which provide information about various aspects of programs. For instance, number and types of instructions, frequency of function calls, time consumed per function call, etc. A lot of work is available in literature for profilers that focus at the fine granularity of instructions or at the coarse granularity of individual functions [13, 61, 62]. Cache profiling, which is a kind of

memory profiling [63], has also been studied extensively. However, very few tools exist which provide intra-application data-communication information. Therefore, in this work our focus is on memory profiling tools with a focus on those which provide intra-application data-communication information. Even though there is no generally acknowledged classification of memory profilers, we propose to organize the discussion based on the following three aspects of profiling:

- Profiling objective (Section 2.2)
- Profiling input (Section 2.3)
- Profiling technique (Section 2.4)

The remainder of this paper is organized as follows. We start by detailing the proposed classification in Section 2.2, Section 2.3 and Section 2.4. In Section 3.7, we compare existing memory profilers which provide the data-communication information. To the best of our knowledge, we have included all the tools in this regard. Based on this study, we provide some recommendations for the improvements of the existing data-communication profilers or design of the future ones, in Section 2.6. Finally, Section 2.7 concludes the paper.

2.2. Memory Profilers based on Profiling Objective

Based on the profiling objective, we further classify memory profilers into four classes, namely, memory-access optimization, memory debugging, dependence analysis and workload characterization, depicted in Figure 2.1. Furthermore, memory-access optimization profilers are further classified into cache/locality profilers and data-communication profiling. The focus of this work is on profilers designed for memory-access optimization. We believe that the other classes of profilers have been discussed in other studies and a number of example tools are available.

2.2.1. Memory Profiling for Memory-access Optimization

Profilers in this class analyze performance issues related to memory accesses in applications. For instance, the performance of an application may suffer because of pure locality of memory accesses. Cache/locality profilers can highlight the parts of the application responsible for such behavior. Another aspect of performance optimization is the communication among the parts of applications running on separate homogeneous/heterogeneous cores. Data-communication profilers provide such information and highlight communication related performance bottlenecks.

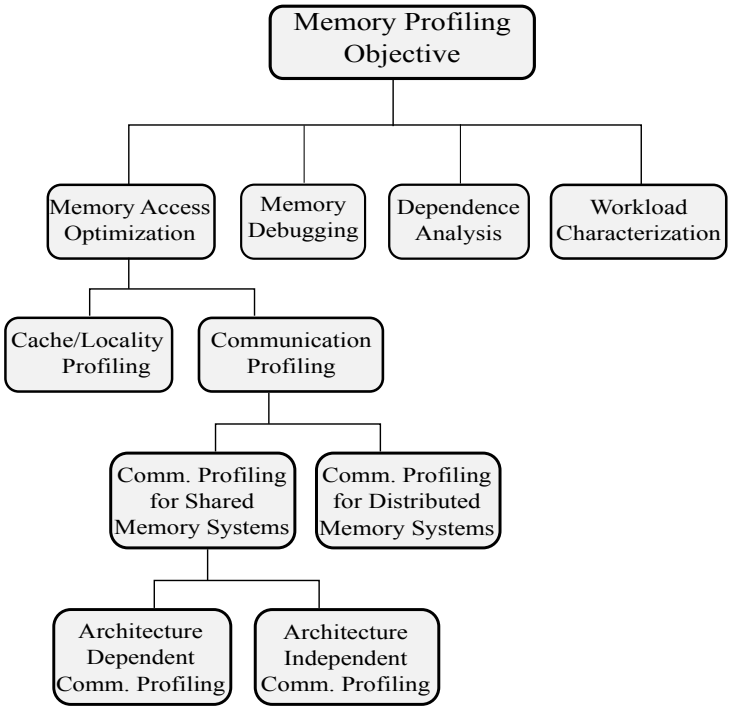

Figure 2.1: Classification of memory profilers based on profiling objective.

Table 2.1: Cache/Locality profilers

Profiler	Input			Output	CL/IDE	Technique	Based on	Availability	Supported Platform	
	Src/Binary	Language	ST/MT						OS	Architecture
Cachegrind[64]	Binary	NA	MT	Text Reports, Graphical Reports	CL, IDE	DBI	Valgrind	Open-Src	Linux, OSX Android	Intel, AMD, PPC
Oprofile[14]	Binary	NA	MT	Text Reports	CL	HWC	Arch. Perf. Counters	Open-Src	Linux	Intel, AMD, ARM, PPC, IBM
NUMATop[65]	Binary	NA	MT	Text Reports	CL	HWC	Arch. Perf. Counters (Intel PMU)	Open-Src	Linux	Intel
Dprof[66]	Binary	NA	MT	Text Reports	CL	HWC	Arch. Perf. Counters (AMD IBS)	Open-Src	Linux	AMD
Zoom[20]	Binary	NA	MT	Graphical Reports	CL, IDE	HWC	Arch. Perf. Counters	Free	Linux, OSX Windows	Intel, AMD, ARM
Vtune[15]	Src for detailed graphical reports	C, C++, C#, Java Fortran, OpenMP, MPI, OpenCL	MT	Graphical Reports	CL, IDE	HWC, DBI	Arch. Perf. Counters (Intel PMU), Pin	Commercial (Intel)	Linux, Win	Intel
Graphical Program Analysis Toolkit[67]	Src for detailed graphical reports	C, C++, pthreads	MT	Graphical Reports	CL, IDE	SBI	ATOM	Commercial (HP)	Tru64	Intel
Caliper[68]	Src for detailed graphical reports	C, C++	MT	Graphical Reports	CL, IDE	DBI	-	Commercial(HP) Free(Non-Commercial)	HP-UX	HP Integrity Servers
CodeXL [69](successor (of Code Analyst)	Src for detailed graphical reports	C, C++, OpenCL	MT	Graphical Reports	CL, IDE	HWC	Arch. Perf. Counters (AMD IBS, TBS)	Commercial(AMD), Open-Src (Linux)	Linux, Win	AMD
Visual Profiler[70]	Src for detailed graphical reports	C, C++, C#, C++ AMP, Visual Basic, Visual F#, Java Script, OpenMP	MT	Graphical Reports	CL, IDE	HWC	Arch. Perf. Counters	Free	Win	Intel, AMD
Solaris Studio[71]	Src for detailed graphical reports	C, C++, OpenMP, MPI, Java	MT	Graphical Reports	IDE	HWC	Arch. Perf. Counters, VampirTrace(MPI)	Free	Solaris, RHEL	SPARC, X86-64

Src: Source, **ST:** Single Threaded, **MT:** Multi threaded, **DBI**: Dynamic Binary Instrumentation, **SBI**: Static Binary Instrumentation, **CL**: Command line, **IDE**: Integreted Development Environment, **HWC**: Hardware Counters, **IBS**: Instruction Based Sampling, **TBS**: Time Based Sampling, **PMU**: Performance Monitoring Unit

Locality/Cache Profilers

To decrease the gap between processor and main memory, small but very fast memories, known as caches, are used. As the size of these memories is small, only the most frequently used data can be stored in these memories based on the prediction of the algorithm in cache controller. Hence, analysis of cache behavior is crucial to increase the performance of the programs, and/or design new cache algorithms. Various tools exist which profile applications to analyze cache behavior as listed in Table 2.1. A few well-known open-source tools are detailed below.

Cachegrind [64] is Valgrind tool which can detect first and last level instruction and data cache misses for C/C++ programs. Cachegrind tracks cache statistics (I1, D1 and L2 hits and misses) for every individual line of source code executed by the program. At program termination, it prints a summary of global statistics, and dumps the line-by-line information to a file. This information can then be used by an accompanying script to annotate the original source code with per-line cache statistics. KCachegrind, a visualization tool for the profiling data generated by Cachegrind, is also available.

Oprofile [14] is a hardware dependent, open-source profiling tool that works on recording events from hardware performance measurement units. Apart from various other performance events, it can sample events related to $L1$, $L2$ instruction and data caches to provide information about cache hits/misses by an application on a certain platform. The profiler is controlled by using the $opcontrol$ and the reports are generated by the $opreport$ utility.

NumaTOP [65] is an open-source tool for runtime memory locality characterization and analysis of processes and threads running on a NUMA system. It utilizes Intel hardware performance counters sampling technologies to identify where NUMA related performance bottlenecks reside. This performance data is associated with Linux runtime information to provide real-time analysis in a GUI on production systems.

Data-communication Profilers

Memory profilers in this class, profile applications to measure communication among various parts of an application. These profilers are further classified in to the following two classes.

Shared Memory Data-communication Profilers Table 2.2 provides a summary of such profilers. Quad (Quantitative Usage Analysis of Data) [72] provides dynamic information regarding data usage between any pair of co-operating functions in an application. This tool is based on Pin [76] and it tracks the reads and writes to a memory location at the granularity of byte. When a function writes to a memory location, it is saved as a producer of this memory location in a Trie

Table 2.2: Data-communication profilers

Profiler	Input			Output	CL/IDE	Technique	Based on	Availability	Supported Platform	
	Src/binary	Language	ST/MT						OS	Architecture
QUAD[72]	binary	NA	ST	DOT, XML	CL	DBI	Intel Pin	Open-Src	Win, Linux, Android, OS X	Intel, ARM
Pincomm[73]	binary	NA	MT	CSV	CL	DBI	Intel Pin	Open-Src	Win, Linux, Android, OS X	Intel, ARM
CETA[74]	binary	NA	ST	DOT	CL	architecture simulation	Virtutech Simics	Open-Src	Win, Linux	Intel
Redux[75]	binary	NA	ST	text	CL	DBI	Valgrind	Open-Src	Linux, Android, OS X	Intel, AMD, ARM, PPC

Src: Source, **ST:** Single Threaded, **MT:** Multi threaded, **DBI**: Dynamic Binary Instrumentation, **CL**: Command line, **IDE**: Integreted Development Environment

data structure. The function reading this memory location is called the consumer and by getting the information from the Trie, a producer-consumer communication relationship is established. Apart from providing the quantitative information about the number of bytes, two other metrics are also reported. The first metric is the number of unique memory addresses, while the second metric is the number of data values uniquely communicated from producer to consumer.

Pincomm [73, 77] is a tool based on Pin [76] which constructs Dynamic Data Flow Graph (DDFG) to report the communication flow between various parts of the program. The parts can be functions, data structures, threads etc. which are represented on DDFG. The communication is reported in the form of producer-consumer relationship. The information can also be provided in terms of marked region in the code which appear on the DDFG. These markers can also be used to start and stop communication. The dynamic objects allocated during the execution of the program are also detected to report the communication through these objects.

CETA (Communication Extraction from Threaded Applications) [74] provides data-flow information between multiple threads. Memory reads and writes are tracked at runtime using Simics multiprocessor architecture simulator [78]. Hash table is utilized to record the writing thread of an address. When the read is performed the communication is updated in another hash table. After the completion of simulation, Python scripts report the collected information as a DOT graph.

Redux [75] is a Valgrind based tool for drawing the detailed dynamic data flow graphs of programs. Because of these details, it can only be used for small kernels or parts of programs, as discussed by authors. Secondly, the purpose of the tool as reported by authors is to represent the computational history of a program and not the communication behavior.

Distributed Memory Data-communication Profilers Message Passing Interface (MPI) [79] is a popular example of distributed memory programming model. The MPI provides communication functionality between a set of processes in a language independent way. This explicit communication is carried out through routines like `MPI_send` and `MPI_recieve`. Various commercial [71, 80, 81] and well maintained open-source [82–85] tools exist which track these routines to characterize communication in MPI programs. We refer the reader to the comparative studies [86, 87] for further details. We would like to highlight here that these tools are not designed to provide the communication profile of sequential applications. These tools are based on the technique which requires MPI parallel program as input. Hence, it only helps in validating the parallel program written only in MPI, rather than constructing one.

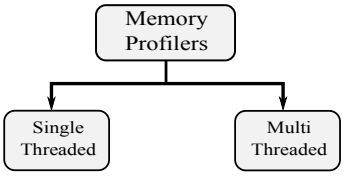

Figure 2.2: Classification of memory profilers based on profiling input.

2.3. Memory Profilers based on Input Application

Memory profilers can take sequential or parallel application as an input (Figure 2.2) to provide the memory access behavior of an application. Profilers also exist which can profile both sequential and parallel applications.

2.3.1. Profiling Sequential Applications

Profilers in this class profile sequential applications for performance analysis, report communication, trace bugs, detect data-races etc. QUAD [72] is an example of such profilers.

2.3.2. Profiling Parallel Applications

Profilers in this class provide information about the memory access behavior of the parallel applications. ParallelTracer [88] is a trace-based performance analysis framework for heterogeneous multicore systems. It instruments source code to trace various events in the application. It is an extension of Trace Collection and Trace Post Processing (TCPP) framework [89]. Furthermore, pin based tools [90], such as Parallel Amplifier, are available for the analysis and optimization of parallel C/C++ programs.

2.4. Memory Profilers based on the Profiling Technique

Based on the technique, memory profilers are broadly classified as static and dynamic analysis tools as shown in Figure 2.3. *Static analysis* tools provide the information based on the source-code without running the application. These tools can predict the communication in regularly structured programs. The polyhedral model is usually imposed in this analysis to compute the communication and data-dependencies analytically. For instance, the work presented in [91] uses exact data-dependence analysis provided by the polyhedral model to automatically explore the opportunities for communication/computation overlap. This kind of anal-

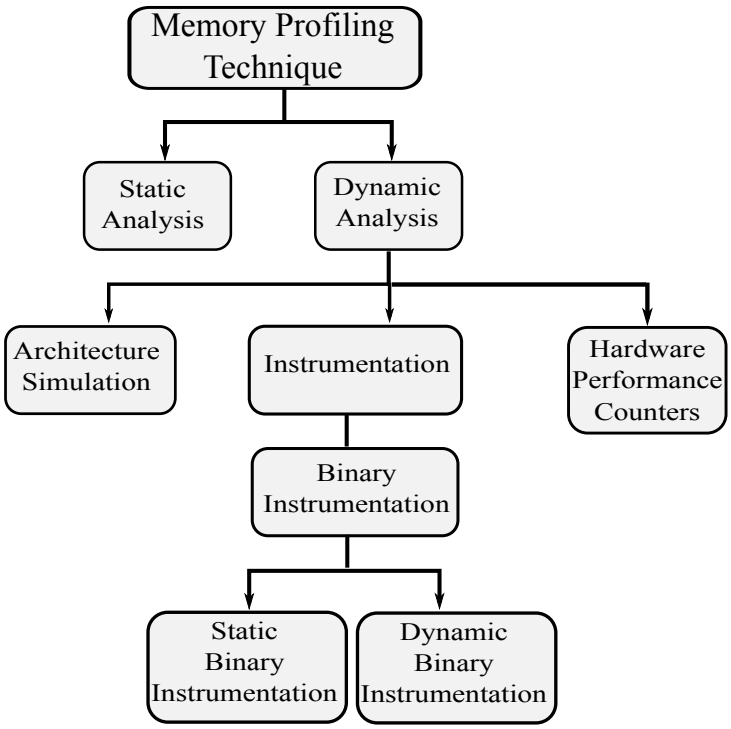

Figure 2.3: Classification of memory profilers based on profiling technique.

ysis is infeasible for a large number of existing and emerging applications as these programs have irregular structure. Furthermore, problems such as pointer analysis, is still very difficult, even exponential-time algorithms do not always produce sufficiently precise results [92].

Tools based on *dynamic analysis* collect information by running the application in a simulator or on the target platform. These are further classified as architecture simulation and instrumentation. *Architecture simulation* involves modeling a virtual computer system with CPU and memory hierarchy. SimpleScalar [93] is an example in this class, which can simulate various architectures with non-blocking caches, speculative execution, and state-of-the-art branch prediction. A drawback of this technique is that it is computationally intensive, which limit its use to small data inputs. Furthermore, simulation with these small data inputs may not exhibit the realistic memory-access patterns.

Binary instrumentation is a widely used instrumentation technique in which an instrumentation tool injects instrumentation code to the compiled binary. This type of instrumentation can be done statically or dynamically.

Static binary instrumentation was pioneered by ATOM [17]. ATOM organizes the final executable such that the application program and user's analysis routines run in the same address space. Hence, there is a possibility to mix code and data in an executable. Third Degree [94] and Graphical program analysis toolkit [67] by HP are example of tools in this regard.

Dynamic binary instrumentation involves the dynamic compilation of binary of an application to insert the instrumentation code anywhere in it. The program binary is instrumented just before its execution. Examples of tools utilizing this technique are Quad [72] and Pincomm [73, 77] which are based on Pin [76]. Similarly, Memcheck [18] and Redux [75] are examples of the tools based on Valgrind [16]. Dr. Memory [95] is another example of dynamic binary instrumentation based memory checking tool, based on open-source DynamoRIO [96] platform.

2.5. Comparison of Data-Communication Profilers

Table 2.3 lists the profilers which can be utilized for memory-access optimizations. To present a combined view, this table depicts the classification of these profilers on the all the three aspects of the proposed criteria. An important observation that can be made from this table is that hardware performance counters and dynamic binary instrumentation is the most widely used technique utilized by these profilers. Another observation is that most of the existing tools focus on cache-access optimizations. Similarly, a number of tools exist with perform communication profiling for distributed memory systems where communication is explicit. However, very few tools provide architecture independent data-communication profiling information. Therefore, these tools are studied and there strengths and weaknesses are

Table 2.3: Classification summary of memory-access optimization profilers based on the proposed criteria.

Memory-access Optimization Profilers				ST/MT	Profiling Technique			
					Architecture Simulation	Static Binary Instrumentation	Dynamic Binary Instrumentation	Hardware Counters
Profilers	Cache/ Locality Profilers			ST + MT		HP Graphical Program Analysis Toolkit[67]	Cachegrind[64], HP Caliper[68]	Oprofile[14], MS Visual Profiler[70], Zoom[20], NUMATop[65], Vampir [81], Intel Vtune[15], AMD CodeXL[69], Dprof[66], Oracle Solaris Studio[71]
	Comm. Profilers	Mem. Shared	Arch. Depend	MT	CETA[74]			NUMATop[65], Intel Parallel Studio XE[80], AMD CodeXL[69], Nvidia NVVP[97]
			Arch. Indep.	ST			QUAD[72], Pincomm[73], Redux[75]	
				MT			Pincomm[73]	
		Distributed Memory		MT			TAU[82], mpiP[83], Scalasca[84], periscope[85]	Vampir Toolset[81], TAU[82], Intel Parallel Studio XE[80], Oracle Solaris Studio[71]

Table 2.4: Comparitive summary of QUAD and Pincomm.

Category	QUAD	Pincomm
Input	Binary	Binary
Input Type	ST	ST/MT
Output	dot, xml	csv
Technique	DBI	DBI
Internal Data Structure	Trie	Hash table
Availability	Open source	Open source
Based on	Intel Pin	Intel Pin
Supported OS	Win, Linux, OS X	Win, Linux, OS X
Supported Architecture	Intel, ARM	Intel, ARM
Reported Metrics	+ (Bytes, UNMA, UNDV)	- (Bytes only)
Profiling Granularity	- (8-bit only)	+ (8,16,32,64-bit)
Execution-time Overhead	+	-
Memory-usage Overhead	-	+
Documentation	+	-

+ indicates profiler is better in this category

discussed and compared in this section.

Redux provides the communication information at a fine-granularity of operations. Due to the amount of the details involved, it can only be used for very small toy applications, as indicated by the authors.

CETA utilizes architecture simulation, which is a computationally intensive approach. To give the reader an idea, Gem5 [98] achieves a simulation speed of 200 KIPS. With this speed, simulating a single core will take around 8 hrs. Hence, this slow simulation speed limits the use of such tools to small data inputs. Simulation with small data inputs may not exhibit the realistic memory-access and data-communication patterns. Furthermore, it requires the design and development of a cycle accurate simulator of these architectures as CETA's implementation is necessarily specific to the processor-architecture, simulator, and OS in use, as reported by the authors. Therefore, our focus in this comparison is limited to Quad and Pincomm which are especially designed to provide data-communication information. A summary of various characteristics of both the tools is provided in Table 2.4. For the following detailed comparison, we performed tests on a 2.66 GHz *Intel(R) Core(TM)2 Quad* CPU with 12 GB of main memory. We used *Pin v2.12* running on *Ubuntu 12.04 LTS* with Linux kernel *3.5.0-45-generic*.

2.5.1. Comparison of the Generated Profiles

Both Quad and Pincomm are based on Pin DBI framework and take application binary as an input to generate data-communication information. Pincomm generates a trace file which is processed by a Perl script to generate user readable information. The advantage of generating this trace file is that temporal aspect of data-communication is preserved. In this way, various phases during the application run can be characterized. The disadvantage is that the size of this trace file grows very large for applications with realistic workloads.

Table 2.5: Overhead comparison of Quad and Pincomm

Domain	Application	Execution-time Overhead		Memory-usage Overhead	
		QUAD	Pincomm	QUAD	Pincomm
Img. Proc.	canny	342.8	712.8	1209.3	204
	KLT	1596.5	3580.2	751.3	152.5
SPLASH-2	ocean-NC	2503.1	3774.6	377.7	64.4
	fmm	2100.8	2657.7	340.2	55.9
	raytrace	2897.3	6690.7	361.3	61.3
Bio Inform.	bwa-mem	1693.3	3765	410.5	73.5
	Average	1855.63	3530.17	575.05	101.93

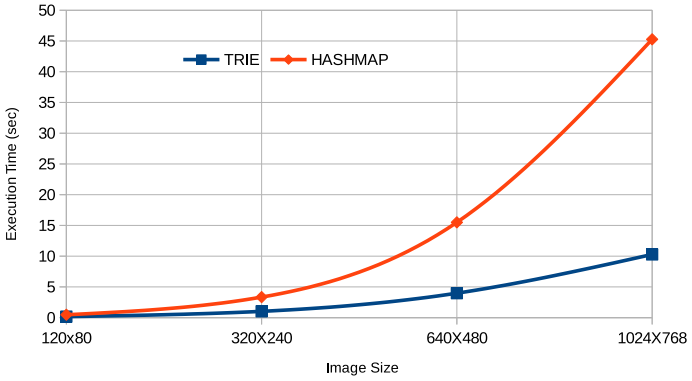

Figure 2.4: Classification of memory profilers based on profiling objective.

Quad provides its output graph in dot and XML formats. The nodes in this graph represent the functions in the application and edges correspond to the data-communication between functions. In each communication relationship, the number of bytes (Bytes), Unique Memory Addresses (UnMAs) and number of Unique Data Values (UnDVs) are reported on the edges. Furthermore, the intensity of the data-communication is also depicted by the color of the edge in the descending order of Red, Brown, Green etc.

Quad only supports sequential applications while Pincomm can also profile multi-threaded applications. Inter-thread data-communication information can be utilized to see the effect of parallelization and drive the mapping of threads to cores for reduced inter-core data-communication.

2.5.2. Overhead Comparison

Both the tools are utilizing dynamic analysis to report the data-communication information, hence large overhead is expected. In order to perform a comparison of the execution-time and memory-usage requirements of Quad and Pincomm, we run the two profilers on the same machine to generate the data-communication information. The execution-time is the wall-clock time measured in seconds by the Linux

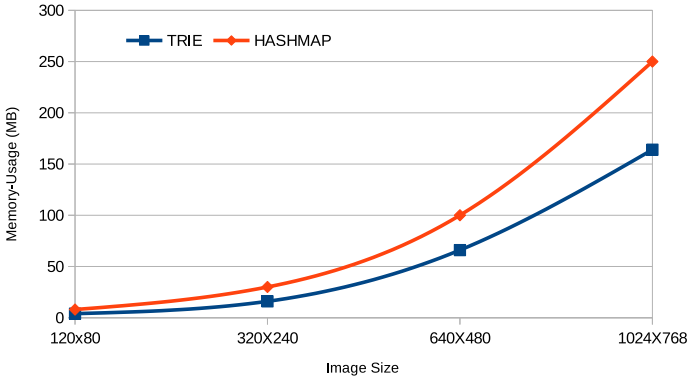

Figure 2.5: Classification of memory profilers based on profiling objective.

time utility. The memory usage is the peak resident set size (`VmHWM`) measured in Mega Bytes (MB) by using the Linux `/proc/<pid>/status`.

Table 2.5 presents the execution-time and memory-usage overhead of Pincomm and Quad. These results are provided for several applications from different domains as depicted in the first two columns of the table. The numbers in Columns 3 and 4 present the ratios of application execution-time with the native application execution time. These numbers represent the slow-down caused by the application execution because of the profiling. For example, Pincomm and Quad slow the execution of $canny$ application by a factor of 712.8 and 342.8, respectively. In order to compare the two profilers, Column 5 lists the ratio of the Pincomm overhead compared to Quad. Similarly, Columns 6 and 7 report the memory-usage overhead caused by profiling. Column 8 reports the ratio of the Quad memory-usage overhead compared to the Pincomm.

It can be seen from the results in Table 2.5 that, on the average, Pincomm has about $2\times$ higher execution-time overhead than Quad to generate the same information. Main source of the overheads in these tools is the data-structure which stores and retrieves the information about the producer of a memory address. This data-structure is critical to the performance of these tools as it is accessed on each memory read/write performed in the application. In this regard, Pincomm uses the STL map, whereas Quad uses a Trie data-structure. Access time increases linearly with the increase in number of accesses for STL map, whereas, in the case of Trie, it stays constant. This means the performance of STL map suffers with the growing size of map, due to the growing application complexity.

Another important point to mention here is that overheads of Trie and Hash map utilized by Quad and Pincomm, respectively, scale with increase in input data. In order to clearly illustrate this, we have plotted the execution-time and memory-usage of accessing only the data-structures of these tools in Figure 2.4 and Figure 2.5,

for the canny application for various image sizes.

Second reason for the variation in overheads of these tools is that Pincomm writes the gathered information to the disk, whereas Quad keeps this information in the memory. Therefore, on the average, the memory-usage of Quad is about $5\times$ higher than Pincomm. This is because of the space required for extra metrics reported by Quad, which are stored in the internal data-structure in the memory, resulting in the higher memory-usage overhead than Pincomm. In short, Quad makes a trade-off in order to be time-efficient, by keeping information in the memory, while Pincomm is space-efficient as it commits the information to the disk.

2.6. Discussion and Recommendations

In this section, we summarize our recommendations to improve existing data-communication profilers and/or design the future ones

1. **Reduction of overheads:** is critical for the usability of the tools. Following suggestions are proposed:
   - Efficient trace collection by utilizing hardware performance counters
   - Efficient storage of producer consumer relationship
   - Efficient design to shift computation from analysis to instrumentation.
   - Configurable Profiling granularity
2. **Architecture independent communication characterization:** is important as it can be utilized to port existing sequential applications to the emerging parallel architectures. Secondly, this approach does not require time-consuming task of architecture simulators.
3. **Detection of communication patterns:** Spatial and temporal data-communication information is important. This will also provide insights in mapping the data-structure in an application to the architecture memory hierarchy.
4. **Source-code related profiling information:** increases the usability of the generated profile by the programmers. Furthermore, this is also important for the automation of communication-aware optimizations of applications.

2.7. Conclusion

Both the memory bottleneck and the multi-core trend create the need for detailed data-communication profiling. In this work, we have discussed various memory profilers, with a deeper focus on data-communication profiling. We have proposed

a categorization of memory profilers based on the profiling objective and profiling technique. In addition, we have provided a detailed comparison of the existing data-communication profilers. The important features of these profilers have been extensively discussed. Furthermore, the shortcoming in these tools are highlighted, which serve as recommendations for the improvement of the existing and/or the design of future data-communication profilers.

Note. The content of this chapter is based on the following article:

[1] Imran Ashraf, Mottaqiallah Taouil, and Koen Bertels. **Memory Profiling for Intra-application Data-Communication Quantification: A Survey**. In *Proceedings of 10th IEEE International Design and Test Symposium*, Dead Sea, Jordan, Dec 2015.

3

MCPROF: Memory and Communication Profiler

The growing demand of processing power is being satisfied mainly by an increase in the number of computing cores in a system. One of the main challenges to be addressed is efficient utilization of these architectures. This demands data-communication aware mapping of applications on these architectures. Appropriate tools are required to provide the detailed intra-application data-communication information and highlight memory-access patterns to port existing sequential applications efficiently to these architectures or to optimize existing parallel applications.

Based on the study in the Chapter 2, in this chapter, we present MCProf, *an efficient memory-access and data-communication profiler. In contrast to other tools,* MCProf *reports such information with manageable overheads for realistic workloads. Experimental results show that on the average, the proposed profiler has at least an order of magnitude less overhead as compared to other state-of-the-art data-communication profilers for a wide range of benchmarks.*

3.1. Introduction

Efficient application partitioning demands the understanding of the data-flow among various parts of a program in an application. With the growing program complexity, driven by an increasing demand of processing, it is time-consuming, tedious and error-prone to manually analyze these complex applications. Hence, program analysis tools are required to identify the hot-spots and/or bottlenecks pertaining

to the target platform[10].

Well maintained open-source and commercial performance analysis tools exist which report communication in programs where communication is explicit, such as MPI [86, 87]. We would like to highlight here that these tools are not designed to provide the communication profile of sequential applications. These tools are based on the technique which requires MPI parallel program as input. Hence, it only helps in validating the parallel program written only in MPI, rather than constructing one.

Data-communication profilers based on static-analysis tools can be developed [92], but they can only be used for regularly-structured applications and are inapplicable for most of the real-world programs due to their irregular structure. Additionally, pointer-analysis and their dynamic nature makes it hard to track the data-communication statically. Hence, dynamic methods are required to characterize the data-communication for such programs. Dynamic analysis tools generally have a high overhead as compared to static ones. To generate a realistic profile of applications dynamically requires the use of realistic workloads, which results in an increase in overhead. Another challenge with such tools is the difficulty of pinpointing the exact source-code location that is responsible for the data-communication. This information, though very useful for developers, makes the design of such tools challenging and increases their overhead.

In this work, we present an open-source memory-access and data-communication profiler which addresses these issues. The proposed tool provides detailed information which is not provided by existing tools. As a case-study, we analyze memory-access patterns and data-communication bottlenecks for a feature tracking application. Experimental results show that the proposed tool generates this information at considerably reduced execution-time and memory-usage overhead due to its well-thought design. The remainder of this chapter is structured as follows. We start by providing the related work in Section 4.2 and design considerations in Section 3.3. The design of the proposed profiler is detailed in Section 3.4. In Section 5.3.3, we discuss the use of generated information as a case-study to map an application on a platform using GPU as accelerator and the experimental results are described in Section 5.3.4. An empirical comparison of the overheads is presented in Section 3.7, followed by conclusions in Section 5.4.

3.2. Related Work

Various open-source [14, 64, 66] and propriety [15, 99] tools exist which perform memory profiling. However, these tools only provide the information about the cache misses and do not report data-communication information in an application to perform partitioning or communication-aware mapping.

Though static-analysis tools [92] can also track data-communication, a large number of tools utilize dynamic-analysis to collect accurate information at runtime. Ar-

chitecture simulation is one of the dynamic analysis technique which has been used to track the data-communication among threads in parallel applications [23] by using a cycle-accurate architecture simulator. However, simulation is generally computationally intensive which limits its use to small data inputs. Furthermore, it requires the design and development of a cycle-accurate simulator of these architectures. A well-known dynamic-analysis technique used by large number of tools is instrumentation, which is performed either at compile-time [100] or at run-time [18], [101]. Various tools based on this technique are used for finding memory-management [18] and threading bugs [18, 100]. However, very few tools exist, as discussed in this section, which perform detailed data-communication characterization, especially in sequential applications for efficient application partitioning.

Redux [75] a Valgrind based tool, draws the detailed Dynamic Data-Flow Graphs (DDFGs) of programs at the instruction-level. This tool has huge overhead as it generates fine-grained DDFGs. Hence, it can only be used for very small programs or parts of programs, as discussed by authors. Secondly, the purpose of the tool as reported by authors is to represent the computational history of a program and not to report its communication behavior.

Pincomm [77] reports the data-communication and it it is based on Intel Pin Dynamic Binary Instrumentation (DBI) framework [76]. Pincomm uses a hash-map to record the producer of a memory location. Due to this map, the tool has high memory overhead. Due to this overhead, Pincomm stores the intermediate information to the disk and reads it later by a script to generate the communication graph. This disk writing incurs high execution-time overhead. Furthermore, the authors also mention the use of markers in the source-code to reduce the overhead and manage the output complexity. However, in complex applications inserting these markers manually is time-consuming. Secondly, this marking requires knowledge of the application to understand what are the important parts of the program, which is not trivial.

Quad (Quantitative Usage Analysis of Data) [72], also based on Pin [76], provides data-communication information between functions by tracking memory access at byte-granularity. Trie is used to store producer-consumer relationships and it does so with less memory overhead as memory allocations in the Trie are done on demand at granularity of each byte. However, this approach has high execution-time overhead, mainly because of the access-time of Trie and the frequent memory allocations. Furthermore, the cumulative information is reported at the application-level, which makes it difficult to utilize. In addition, the information generated is not really useful when the application has different memory access behavior per call. Moreover, the provided information is without suitable relationship to the application source-code, which makes its use tedious for developers.

Summarizing, existing approaches have high execution-time and memory-usage overhead, which limits their use for realistic workloads. This may affect the quality of the generated profile. Furthermore, the provided information lacks necessary

dynamic details and is not linked to the source-code, making it hard to utilize this information.

3.3. Practical Considerations

In order to record memory-access behavior of an application and generate an inter-function data-communication profile, an important requirement is to know the producer-consumer relationship among functions. Consumer of a memory location is trivial to determine, as it is the currently executing function. On the contrary, efficiently obtaining the information about the producer is not trivial, as this requires recording the producer of each memory location at a certain granularity. The main reason for this difficulty is the huge user space, for example, on a 64-bit system, with 48-bit virtual addressing, memory addresses can be anywhere in the 128 TB memory-map in the user space. Hence, the problem boils down to efficiently recording the producer of a memory location, such that the profiling approach has a balanced trade-off between execution-time and memory-usage overhead. Furthermore, this mechanism should be flexible and portable, thereby making almost no assumptions about the memory map or other Operating System (OS) specific functionalities.

3.4. MCPROF: Memory and Communication PROFiler

In this section, we present the design of *MCProf* [1] which can conceptually be divided into three main blocks as depicted in Figure 3.1 and detailed in this section.

3.4.1. Memory Access Tracer

The memory access tracer uses Intel's Pin [76] DBI framework to trace memory reads and writes performed by the application. Pin provides instrumentation APIs to instrument at various granularity levels, such as instructions, basic-blocks, routines and image level. The instrumentation APIs allow the user to register callback routines, known as analysis routines, which are called when a certain event happen. For instance, registered instruction-level analysis routine will be triggered on each executed instruction. We utilize instruction-level instrumentation to track memory reads and writes by each instruction. Furthermore, routine-level instrumentation is utilized to keep track of the currently executing function. These are tracked by maintaining a call-stack of the functions executing in the application. Static symbols are obtained by reading Executable and Linkable Format (ELF) [102] header. To track the dynamic allocations, image-level instrumentation is utilized to selectively

[1] https://bitbucket.org/imranashraf/mcprof/downloads

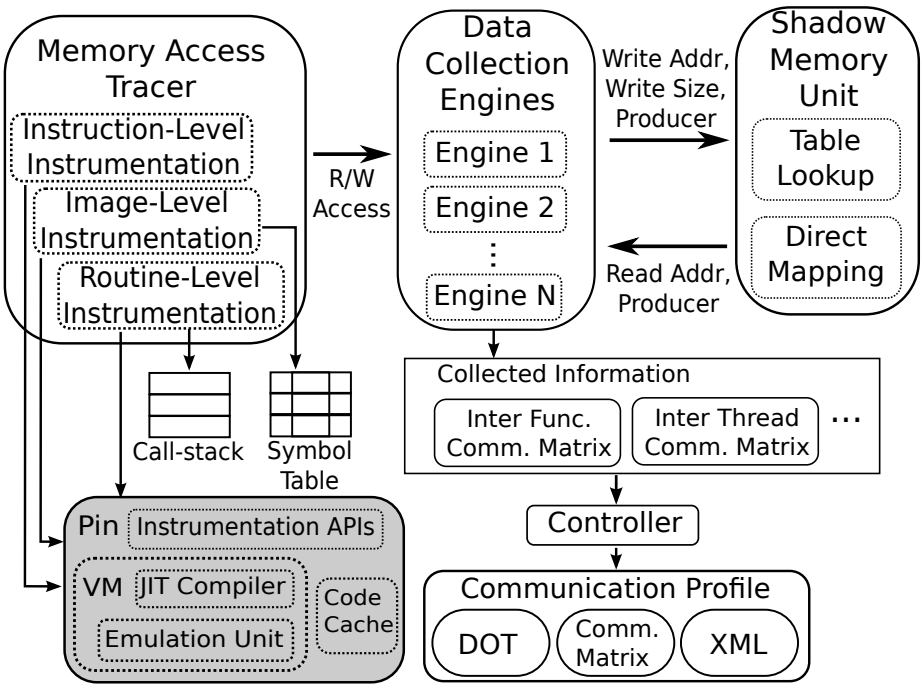

Figure 3.1: Main blocks of *MCProf* and their important sub-blocks. Application (binary) to be profiled is given as input to obtain data-communication profile in various formats at the output. Shaded block is the Intel's Pin DBI framework.

instrument library images for memory (re)allocation/free routines.

An important point to mention here is that although in the current implementation we have used the Pin framework to trace memory accesses, in the future, if desired, with minor modifications, it is possible to use any other DBI framework or any other technique to trace memory accesses.

3.4.2. Data Collection Engines

On each memory access traced by the Memory Access Tracer, a specific callback function is triggered based on the selected engine. In the case of a write, the producer of the memory address is recorded in the shadow memory. On a read access, the producer is retrieved from the shadow memory, while the consumer of the memory access is the function at the top of the call-stack. Furthermore, based on the information required by each engine, extra information is also recorded. For instance the source-line and filenames of the allocated blocks as well as the allocation size, which is stored in the symbol-table. Currently we have implemented the following three engines in *MCProf*.

- Engine-1: This engine reports the memory-intensive functions and objects in the application. This information, combined with the execution profile of the application, can be automatically used, if desired, to reduce the overhead by performing selective instrumentation and also reduce the complexity of generated profile.
- Engine-2: This engine records inter-function/inter-thread data-communication at the application level. The data-communication information is stored in a data-communication matrix, where indices of the matrix are the producer and consumer function/threads. When object-tracking is enabled, the data-communication is reported to/from the objects in the source-code.
- Engine-3: This engine generates per-call data-communication information. This is important for applications with irregular memory access behavior per-call. Each call is also given a unique sequence number which helps in identifying the temporal information of each call.

The complexity of these engines varies with the variation in the amount of details collected in the profile, hence, command-line switches are provided to select the desired engine. Furthermore, the modular design of the tool helps in easily adding new engines by modifying the existing engines to generate the desired information or produce the output in the desired format.

3.4.3. Shadow Memory

This block is responsible for recording the producer of each byte. On each write access, the selected engine sends the address, size, thread ID and the function at the top of the stack, which is the writer (producer) of this byte to the shadow memory unit. When a function reads a byte, the reader (consumer) is the currently executing function, while the the producer is retrieved from the shadow memory unit. These reads and writes can happen anywhere in the 128 TB user address space, so keeping track of the producer efficiently, is not trivial. Hence, the design of this shadow memory block has a great impact on the execution-time and memory-usage overheads of a profiler. Hence, we have combined the following two techniques in the design of the shadow memory unit.

- **Direct Mapping** in which an application's address is translated to a shadow memory address by using a *Scale* and *Offset*. Given an address *Addr*, its shadow address will be $(Addr \times Scale) + offset$. Although this address translation is fast, it assumes a particular OS memory layout and requires the reservation of a huge amount of virtual memory at fixed addresses.
- **Table-lookup** in which multi-level tables are used to map addresses in an application to their shadow addresses. This is similar to the page look-up tables utilized in OSes. This approach is more flexible as it does not require

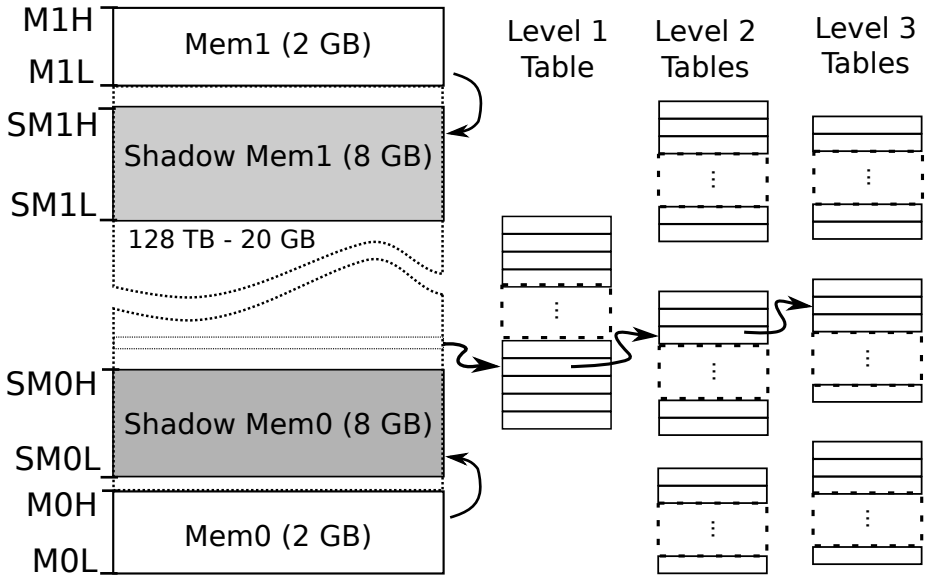

Figure 3.2: Hybrid shadow memory scheme utilized by *MCProf*. Most frequently accessed regions of memory (Mem0 and Mem1) use Direct Mapping. Rest of the memory map is shadowed by 3-Level Table Lookup.

neither a fixed memory layout, nor an initial reservation of huge memory, as tables are allocated on demand. The downside of this approach is that the multi-level table look-up is slower than the address translation in direct mapping.

In order to make a well-informed trade-off between flexibility, execution-time and memory-usage overheads, we have utilized a hybrid design of the shadow memory unit as shown in Figure 3.2. We analyzed the access frequency in the memory map and found out that the most frequently accessed memory is the bottom (Mem0) and the top (Mem1) regions in the memory map. For most of the applications, N and M can be 2 GB as shown in Figure 3.2. To make accesses to these regions faster, we reserve[2] in advance two shadow memories corresponding to these two memory regions, shown as *Shadow Mem0* and *Shadow Mem1*, respectively. This results in a simpler mapping of addresses in these regions to the shadow addresses by Equation (3.1), without requiring any lookup.

$$Addr_{sh} = ((Addr\&M0H) << log_2(SCALE)) + (Addr\&(SM1L + SM0L)) + SM0L \tag{3.1}$$

[2]These regions are only reserved in the memory map, actual memory-usage is 4 B for each byte of memory used by the program.

where, $Addr$ is the address of the original byte, $Addr_{sh}$ is the address of the corresponding shadow bytes, $SCALE$ is 4, and $M0H$, $SM1L$ and $SM0L$ are constants as shown in Figure 3.2.

A point worth mentioning here is that in the Windows/Linux OS, heap grows from lower to higher addresses, while stack grows from higher toward lower addresses. The shared libraries are mapped by the OS somewhere between these bounds. A simpler and faster approach could have been to restrict the address mapping of a program to the lower half and use the upper half for the shadow memory. However, to the best of our knowledge, there is currently no (portable) method to restrict the address allocation in this manner. So, for the middle (128 TB $-$ 20 GB) less frequently used region, we utilize a 3-level table-lookup scheme as shown in Figure 3.2 to cover this address space.

Initially, the level-1 table is created and all its entries are marked as *UNACCESSED*. Tables in the remaining two levels are created on demand when the address in that range is touched for the first time. The address of the memory accessed in this region is used to index these tables to reach level-3 where 4 shadow bytes are written for each byte memory accessed in the original program. One byte for the function ID, one byte for thread ID and 2 bytes for the ID of the object this address belongs to. Therefore, we currently restrict the number of function and thread IDs to 256. In the future we will investigate more applications, and if required, increase the number of bytes to store the IDs, as it is simply a parameter in the tool.

3.5. Case-study

The focus of this case study is on the utilization of data-communication information provided by *MCProf* to map an application onto the GPU, without performing algorithmic modifications. The use case involves Kanade-Lucas-Tomasi Feature Tracker (KLT) application [103]. This application detects interesting features in a frame and tracks them in the subsequent frames. We have used version $1.3.4$, which is the latest version of KLT [104]. This *C* implementation has 102 functions in 17 source-files making up 5033 lines of code.

For the experiments performed in this case study, we used 64 bit, 2.5 GHz Intel(R) Xeon(R) CPU with 32 GB RAM. Nvidia GeForce GT 640 GPU, with 2 GB memory, is used as an accelerator which is connected to the PCIe slot of the CPU. Ubuntu 12.04 is running on the machine with Linux kernel 2.6.32-24-server and Nvidia driver version 319.37. Nvidia CUDA toolkit $V6.0$ is used to program the GPU.

Table 3.1: *gprof* flat profile for the *KLT* application.

Function Name	%Time
KLTSelectGoodFeatures	54.07
convolveImageVert	19.65
convolveImageHoriz	10.17
trackfeature	7.81
%Total Contribution	**91.7**

3.5.1. Implementation without Data-communication Optimization

In order to efficiently map an application onto an accelerator based platform, compute intensive functions, known as kernels, are off-loaded to the accelerator. We used *gprof* [13] to identify the kernels in the application as shown in Table 5.1. For this run, 30 frames have been used with frame size chosen as 1024×768, to accumulate enough number of samples to generate representative profile of the application. The total percentage contribution of these kernels 91.7% (0.917 per unit).

As a first step in the mapping process, we mapped these kernels to the GPU. Table 3.2 provides the timing results of the first mapping step. For these experiments, 1024 features were tracked from frames of size 1024×768. Column 1 contains the names of the compute-intensive kernel. Column 2 lists the execution-time of these kernels on CPU (t_{cpu}) in seconds. $t_{gpu_{comp}}$ is the time spent in performing the computation on GPU which is shown in Column 3. The communication time $t_{gpu_{comm}}$ is listed in Column 4 which is the time spent in transferring data to GPU before computation and reading the results back, after the computation is complete. The execution-time speedup is the ratio of the execution-time on CPU and GPU. Total kernel speedup ($S_{K_{total}}$) is reported in Column 5, which is calculated as $\frac{t_{cpu}}{t_{gpu_{comp}}+t_{gpu_{comm}}}$. In order to highlight the effect of data-communication, Column 6 lists the kernel speedup ($S_{K_{comp}}$) for only the computation, calculated as $\frac{t_{cpu}}{t_{gpu_{comp}}}$.

From Column 5 in Table 3.2, it can be seen that speedup has been obtained for all the kernels except for `trackFeature` kernel. Hence, this kernel should not be mapped to GPU. Another important result that can be deduced by comparing Column 5 to Column 6 is that the communication has significantly reduced the achieved speedup. In the next sub-section we will perform the optimization of this data-communication by utilizing *MCProf*.

Table 3.2: Execution Time (sec) and Speedup results for the initial KLT implementation.

Kernel	t_{cpu}	$t_{gpu_{comp}}$	$t_{gpu_{comm}}$	$S_{K_{total}}$	$S_{K_{comp}}$
KLTSelectGoodFeatures	13.53	1.17	0.36	8.8×	11.52×
convolveImageVert	3.93	0.14	0.76	4.35×	28.08×
convolveImageHoriz	1.77	0.18	0.76	1.87×	9.89×
trackFeature	1.96	1.49	0.52	0.96×	1.31×

Table 3.3: Memory Intensive Objects in *KLT* reported by *MCProf*.

Objects	Reads	Writes	Reads/Writes	Total	%Total
tmpimgCS	3.8e8	5.1e7	7.4	4.3e8	26.6
pointlist	1.3e8	1.3e8	1	2.6e8	16.3
pyramidImg	1.3e8	3.5e7	3.8	1.7e8	10.3
grady	1.34e8	3.1e6	42.7	1.3e8	8.3
gradx	1.34e8	3.1e6	42.7	1.3e8	8.3
tmpimgTF	6.7e7	9.4e6	7.1	7.6e7	4.6
guassderiv_kernel	6.7e7	4.7e3	14063.1	6.7e7	4.1
guass_kernel	6.7e7	4.6e3	14500.6	6.7e7	4.1
%Total Contribution					**82.6**

3.5.2. Optimization of Data-communication

Optimization of the data-communication requires understanding of the data-flow in the application. Understanding this data-communication by manual source-code is not trivial, especially for applications like the one we have considered in this case study involving a large number of functions and objects. Furthermore, pointer arithmetic exacerbates this problem making it hard to determine the real producer and consumer of the data.

MCProf provides this production-consumption information in the form of a data-communication graph in various formats. An overview of this information is shown as communication matrix in top right corner of Figure 3.3 representing inter-function communication intensity. *MCProf* also generates the detailed quantitative data-communication information in the form of a directed graph. As there are large number of functions in the KLT application, the complete graph is too large to present here. Secondly, such large graphs are hard to be utilized by developers. Typically, the most compute-intensive functions are selected for analysis. *MCProf* detects memory-intensive objects and the functions communicating with these memory-intensive objects. Table 3.3 lists the memory-intensive objects of the KLT application reported by *MCProf*. Apart from mentioning the reads and writes accesses, the percentage accesses are also reported in the last column. The last row of the table shows that memory accesses through these 9 objects correspond to 82.6% of the total application memory accesses.

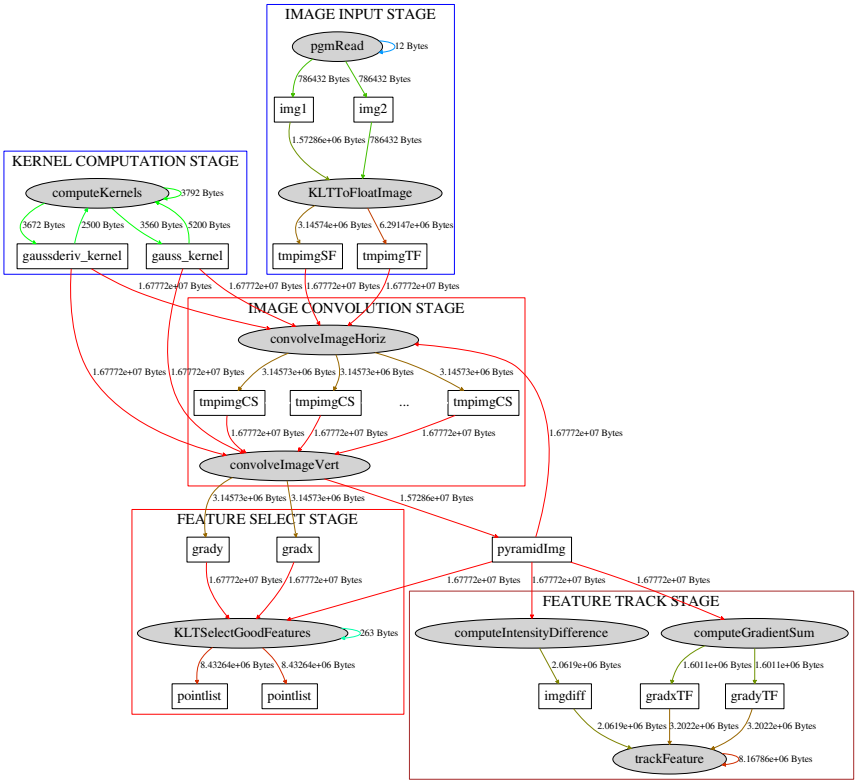

Figure 3.3: KLT communication graph generated by *MCProf*. Functions (ovals), compute-intensive functions (Grey ovals) and the objects(rectangles) involved in the communication are also shown.

Figure 3.3 shows the data-communication graph of KLT application generated by *MCProf* while tracking 256 features in 3 frames of size 1024×768. The ovals represent the functions in the application whereas the objects are represented by rectangles where the number inside the rectangle is the allocation size. The kernels in the application are shown in Grey ovals. The amount of communication in bytes is represented by directed edges, where the color of the edges represent the intensity of the communication. To simplify the discussion, the dotted lines are used to mark the functions in the main stages of applications.

`tmpimgSF` and `tmpimgTF` are generated by `KLTToFloatImage` on CPU and transferred to GPU as an input to `convolveImageHoriz`. `KLTToFloatImage`, though not compute-intensive, still mapping it to GPU will be better as it will make `tmpimgSF` and `tmpimgTF` internal to GPU and reduce CPU-GPU communication.

`guassderiv_kernel` and `guass_kernel` are generated by `computeKernels`

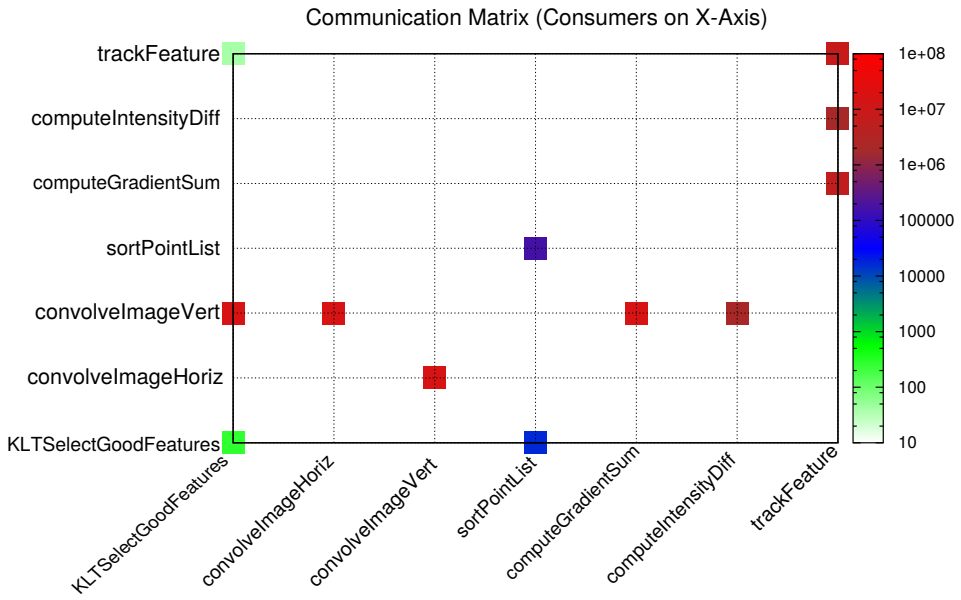

Figure 3.4: KLT communication matrix generated by *MCProf*.

on the CPU and consumed by `convolveImageHoriz` and `convolveImageVert` on GPU. However, mapping the function `computeKernels` to GPU is not required as `guassderiv_kernel` and `guass_kernel` are consumed heavily but produced very infrequently. This is also evident from the very high Reads/Write ratio (Table 3.3) implying very less production and a lot of consumption of data from these objects. Furthermore, these objects are very small in size (284 Bytes), hence can be easily mapped to GPU's constant memory.

Another optimization which can be performed in the convolution stage is the allocation and de-allocation of large number of `tmpimgCS` objects for each frame. Allocating a single object in the start and re-using it in the subsequent frames instead of re-allocating it will reduce the execution-time. Similar optimization can be performed for `pointlist` in the Feature Selection stage.

`gradx` and `grady` generated in the Convolution Stage are consumed in the Feature Selection stage by `KLTSelectGoodFeatures`. Hence, these objects can be kept on the GPU for utilization in these stages. Furthermore, these objects should be mapped to GPU's shared memory. This is because of high Reads/Writes ratio depicted in Table 3.3, suggesting high re-use of these objects. This will result in performance improvement as shared memory has higher bandwidth as compared to global memory. On the contrary, `pointlist` has Reads/Writes ratio of 1, which suggests no reuse, hence it should be kept in the global memory. Mapping `pointlist` to shared memory will only increase the overhead of the data transfer between global memory and shared memory without being reused.

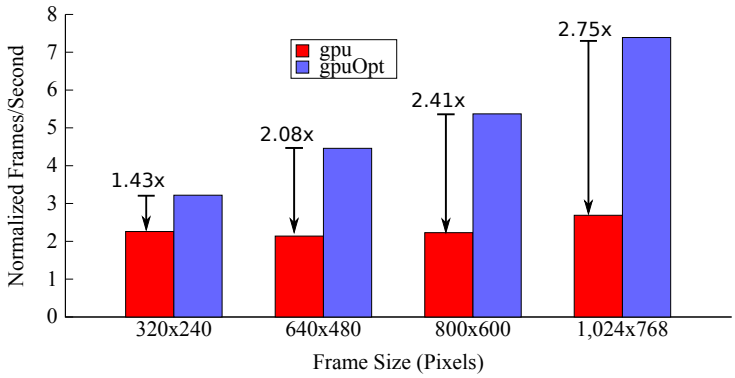

Figure 3.5: Normalized frames per seconds achieved by the GPU and data-communication optimized GPU implementation.

Based on the preliminary results in Table 3.2, it was concluded that `trackFeature` should not be mapped to GPU because of slow-down. Even if this kernel is not so efficient on the GPU, we should still port it to the GPU to avoid the bulk of data-communication regarding the transfer of `pyramidImg` between GPU and CPU. This is clearly shown by the communication edges to the `computeIntensityDifference` and `computeGradientSum` function in the Feature Tracking stage.

3.6. Experimental Results

In this section we provide the performance results for the implementations discussed in the case study. The frame sizes have been selected corresponding to the frame dimensions used in various video standards [105]. The affect of varying the number of tracked features on the achieved application speedup is also discussed. The number of features to be tracked is set to 1024.

After applying these optimizations to the initial GPU implementation (gpu), we obtained a data-communication optimized version of the GPU implementation (gpu_{opt}). Figure 3.5 shows the normalized Frames Per Seconds (fps) achieved by both the implementations for various frame sizes ranging from 320×240 to 1024×768 while the number of tracked features is set to 1024. Increasing the frame size, results in an increase in the amount of computation performed on the GPU. Increased computation results in better utilization of the available resources of the GPU, resulting in higher speedup as can be observed from this figure. On the other hand, increasing the frame size also increases the amount of frame data transferred to the GPU for processing and getting results back. This data-communication has been optimized in the case of gpu_{opt} based on the information provided by *MCProf*. Hence, gpu_{opt} implementation achieves up-to $2.75\times$ higher speedup as compared to gpu implementation where this communication is not optimized.

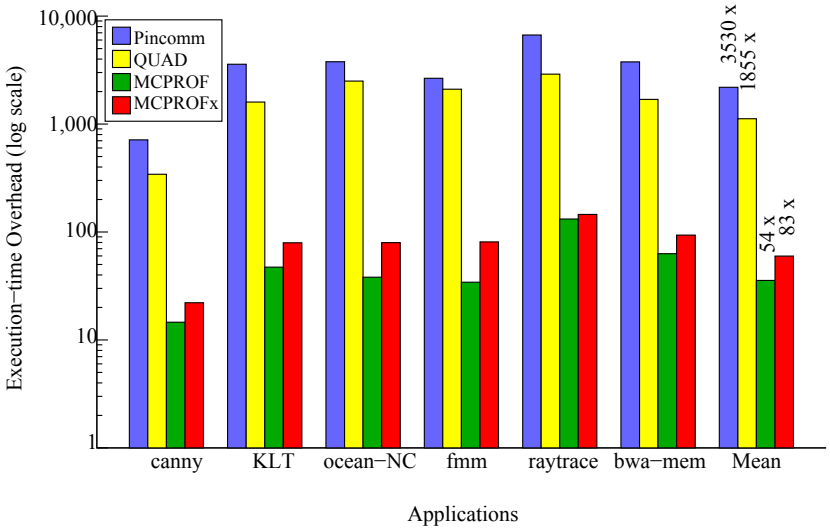

Figure 3.6: Execution-time overheads.

3.7. Overhead Comparison with Existing Profilers

In this section, we present the overhead comparison of *MCProf* with other state-of-the-art data-communication profilers Quad and Pincomm as these are particularly designed to report data-communication. To make a fair comparison, these tools are configured in such a manner that all the profilers generate exactly the same information while running on the same platform. For these experiments, we used Pin v2.13 on the machine used in case-study. Figure 3.6 depicts the execution-time and memory-usage overhead of Pincomm, Quad and *MCProf* for applications from various domains, namely; image-processing (canny, klt) domain, SPLASH-2 benchmarks (ocean-NC, fmm, raytrace) and a bio-informatics application (bwa-mem). Each bar represents the ratios of the application execution-time with and without profiling for each profiler. Similarly, Figure 3.7 reports the ratios of application memory-usage with and without profiling.

We have reported *MCProf* results with two different settings depicted as *MCPROF* and *MCPROFx* in these figures. Results with *MCPROF* legend are overheads while providing the common basic information which Pincomm and Quad can also generate. Whereas, *MCPROFx* report overheads of complex engine while generating the detailed data-communication information with stack recording and object detection. Mean overhead results are also depicted in these figures. These results show that *MCProf* has, on the average, an order of magnitude less execution-time and memory-usage overheads. Main reason for this reduction in overhead is the well-thought design of the shadow memory scheme utilized by *MCProf*. Due to the design, we were able to shift most of the processing from analysis-time to instrumentation-time. Moreover, the access-time and memory-usage overhead of

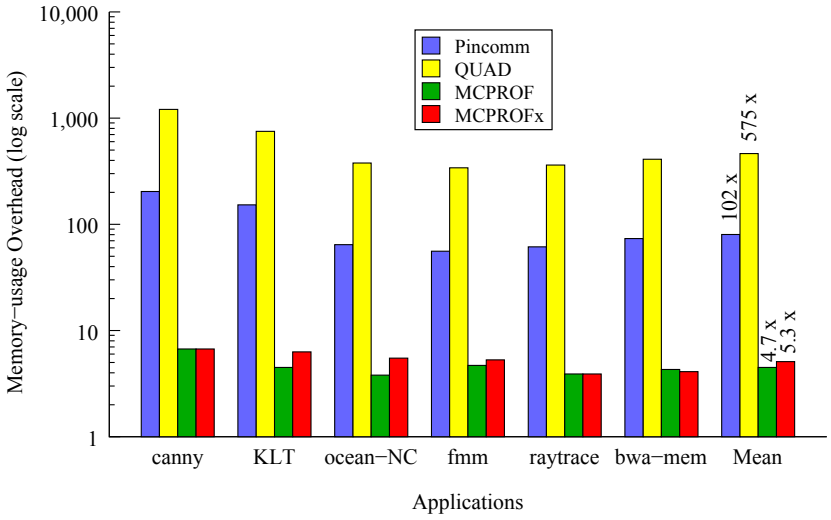

Figure 3.7: Memory-usage overheads.

the hybrid shadow memory scheme is considerably less as compared to Trie or Hash map utilized by Quad and Pincomm, respectively. In order to clearly illustrate this, we have plotted the execution-time and memory-usage of accessing only the data-structures of the three tools in Figure 3.8 and Figure 3.9, for the canny application for various image sizes.

Another important observation from Figure 3.6 is that *MCProf* has an average memory-usage overhead of $4.7-5.3\times$, which is mainly because 4 shadow bytes are allocated for each byte used in the original program, plus some additional memory for storing extra information.

3.8. Conclusion

Both the memory wall and the multi-core trend create the need for detailed data-communication profiling. In this work, we presented the design of *MCProf*, a memory-access and data-communication profiler. The unique design of the proposed profiler resulted in significantly reduced execution-time and memory-usage overheads as compared to the state-of-the-art, making the profiler useful for real applications with realistic workloads. The reduced overheads allowed us to generate additional memory-access and data-communication information which is not provided by existing tools.

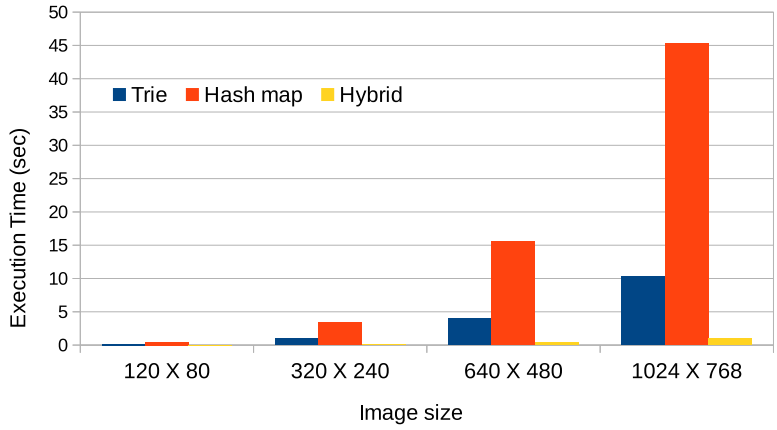

Figure 3.8: Execution-time comparison of data-structure access only.

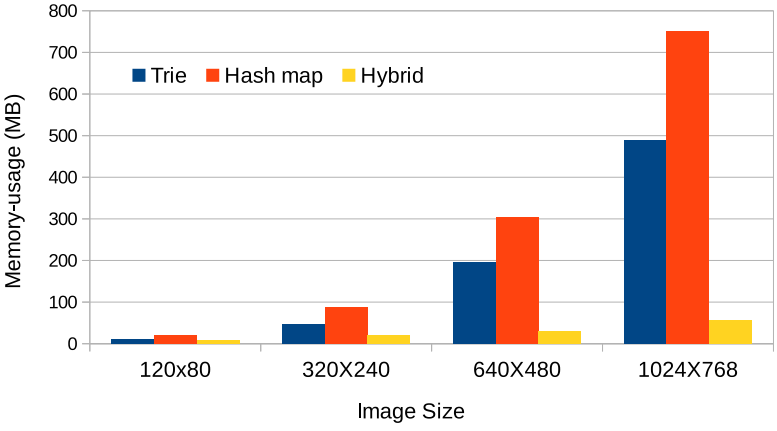

Figure 3.9: Memory-usage comparison of data-structure access only.

Note. The content of this chapter is based on the following article:

[1] Imran Ashraf, Vlad-Mihai Sima, and Koen Bertels. **Intra-Application Data-Communication Characterization**. In *Proceedings of 1st International Workshop on Communication Architectures at Extreme Scale*, Frankfurt, Germany, July 2015.

4

Profile Driven Application Parallelization

In this chapter, we first present a semi-automatic parallelization methodology based on MCProf *to help programmers extract and express parallelization. The* MCProf *tool provides a detailed profile of the data flowing inside an application and the* xpu *programming paradigm provides an intuitive and simple interface to express parallelism as well as the necessary runtime support. We present in detail a use case, Canny Edge Detection, as well as the performance numbers for a second application, fluid animate. Later on, we present a framework which automates the whole process of extracting various forms of parallelism in sequential applications. The framework takes a sequential application, generates the required static and dynamic information and utilizes this information to automatically generate the parallel representation of the application. We demonstrate the working of framework by a case study to highlight the information generated by various tools involved in the developed framework. We also discuss that our framework is able to extract various forms of fine and coarse grained parallelism.*

4.1. Introduction

Despite decades of research and development of parallelizing compilers, automatic parallelization of sequential code using a compiler is standing as the holy grail of parallel computing and has had a limited success. Utilization of tool-chains which assist the programmer by full or partial automation of various parallelization phases presents a better alternative which can offer better performance-productivity trade

off.

In this chapter we present the semi-automatic approach. Section 4.2 discusses some of the available parallelizing compiler frameworks. Section 4.3 presents the results of the Canny Edge Detection algorithm when using two state-of-the-art commercial parallelizing compilers. Section 4.4 presents the proposed parallelization methodology and achieved results. Later in Section 4.5 we introduce the parallelizing framework followed by Section 5.3.3 describing the use of the proposed framework.

4.2. Background and Related Work

Though static-analysis tools [92] can also track data-communication, a large number of tools [72, 73] utilize dynamic-analysis to collect producer-consumer relationship at runtime. These tools have high run-time overhead, which limits their use for realistic workloads affecting the quality of the generated information. Furthermore, the provided information lacks necessary dynamic details and is not linked to the source code, making it hard to utilize this information. A number of automatic parallelization compilers exist such as Par4All [106], Cetus [107], Parallware [108], Polaris [109] or PolyCC/PLUTO [110] which are source to source compilers that can produce parallel code after analyzing the sequential code using different parallelization techniques. The Intel ICC [111] is a popular compiler which provides an automatic parallelization feature which allows both instruction-level and thread-level parallelization of sequential regions of the input code.

Automatic parallelization of sequential code has had limited success [112]. Great advances have been made in automatic parallelism extraction at the instruction level, however, in order to exploit efficiently modern multicore platforms, compilers need to capture parallelism also at thread-level which is a challenging task. Pareon by Vector Fabrics [113] is an example of a tool, which assists the programmer and guides the parallelization process instead of performing automatic code parallelization.

4.3. Parallelization using Existing Commercial Compilers

In this section, we present the parallelization of the Canny application by using two commercial compilers; we refer to them as CC1 and CC2 in the rest of the discussion. CC1 can be used both in automatic and semi-automatic way, whereas, CC2 uses only a semi-automatic approach. We attempted to use PolyCC/PluTo compiler [110] which uses polyhedral analysis to tile and parallelize loops in sequential programs. However, the compiler suggested significant manual modification of the code in

order to make it processable by the compiler. For instance, the compiler requires removing all affine expressions from the inner loop to be able to process it. So the parallelization process is no longer transparent.

4.3.1. CC1

CC1 allows automatic parallelization of sequential program at thread-level using OpenMP and at instruction-level through vectorization using SSE/AVX intrinsics. In order to parallelize a sequential program, the compiler searches for loops which do not expose cross-iteration dependence and are good candidate for parallel execution. A data flow analysis is performed to ensure correct and safe parallel execution. The compiler then uses OpenMP to specify the parallelism.
Default Automatic Mode: is a one shot fully automatic parallelization mode in which the program is parallelized using compile-time static analysis.
Run-time Controlled Mode: collects the run-time information (profiling data) and uses it to guide the program parallelization.
Guided Parallelization Mode: in which compiler can be used to perform run-time analysis to generate an advisory reports, suggesting ways (often code modifications) to the programmer to parallelize loops.

Parallelization of Canny Application using CC1

For the canny application, the default automatic mode and run-time controlled mode did not show any significant speedup over the original sequential application, hence, we used the guided parallelization mode. By using the advisory reports, the loops in the `gaussian_smooth` were successfully parallelized, however, many other loops could not be parallelized due to false data-flow dependences. Listing 4.1 illustrates an example where a small manual modification made the loop parallelizable. We made several manual modifications such as nested loop fusion and iterator duplication to help the compiler. The resulting parallel program achieved better speedups (up to 4× for 16 threads on a platform with two Intel Xeon E5-2670 processors at 2.6 GHz).

CC1 is capable of performing instruction-level parallelism by vectorization. In this regard, we performed three experiments to evaluate the vectorization quality in each case:

1. In order to evaluate the vectorization quality, we measured the achieved performance of one of the two loops of the `gaussian_smooth` function. This loop was reported as auto-vectorized by the compiler.
2. In the original code, this loop contains control branches (`if`) to handle loop bounds, which limit vectorization efficiency. We removed these branches.
3. Finally, we vectorized the code manually using SIMD intrinsics (SSE4.2).

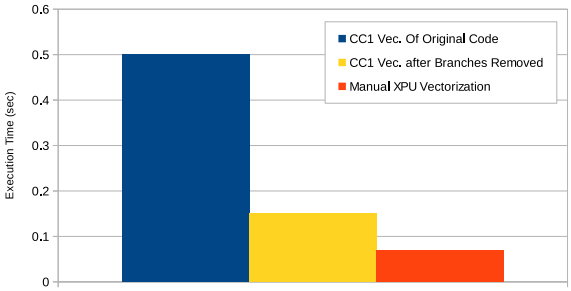

Figure 4.1: Performance comparison of automatically vectorized code using CC1 and manually vectorized code.

The results of these three versions are shown in Figure 4.1. It can be seen that the manual vectorization achieves significantly better performances than the automatic vectorization performed by the compiler even after vectorization-friendly code transformations.

4.3.2. CC2

CC2 aims at the parallelization of sequential C/C++ code for ARM Cortex A9 and x86 as target systems. To use CC2, an application is compiled with a C99 compliant compiler to perform instrumentation of the application. The instrumented application is then executed on the model of the target architecture to generate the profile of the application. The generated profile can be visualized in a GUI to select the loops which have high execution contribution as the candidates for parallelization. Furthermore, the dependencies are also reported in a GUI.

After selecting the number of cores in the architecture, CC2 predicts the achievable speedup at the loop-level as well as the application-level. Based on the selected parallelization, refactoring steps are presented by the tool to be applied on the sequential code by the programmer to parallelize it. CC2 suggests parallelizations in terms of a low level threading Application Programming Interface (API) and OpenMP pragmas.

```
1 // Loop carried dependency on pos in orignal sequential loop
2 for (int pos=0, r=0; r<rows; r++, pos++)
3     for (int c=0; c<cols; c++, pos++)
4         image[pos] = x;
5
6 // equivalent parallelizable loop
7 for (int pos=0; pos<rows*cols; pos++)
8     image[pos] = x;
```

Listing 4.1: CC1 fails to resolve an easily removable data dependencies which prevents parallelization.

Parallelization of Canny Application by CC2

For the loops in `gaussian_smooth`, CC2 detected that there are no loop dependencies and suggested an OpenMP pragma. For a 4-core system, loop speedup of 4× was predicted, whereas the achieved speedup is only 2.58×. Similarly, a 2.2× application speedup was predicted, however, the actual application speedup is 1.64×.

For the loop in `magnitude_xy`, CC2 detected `r` as an induction variable, however, for `pos` variable, synchronization was suggested. This synchronization resulted in slowdown, instead of speedup. By changing the code, as already discussed in Listing 4.1, programmer can avoid this synchronization. Hence, a loop speedup of 2.84× and an application speedup of 1.7× was achieved. Finally, for the third loop in `non_max_supp`, CC2 predicted no speedup, hence, it was not parallelized.

4.3.3. Lessons Learned

It can be summarized from the discussion in this section that Parallel compilers, such as CC1, may significantly improve programmer's productivity by automatic code parallelization, however, such compilers suffers from inherent difficulties. Hence, manual code analysis and modifications are required.

CC2 performs run-time analysis to detect dependencies and suggest parallelization. However, there are cases in which even simple loop carried dependencies are not resolved automatically. Therefore, these dependencies are either resolved manually or synchronization is suggested. This synchronization can result in loop (and hence application) slow-down, instead of speedup, requiring manual inspection and modification of application by the programmer. Furthermore, the information is provided among loop iterations, not across loop nests or functions in the application to exploit coarse-grained application parallelization and data-flow optimizations.

This outlines the need for tools which analyze the programs dynamically to provide data-flow information to highlight real data-dependences. This information should be provided at various granularity levels to extract and express various forms of parallelization available in the application.

4.4. *MCProf*-xpu Semi-automatic Approach

xpu [1][114] is a structured parallel programming framework which aims to easily express and exploit parallelism. xpu allows the expression of different types of parallelism at different levels of granularity. Supported parallelism types includes data parallelism [115], task parallelism [114] and pipeline parallelism [116]. These different parallelism types can be composed hierarchically in the same application [117].

The xpu parallel programming model is designed for easing explicit parallelism expression and therefore requires locating the hot-spots in the program and extracting parallelism by analyzing task-dependencies (producer-consumer relationships). Usually this analysis is performed manually by reading and analyzing the code and profiling the application, which is a time-consuming and error-prone task. *MCProf* can automate this analysis phase and provides a clear picture of the program with all the required information including task-dependencies, compute-intensive and communication-intensive hot-spots in the application. Hence, *MCProf* and xpu can form a parallelization tool-chain which offers a much smoother transition from the sequential-code to the parallel-code. The *MCProf*-xpu parallelization methodology follows the traditional parallelization steps which can be summarized as follows:

1. **Profiling sequential code:** to locate hot-spots.
2. **Extracting parallelism and granularity adjustment:** analyzing data and task dependencies and decomposing tasks to extract more parallelism, if available, at finer grain.
3. **Expressing parallelism**: using the parallel programming API.
4. **Executing the parallel code efficiently:** parallel programming libraries are often based on a run-time which is responsible of efficient scheduling to optimize execution.

4.4.1. Profiling The Sequential Application using *MCProf*

Figure 4.2 shows the *MCProf* output graph of Canny, where nodes represent functions. Each node contains the function name, the percentage of dynamically executed instructions by this function with respect to the whole application, as well as the total number of calls to this function. For instance, `gaussian_smooth` is executed once and it is a computationally intensive function as it covers 80% of the instructions executed by the whole application.

In addition, the inter-function data-flow is represented by edges. Furthermore, the statically and dynamically allocated objects involved in each flow are also detected by *MCProf* and represented by rectangles containing object names and allocation sizes. The communication intensity is quantitatively shown by the number of bytes on each edge and also illustrated by the color the edges from red (highest) to green (lowest). In this way, programmer is able to visualize the computation- and communication-intensive parts of an application, in a single graph, without manual source code inspection.

To extract parallelism, coarse-grained functions should be decomposed to extract potential finer-grain parallelism, we refer to this decomposition step as granularity adjustment.

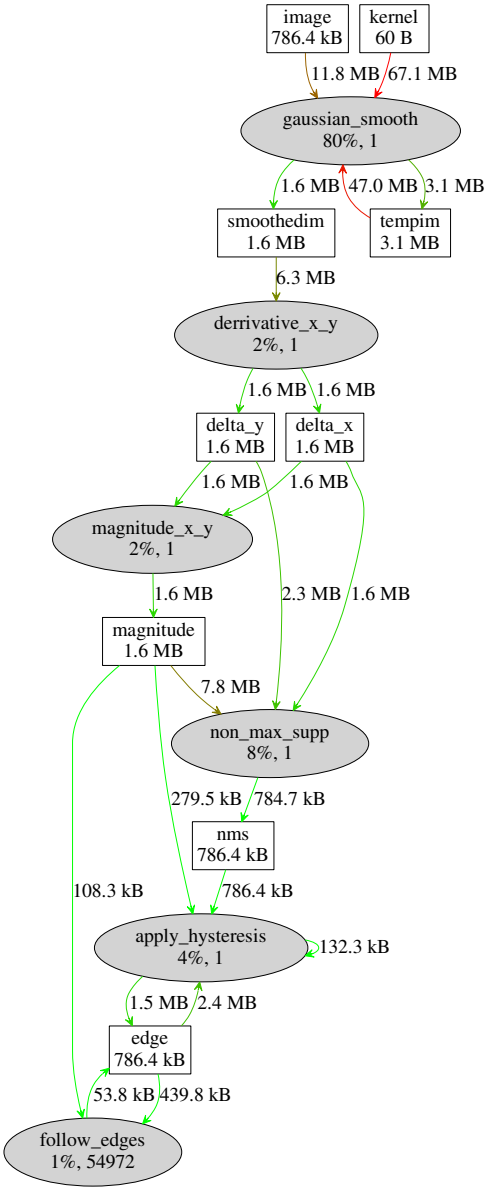

Figure 4.2: Task dependency graph of Canny application generated by *MCProf*.

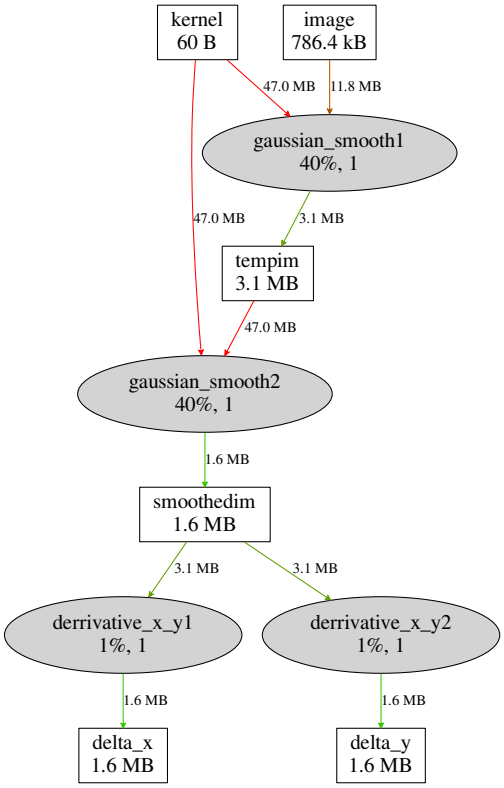

Figure 4.3: *MCProf* profile at loop-level granularity showing independence of the two loop nests inside the *derivative_x_y* function.

4.4.2. Granularity Adjustment

As seen in Figure 4.2, `gaussian_smooth` has 80% execution contribution in the application. In order to extract parallelism inside this function, we use the *MCProf* feature of splitting functions at loop level granularity. Figure 4.3 shows the simplified output graph generated by *MCProf* where the loops in `gaussian_smooth` are split as separate nodes represented by `gaussian_smooth1` and `gaussian_smooth2`. A similar split of `derivative_x_y` is also shown as `derivative_x_y1` and `derivative_x_y2`.

As depicted in the Figure 4.3, the `gaussian_smooth` function exposes dependencies between its two loop nests, thus they cannot be executed concurrently. However, each of these loops can be parallelized individually since they operate independently on columns and rows of the image. On the other hand, the two

```
1 int gaussian_smooth1(char *inImg, float *kernel, char *outImg, /* more args */)
2 {    /* process an image row */ }
3 int main(){
4     // task definition
5     xpu::task gauss_task(gaussian_smooth1, image, kernel, tempim, /* more args */);
6     // parallel loop construction
7     xpu::parallel_for parallel_gauss(0,size,1,&gauss_task);
8     // parallel loop execution
9     parallel_gaussian.run();
10 }
```

Listing 4.2: xpu parallel for loop construction and execution.

parts of the `derivative_x_y` do not expose any dependency and therefore can be executed in parallel, since these loops are using a common read-only input while producing separate outputs. A similar analysis is performed on other functions to extract the available parallelism.

4.4.3. Parallelization Using xpu

After analyzing the data-flow dependencies and extracting parallelism, we use the xpu framework to express the parallelism. In this regard, xpu skeletons are utilized to exploit various forms of parallelism available in the application.

Thread-level parallelism: Thread-level data parallelism can be achieved by parallelizing sequential loops. In xpu, this is expressed by using `parallel_for` pattern. For instance, Listing 4.2 shows how the `gaussian_smooth` is parallelized using the xpu parallel loop skeleton. The task which processes the data elements is constructed and named `parallel_gaussian`. We note that data partitioning and tasks scheduling are handled transparently by the xpu run-time to ensure dynamic forward-scalability and execution-efficiency across different platforms.

Instruction-level parallelism: Beside loop parallelization, vectorization can act as a great performance multiplier by allowing SIMD operations on the pixels of the image. xpu provides transparent vectorization through built-in vectorized types such as `vec4f` or `vec8s`, which are implemented using x86 SSE intrinsics. For instance, using `vec4f` allows vectorization of standard operations on single precision `float` and other composite operations which are not available natively in SSE instruction-sets, such as trigonometric functions. Similarly, by using the `vec8s` type, the programmer can operate implicitly on 4 floats simultaneously.

Understanding the data-access pattern is one of the major challenging task in the vectorization process especially in the case of irregular or non-contiguous memory-accesses (load and stores). Tracking data-accesses across loops iterations can be a time-consuming and error-prone task. To address this issue, *MCProf* can generate a graphical view of the data-access pattern of both input and output data in a user-delimited region of the code, particularly loops. Figure 4.4 shows the data-access

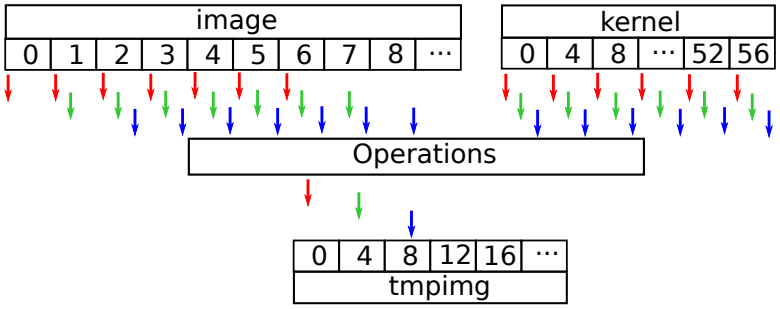

Figure 4.4: Fine-grain access-pattern of $gaussian\_smooth1$ loop reading $image$, $kernel$ and writing $tmpimg$ objects. Accesses in same iteration are represented by the same color.


```
1 xpu::vec4f v1,v2,v3,acc,k1,k2,k3,k4;
2 for (int i=begin; i<end; i++) {
3     v1 = &in[i];    v2 = &in[i+4]; //load inputs
4     v3 = &in[i+8]; v4 = &in[i+12];
5     // vectorized operations :
6     acc = k1*v1 + k2*v2 + k3*v3 + k4*v4;
7     sum = (k1+k2+k3+k4).sum();
8     acc /= sum;
9     out[i-(kernel_size/2)] = acc.sum();  //store output
10 }
```

Listing 4.3: Code Vectorization using xpu.

patterns in the first three iterations of the gaussian loop. Each color corresponds to an iteration. In each iteration, we used the xpu vectorized type `vec4f` to load the required inputs from both `image` and `kernel` at the indicated position. Four multiplications and the sum is then performed simultaneously. This allowed us to achieve a significant speedup over both the sequential code and the automatically vectorized code generated by the compiler. Listing 4.3 shows an example of vectorized code using xpu.

Task Parallelism: As depicted in the *MCProf* output in Figure 4.3, `derivative_x` and `derivative_y` do not expose any producer-consumer dependencies and thus can be executed as parallel tasks. This can be easily specified using xpu as shown in Listing 4.4.

Figure 4.5 shows the speedup achieved for the Canny application parallelized by

```
1 // tasks definition
2 xpu::task dx_t(derrivative_x, smoothed_img, dx);
3 xpu::task dy_t(derrivative_y, smoothed_img, dx);
4 // parallel execution
5 xpu::parallel(&dx_t, &dy_t)->run();
```

Listing 4.4: Task-level parallelism in $derivative\_xy$ expressed in XPU.

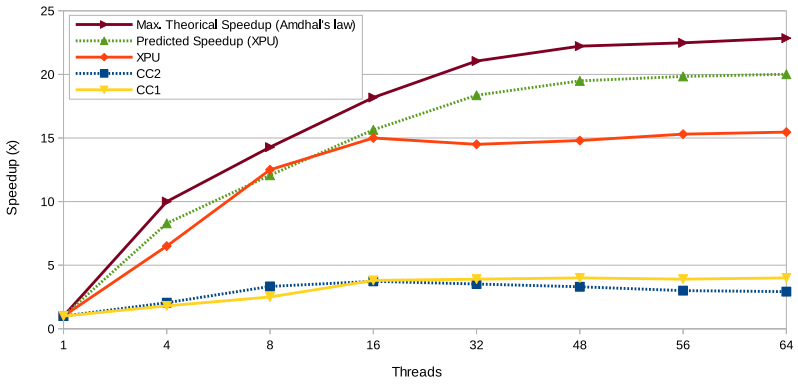

Figure 4.5: Achieved speedups over the sequential execution for the Canny application when parallelized by various approaches.

MCProf-xpu methodology, CC1 and CC2. The theoretical speedup based on Amdahl's law and estimated speedup while considering SIMD support is also shown. It can be seen that a speedup of $15\times$ is achieved for 16 cores on an `Intel Xeon E5-2670`, which is about $4\times$ higher than what achieved by CC1 and CC2. Similar performances are achieved on 64 cores platform with four `AMD Opteron 6274` processors.

Apart from the Canny application, we have also parallelized *fluidanimate* application which is a part of the PARSEC Benchmark [23]. This benchmark includes three versions of the application: a serial version, a parallel version which uses POSIX Thread API and another parallel version which uses Intel Thread Building Blocks. Another parallel version using xpu has been developed in [1] [115]. We developed a new version using the *MCProf*-xpu methodology. In the original xpu-based *fluidanimate* version, all the five processing stages were parallelized. However, the *MCProf* analysis report have shown that some of these stages are not hot-spots which are not worth parallelization. Parallelizing these stages introduces a communication overhead which affects the overall performance. The granularity adjustment using *MCProf* showed that splitting the `computeForces` processing stage in three sub-stages clearly isolate the computationally intensive regions which should be parallelized.

The xpu `parallel_for` skeleton has been used to express the thread-level data parallelism exposed by different processing stages. The later skeleton provides a scalable data partitioning and uses a cache-aware scheduling policy which promote data reuse and improve spatial and temporal data locality. This scheduling techniques appeared to be particularly beneficial in this study case since each fluid cell is processed on the processor core. Furthermore, we replaced the fluid cells arrays by xpu *vec3f* arrays to take advantage of SSE vectorization. In the reference sequential code, these fluid cells are expressed using regular float arrays.

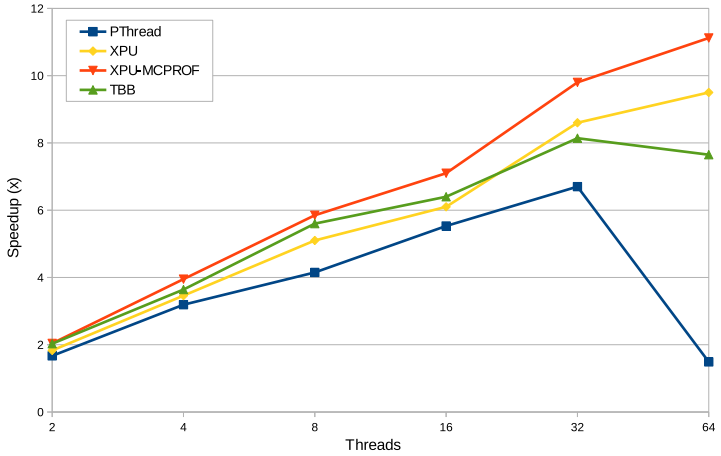

Figure 4.6: Performance comparison of the PThread, TBB, xpu only and xpu-*MCProf* versions of the PARSEC Fluidanimate application.

Figure 4.6 shows the achieved performance by the four versions. We observe that using *MCProf* profiling information improved the performances of the original xpu-only version. We note that the performances of the PThread version suffers from significant degradation when more than two processors (32 threads) on the AMD platform (64 cores/4 processors) are used, our investigation have shown that this degradation is caused by the use of barriers which results to an expensive many-to-many communication which affect the performances especially when the four processors are used. The xpu version uses a synchronization mechanism that follows a less-expensive one-to-many communication pattern.

4.5. MCPROF-XPU Parallelizing Framework

This section briefly presents the *MCProf*-xpu parallelizing framework. Figure 4.7 presents an overview of the framework.

4.5.1. ROSE Compiler

ROSE [118] is an open-source, object-oriented compiler infrastructure facilitating ease of building tools for analysis and transformation of programs in various languages including C/C++. We have used ROSE for the following tasks.

1. Inserting markers in the source-code to mark loop boundaries.

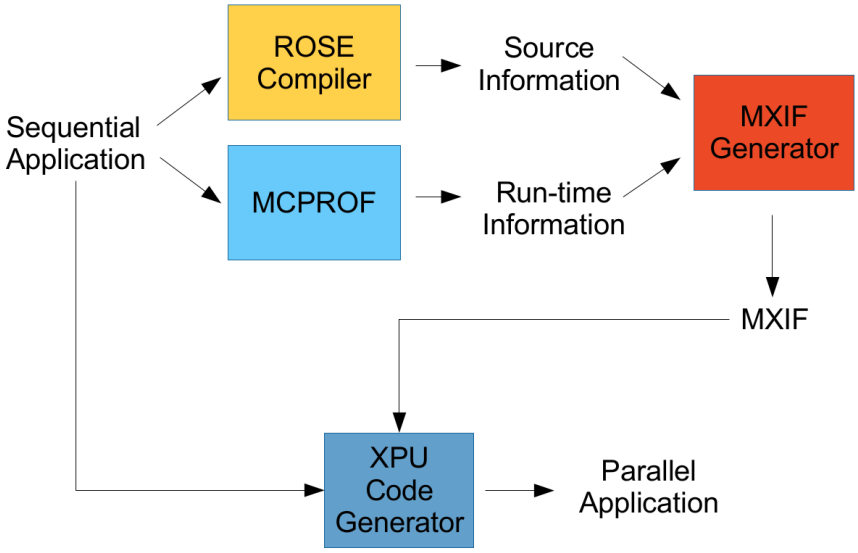

Figure 4.7: *MCProf*-xpu parallelizing framework.

2. Outlining loops to consider them as separate functions.
3. Generating source location information like starting and ending position of loops, variable names etc.

4.5.2. MCPROF Extensions

For this work, we extended *MCProf* to generate the following information:

- Runtime callgraph to show the loop nests as well in the callgraph. This is important for function splitting at the coarser granularity of loop nests.
- Coarse-grained dependence information, for instance, dependence between functions and loop nests.
- Fine-grained dependence information, for instance, dependence among loop iterations.

4.5.3. MXIF Generator

This block takes static and dynamic information generated by various tools to generate parallel application. In order to help programmers easily apply the parallelization manually, or help a tool automatically generate the parallel application,

we generate a representation of the parallel application in *MCProf*-xpu Interchange Format (MXIF).

4.5.4. Case-study

The focus of this case-study is to show various intermediate steps taken by the parallelizing framework to parallelize a sequential application presented in Listing (4.5) and show the information generated by various tools in the framework. Though this is a simple application, it contains various forms of parallelisms:

Coarse-grained: Function calls at Lines 28-29 can be executed in parallel.

Coarse-grained: Both loops in function *addsub* can be executed in parallel to each other.

Coarse-grained: Both loops in function *muldiv* can be executed in parallel to each other.

Fine-grained: Iterations of all the *for* loops can be executed in parallel.

```
1  void addsub(float* sum, float* diff, float* in1, float* in2, int N)
2  {
3      for (int i=0; i<N; i++)
4          sum[i] = in1[i] + in2[i];
5
6      for (i=0; i<N; i++)
7          diff[i] = in1[i] - in2[i];
8  }
9
10 void muldiv(float* prod, float* qout, float* in1, float* in2, int N)
11 {
12     for (int i=0; i<N; i++)
13         prod[i] = in1[i] * in2[i];
14
15     for (int i=0; i<N; i++)
16         qout[i] = in1[i] / in2[i];
17 }
18
19 int main(int argc, char **argv)
20 {
21     int i, N = 5000;
22     /* allocation of a,b,c,d,e and f */
23
24     for (i=0; i<N; i++){
25         e[i] = i+1.7;
26         f[i] = i+1.7;
27     }
28     addsub(a,b,e,f,N);
29     muldiv(c,d,e,f,N);
30
31     /* de-allocation of a,b,c,d,e and f */
32 }
```

Listing 4.5: Example Sequential Application.

```
sequential main_0_1
    parallel_for main_15_2_pf
        task main_15_2
    parallel dummy_par_3
        parallel addsub_23_3
            parallel_for addsub_7_4_pf
                task addsub_7_4
            parallel_for addsub_9_5_pf
                task addsub_9_5
        parallel muldiv_24_6
            parallel_for muldiv_11_7_pf
                task muldiv_11_7
        parallel_for muldiv_13_8_pf
            task muldiv_13_8
```

Figure 4.8: Simplified representation of parallel application.


```
( task_graph dummy_par_3
    ( parallel
        ( task_graph addsub_23_3)
        ( task_graph muldiv_24_6)
    )
)
( task_graph addsub_23_3
    ( parallel
        ( task_graph addsub_7_4_pf)
        ( task_graph addsub_9_5_pf)
    )
)
( task_graph addsub_7_4_pf
    ( parallel_for
        ( task addsub_7_4)
    )
)
( task addsub_7_4
    ( mapping_of
        ( chunk_of addsub_7
            ( function addsub
                ( file "test.c" )
                ( lines 5 22 )
            )
            ( lines 9 13 )
            ( input sum " float * " )
            ( input in1 " float * " )
            ( input in2 " float * " )
            ( input N " int " )
            ( input i " int " )
            ( output NULL )
        )
    )
    ( call_file "test.c" )
    ( call_line 9 )
)
```

Listing 4.6: Part of the Generated MXIF.

Figure 4.8 shows the result of parallelization in simplified form. It can be seen from this figure that the framework is able to extract the available parallelism in the application. Listing (4.6) shows the generated mxif.

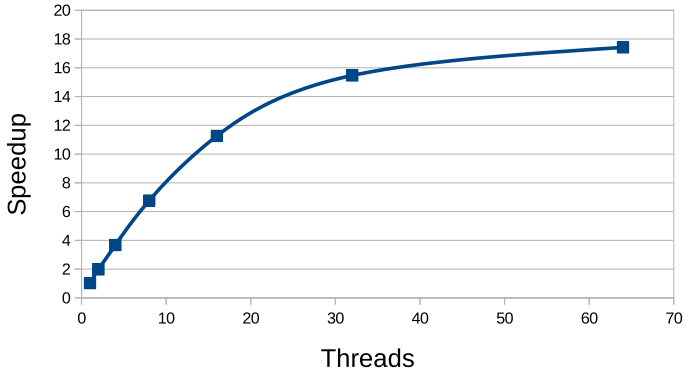

Figure 4.9: Speedup results on Machine 1.

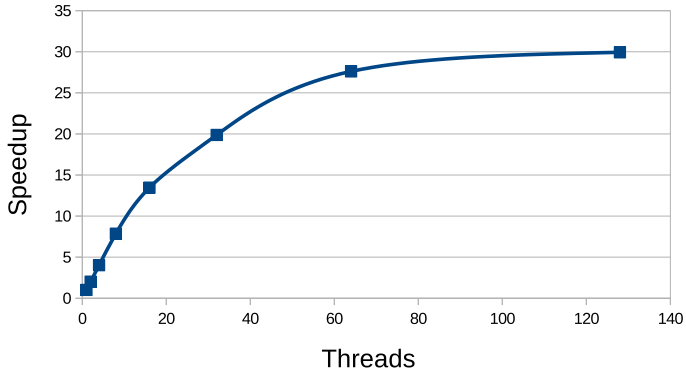

Figure 4.10: Speedup results on Machine 2.

In order to check the scalability of the parallel application, we have executed it on two machines. Machine 1 has 40 core (20 core 2 way hyper threaded) on Intel(R) Xeon(R) CPU E5-2670 v2 running at 2.50 GHz having 96 GB of main memory. Machine 2 has 64 core (16 core 4 way hyper threaded) IBM Power 7 v2.3 running at 3.3 GHz containing 164 GB main memory. Figure 4.9 and Figure 4.10 depict the speedup results for both the machines showing the scalability upto 64 and 128 threads respectively.

4.6. Conclusion

With the emergence of multicore processor architectures, we can no longer avoid parallelizing applications and making parallelizing compilers more efficient is an im-

portant objective. In this chapter, we presented the integrated use of two main tools which are very complementary in their functionality. *MCProf* provides such a detailed profile which can then be used by xpu, a parallel middle layer and programming approach which provides minimally invasive code changes to express and exploit in a natural way the available parallelism in an application. Not only does the combined approach provide better performance it also reduces substantially the overall time needed to parallelize sequential applications. Later on we also presented a framework to automate the approach. We extended *MCProf* to generate required extra run-time information, augmented it with static information generated by ROSE compiler to generate parallel representation of the application in mxif. This mxif is then used to generate the parallel code using xpu. We demonstrated through a case-study that our framework is able to extract various forms of fine and coarse grained parallelism.

Note. The contents of this chapter are based on the following articles:

[1] Imran Ashraf, Koen Bertels, Nader Khammassi, and Jean-Christophe Le Lann. **Communication-aware Parallelization Strategies for High Performance Applications**. In *Proceedings of IEEE Computer Society Annual Symposium on VLSI*, Montpellier, France, July 2015.

5

Data-communication Optimization for Accelerator-based Platforms

Accelerators based platforms are being utilized to satisfy the growing demand of processing in various application domains. However, programming these accelerators poses specific challenges regarding their programmability. One of the main challenges deals with assignment of data-structures to the available memory hierarchy. Second main challenge is the data-communication between the accelerator and the main processor.

In this chapter we address these challenges by utilizing the detailed memory access and data-communication profile of an application. In Chapter 3 we presented a case-study where the information generated by MCProf *was utilized to perform communication-aware mapping of a sequential application on a platform using GPU as an accelerator. In this chapter, we present two case-studies where the software and hardware based optimizations are performed by using the information generated by* MCProf *for platforms utilizing FPGA as accelerator.*

The third case-study details the utilization of data-communication profile of applications to perform data communication-aware evaluation of partitioning solutions. Results are provided for real applications as well as well-known benchmarks from various domains.

5.1. Case Study 1: Software-based Optimizations

In this case-study, we present a detailed discussion of a use case involving Kanade-Lucas-Tomasi Feature Tracker (KLT) application [103]. This application detects interesting features in a frame and tracks those features in the subsequent frames. We have used version 1.3.4, which is the latest version of KLT [104]. This *C* implementation has 102 functions in 17 source files. The focus of this case study is on the utilization of information provided by *MCProf* to map the application onto the *Molen* heterogeneous reconfigurable platform.

5.1.1. Research Context and Experimental Setup

The work presented in this case-study, although not restricted to any specific architecture, has been developed in the context of the *Molen* [52] polymorphic processor. The *Molen* architecture is based on the shared memory, processor, co-processor architectural paradigm [119]. It couples a General-Purpose Processor (GPP) and one or more Custom Computing Units (CCUs). Each CCU has its own set of registers and local memory for processing. The GPP controls the execution and (re)configuration of the CCUs. We utilized *MCProf* to provide a comprehensive overview of the memory access behavior of an application.

All the experiments were performed on two different platforms. The general profiling of the KLT application with *gprof* was done on an Intel 32-bit Core2 Duo E8500 @3.16GHz with 4GB of RAM, running the Linux kernel v2.6.34.10-0.6-pae. The application source code was compiled with *gcc* v4.5.0 with level two optimizations and without function inlining. The target platform is the *Molen* heterogeneous reconfigurable platform on Xilinx ML510, Virtex5 FX 130T with 2 MB Block RAM (BRAM) FPGA board. A PowerPC 440 @400 MHz with 512 MB DRAM, is used as a GPP, and CCUs are implemented as HW modules on FPGA. 30 K slices are available for (re)configuration and there can be a maximum of 5 RUs on the FPGA, where each CCU has 256 KB of local Memory. A number of design choices can be made in mapping applications onto *Molen*, which are guided by the information provided by the *MCProf*.

5.1.2. Mapping Steps

Table 5.1 shows the flat profile of the KLT application generated by *gprof* on the Intel x86 architecture. For this run, 10 frames have been used for feature tracking. The frame size has been chosen as 80×60 to be able to satisfy the memory requirement of the platform. It can be seen from this profile that we can map the top three kernels, namely *interpolate*, *convolveImageHoriz* and *convolveImageVert* on each of CCUs in the platform. The combined execution time of these three kernels is 0.805 p.u (80.5%). Using Amdahl's law, the theoretical application speedup,

Table 5.1: *gprof* flat profile for the *KLT* application on the Intel x86 architecture.

Kernel	%time	self sec	calls	self ms/call	total ms/call
interpolate	48.5	0.97	26.26M	0.00	0.00
convolveImageHoriz	16.0	0.32	183	1.75	1.75
convolveImageVert	16.0	0.32	183	1.75	1.75
KLTSelectGoodFeat.	6.0	0.12	1	120.0	141.14
computeGradientSum	5.0	0.10	17249	0.01	0.04
computeIntensityDiff.	2.5	0.05	23871	0.00	4.02

assuming an unlimited speedup for the kernel(s) in question, can be calculated as follows:

$$\lim_{p\to\infty} \frac{p}{1-f(p-1)} = \frac{1}{f} = \frac{1}{1-s} = \frac{1}{1-0.805} = 5.13, \tag{5.1}$$

where p is the speedup factor of the accelerated part, f is the percentage contribution of the sequential part, and s is the original percentile contribution of the accelerated part.

The mapping of the aforementioned kernels will result in performance improvements, but this performance can be improved by reducing the communication among all the computing elements. *gprof* and other traditional profilers do not provide information about the data communication in an application. In simple applications, it may be easy to analyze communication among various functions. However, in a complex application as in this case study (102 functions), it can be a tedious and time consuming task to manually understand the intensity of the communication, the addresses, and the amount of unique data involved in each communication. In short, an automatic tool like *MCProf* can provide this information and guide us in the mapping process based on the communication of functions with the top contributing kernels. We describe the mapping process below.

Step 1: In the original application, *gprof* shows that the $self$ contribution of *interpolate* per call is quite low (0.00 in Table 5.1). Mapping *interpolate* as it is onto a CCU will result in performance degradation, due to the large number of calls (about 24.2M) to this CCU. This is because the overhead of each call to CCU will be more than the execution time of this function. So, we modified *interpolate* to process a complete frame per call, resulting in reduced number of total calls.

Step 2: The complete communication graph generated by *MCProf* is quite complex due to the large number of functions in this application. So, a reduced graph of the top contributing kernels (dark Grey ovals) and the functions communicating with these kernels is shown in Figure 5.1. The objects allocated inside the application are represented by rectangles. The amount of data communication is shown in bytes. Furthermore, the intensity of the communication is indicated by the color of

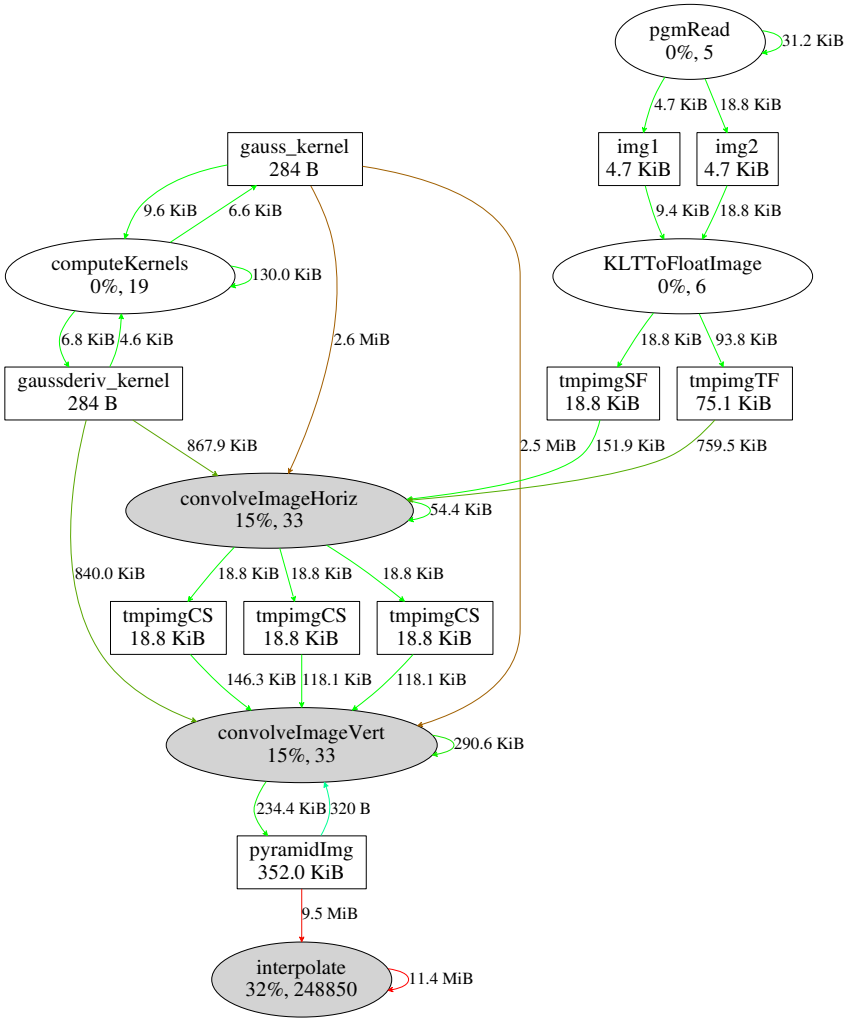

Figure 5.1: Communication graph generated by *MCProf* for the original KLT application.

the links in the descending order of red, brown, dark green, green and blue.

Figure 5.1 depicts that the image is read by *pgmread* and this image data is fed to *KLTToFloatImage*. The bytes transferred by *KLTToFloatImage* to *convolveImage-Horiz* are roughly 4 times higher than the image data input to *KLTToFloatImage*, which shows some kind of expansion being performed here. If we look into the code, it becomes clear that the image data is converted from $char$ to $float$ data type. As *convolveImageHoriz* is one of the top contributing kernels, to further re-

duce the communication, we can transfer the *char* data to *convolveImageHoriz* and cast it inside this function to reduce the external communication.

Step 3: The *convolveImageHoriz* and *convolveImageVert* are also communicating heavily with each other through *tmpimgCS* objects. If we merge these two functions together as a single *convolveImage* function, this communication will be performed locally. *convolveImageHoriz* and *convolveImageVert* are also consuming a lot of data from *gauss_kernel* and *gaussderiv_kernel* objects. The size of these objects is not big and they are written fewer times and read multiple times. Therefore merging *computeKernel* with convolution functions will also make this communication local.

Step 4: *interpolate* is communicating heavily with *convolveImageVert*. In this case, we cannot merge these two functions together as this will require *pyramidImg* to be allocated on CCU. The size of *pyramidImg* is 352 KB which is greater than the the memory available per CCU (256 KB). Finally, we have two modified kernels performing the functionality of *convolveImage* and *interpolateImg*, which can be mapped onto two CCUs.

Table 5.2: Results of various intermediate implementation steps performed in mapping.

Entry	Implementation	Kernel	SW Time(μsec)		HWTime (μsec)		Speedup	
			Kernel	Application	Kernel	Application	Kernel	Application
1	Original SW	interpolate	4.75	12154566	NA	NA	NA	NA
2	Original HW	interpolateImg	1310	11786160	831	7110129	1.58	1.71
		convolveImageHoriz	16132		4007		4.03	
		convolveImageVert	16491		4016		4.11	
3	Modified HW 1	convolveImageHoriz	16689	11654476	4013	7015287	4.16	1.73
4	Modified HW 2	ConvolveImg	32125	11154476	6683	6797151	4.81	1.79
5	Final		2747	11025172	957	5429859	2.87	2.24

5.1.3. Experimental Results

Table 5.2 shows the experimental results of the intermediate steps performed during the process of mapping the KLT application onto the *Molen* platform. The third column contains the name of kernels under discussion in the corresponding step, as shown in communication graph in Figure 5.1.

The first row is the original software implementation which is provided here for comparison. It does not involve a HW implementation, hence, mentioned Not Applicable (**NA**) in HW execution times. The second row is the HW implementation based on the *gprof* information, giving a total speedup of 1.71. The third row corresponds to Step 2 in Section 5.3.3, where *KLTToFloatImage* was merged with the *convolveImageHoriz* to reduce the data communication. It can be seen that we have achieved a speedup of 1.73 by this communication reduction. Furthermore, compiler was also able to optimize the code efficiently when most of the functionality was placed in a single function.

The fourth entry corresponds to Step 3 where we achieved an overall speedup of 1.79×, by using merged *convolveImage* and reducing external expensive communication. Row 5 corresponds to the final result obtained by applying all the opti-

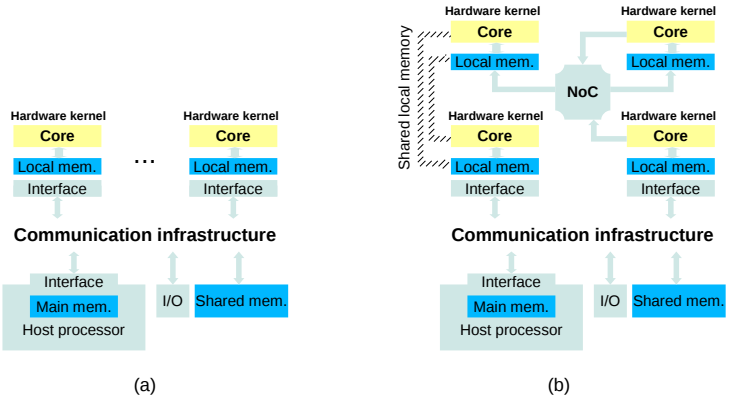

Figure 5.2: (a) The generic hardware accelerator architecture. (b) The generic hardware accelerator system with hybrid interconnect.

5

mizations mentioned in Section 5.3.3 showing that the overall speedup obtained is 2.24×.

5.2. Case Study 2: Hardware-based Optimizations

In this section we discuss the optimizations which can be performed by utilizing the data-communication profile of *MCProf* to generate an efficient hybrid interconnect [120]. These are basically hardware implementations targeted to specific platforms to speed up the software-hardware or hardware-hardware intercommunication custom to specific application. We use Canny Edge Detection application [121] as a case-study. The detailed profile of the data communication patterns is used to define a hybrid interconnect to alleviate the data communication bottleneck and improve the system performance. Based on the detailed profile, a kernel knows the consumers of its data.

In a generic hardware accelerator system the *communication infrastructure* is a predefined system backbone upon which data is transferred between the host and the kernels as well as among the kernels. The communication infrastructure is different from system to system. It can be a bus, a Network on Chip (NoC), shared memory or a crossbar. Figure 5.2(a) depicts an example of a generic accelerator system.

Hybrid interconnect refers to the communication infrastructure used considering the data communication among the hardware accelerator kernels to speed up system performance. The infrastructure consists of a NoC and shared local memory. The accelerator system using our approach includes both the original communication infrastructure to exchange the parameters and the data between the host and the kernels and the hybrid interconnect for inter-kernel data transfer. Figure 5.2(b)

illustrates the generic accelerator system with our hybrid interconnect.

5.2.1. Design Choices

Depending on the communication patterns, there are different ways to connect a kernel and its local memory to the NoC (to communicate with other kernels) and the communication infrastructure (to communicate with the host). As mentioned earlier, *MCProf* is utilize to generate the communication profile to make these design choices.

A simpler solution is to map all the kernels and all their local memories to both the NoC and the system communication infrastructure. However, this mapping solution requires the maximum number of routers as well as network adapters. To reduce the NoC latency, a kernel and its communicating local memories should be mapped to the NoC routers in such a way that the distance of these routers is shortest. For instance, if a kernel is mapped to a router at the coordination (x, y) then the ideal location for the local memory to which it communicates is either $(x-1, y)$, $(x+1, y)$, $(x, y-1)$, or $(x, y+1)$.

When NoC is used, four routers and four adapters (two for kernels and two for their local memories). Keeping in mind that the hardware resources usage for those routers and adapters is $6\times$ larger than the hardware resources usage for the shared local memory solution (in the Xilinx xc5vfx130t FPGA device, the four routers and adapters requires 1221 Look-up Table (LUT) while it is 201 LUT for the crossbar; more details in [122]).

An efficient mapping method reduces the hardware resource usage while keeping the communication time minimal. The shared local memory is another option which connects the local memories of two kernels. Shared local memory solution requires less hardware resources compared to the NoC. Therefore, the simpler solution is customized to connect the kernels which are communicating with each other by shared memory solution. Furthermore, the routers of the kernels which are not communicating the host can be removed to reduce resource usage. When implemented on FPGAs, most accelerator systems use BRAM as the local memory. BRAM in modern FPGA usually have only two ports while they may be accessed by three different components (the two communicating kernels and the host). Therefore, based on the communication topology of each specific application, we define a connection topology of the kernels and the local memories to the NoC so that the number of routers and adapters required is as low as possible.

Figure 5.3 depicts the data communication profiling graph for the Canny application with a 1024×1024 pixels input image. This figure shows that *gaussian_smooth* takes an image as input from the host machine. So this kernel needs a connection to the host. The output produced by *gaussian_smooth* is consumed by *derrivative_x_y* hence a shared memory connection can be used to have a direct connection be-

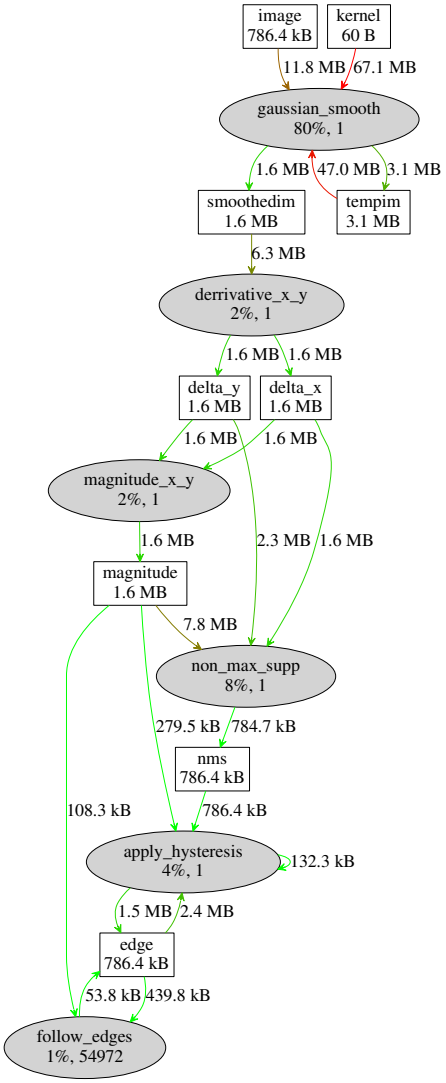

Figure 5.3: Communication graph for the canny application generated by *MCProf*; Functions (ovals), compute-intensive functions (Grey ovals) and the objects(rectangles) involved in the communication are also shown.

tween these two kernels.

derrivative_x_y communicates with three functions (*gaussian_smooth*, *magnitude_x_y* and *non_max_supp*). As BRAMs have two ports, we cannot use shared memory solution in this case. Hence, these kernels are configured to use NoC.

By applying these optimizations, an application speed up of 1.83× was achieved compared to baseline. This also resulted in about 50% reduction in energy consumption. The detailed experimental results can be found in [123].

5.3. Case-study 03: Evaluation Methodology for Data Communication-aware Application Partitioning

In this section, we propose a data communication-aware methodology for evaluating the quality of application partitions as well as partitioning algorithms. We also present an open source tool which implements the proposed methodology. Moreover, we evaluate several heuristic algorithms to further substantiate the applicability of the proposed methodology and the utilization of the developed tool. The modularity of the tool allows easy integration of new partitioning algorithms as well as the addition of benchmark applications. The applications are used as test inputs for the sake of comparisons in terms of relative and absolute quality measurements of the partitioning solutions.

The criteria that drive application partitioning include, among others, the nature of computations, execution times on Processing Elements (PEs), memory requirements, area available on PEs, etc. Due to the huge size of the search space, finding an optimal *partition* is an NP-hard problem [124]. Therefore, heuristic algorithms are commonly utilized to find solutions in short time. Nevertheless, the *partition* found by a heuristic may or may not be close to the optimal solution. Hence, it is very important to be able to evaluate the quality of the solution(s) found by a heuristic method and to make a robust comparison with the solutions found by other existing or future partitioning algorithms.

Due to a large variety of architectures and the lack of proper benchmarks, it is hard to reproduce experimental results, for fair comparison, on the target platforms [125]. Furthermore, the complexity furthers in case of reconfigurable architectures as the application development process involves building and synthesizing hardware blocks. Hence, a proper methodology is required to quickly evaluate the quality of the partitioning algorithms.

An interesting observation is that only a small number of functions primarily contribute to the overall application execution time. Similarly, a limited number of functions are responsible for the main inter-task data communication. By considering these two observations, we can potentially reduce the huge design space exploration effort. Thus, the comparison with optimal partitions found by an exhaustive search becomes feasible, in spite of the general intractability of the application partitioning problem. In the following, we present the proposed methodology and the design of PET tool to evaluate the quality of partitions and compare various partitioning algorithms.

5.3.1. The Methodology

The proposed evaluation methodology is a 4-step process as described below.

Step 1: Formulation of the cost function. Partitions can be evaluated by calculating their costs using a cost function. A cost function takes a *partition* as input and assesses its quality. Various factors can contribute to the quality of a *partition*, for instance, how well the clusters are balanced, inter-cluster data communication, etc. In this step, the factors of interest are determined to formulate the cost function.

Step 2: Implementation of the partitioning algorithm. The partitioning algorithm to be evaluated is implemented in this step. The algorithm takes the application as input and outputs a *partition* of the application.

Step 3: Specification of the input set. In this step, the applications that are input to the partitioning algorithm are specified. These applications are represented as graphs, where the values of graph vertices represent the characteristics of applications, for instance, execution contribution, memory requirements, area estimates, etc. The edges in these graphs represent the data communication between functions. The specification of an input application boils down to selecting numbers for vertices and edges, which can be obtained randomly or by profiling real applications.

Step 4: Evaluation and comparison. In this step, the cost of the *partition* found by the *partitioning algorithm* is evaluated by the cost function. The evaluated cost can be used to perform comparisons and to rank the *partition* based upon its quality. This ranking can be:

Absolute — where the comparison is performed with the optimal *partition* found by an exact solution or an exhaustive search (only for applications with small number of functions).

Relative — when optimal *partitions* cannot be found, thus the cost is compared in relation to the costs of *partitions* found by other algorithms. In the next section, we present the details of the tool that implements the proposed methodology.

5.3.2. PET Implementation

We developed the Partition Evaluation Tool (PET[1]), to evaluate the quality of partitions and compare various partitioning algorithms. This tool is implemented in C++ in a flexible way to allow the partitioning strategies to be evaluated easily. We define the relevant concepts used in the PET discussion as follows. Figure 5.4 shows the block diagram of the PET tool.

[1]Sources available at http://imranashraf.github.com/PET

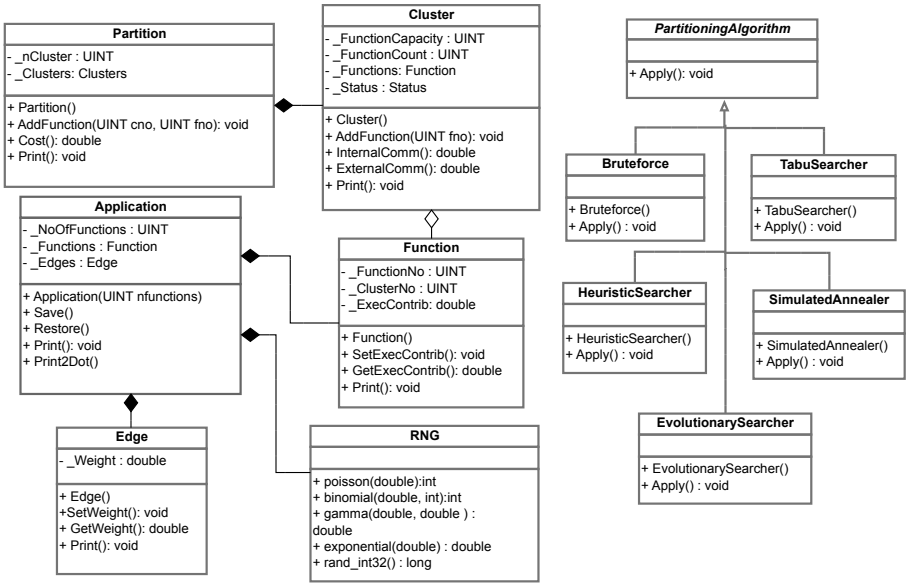

Figure 5.4: Bock diagram of the PET tool.

Application class models the concept of the application which needs to be partitioned by the partitioning algorithm. An application has two important members, namely functions and edges.
Function represents the subroutine in an application, which performs a certain task of the application. Each function has an execution contribution depending upon the task assigned to it.
Edge is a directed link denoting the communication between a pair of functions in an application. The amount of data that is communicated is represented by the weight of the edge.

Cluster represents the collection of functions which can be mapped onto a single core/PE of the target platform. The number of functions in a cluster is controlled by the capacity of that cluster. As an example, in reconfigurable systems, this capacity may represent the maximum number of slices reserved for a single PE. When a cluster is full, its status is changed from *UnFinished* to *Finished*.
Partition refers to the union of clusters, where each cluster contains various functions. A viable partition holds the semantics of the application with the combination of functions in clusters that can be mapped onto various cores/PEs in a multicore platform. The cost of a *partition* can be evaluated by a call to the *Cost()* method of this class.
PartitioningAlgorithm is an abstract class that can be used to implement the algorithm(s) used to perform partitioning. This can be achieved by implementing the *Apply()* virtual function of the *PartitioningAlgorithm* class for some algorithm

of interest. *Bruteforce* is a derived class of *PartitioningAlgorithm*, which is used to find all the partitions of an application by an exhaustive search. This is required to compare the results of a heuristic algorithm under test with the optimal solution found by the exhaustive search for small number of functions. Similarly, *HeuristicSearcher*, *SimulatedAnnealer*, *EvolutionarySearcher* and *TabuSearcher* are also derived classes of *PartitioningAlgorithm* and implement Heuristic Search (example algorithm used for comparison/evaluation in this work), Simulated Annealing, Evolutionary search and Tabu search, respectively, to find an optimal partition.

RNG is an efficient random number generator class, which is based on the Ziggurat Method [126]. It generates high-quality random numbers that are able to pass all the commonly-used tests for randomness [126]. A number of functions are available in RNG class, which can generate random numbers following a number of well-known distributions. This class has been used to generate random graphs and cost values in PET.

In the next section, we detail the application of the proposed methodology and the utilization of PET tool to evaluate the output *partitions* and, hence, the *partitioning algorithm*.

5.3.3. Evaluation of Multi-objective Task Clustering Algorithm

In this section, we present the evaluation of a multi-objective task clustering algorithm [72], to illustrate our partition evaluation methodology and the utilization of PET in this regard. This algorithm initiates task clustering at the function-level based on dynamic profiling information. This includes the data communication among functions and the information about their execution time. The overall goal of the heuristic algorithm is to get a well-balanced cluster containing tightly interconnected functions. The steps of the evaluation methodology are described below:

Step 1: Formulation of the Cost Function. We assume an application containing n functions to be partitioned into k clusters. These functions can be specified as vertices $v_i, 1 \leq i \leq n$ in a graph. The execution cost EC_{v_i} of a vertex v_i, is the cost of executing this function on a PE when it is mapped as a part of a cluster. The execution cost of a cluster C_i is defined as: $EC_{C_i} = \sum_{v_j \in C_i} EC_{v_j}, 1 \leq i \leq k, 1 \leq j \leq n.$

Vertices v_i and v_j have an edge e_{ij} connecting them, with the weight w_{ij} representing the amount of the data that is communicated. The set of all the edges in a graph makes an edge set E of the graph. The set of edges which cross the boundary of a cluster C_i form the set of external edges $Eext_{C_i}$ of this cluster with respect to the rest of the partition. This edge set represents the communication cost of the cluster C_i with respect to the other clusters. Furthermore, the set of edges internal to a cluster C_i form the internal edge set $Eint_{C_i}$, as it represents the communication among functions inside C_i. For this case study, we consider the following two metrics to formulate the total cost of a *partition*:

1. **Balancing Penalty (BP)** accounts for the load balancing among clusters in a partition. It basically depends upon the distance between the execution cost EC_{C_i} of cluster C_i and the average execution cost of all the clusters in a partition $BP_{C_i} = \left| \frac{\sum_{j=1}^{k} EC_{C_j}}{k} - EC_{C_i} \right|, 1 \leq i \leq k$. Balancing penalty BP_P of a partition P is defined as $BP_P = \sum_{i=1}^{k} BP_{C_i}$.

2. **Communication Cost (CC)** is the cost associated with external communication of a cluster with respect to the other clusters in the partition. Communication cost CC_{C_i} of a cluster C_i is $CC_{C_i} = \sum_{e_{ij} \in Eext_{C_i}} w_{ij}$. The communication cost CC_P of a partition P is defined as $CC_P = \sum_{i=1}^{k} CC_{C_i}$.

The total cost TC_P of a partition P is then defined as:

$$TC = \alpha BP_P + \beta CC_P;\ 0 \leq \alpha \leq 1, 0 \leq \beta \leq 1\ and\ \alpha + \beta = 1, \tag{5.2}$$

where α and β are relative weights associated with BP and CC metrics, respectively. These weights can be selected by the designer based on the platform at hand, to stress one metric relative to the other. For instance, if workload balancing is important then, one can select a higher value of α than β. On the other hand,when communication is expensive, β can get a higher value than α.

It is worth mentioning here that this is just one way of specifying the cost function as the weighted sum of the objectives of interest. Our approach is similar to the one discussed in [127], where the difference of values of each objective is explained in detail. On the same lines, we have assigned the values to each objective as their percentage contribution in the total execution and communication cost for BP and CC, respectively, instead of using their actual values.

Step 2: Implementation of the Partitioning Algorithm. We implemented the clustering algorithm as a derived class of the abstract *PartitioningAlgorithm* class of PET. The *Apply()* function implements the functionality of this heuristic algorithm on an input application. The result is a partition of the input application containing the clusters with various functions.

Step 3: Specification of the Input Set. We generated a number of synthetic application graphs to be used as test inputs. Due to space limitation, we will only discuss the real applications. We used applications from Standford Parallel Applications for Shared Memory Architectures (SPLASH-2) Benchmark Suite [128]. We used the *MCProf* to extract the percentage contribution of functions and to get the communication graphs. It should be noted that getting the profiling data by using any other profilers will not change the methodology or the evaluation process.

Step 4: Evaluation and Comparison. We performed the absolute ranking by finding the optimal *partition* determined by the exhaustive search. We compared

Table 5.3: Specifications of the real benchmark applications used in the evaluation

#	Applic.	No. of Functions	% Exec. Covered	% Comm. Covered
1	Barnes	38	99.19	99.45
2	Fmm	70	91.86	84.12
3	Raytrace	85	75.32	77.55
4	Clustalw	89	96.06	86.17
5	KLT	103	99.85	98.73

the cost of *partition* found by the heuristic algorithm against the best possible partition, using *bruteforce()* class to generate all the possible partitions for the generated applications. The cost of each *partition* is evaluated by the *Cost()* method of the *Partition* class. Apart from the absolute ranking, we also performed relative ranking of the *partitioning algorithm* against the best solutions found by simulated annealing, evolutionary and tabu search. The detailed results of these evaluations and comparisons are provided in the following section.

5

5.3.4. Experimental Results

In this section, we provide the results of the experiments performed to evaluate the cost of heuristic algorithm and the cost of *partitions* found by the exhaustive search. We performed these experiments on a $2.66\ GHz$ *Intel(R) Core(TM)2 Quad* CPU, with $4\ GB$ RAM. We used the value of 0.5 for α and β to give equal weights to both cost metrics. Values given to these relative weights should be based on the target platform and care must be taken while revising them.

In order to present the evaluation for real benchmark applications, we selected three applications from SPLASH-2 benchmark suite, namely *Barnes*, *Fmm* and *Raytrace*. We also selected *Clustalw* [129] sequence alignment application and *KLT* [103] feature tracking application for the evaluation. Figure 5.5 presents the comparison of the cost of the partition found by Heuristic Search (HS) with the costs of the solutions found by Simulated Annealing (SA), Tabu Search (TS) and Evolutionary Search (ES). It can be seen that HS performs close to other algorithms. For *Barnes* and *Fmm*, which are relatively smaller applications, SA performs better, but for the remaining three bigger applications, TS outperforms all the other algorithms.

An important point worth mentioning is that HS performs comparable to SA, TS and ES in terms of cost.However, HS takes very less time to find the solution compared to other algorithms. To give an idea, for the *KLT* application, HS took 815μs to find the solution, whereas, TS took $18m$.

In the above experiments, we performed evaluations relative to other algorithms. Absolute ranking can also be performed by comparing the costs against the cost of the optimal solution found by Brute Force (BF). Performing this exhaustive search to find the optimal *partition* of four clusters will require about 8 years on the computer

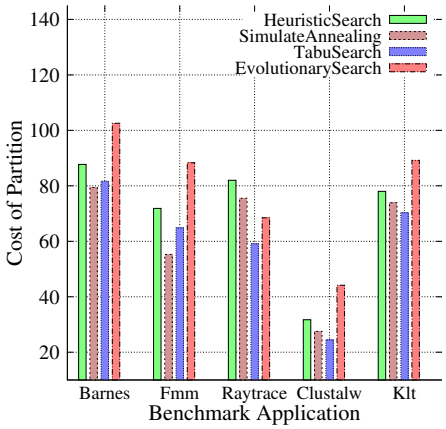

Figure 5.5: Comparison of cost of partition found by HS against the cost of partition found by SA, TS and ES for the real applications.

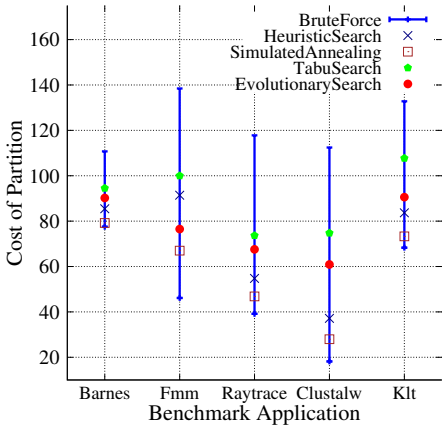

Figure 5.6: Absolute ranking of the cost of the partition found by HS, SA, TS and ES from the real applications.

used in these experiments for an application with 25 functions. The considered applications contain 38 to 103 functions, hence, performing the exhaustive search for the optimal *partition* is not practical. Thus, we limited the number of functions to a threshold value of 18, picking the top functions according to the execution time and data communication contribution. This feature is also supported by the tool, where we can specify this threshold value in terms of the number of functions for execution contribution and communication weights for filtering the top kernel functions in the application. Table 5.3 provides detailed specifications of the five applications. Column 3 provides the number of functions in each application. As it can be seen in Column 4 and 5, considering a subset of the top contributing functions covers most of the heavily executing and heavily communicating functions in these applications. The results of the absolute ranking are provided in Figure 5.6. The vertical lines show the range of the cost of solutions found by BF for each application. It can be seen that HS is able to find close-to-optimal solutions. The costs of solutions found by SA, TS and ES are also marked for comparison. Overall, SA outperforms all the other algorithms, as it is able to find close to optimal solutions.

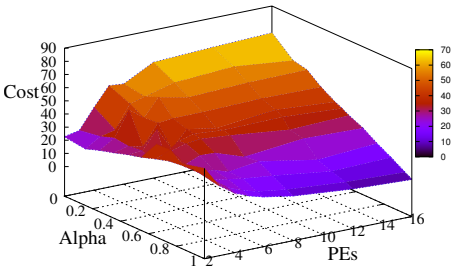

Figure 5.7: Effect of variation of α and No. of PEs on cost of the partitions found by tabu search for the *Raytrace* application from SPLASH-2 benchmark suite.

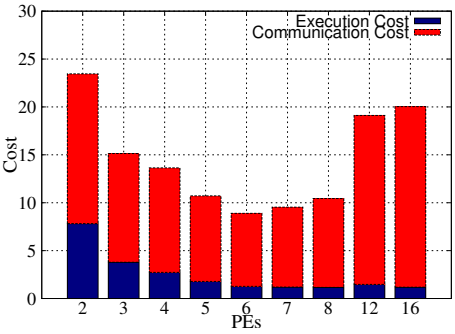

Figure 5.8: Variation of execution and communication cost of Clustalw application. Minimum cost values correspond to the optimal number of PEs (at $PEs = 6$) for the given α and β.

In the above experiments, we have kept α and β fixed. Figure 5.7 presents the variation of the costs of partitions found by Tabu Search algorithm for *Raytrace* application from SPLASH-2 benchmark suite. It can be seen in this figure that for lower values of α (higher values of β) increasing more PEs, increases cost because of the increased communication among PEs. Higher values of α (lower values of β) implies that the communication is not expensive on the platform at hand and adding more PEs will decrease the total cost of partition. Similar analysis can also be performed for various values of α and β and the number of PEs for other applications.

Another important result which can be obtained from this evaluation is the optimal number of PEs required for a given application for a given platform (values of α and β). Adding more PEs may reduce the execution time of an application by performing the job in parallel, however, the disadvantage is the increase in data communication among the PEs, which may kill the anticipated speedup. This analysis can especially be important for platforms where number of PEs can be reconfigured. For instance, in Molen architecture [52], the number of cores can vary from 2-5 with an upper limit imposed by the resources on FPGA. Even for non-reconfigurable platforms, where number of cores cannot be altered, the additional PEs can be switched-off to save the energy consumption.

In order to perform the analysis for optimal number of PEs, we have modified the equation to calculate the total cost of the *partition* to consider the effect of the addition of PEs as follows:

$$TC = \alpha(EC_s + EC_p/N) + \beta CC \; ; \; 0 \leq \alpha \leq 1, 0 \leq \beta \leq 1 \; and \; \alpha + \beta = 1, \qquad (5.3)$$

where EC_s is the execution cost of the serial region of the application, EC_p is the execution cost of the parallel region of the application and N is the number of PEs. In this simple equation, we have assumed that the cost of the parallel region of the application scales with the number of the available PEs. Furthermore, we did not consider the overhead of parallelization, etc, in this analysis, as the main objective

here is to show the effect of increasing the number of PEs on the execution and the communication costs of a *partition* for an application for certain values of α and β. Figure 5.8 provides a plot of the execution and the communication cost of the *Clustalw* application for a number of values of PEs. For this graph, we have given equal weights to α and β. It can be observed from this plot that increasing the number of PEs reduces the execution cost, but the interaction among PEs in the form of data communication increases resulting in an increase in the communication cost. The optimal number of PEs corresponds to the minimal cost values, which in this case happens when the number of PEs is equal to 6.

5.4. Conclusions

In Chapter3 we presented a case-study dealing with communication-aware mapping of a sequential application on a platform using GPU as an accelerator. In this chapter, we presented further use cases showing the utilization of the information generated by *MCProf* for platforms using FPGA as accelerator. In the first case-study we detailed the partitioning and mapping a feature tracking application on *Molen* platform.

In the second case-study we described how the information generated by *MCProf* can be utilized to customize the hardware to generate hybrid interconnect. The idea was to use the detailed communication profile of the application to generate custom interconnect specific to the communication demands of application.

In the third case-study, we utilized *MCProf* information to present a methodology to evaluate application *partitions* for multicore architectures. Functions are assumed as vertices in a graph and the communication among functions is represented by the weights of the edges connecting them. We presented PET— a flexible and modular partition evaluation tool— implementing the proposed methodology. We provided the detailed relative and absolute evaluations of a heuristic task clustering algorithm, as a case-study to prove the effectiveness of the methodology.

5

Note. The contents of this chapter are based on the following articles:

[1] Imran Ashraf, S. Arash Ostadzadeh, Roel Meeuws, and Koen Bertels. **Communication-Aware HW/SW Co-design for Heterogeneous Multi-core Platforms**. In *Proceedings of the 2012 Workshop on Dynamic Analysis*, WODA 2012, pages 36–41, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1455-8. doi: 10.1145/2338966.2336806.

[2] Cuong Pham-Quoc, Imran Ashraf, Zaid Al-Ars, and Koen Bertels. **Heterogeneous Hardware Accelerators with Hybrid Interconnect: an Automated Design Approach**. Ho Chi Minh City, Vietnam, November 2015.

[3] Imran Ashraf, S.A. Ostadzadeh, R.J. Meeuws, and Koen Bertels. **Evaluation Methodology for Data Communication-aware Application Partitioning**. In *Proceedings of 1st Workshop on Runtime and Operating Systems for the Many-core Era*, Aachen, Germany, August 2013.

6

Conclusion and Future Work

This chapter summarizes the overall contributions of this thesis and highlights some future research directions.

6.1. Conclusion

Chapter 1 "Introduction" briefly introduced the field of homogeneous and heterogeneous multi-core computing and presented current computing trends. To port an existing sequential application to multi-core platform, applications must be divided into smaller parts which are mapped to the available cores in the architecture. This is a critical task, as an improper partitioning and mapping may result in performance degradation. Tools can help programmers by performing all or most of the steps of the porting process. This chapter presented various research challenges in the field of application mapping to multicore architectures. The research questions focused in this dissertation were formulated, followed by the main contributions of this thesis.

Chapter 2 "Background and Related Work" surveyed the profilers for memory-access optimization and presented a classification of these profilers. Data-communication profilers which are a sub-class of memory-access optimization profilers, were the focus of this chapter. A detailed comparison of data-communication profilers was provided to highlight their strong and weak aspects. Finally, recommendations for improving existing data-communication profilers and/or designing the future ones were discussed.

Chapter 3 "*MCProf*: Memory and Communication Profiler" presented the design of *MCProf*, an efficient memory-access and data-communication profiler based on the study in the Chapter 2. It was shown that this tool profiles the application at various granularity levels with manageable overheads for realistic workloads. Experimental results showed that on the average, the proposed profiler has at least an order of magnitude less overhead as compared to the state-of-the-art data-communication profilers for a variety of benchmarks.

Chapter 4 "Profile Driven Application Parallelization" started by presenting a semi-automatic parallelization methodology based on *MCProf* to help programmers extract parallelism. xpu, a parallel programming library, was used to manually express the extracted parallelism. Not only does the combined approach provided better performance (up to $4\times$ higher than the existing commercial compilers), it also reduced substantially the overall time needed to parallelize sequential applications. Later on, a framework was presented which automated the whole process of application parallelization. We demonstrated through a case-study that our framework is able to extract various forms of fine and coarse grained parallelism available in the application.

Chapter 5 "Data-communication optimization for Accelerator-based Platforms" addressed the challenges and the solutions of mapping an application onto an accelerator-based platforms by utilizing the detailed memory access and data-communication profile of an application. We presented both the software and hardware based optimizations for platforms utilizing GPU and FPGA as accelerator. Results were provided for real applications as well as well-known benchmarks from various domains. In the case of GPU, we achieved up-to

$2.75\times$ higher speedup as compared to the GPU implementation where this communication is not optimized. For FPGA based platforms, *MCProf* driven software-based optimizations resulted in an overall speedup of $2.24\times$. Applying hardware-based optimizations for FPGA resulted in speed up of $1.83\times$ compared to the baseline. This also resulted in about 50% reduction in energy consumption. Finally, we also presented a case-study showing the utilization of the information generated by *MCProf* to perform data-communication aware evaluation of partitioning algorithm and partitioning solutions.

6.2. Future Research Directions

In this section, we suggest several recommendations to further the research on the challenges addressed in this dissertation.

- Directive based programming languages appear to be an easy way to offload the compute intensive part of an application to the accelerator. However, programmer has to analyze the memory access patterns to perform the memory assignment. Automatic mapping of data-structures to memory-spaces available in the hierarchy can be another area of future research.
- Both the bandwidth and latency considerations are important for optimized mapping of application. In certain cases, if the required bandwidth is satisfied but latency is high, then for each execution of the kernel, this high latency will become a bottleneck in performance. Therefore, bandwidth and latency both should be readily reported by the profiler.
- *MCProf* does not generate temporal information. Generation of temporal information while keeping the overheads manageable, can be another area of investigation. This information can be utilized to implement pipelining.
- Dynamic binary instrumentation has been used in our work to generate the detailed application profile. The recent architectures have better support for hardware performance counters. Profiling based on hardware performance counters have less overhead as compared to dynamic binary instrumentation. So these architectural facilities can be investigated to reduce the execution-time overhead of profiling.
- In order to help programmer with the mapping process, some kind of database management can be very interesting to store all the information represented by various profiling tools so that queries can be run to see the results rather than multiple passes of modification and execution. Example queries can be; give me the list of functions with X amount of communication, or area greater or less than Y, etc. In this way, all the information can be merged at one place and the output should not only be based per tool but it should be based per requirement/goal.

- Effect of various architectures on the generated application profile can also be another interesting study which can be performed in the future.
- In order to reduce the overheads further, parallel version of *MCProf* can also be implemented.
- Support for multi-threaded applications will also be interesting to optimize existing parallel applications.

Bibliography

[1] N. Khammassi, *High-level Structured Programming Models for Explicit and Automatic Parallelization on Multicore Architectures*, Theses, Université de Bretagne Sud (2014).

[2] M. Horowitz and W. Dally, *How Scaling Will change Processor Architecture,* in *ISSCC*, Vol. 1 (2004) pp. 132–133.

[3] Y. Taur, *CMOS Design near the Limit of Scaling,* IBM Journal of Research and Development **46**, 213 (2002).

[4] K. Olukotun, L. Hammond, and J. Laudon, *Chip Multiprocessor Architecture: Techniques to Improve Throughput and Latency,* Synthesis Lectures on Computer Architecture **2**, 1 (2007).

[5] AMD, *AMD A10-7850K APU,* http://www.amd.com/us/products/desktop/processors/a-series/Pages.

[6] Xilinx, *Zynq-7000 All Programmable SoC,* http://www.xilinx.com/products/silicon-devices/soc/zynq-7000.

[7] Texas Instruments, *OMAP3530 Application Processors,* http://www.ti.com/product/omap3530.

[8] *Hybrid Core Computer by Micron,* http://www.conveycomputer.com (2012).

[9] W. A. Wulf and S. A. McKee, *Hitting the Memory Wall: Implications of the Obvious,* SIGARCH Comput. Archit. News **23**, 20 (1995).

[10] G. Martin, *Overview of the MPSoC design challenge,* in *43rd ACM/IEEE DAC* (2006) pp. 274–279.

[11] S. Borkar and A. A. Chien, *The Future of Microprocessors,* Commun. ACM **54**, 67 (2011).

[12] J. Jeffers and J. Reinders, *Intel Xeon Phi Coprocessor High Performance Programming*, 1st ed. (Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2013).

[13] S. L. Graham, P. B. Kessler, and M. K. Mckusick, *Gprof: A Call Graph Execution Profiler,* SIGPLAN Not. **17**, 120 (1982).

[14] W. Cohen, *Multiple Architecture Characterization of the Build Process with OProfile,* (2003).

[15] *vTune by Intel,* http://software.intel.com/en-us/intel-vtune.

[16] N. Nethercote and J. Seward, *Valgrind: a Framework for Heavyweight Dynamic Binary Instrumentation,* SIGPLAN Not. **42**, 89 (2007).

[17] A. Srivastava and A. Eustace, *ATOM: a System for Building Customized Program Analysis Tools,* SIGPLAN Not. **29**, 196 (1994).

[18] J. Seward and N. Nethercote, *Using Valgrind to Detect Undefined Value Errors with Bit-precision,* in *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05 (USENIX Association, Berkeley, CA, USA, 2005) pp. 2–2.

[19] M. Martonosi, A. Gupta, and T. Anderson, *MemSpy: Analyzing Memory System Bottlenecks in Programs,* SIGMETRICS Perform. Eval. Rev. **20**, 1 (1992).

[20] *Zoom by Rotate Right,* http://www.rotateright.com/zoom.

[21] A. Lebeck and D. Wood, *Cache Profiling and the SPEC Benchmarks: a Case Study,* Computer **27**, 15 (1994).

[22] A. R. Lebeck and D. A. Wood, *Cache Profiling and the SPEC Benchmarks: A Case Study,* IEEE Computer **27**, 15 (1994).

[23] N. Barrow-Williams, C. Fensch, and S. Moore, *A Communication Characterisation of Splash-2 and Parsec,* in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on* (2009) pp. 86–97.

[24] M. Själander, M. Martonosi, and S. Kaxiras, *Power-Efficient Computer Architectures: Recent Advances,* Synthesis Lectures on Computer Architecture **9**, 1 (2014).

[25] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach (The Morgan Kaufmann Series in Computer Architecture and Design)* (Morgan Kaufmann, 2007).

[26] S. Manakkadu and S. Dutta, *Bandwidth Based Performance Optimization of Multi-threaded Applications,* in *Parallel Architectures, Algorithms and Programming (PAAP), 2014 Sixth International Symposium on* (2014) pp. 118–122.

[27] M. Hill and M. Marty, *Amdahl's Law in the Multicore Era,* Computer **41**, 33 (2008).

[28] M. A. Suleman, Y. N. Patt, M. A. S. Y. Patt, E. Sprangle, A. Rohillah, A. Ghuloum, D. Carmean, and I. Corporation, *Asymmetric Chip Multiprocessors: Balancing Hardware Efficiency and Programmer Efficiency,* .

[29] W. Blume and R. Eigenmann, *Performance Analysis of Parallelizing Compilers on the Perfect Benchmarks Programs,* IEEE Trans. Parallel Distrib. Syst. **3**, 643 (1992).

[30] A. Darte, Y. Robert, and F. Vivien, *Scheduling and Automatic Parallelization*, 1st ed. (Birkhauser Boston, 2000).

[31] M. A. Suleman, M. K. Qureshi, and Y. N. Patt, *Feedback-driven Threading: Power-efficient and High-performance Execution of Multi-threaded Workloads on CMPs,* SIGARCH Comput. Archit. News **36**, 277 (2008).

[32] G. Vishal, K. Hyesoon, and K. Schwan, *Evaluating Scalability of Multi-threaded Applications on a Many-core Platform*, Tech. Rep. (2012).

[33] L. Liu, Z. Li, and A. H. Sameh, *Analyzing Memory Access Intensity in Parallel Programs on Multicore,* in *Proceedings of the 22Nd Annual International Conference on Supercomputing*, ICS '08 (ACM, New York, NY, USA, 2008) pp. 359–367.

[34] Z. Guz, E. Bolotin, I. Keidar, A. Kolodny, A. Mendelson, and U. Weiser, *Many-Core vs. Many-Thread Machines: Stay Away From the Valley,* Computer Architecture Letters **8**, 25 (2009).

[35] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, *The NAS Parallel Benchmarks*, Tech. Rep. (The International Journal of Supercomputer Applications, 1991).

[36] S. Borkar, *Exascale Computing - A Fact or a Fiction?* in *IPDPS* (2013).

[37] O. Mutlu, *Memory Scaling: A Systems Architecture Perspective,* in *MEMCON* (2013).

[38] M. Horowitz, *Computing's Energy Problem (and what we can do about it),* in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2014 IEEE International* (2014) pp. 10–14.

[39] R. Nair, *Active Memory Cube: A Processing-in-Memory Approach to Power Efficiency in Exascale Systems,* in *WoNDP* (2014).

[40] J. Balfour, W. Dally, D. Black-Schaffer, V. Parikh, and J. Park, *An Energy-Efficient Processor Architecture for Embedded Systems,* Computer Architecture Letters **7**, 29 (2008).

[41] V. Saripalli, G. Sun, A. Mishra, Y. Xie, S. Datta, and V. Narayanan, *Exploiting Heterogeneity for Energy Efficiency in Chip Multiprocessors,* Emerging and Selected Topics in Circuits and Systems, IEEE Journal on **1**, 109 (2011).

[42] *TOP500 List of World's Fastest Supercomputers,* `http://www.top500.org`.

[43] National University of Defense Technology, China, *Tianhe-2 (milkyway-2),* http://www.top500.org/system/177999.

[44] N. Teimouri, H. Tabkhi, and G. Schirner, *Revisiting Accelerator-rich CMPs: Challenges and Solutions,* in *Proceedings of the 52Nd Annual Design Automation Conference*, DAC '15 (ACM, New York, NY, USA, 2015) pp. 84:1–84:6.

[45] M. A. Kim and S. A. Edwards, *Computation vs. Memory Systems: Pinning Down Accelerator Bottlenecks,* in *Proceedings of the 2010 International Conference on Computer Architecture*, ISCA'10 (Springer-Verlag, Berlin, Heidelberg, 2012) pp. 86–98.

[46] W. J. Dally and B. Towles, *Principles and Practices of Interconnection Networks.* (Morgan Kaufmann Publishers, 2007).

[47] S. G. Kavadias, M. G. Katevenis, M. Zampetakis, and D. S. Nikolopoulos, *On-chip Communication and Synchronization Mechanisms with Cache-integrated Network Interfaces,* in *International conference on Computing frontiers* (2010) pp. 217–226.

[48] N. Vijaykumar, G. Pekhimenko, A. Jog, A. Bhowmick, R. Ausavarungnirun, C. Das, M. Kandemir, T. Mowry, and O. Mutlu, *A case for Core-Assisted Bottleneck Acceleration in GPUs: Enabling Flexible Data Compression with Assist Warps,* in *Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on* (2015) pp. 41–53.

[49] M. Chu, R. Ravindran, and S. Mahlke, *Data Access Partitioning for Fine-grain Parallelism on Multicore Architectures,* in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40 (IEEE Computer Society, Washington, DC, USA, 2007) pp. 369–380.

[50] J. Hu and R. Marculescu, *Energy-aware Communication and Task Scheduling for Network-on-chip Architectures under Real-time Constraints,* in *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, Vol. 1 (2004) pp. 234–239.

[51] D. Pham, S. Asano, M. Bolliger, M. Day, H. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, and K. Yazawa, *The Design and Implementation of a First-generation CELL Processor,* in *Solid-State Circuits Conference, 2005. Digest of Technical Papers. ISSCC. 2005 IEEE International* (2005) pp. 184–592 Vol. 1.

[52] S. Vassiliadis and et al., *The MOLEN Polymorphic Processor,* IEEE Transactions on Computers **53**, 1363 (2004).

[53] J. Stuecheli, *Next Generation POWER Microprocessor,* in *HotChips 2013* (August 2013).

[54] Nvidia, *GeForce GT 640 Specifications,* http://www.geforce.com/hardware/desktop-gpus/geforce-gt640/specifications (2012).

[55] O. Pell and O. Mencer, *Surviving the End of Frequency Scaling with Reconfigurable Dataflow Computing,* SIGARCH Comput. Archit. News **39**, 60 (2011).

[56] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan, *Sequoia: Programming the Memory Hierarchy,* in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06 (ACM, New York, NY, USA, 2006).

[57] R. Willenberg and P. Chow, *A Remote Memory Access Infrastructure for Global Address Space Programming Models in FPGAs,* in *FPGA*, FPGA '13 (2013) pp. 211–220.

[58] Y. Afek, D. Dice, and A. Morrison, *Cache Index-aware Memory Allocation,* SIGPLAN Not. **46**, 55 (2011).

[59] J. van Eijndhoven, *Embedded Systems: map to FPGA, GPU, CPU?* Presentation at Bits and Chips Embedded Systems (2012).

[60] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, *Demystifying GPU Microarchitecture through Microbenchmarking,* in *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on* (2010) pp. 235–246.

[61] B. Wun, *Survey of Software Monitoring and Profiling Tools*.

[62] J. Tong and M. Khalid, *Profiling Tools for FPGA-Based Embedded Systems: Survey and Quantitative Comparison,* Journal of Computers **3** (2008).

[63] A. Jaleel, R. S. Cohn, C.-k. Luk, and B. Jacob, *CMP$im: A Pin-Based On-The-Fly Multi-Core Cache Simulator*.

[64] N. Nethercote, *Dynamic Binary Analysis and Instrumentation*, Ph.D. thesis, University of Cambridge, UK (2004).

[65] Y. Jin, *NumaTOP: A Tool for Memory Access Locality Characterization and Analysis,* https://01.org/numatop (2013).

[66] A. Pesterev, N. Zeldovich, and R. T. Morris, *Locating Cache Performance Bottlenecks Using Data Profiling,* in *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10 (ACM, New York, NY, USA, 2010) pp. 335–348.

[67] *Program Analysis Toolkit by Hewlett Packard,* http://h30097.www3.hp.com/developerstoolkit/tools.html.

[68] *Caliper by Hewlett Packard,* https://h20392.www2.hp.com/portal/swdepot/displayProductInfo.do?productNumber=CALIPEREVAL.

[69] *CodeXL by AMD,* http://developer.amd.com/tools-and-sdks/opencl-zone/codexl.

[70] *Visual Profiler by Microsoft,* http://msdn.microsoft.com/en-us/library/aa969767%28v=vs.110%29.aspx.

[71] *Solaris Studio by Oracle,* http://www.oracle.com/technetwork/server-storage/solarisstudio/overview/index-jsp-142272.html.

[72] S. Ostadzadeh, *Quantitative Application Data Flow Characterization for Heterogeneous Multicore Architectures*, Ph.D. thesis, Delft University of Technology, Delft, Netherlands (2012).

[73] W. Heirman, D. Stroobandt, N. R. Miniskar, and R. Wuyts, *A communication profiler to optimize embedded resource usage,* Annual Workshop on Circuits, Systems and Signal Processing, 20th, Proceedings , HASH(0x57aafb8) (2009).

[74] A.-H. Liu and R. Dick, *Automatic run-time Extraction of Communication Graphs from Multithreaded Applications,* in *Hardware/Software Codesign and System Synthesis, 2006. CODES+ISSS '06. Proceedings of the 4th International Conference* (2006) pp. 46–51.

[75] N. Nethercote and A. Mycroft, *Redux: A Dynamic Dataflow Tracer,* Electronic Notes in Theoretical Computer Science **89**, 149 (2003).

[76] C. Luk and et al., *Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation,* in *PLDI '05* (ACM, New York, NY, USA, 2005) pp. 190–200.

[77] W. Heirman, D. Stroobandt, N. Miniskar, R. Wuyts, and F. Catthoor, *PinComm: characterizing intra-application communication for the many-core era,* in *2010 IEEE 16th International Conference on Parallel and Distributed Systems (ICPADS)* (2010) pp. 500–507.

[78] P. Magnusson and B. Werner, *Efficient Memory Simulation in SimICS,* in *Simulation Symposium, Proceedings of the 28th Annual* (1995) pp. 62–73.

[79] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, *MPI-The Complete Reference, Volume 1: The MPI Core*, 2nd ed. (MIT Press, Cambridge, MA, USA, 1998).

[80] *Parallel Studion XE by Intel,* https://software.intel.com/en-us/intel-parallel-studio-xe ().

[81] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, and W. E. Nagel, *The Vampir Performance Analysis Tool-Set,* in *Tools for High Performance Computing*, edited by M. Resch, R. Keller, V. Himmler, B. Krammer, and A. Schulz (Springer Berlin Heidelberg, 2008) pp. 139–155.

[82] *TAU Performance System,* http://www.cs.uoregon.edu/research/tau/home.php.

[83] *mpiP: Lightweight, Scalable MPI Profiling,* mpip.sourceforge.net.

[84] M. Geimer, F. Wolf, B. J. N. Wylie, E. Ábrahám, D. Becker, and B. Mohr, *The Scalasca Performance Toolset Architecture,* Concurr. Comput. : Pract. Exper. **22**, 702 (2010).

[85] M. Gerndt and M. Ott, *Automatic Performance Analysis with Periscope,* Concurrency and Computation: Practice and Experience **22**, 736 (2010).

[86] I.-H. Chung, R. Walkup, H.-F. Wen, and H. Yu, *MPI Performance Analysis Tools on Blue Gene/L,* in *SC* (2006).

[87] H. Brunst and B. Mohr, *Performance Analysis of Large-Scale OpenMP and Hybrid MPI/OpenMP Applications with Vampir NG,* in *OpenMP Shared Memory Parallel Programming*, Lecture Notes in Computer Science, Vol. 4315, edited by M. Mueller, B. Chapman, B. de Supinski, A. Malony, and M. Voss (Springer Berlin / Heidelberg, 2008) pp. 5–14.

[88] S.-H. Hung, C.-H. Tu, and T.-S. Soon, *Trace-based Performance Analysis Framework for Heterogeneous Multicore Systems,* in *Proceedings of the 2010 Asia and South Pacific Design Automation Conference*, ASPDAC '10 (IEEE Press, Piscataway, NJ, USA, 2010) pp. 19—24.

[89] S.-H. Hung, S.-J. Huang, and C.-H. Tu, *New Tracing and Performance Analysis Techniques for Embedded Applications,* in *Proceedings of the 2008 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, RTCSA '08 (IEEE Computer Society, Washington, DC, USA, 2008) pp. 143–152.

[90] M. Bach, M. Charney, R. Cohn, E. Demikhovsky, T. Devor, K. Hazelwood, A. Jaleel, C.-K. Luk, G. Lyons, H. Patil, and A. Tal, *Analyzing Parallel Programs with Pin,* Computer **43**, 34 (2010).

[91] S. Pellegrini, T. Hoefler, and T. Fahringer, *Exact Dependence Analysis for Increased Communication Overlap,* in *Recent Advances in the Message Passing Interface*, Lecture Notes in Computer Science, Vol. 7490, edited by J. Träff, S. Benkner, and J. Dongarra (Springer Berlin Heidelberg, 2012) pp. 89–99.

[92] M. D. Ernst, *Static and Dynamic Analysis: Synergy and Duality,* in *WODA 2003: Workshop on Dynamic Analysis* (Portland, Oregon, 2003) pp. 24–27.

[93] T. Austin, E. Larson, and D. Ernst, *SimpleScalar: an Infrastructure for Computer System Modeling,* Computer **35**, 59 (2002).

[94] *Third Degree by Hewlett Packard,* http://h30097.www3.hp.com/developerstoolkit/tools.html.

[95] D. Bruening and Q. Zhao, *Practical Memory Checking with Dr. Memory,* in *Code Generation and Optimization (CGO), 2011 9th Annual IEEE/ACM International Symposium on* (2011) pp. 213–223.

[96] D. L. Bruening, *Efficient, Transparent and Comprehensive Runtime Code Manipulation*, Tech. Rep. (MIT, 2004).

[97] Nvidia, *NVPROF and NVVP, Nvidia Command-line and Visual Profilers,* http://docs.nvidia.com/cuda/profiler-users-guide.

[98] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, *The Gem5 Simulator,* SIGARCH Comput. Archit. News **39**, 1 (2011).

[99] *Purify by IBM,* http://www-03.ibm.com/software/products/us/en/rational-purify-family.

[100] K. Serebryany and T. Iskhodzhanov, *ThreadSanitizer: Data Race Detection in Practice,* in *Proceedings of the Workshop on Binary Instrumentation and Applications*, WBIA '09 (ACM, New York, NY, USA, 2009) pp. 62–71.

[101] N. Nethercote and J. Seward, *How to Shadow Every Byte of Memory Used by a Program,* in *Proceedings of the 3rd International Conference on Virtual Execution Environments*, VEE '07 (ACM, New York, NY, USA, 2007) pp. 65–74.

[102] Committee, T.I.S., *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2,* (1995).

[103] B. D. Lucas and T. Kanade, *An Iterative Image Registration Technique with an Application to Stereo Vision,* (1981) pp. 674–679.

[104] *KLT: An Implementation of the Kanade-Lucas-Tomasi Feature Tracker,* http://www.ces.clemson.edu/~stb/klt/installation.html.

[105] R. L. Myers, *Display Interfaces : Fundamentals and Standards* (New York ; Chichester : Wiley, 2002).

[106] *Par4All: An Automatic Parallelizing and Optimizing Compiler by HPC Project,* http://www.par4all.org ().

[107] C. Dave, H. Bae, S. Min, S. Lee, R. Eigenmann, and S. Midkiff, *Cetus: A Source-to-Source Compiler Infrastructure for Multicores,* Computer **42**, 36 (2009).

[108] Appentra, *Parallware,* http://www.appentra.com/products/parallware.

[109] W. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. A. Padua, P. Petersen, W. M. Pottenger, L. Rauchwerger, P. Tu, and S. Weatherford, *Polaris: Improving the Effectiveness of Parallelizing Compilers,* in *Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing*, LCPC '94 (Springer-Verlag, London, UK, UK, 1995) pp. 141–154.

[110] U. Bondhugula, J. Ramanujam, and P. Sadayappan, *PLuTo: A Practical and Fully Automatic Polyhedral Parallelizer and Locality Optimizer*, Tech. Rep. (The Ohio State University, 2007).

[111] Intel, *Automatic Parallelization with Intel Compilers,* https://software.intel.com/en-us/articles/automatic-parallelization-with-intel-compilers.

[112] J. Shen and M. Lipasti, *Modern Processor Design: Fundamentals of Superscalar Processors* (McGraw-Hill Higher Education, 2002).

[113] *Pareon by Vector Fabrics B.V,* http://www.vectorfabrics.com/products.

[114] N. Khammassi, J. Le Lann, J. Diguet, and A. Skrzyniarz, *MHPM: Multi-Scale Hybrid Programming Model: A Flexible Parallelization Methodology,* in *Proceedings of the 2012 IEEE 14th International Conference on High Performance Computing and Communication*, HPCC '12 (IEEE Computer Society, Washington, DC, USA, 2012) pp. 71–80.

[115] N. Khammassi et al., *Design and Implementation of a Cache Hierarchy-aware Task Scheduling for Parallel Loops on Multicore Architectures,* in *PDCTA* (Sydney, Australia, 2014).

[116] N. Khammassi and J.-C. Le Lann, *A High-level Programming Model to Ease Pipeline Parallelism Expression on Shared Memory Multicore Architectures,* in *Proceedings of the High Performance Computing Symposium*, HPC '14 (Society for Computer Simulation International, San Diego, CA, USA, 2014) pp. 9:1–9:8.

[117] N. Khammassi and J. Le Lann, *Tackling Real-Time Signal Processing Applications on Shared Memory Multicore Architectures Using XPU,* (2014).

[118] *ROSE Compiler Infrastructure,* http://www.rosecompiler.org.

[119] S. Vassiliadis, G. Gaydadjiev, K. Bertels, and E. Moscu Panainte, *The Molen Programming Paradigm,* in *Computer Systems: Architectures, Modeling, and Simulation*, Lecture Notes in Computer Science, Vol. 3133, edited by A. Pimentel and S. Vassiliadis (Springer Berlin / Heidelberg, 2004) pp. 1–10.

[120] P. Pham-Quoc Cuong, *Hybrid Interconnect Design for Heterogeneous Hardware Accelerators*, Ph.D. thesis, Delft University of Technology, Delft, Netherlands (2015).

[121] *Canny Edge Detector, Image Analysis Research Lab., USF,* http://marathon.csee.usf.edu/edge/edge_detection.html.

[122] C. Pham-Quoc, Z. Al-Ars, and K. Bertels, *Automated Hybrid Interconnect Design for FPGA Accelerators Using Data Communication Profiling,* in *Parallel Distributed Processing Symposium Workshops* (2014) pp. 151–160.

[123] C. Pham-Quoc, I. Ashraf, Z. Al-Ars, and K. Bertels, *Heterogeneous Hardware Accelerators with Hybrid Interconnect: an Automated Design Approach,* (Ho Chi Minh City, Vietnam, 2015).

[124] S.-A. Tahaee and A. H. Jahangir, *A Polynomial Algorithm for Partitioning Problems,* ACM Trans. Embed. Comput. Syst. **9**, 34:1 (2010).

[125] M. López-Vallejo and J. C. López, *On the Hardware-software Partitioning Problem: System Modeling and Partitioning Techniques,* ACM Trans. Des. Autom. Electron. Syst. **8**, 269 (2003).

[126] G. Marsaglia and W. W. Tsang, *The Ziggurat Method for Generating Random Variables,* Journal of Statistical Software **5**, 1 (2000).

[127] K. Schloegel, G. Karypis, and V. Kumar, *A New Algorithm for Multi-objective Graph Partitioning,* in *In Proceedings of Europar* (Springer Verlag, 1999) pp. 322–331.

[128] J. P. Singh, W.-D. Weber, and A. Gupta, *SPLASH: Stanford Parallel Applications for Shared-memory,* SIGARCH Comput. Archit. News **20**, 5 (1992).

[129] J. D. Thompson, D. G. Higgins, and T. J. Gibson, *CLUSTAL W: Improving the Sensitivity of Progressive Multiple Sequence Alignment through Sequence Weighting, Position-specific Gap Penalties and Weight Matrix Choice,* Nucleic Acids Research (1994).

Summary

Though transistor scaling yields more transistors per chip, however, the consistent performance gain due to frequency scaling is no more feasible due to physical limits. These trends shifted the computational paradigm towards integration of more and more processing cores. Multicore computing is challenging, not only because applications need to be parallelized, but also because memory access patterns and inter-core communication need to be carefully analyzed for scalable performance gain.

Another trend in computing is the utilization of heterogeneous cores in the systems, especially in the big data era. Efficient utilization of these heterogeneous architectures is not possible in an architecture agnostic way. Secondly, these systems normally have a deep memory hierarchy which makes the assignment of data-structures to the available memory spaces even more challenging. Developers need to carefully understand and match the inherent memory access patterns of the application to the architecture facilities to gain performance. Manual analysis of applications is tedious and error prone. Therefore, tools are required to characterize the data-communication in an application and highlight the communication hot spots.

In this thesis we present the design of *MCProf*, a runtime memory-access and data-communication profiler which helps programmers to perform communication-aware partitioning and mapping decisions based on the detailed quantitative profile of an application. *MCProf* provides a detailed insight of the data flow in the application and highlights not only the compute intensive parts but also the memory-intensive parts. Experimental results show that on the average, the proposed profiler has at least one order of magnitude less overhead as compared to the state-of-the-art data-communication profilers for a variety of benchmarks. Furthermore, the provided information is in relationship to the source-code, making it easy for the developers to utilize the generated information. We present a semi-automatic parallelization methodology based on *MCProf* to help programmers extract and express parallelism. Later on, we present a framework which automates this process.

To validate the proposed tool, we present the acceleration of several applications as case studies targeting both homogeneous and heterogeneous multicore platforms. In the case of homogeneous multicores, we demonstrate that better performance up to $4\times$ can be achieved by the proposed parallelization methodology when compared to available commercial compilers. In the case of heterogeneous systems using GPU and FPGA as an accelerator, experimental results show significant performance gains due to communication-aware application mapping. For instance, in

the case of GPU up to $3\times$ speedup was achieved over the optimized parallel version by utilizing the information generated by *MCProf*. In the case of reconfigurable platforms, we presented software and hardware based optimizations based on the detailed application's profile generated by *MCProf*. Software-based optimizations resulted in an overall speedup of $2.24\times$. Applying hardware-based optimizations for FPGA resulted in speed up of $1.83\times$ compared to the baseline. This also resulted in about 50% reduction in energy consumption. Finally, we also presented a case-study showing the utilization of the information generated by *MCProf* to perform data-communication aware evaluation of partitioning algorithm and partitioning solutions.

Samenvatting

Hoewel het schalen van transistors meer transistors per chip oplevert, is daarentegen de consistente prestatieverhoging als gevolg van het schalen van de frequentie niet meer haalbaar als gevolg van fysieke beperkingen. Deze trends verschoven de computationele paradigma richting de integratie van meer en meer processorkernen. Multi-core computing is uitdagend, niet alleen omdat applicaties geparallelliseerd moeten worden, maar ook omdat geheugen access patterns en inter-core communicatie zorgvuldig moeten worden geanalyseerd voor schaalbare prestatieverhogingen.

Een andere trend op het gebied van computers is het gebruik van heterogene kernen in de systemen, met name in de big data tijdperk. Het efficiënt gebruik van deze heterogene architecturen is op een architectuur agnostische manier niet mogelijk. Ten tweede hebben deze systemen doorgaans een diepe geheugenhiërarchie die de toewijzing van datastructuren aan beschikbare geheugenruimtes nog uitdagender maakt. Ontwikkelaars moeten zorgvuldig inherente geheugen access patterns begrijpen en deze matchen met de faciliteiten van de architectuur om prestatieverhogingen te verkrijgen. Handmatige analyse van applicaties is saai en foutgevoelig. Daarom zijn tools om de datacommunicatie in een applicatie te karakteriseren en de communicatie hot spots te markeren nodig.

In dit proefschrift presenteren we het design van *MCProf*, een runtime geheugentoegang en datacommunicatie profiler die programmeurs helpt bij het uitvoeren van communicatiebewuste partitionering en mapping beslissingen op basis van de gedetailleerde kwantitatieve profiel van een applicatie. *MCProf* geeft een gedetailleerd inzicht van de datastroom in de applicatie en wijst niet alleen de intensieve rekenkundige delen aan, maar ook de intensieve geheugen delen. Experimentele resultaten tonen gemiddeld gezien aan dat de voorgestelde profiler ten minste een orde van grootte minder overhead heeft vergeleken met state-of-the-art datacommunicatie profilers voor diverse benchmarks. Bovendien is de verstrekte informatie gerelateerd aan de source-code, waardoor het gemakkelijk voor ontwikkelaars is om gebruik te maken van de gegenereerde informatie. We presenteren een semi-automatische parallellisatie methodologie gebaseerd op *MCProf* die programmeurs helpen parallellisme te extraheren en uit te drukken. Later presenteren we een framework dat dit proces automatiseert.

Om de voorgestelde tool te valideren presenteren we de acceleratie van verscheidene toepassingen als case studies gericht op zowel homogene en heterogene multicore platforms. In het geval van homogene multicores demonsteren we dat maxi-

maal $4\times$ betere prestaties kunnen worden bereikt door de voorgestelde parallellisatie methode, vergeleken met beschikbare commerciële compilers. In het geval van heterogene systemen waar GPU en FPGA als een accelerator worden gebruikt, tonen de experimentele resultaten aanzienlijke prestatieverbeteringen aan, te danken aan de communicatie-bewuste applicatie mapping. Bijvoorbeeld, in het geval van GPU werd maximaal $3\times$ speedup over de geoptimaliseerde parallelle versie bereikt door het gebruikmaken van de door *MCProf* gegenereerde informatie. Voor herconfigureerbare platforms presenteerden we software en hardware optimalisaties op basis van een gedetailleerde profiel van de toepassing die gegenereerd is door *MCProf*. Software gebaseerde optimalisaties hebben geresulteerd in een totale speed up van $2.24\times$. Het toepassen van hardware optimalisaties voor FPGA heeft geresulteerd in een speed up van $1.83\times$ vergeleken met de basislijn. Dit heeft bovendien geresulteerd in een reductie van ongeveer 50% in energie consumptie. Tot slot, presenteerden we ook een casestudie die de informatie gegenereerd door *MCProf* gebruikt om een datacommunicatie-bewuste evaluatie van partitionering algoritme en oplossingen uit te voeren.

List of Publications

Publications related to this thesis

[1] Imran Ashraf, Mottaqiallah Taouil, and Koen Bertels. **Memory Profiling for Intra-application Data-Communication Quantification: A Survey**. In *Proceedings of 10th IEEE International Design and Test Symposium*, Dead Sea, Jordan, Dec 2015.

[2] Imran Ashraf, Koen Bertels, Nader Khammassi, and Jean-Christophe Le Lann. **Communication-aware Parallelization Strategies for High Performance Applications**. In *Proceedings of IEEE Computer Society Annual Symposium on VLSI*, Montpellier, France, July 2015.

[3] Cuong Pham-Quoc, Imran Ashraf, Zaid Al-Ars, and Koen Bertels. **Heterogeneous Hardware Accelerators with Hybrid Interconnect: an Automated Design Approach**. Ho Chi Minh City, Vietnam, November 2015.

[4] Imran Ashraf, Vlad-Mihai Sima, and Koen Bertels. **Intra-Application Data-Communication Characterization**. In *Proceedings of 1st International Workshop on Communication Architectures at Extreme Scale*, Frankfurt, Germany, July 2015.

[5] Imran Ashraf, S.A. Ostadzadeh, R.J. Meeuws, and Koen Bertels. **Evaluation Methodology for Data Communication-aware Application Partitioning**. In *Proceedings of 1st Workshop on Runtime and Operating Systems for the Many-core Era*, Aachen, Germany, August 2013.

[6] Imran Ashraf, S. Arash Ostadzadeh, Roel Meeuws, and Koen Bertels. **Communication-Aware HW/SW Co-design for Heterogeneous Multicore Platforms**. In *Proceedings of the 2012 Workshop on Dynamic Analysis*, WODA 2012, pages 36–41, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1455-8. doi: 10.1145/2338966.2336806.

Other publications

[7] S. Arash Ostadzadeh, Roel Meeuws, Imran Ashraf, Carlo Galuzzi, and Koen Bertels. **The Q² Profiling Framework: Driving Application Mapping for Heterogeneous Reconfigurable Platforms**. In *Reconfigurable Computing: Architectures, Tools and Applications*, volume 7199 of *Lecture Notes in Computer Science (LNCS)*, pages 76–88. Springer Berlin / Heidelberg, March 2012. ISBN 978-3-642-28364-2. doi: 10.1007/978-3-642-28365-9_7.

[8] S. Arash Ostadzadeh, Roel Meeuws, Imran Ashraf, Carlo Galuzzi, and Koen Bertels. **Profile-Guided Application Partitioning for Heterogeneous Reconfigurable Platforms**. In *Proceedings of the 16th CSI International Symposium on Computer Architecture and Digital Systems*, CADS'12, pages 37–43, Shiraz, Iran, May 2012. ISBN 978-1-4673-1481-7. doi: 10.1109/CADS.2012.6316416.

[9] M. Faisal Nadeem, Imran Ashraf, S. Arash Ostadzadeh, Stephan Wong, and Koen Bertels. **Task Scheduling in Large-scale Distributed Systems Utilizing Partial Reconfigurable Processing Elements**. In *Proceedings of the 26th IEEE International Parallel and Distributed Processing Symposium Workshops*, IPDPSW'12, pages 79–90, Shanghai, China, May 2012. IEEE Computer Society. ISBN 978-1-4673-0974-5. doi: 10.1109/IPDPSW.2012.6.

[10] Imran Ashraf. **Memory and Power Efficient Architecture for Embedded Microcontrollers**. Msc thesis, Delft University of Technology, Delft, Netherlands, July 2011.

[11] Imran Ashraf, G.M. Hassan, K.M. Yahya, S.A.A. Shah, S. Ullah, A. Manzoor, and M. Murad. **Parameter Tuning of Evolutionary Algorithm by Meta-EAs for WCET Analysis**. In *Emerging Technologies (ICET), 2010 6th International Conference on*, pages 7–10, Oct 2010. doi: 10.1109/ICET.2010.5638389.

Curriculum Vitæ

Imran Ashraf

Imran Ashraf was born in Mansehra, Pakistan, on 23 July, 1982. He received a Bachelor of Science degree in Electrical Engineering from University of Engineering and Technology, Peshawar, Pakistan, in 2006. Afterwards, he continued his studies towards a Master of Science degree in Electrical Engineering, majoring Electronics and Communication Engineering, at the same university. He was awarded the M.Sc. degree in 2008. In 2009, Imran got a scholorship from Higher Education Commission (HEC) Pakistan, and Top Talent scholorship from TU Delft, for Master studies in Embedded Systems.

In October 2011, he joined the Computer Engineering Lab, Department of Software and Computer Technology, Faculty of Electrical Engineering, Mathematics and Computer Science (EEMCS/EWI), Delft University of Technology, The Netherlands, to pursue his Doctoral Research under the supervision of Prof. dr. Koen Bertels. His research focused on advanced profiling, code parallelization, communication driven mapping of applications on multicore platforms.

His research work was mainly funded by Embedded Multi-Core systems for Mixed Criticality applications (EMC^2) project in the ARTEMIS Innovation Pilot Programme 'Computing platforms for embedded systems' (AIPP5) and HEC, Pakistan. He has also contributed to the EU-ICT FWP7 REFLECT and the SMECY (Smart Multicore Embedded Systems) projects.