TOWARD HIGH PERFORMANCE AND ENERGY EFFICIENCY ON MANY-CORE ARCHITECTURES

by

Elkin Garcia

A dissertation submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Electrical and Computer Engineering

Summer 2014

TOWARD HIGH PERFORMANCE AND ENERGY EFFICIENCY ON MANY-CORE ARCHITECTURES

by

Elkin Garcia

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: ____

Guang R. Gao, Ph.D.
Professor in charge of dissertation

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: ____

Xiaoming Li, Ph.D.
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: ____

Hui Fang, Ph.D.
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: ____

Jingyi Yu, Ph.D.
Member of dissertation committee

To my mom and grandma:

For teaching me by example to do the right things and to do my best effort no matter the difficulties.

ACKNOWLEDGEMENTS

Approaching the end of this journey, there is so much people to acknowledge. First, I want to thank my family, specially my mom. They have encouraged me to follow my dreams and they have been supportive during all these years far from home. I also want to thank Prof. Fernando Lozano, he encouraged me to explore the world and he showed me the path for pursuing a Ph.D since I was an undergrad.

During these years in the US, I have met so many people. I want to deeply express my appreciation to Prof. Guang R. Gao, he has guided me and taught me on many aspects of life during these years. He also gave me the freedom to explore and progress on my research interest. I want to thank my dissertation committee for your comments and suggestions in order to improve the quality of this work.

I am so thankful to be part of the Computer Architecture and Parallel System Laboratory (CAPSL). All CAPSL members (It is a very long list) are very talented and I enjoyed not just the academic and technical discussions but also the opportunity to learn about their culture and personal life. I particularly want to mention my mentors, close collaborators and friends: Dr. Ioannis Venetis, Dr. Joseph Manzano, Dr. Daniel Orozco, Sunil Shrestha, Robert Pavel, Jaime Arteaga and Sergio Pino. I am also grateful from the close collaboration I had with ET International and particularly with Dr. Rishi Khan.

During these years I was also surrounded by very good friends that make my life pleasant in the US. I was able to share with many different cultures and nationalities. It enhanced my Ph.D. experience. I particularly enjoyed all the support from a close group of friends known as 'Los Cheveres'.

Finally, I want to thank all the staff in the ECE Department at U. of Delaware for their valuable guidance and help through all the stages I had to pass during these years as an International Ph.D student. Particularly, Kathy Forwood has been extremely helpful from the first day I joined U. of Delaware.

TABLE OF CONTENTS

Appendix

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Recent attempts to build peta-scale and exa-scale systems have precipitated the development of new processor with hundreds, or even thousands, of independent processing units. This many-core era have brought new challenges on several fields including computer architecture, algorithm design and operating systems among others. Addressing these challenges implies new paradigms over some well-established methodologies for traditional serial architectures.

These new many-core architectures are characterized not only by the large amount of processing elements but also by the large number and heterogeneity of resources. This new environment has prompted the development of new techniques that seek finer granularity and a greater interplay in the sharing of resources. As a result, several elements of computer systems and algorithm design need to be re-evaluated under these new scenarios; it includes runtime systems, scheduling schemes and compiler transformations.

The number of transistors on a chip continues to grow following Moore's law, but single processor architectures manufactured by main vendors in the late 90's were in trouble taking advantage of the increasing number of transistors. As a consequence, Computer Architecture has become extremely parallel at all levels. It has been preferred to have several simpler processing elements than fewer more complex and powerful ones. Two main challenges in the algorithms implemented on these modern many-core architectures have arisen: (1) Shared resources have become the norm, ranging from the memory hierarchy and the interconnections between processing elements and memory to arithmetic blocks such as double floating point units, different mechanism at software and hardware levels are used for the arbitration of these shared resources and need to be considered on the scheduling and orchestration of tasks. (2) In order to

take advantage of the increasing amount of parallelism available, the number of tasks has increased and tasks have become finer, imposing new challenges for a light and balanced scheduling subject to resource and energy constraints.

The research proposed in this thesis will provide an analysis of these new scenarios, proposing new methodologies and solutions that leverage these new challenges in order to increase the performance and energy efficiency of modern many-core architectures. During the pursue of these objectives, this research intends to answer the following question:

1. Which is the impact of low-level compiler transformations such as tiling and percolation to effectively produce high performance code for many-core architectures?
2. What are the tradeoffs of static and dynamic scheduling techniques to efficiently schedule fine grain tasks with hundreds of threads sharing multiple resources under different conditions in a single chip?
3. Which hardware architecture features can contribute to better scalability and higher performance of scheduling techniques on many-core architectures on a single-chip?
4. How to effectively model high performance programs on many-core architectures under resource coordination conditions?
5. How to efficiently model energy consumption on many-cores managing tradeoffs between scalability and accuracy?
6. Which are feasible methodologies for designing power-aware tiling transformations on many-core architectures?

So far, this thesis establishes a clear methodology in order to answer these questions. This thesis addresses the research questions raised and support the claims and observations made through this document with several experiments.

We have shown the importance of tiling using dense matrix multiplication on the Cyclops-64 many-core architecture as an example. This technique alone is able to increase the performance from 3.16 GFLOPS to 30.42 GFLOPS. This performance was further improved using Instruction Scheduling and other Architecture specific optimizations reaching 44.12 GFLOPS. Later, with the use of Percolation, the new performance was 58.23 GFLOPS. We have also shown how Dynamic Scheduling can overcome a highly balanced Static Scheduling on a Matrix Multiplication. For this case, we were able to increase the performance from 58.23 GFLOPS to 70.87 GFLOPS on SRAM and from 38.73 GFLOPS to 56.26 GFLOPS on DRAM using Dynamic Percolation. These results are by far greater than any other previous published result for this architecture and it approaches the 80 GFLOPS of theoretical peak performance.

We demonstrated how Dynamic Scheduling can overcome Static Scheduling with regard to performance with other two additional applications. First, the tradeoffs of Static Scheduling (SS) vs. Dynamic Scheduling (DS) are exposed using a Memory Copy microbenchmark. Under scenarios with small amount of Hardware threads (e.g. less than 48), SS overcome DS because SS is able to produce a balanced workload with minimum overhead. However, increasing the number Thread Units makes SS schedule highly unbalanced, loosing performance. DS is a feasible solution to manage these complex scenarios and produces balanced workloads under more than a hundred Thread Units with light overhead that allows doubling the performance in some cases. Second, Sparse Vector Matrix Multiplication (SpVMM) was used to show the tradeoffs of SS vs DS under heterogeneity of task controlling the variance of the sparsity distribution for the matrix. In addition, we explained how the advantages of DS are further improved by a low-overhead implementation using mechanisms provided by the architecture, particularly *in-memory* atomic operations, diminishing the overall overhead of DS. As a result, DS can remain efficient for finer task granularities.

We have demonstrated a technique to model the performance of parallel applications on many-core architectures with resource coordination conditions. Our approach, based on timed Petri nets, results in algorithm specific models that allow us to account for the resource constraints of the system and the needs of the algorithm itself. With our approach, we were able to model the performance of a dense matrix multiplication algorithm and a finite difference time-domain (FDTD) solution in 1D and 2D with

a very high degree of accuracy, an average error of 4.4% with respect to the actual performance of the algorithms. Finally, we demonstrated how to use our approach to performance modeling to investigate, develop, and tune algorithms for modern many-core architectures, we compared two different tiling strategies for the FDTD kernel and we tested two different algorithms for LU Factorization.

We also proposed a general methodology for designing tiling techniques for energy efficient applications. The methodology proposed is based on an optimization problem that produces optimal tiling and sequence of traversing tiles minimizing the energy consumed and parametrized by the sizes of each level in the memory hierarchy. We also showed two different techniques for solving the optimization problem for two different applications: Matrix Multiplication (MM) and Finite Difference Time Domain (FDTD). Our experimental evaluation shows that the techniques proposed reduce the total energy consumption effectively, decreasing the static and dynamic component. The average energy saving for MM is 61.21%, this energy saving is 81.26% for FDTD compared with the naive tiling.

We studied and implemented several optimizations to target energy efficiency on many-core architectures with software managed memory hierarchies using LU factorization as a case of study. Starting with an implementation optimized for High Performance. We analyzed the impact of these optimizations on the Static Energy E_s, Dynamic Energy E_d, Total Energy E_T and Power Efficiency using the energy model previously developed. We designed and applied further optimizations strategies at the instruction-level and task-level to directly target the reduction of Static and Dynamic Energy and indirectly increase the Power Efficiency. We designed and implemented an energy aware tiling to decrease the Dynamic Energy. The tiling proposed minimizes the energy contribution of the most power hungry instructions. The proposed optimizations for energy efficiency increase the power efficiency of the LU factorization benchmark by 1.68X to 4.87X, depending on the problem size, with respect to a highly optimized version designed for performance. In addition, we point out examples of optimizations that scale in performance but not necessarily in power efficiency.

Finally, we showed tradeoffs between performance and energy optimizations for Many-core architectures. We explained the partial relation between performance and energy consumption through the Static Energy and execution time. We detailed some reasons that explain why energy optimization are more challenging than performance optimizations including: a) Performance optimizations just target directly the Static Energy component, with diminishing benefits for the total energy consumption. b) Some performance optimizations can affect negatively the Dynamic Energy component diminishing even more the benefits for total energy; and c) Latency can be hidden but energy cannot be; while multiple performance optimizations target a better use of resources by reordering instructions, computations or tasks in order to hide latency, the amount of work performed and the energy associated keep the same. All these reasons motivate a deeper look at strategies that optimize Dynamic Energy such as the Power Aware Tiling. Last, we showed how to exploit tradeoffs between performance and energy using a parametric power aware tiling on a parallel matrix multiplication. We reached 42% energy saving allowing a 10% decrease in performance using a rectangular tiling instead of an square tiling.

Chapter 1
INTRODUCTION

The first stage of the evolution in microprocessors were dominated by an uninterrupted increase in processor frequency, transistor count and processor functionality. As a consequence, programmers had needed minimum effort to increase the performance of their programs because all the new improvements in hardware reflected on automatic gains in performance. Unfortunately, these automatic gains can not be sustained anymore because limitations in frequency and single processor functionality have reach their limits.

These limitations are better known as *walls*. They are related to physical constrains such as the *frequency wall* and the *power wall* or architectural constrains such as the *memory wall* and the *ILP wall*

1.1 Frequency Wall

Frequency of processors had followed and exponential increase from early 70's to approximately 2004 when it reached a threshold around 4GHz as can be seen in Figure 1.1. Unfortunately, several factors stop the frequency increase and motivate to decrease the frequency on future generations of processors. The main factor is the relation between frequency and dynamic power given by Eq. 1.1

$$P_d = f \cdot C \cdot V_{DD}^2 \tag{1.1}$$

The increase in density of transistors for a single chip and the increasing in frequency has pushed silicon to is thermal limit before melting. In addition, the electromagnetic side effects produced by higher frequencies introduce additional difficulties in the hardware design process.

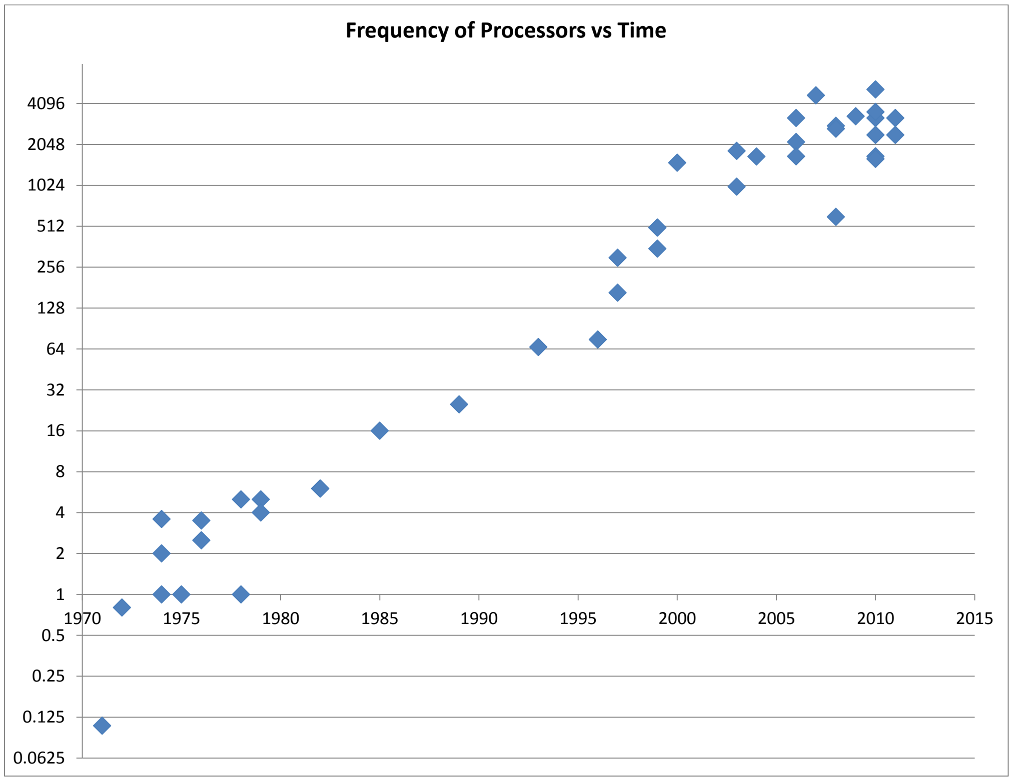


Figure 1.1: Processor Frequency over time

1.2 Power Wall

Power consumption in processors grew rapidly thanks to the increasing frequency of processors and complexity of single processor cores. Heat generated by processor makes necessary the use of passive heat sinks but later cooling systems where required in order to keep the silicon at safe temperatures. Unfortunately it produces additional increases in the power budget of a single-processor system. At the end, power density is too high for a cost-effective cooling [1]

Unfortunately, this increase in power is not scalable in the supercomputer field. Electrical energy cost and the infrastructure required to provide the electrical power to modern supercomputers make economically and physically unfeasible a continuous increase on power consumption. Power budget is a new variable on modern architectures that limits not only the frequency of operation but also the architecture itself.

1.3 Instruction Level Parallelism (ILP) Wall

The increased capabilities of processors have played a significant role in the advances of Computer Architecture. The main objective of Instruction Level Parallelism (ILP) is to increase the number of Instructions completed Per Cycle (IPC) or decreasing its reciprocal, the average cycles to complete an instruction - Clocks Per Instruction (CPI). The development of Vector Processing highly benefit the ILP, favoring specially the field of High Performance Computing (HPC). However, it failed to penetrate the processor market at large due to their high cost at that time.

Starting in the late 60's, the development of new features provided greatest impact in the consumer market, increasing the IPC of single processor architectures at that time. These new features include Scoreboarding [2] that allows out of order completion of instructions, the Tomasulo's algorithm [3] that enables multiple instructions to be issued at the same time and the Reorder Buffer that allowed speculative execution, among others.

However, the complexity of processors with higher IPC increases quadratically with the number of simultaneous instructions supported [4]. For example, the Intel Pentium II achieved only a 1.44X speedup over its predecessor, the Intel Pentium, which uses half the number of transistors. In addition, if hardware is able to execute multiple instructions at the same time, programs are not able to provide sufficient independent instructions. Despite the efforts from compiler and computer architecture sides, the maximum IPC has been constant in around 4 instructions per cycle during the past few years. This is due to the complexity in processor architecture and the data and control dependencies in programs.

1.4 Memory Wall

The increasing number of transistors per die area has allowed to include additional function units in processors and also larger memories.

Today, memory sizes continue increasing, they can store billions of bits, but their speed has not increased as rapidly as the speed of processor chips. This has created a gap between the speed of processors and the speed of memories.

Several techniques has been used to leverage this disparity between processors and memories. They include additional levels of automatic cache that take advantage of locality or prefetching units and context switches that try to hide the latency of data movement. However, the memory wall is still challenging and several programs, including HPC ones, are memory bounded. The limitation is the memory bandwidth required to keep busy the computing units.

1.5 Moore's Law

Despite the limitations reached on modern computer architecture for single processors, the increase in transistor count per die following Moore's law is still valid [5]. The number of transistors per die area continue doubling every 18 months following the observation of Gordon E. Moore in his 1965 paper. Figure 1.2 shows how the number of transistors per die grows exponentially, even after frequency of single processors stop to increase.

Now, in order to overcome and leverage the effects of the Walls that have been explained, the extra transistors are used to include additional processing elements, bringing a parallel computing revolution.

1.6 Parallel Computing Era

Parallel Computing is not new. Countless efforts to design and build parallel machines have been attempted at several points in the past.

In 1962, Borroughs Corporation introduced the D825, a computer with 4 processors and 16 memory modules accessed through a crossbar switch. Also in 1969, Honeywell introduced a symmetric multiprocessor system with 8 processors in parallel: the first Multics system.

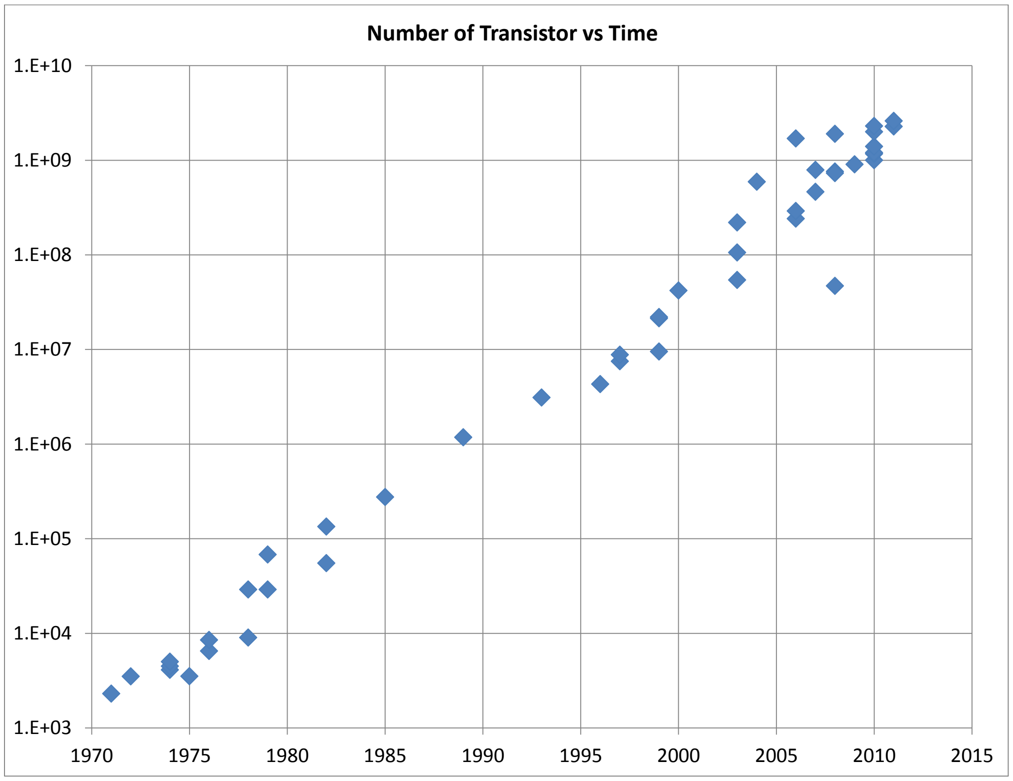


Figure 1.2: Number of Transistors over time

Single Instruction Multiple Data parallel computers started with the ILLIAC IV in 1964, from University of Illinois. It was able to work with 256 processors and large data sets, taking advantage of vector processing. Later, Seymour Cray - chief engineer and co-founder of Cray Research - was the architect of the Cray-1 in 1976. They were followed by the Cray X-MP and Cray Y-MP in 1982 and 1988.

Dataflow has lead another one of the first efforts, proposed a fully parallel machine using the concept of Dataflow Procedure Language [6]. Later the architecture of a highly concurrent multiprocessor which runs programs expressed in data flow notation was proposed [7]. It was followed by other related efforts such as the U-interpreter in 1982 [8], and the Manchester Computer in 1986 [9].

These early successful initiatives started in the academia but failed to penetrated the market. The trend of serial processors and serial programming models continued for several years with a continuous and stable increase in performance until the walls described in Sections 1.1, 1.2, 1.3 and 1.4 forced to rethink about Parallel Computer Architecture. The increase in performance comes from the use of several processing elements inside a single chip. This new paradigm on the common scenarios found in many-core architectures (e.g. large number and heterogeneity of resources) requires the study and development of new techniques that seek finer granularity and a greater interplay in the sharing of resources. As a result, several elements of computer systems and algorithm design need to be re-evaluated under these new scenarios, it includes runtime systems, scheduling schemes and compiler transformations.

1.7 Document Organization

The rest of this thesis is organized as follows: Chapter 2 presents an overview of Dataflow. Chapter 3 introduces the Problem Formulation and Research Questions developed along this thesis. Chapter 4 explains the IBM Cyclops-64 (C64) many-core architecture used in this work. Chapter 5 presents Low-level compiler transformations for high performance on many-cores. Chapter 6 discusses Dynamic Techniques for fine-grain, highly parallel programs. Chapter 7 explains a method to model the performance of Many-core Architectures under Dynamic Scheduling and Resource Constraints. Chapter 8 proposes an scalable Energy Consumption Model for many-cores and Power Aware Tiling Transformations. Chapter 9 discusses Energy Optimizations in the context of Many-core Architectures. Chapter 10 explains some trade offs between Performance and Energy optimizations. Chapter 11 outlines the related work and extensions of this research. Finally, Chapter 12 presents the summary and conclusions of this thesis.

Chapter 2
AN OVERVIEW OF DATAFLOW

The dataflow program execution model, also know as dataflow for short, is an alternative to the traditional von Neumann execution model. Dataflow relies on a graph representation of the program and its advantages are the complements of stored-program in von Neumann's model. During the las forty or so years since it was proposed, this model of computation has been used and developed in several areas of computing research ranging from programming languages to processor design, including applications to signal processing and reconfigurable computing. This chapter summarizes the current state-of-the-art in the evolution of dataflow, focusing on multi-threaded computing.

Dataflow is a very simple but powerful model for parallel computation. Dataflow programming and architectures do not have the notion of program counter or control flow such as a conventional sequential computer. In a dataflow model, computation is described in terms of local events that correspond to the firing of an actor. An actor is a single instruction or a sequence of instructions, the dataflow model does not impose a limitation on the size or the complexity of an actor. An actor can fire when all the inputs are available, in consequence many actors can fire at the same time, representing asynchronous concurrent computation events [10].

The dataflow model of execution has its roots in the work of several researchers [11, 12, 13, 6]. In the early 70's, dataflow computer architecture emerged with the use of dataflow graphs in programs to represent and exploit parallelism in them.

Dataflow allows distributed scheduling (each actor can decide "on its own" whether it is ready to fire or not), rather than relying upon a central controller to do so. In addition, the model of execution do not require a central memory system for the data elements. They are passed directly from the instructions that produce them to those that consume them.

2.1 The Static Model

Static dataflow machines share the property that in the dataflow graphs on which they are based, an arc can only hold one token. As a consequence, if there is a section of a program which is executed repeatedly (e.g., a loop body or a subroutine), the corresponding section of the dataflow graph cannot allow simultaneous execution of more than one instance of that code. There are two ways to solve this problem:

1. Pipeline the execution of the graph: Pipelining the graph for maximal parallelism requires that the graph have a structure analogous to pipelined processors. The graph must be organized neatly into stages, with no internal cycles, and all paths through the graph must have the same length. Thus, shorter paths need to be filled with identity actors that simply pass tokens along [14]

2. Replicate the graph: Replicating the graph works well when the number of iterations can be determined at compile time, as in regular numerical applications, but not when iteration counts are determined dynamically, e.g., irregular loops or binary recursion.

An architecture to execute dataflow graphs was proposed by Dennis and Misunas at MIT [11]. Their idea was to convert a dataflow graph into an essentially isomorphic structure which would be more amenable to execution on real hardware. On dataflow graphs, the arguments of actors flow on the arcs as tokens, this type of machines similar to the Dennis-Misunas architecture are known as *argument-flow machines*. Other argument-flow machines were later proposed [13, 15]

A shortcoming of the argument-flow implementations is the need for extra storage for copying operands. As an alternative, The *argument-fetch machines* were proposed to address this issue. In argument-fetch machines, data values are not attached to a specific actor, but can be stored anywhere in the Operand Memory. This means that instructions must contain references to those locations. Data no longer flows from one actor to another; only signals flow. The program in an argument-fetch machine looks much less like a dataflow graph, though it is functionally equivalent if constructed properly.

2.2 The Unraveling Interpreter

While the static interpreter allows only one instance of an instruction at a time, the U-interpreter proposed by Arvind [12] allows an indefinite number of instances of a specific instruction to exist, as a loop unravels. This is done by adding iteration tags to each data element and allowing execution of an instruction when its input operands have arrived and they all carry identical iteration tags. Among other parameters, a tag contains the iteration to which the data token belongs (other fields are used for nested loops and recursive calls). The iteration field of the tag does not change within the body of a loop but may be modified by special operators when a data element is transferred from one iteration to another.

This interpretation model is quite dynamic because it exposes, at runtime, all the parallelism that is inherent to the program graph being executed. However, this comes at a price. First, an undesirable situation could develop as the program unravels, It will happen when some parts of the program "run ahead," while others are bogged down since there is no central control. This could result in some local resources being overwhelmed. In a related issue, the size of the tag could grow indefinitely in a situation not unlike. However, again because of the lack of control, there would be no way to keep this in check.

Despite the interesting possibilities presented by the U-interpreter, its generality causes it to be impractical. A better understanding of the program structure and control on the granularity are required.

2.3 Architecture Prototypes and Implementations

The initial research on dataflow scheduling spawned several architecture implementation projects. One of the firsts was the LAU machine [16] which relied upon a single assignment language at the high level and upon associative memories to identify ready instructions.

In another project, the Hughes Data-flow Machine (HDFM) [17] offered several interesting hybrid concepts such as a higher granularity for acknowledgment between blocks, update-in-place for large structures, as well as specialized handling for lower dimensional arrays. Simulation results were promising and emphasized the need to be "application-driven."

One of the most important projects in the area, the Monsoon Dataflow Machine [18, 19], resulted in actual machines manufactured my Motorola which were distributed to various research groups for evaluation. Monsoon was based on the Unraveling Interpreter and actually implemented a mapping from the "virtual" tag to a "physical" tag through a conversion process which entailed partitioning the code into blocks of execution. One of the guiding principles behind the design of Monsoon was the close mapping which existed between the high-level language Id (functional, single assignment) and the low-level data-flow principles of execution. Monsoon was eventually superseded by a more multi-threaded version, the StarT [20].

At that time, the technology did not always allowed a large amount of parallelism, also the level of granularity used on the partition, allocation and scheduling did not take into account well-known program constructs, causing an increasing sequential overhead. Finally, the increasing performance of single processor machines, through frequency scaling, decreased the interest on this new type of parallel machines.

2.4 Dataflow and Multithreaded Execution

Challenges for suitable multithreaded computer architecture in general purpose parallel computers are the subject of intensive debate. They depend heavily on the program execution model selected, affecting the programming model, the organization of the system and the development and support of applications implemented in the architecture (e.g. compiler, runtime system and tools) [21, 22].

In this context, a program execution model defines a basic low-level layer of programming abstraction of the underlaying system architecture upon which the architecture model, programming model, compilation strategy, runtime system, and other software components are developed. It serves as an interface between the architecture and the software [10]. Program execution model is in a broader (or higher) level than instruction set architecture (ISA) specification. An ISA usually provides a description of an instruction set for a particular machine that involves specific details of instructions such as instruction encoding and the organization of the machine registers set. In the context of this work, a program execution model for multithreaded machines includes the following aspects: the thread model, the memory model and the synchronization [10].

Traditionally, the dataflow model and von Newmann serial control flow model are viewed as two execution models on opposite sites used to design a spectrum of architecture models. However, they are not orthogonal: Based on the operational model of a pure dataflow graph, it can be extended to support von Newmann program execution style. A region of actors in a dataflow graph can be grouped as a thread and executed sequentially under a private program counter; at the same time, activation and syncronization of threads can be data-driven. This hybrid model is flexible and it can combine dataflow and control-flow evaluation, exposing parallelism at a desired level.

As an example, the McGill Dataflow Architecture Model (MDAM) [23, 24] has been proposed based on the argument-fetching principle. The architecture departs from a direct implementation of dataflow graphs by having instructions fetch data from memory or registers instead of having instructions deposit operands (tokens) in "operand receivers" of successor instructions. The completion of an instruction will post an event (called a signal) to inform instructions that depend on the results of the instruction. This implements a modified model of dataflow computation called dataflow signal graphs [25].

The ideas of Iannucci, inspired by his experience on dynamic dataflow architectures, combined dataflow models with sequential thread execution on a hybrid computation model that later evolved into a multithreaded architecture at IBM [26, 27]. This architecture includes features such as cache memory with synchronization controls, prioritized processor ready queues and features for efficient process migration to facilitate load balancing.

Multithreaded execution models with dataflow origin provide support for fine-grain threads at two levels. For example, under the EARTH model [28, 29, 30], the first level of threads is called threaded function invocation: parallel function invocation forks a thread to execute the function in parallel. Note that the caller continues its own execution without waiting for the return of the forked threaded function. At a lower (finer) level, the body of a threaded function can be further partitioned (by a user or a compiler) into fibers: a collection of operations that can be forked as a separate thread. In this way, it is possible to define a thread model combining the advantages from both the static and dynamic dataflow models: the thread function invocation provides full generality as in the dynamic data flow model, while the finer level of threads maintains the simplicity of the static dataflow through software pipelining.

Recently, another hierarchical multithreading model featuring a novel event-driven, fine-grain, multithreading model has been proposed. The Codelet Execution Model is a new model centered on the concepts and semantics of codelets [31, 32]. The execution model is based on the EARTH model [33] and has been extended by Dennis's Fresh Breeze Tree-Based Memory Model [34] and also well explored by the ongoing DOE X-Stack execution model [31]. The codelet-based model has good features for exascale systems design. Even though codelets have a higher granularity than individual instructions, their granularity remains finer than that used by the execution models of the commercial CPUs. The Codelet Execution Model is a hybrid model that incorporates the advantages of macro-dataflow[26, 23, 10] and the Von Neumann model. The Codelet Execution Model can be used to describe programs in massively parallel systems, including hierarchical or heterogeneous systems. The Codelet Execution Model extends traditional macro-dataflow models in the way shared system resources are managed. The management of such resources is accomplished through events created by threads. As in macro-dataflow, computation is done through units of small serial code known as *codelets*, and execution is based solely on the availability of the data required. Codelets are executed based on required events. An event can consist of the availability of (shared) resources including data, processing elements, bandwidth, energy, etc. Codelets are tagged with resource requirements and linked together by data dependencies to form a graph (analogous to a dataflow graph [35]). This graph is further partitioned into asynchronous procedures which are invoked in a controlled flow manner. Moreover, the definition of events and the explicit expression and inclusion of system resources in the parallel execution model makes the Codelet Execution Model a promising model capable of addressing the power issues faced by future large-scale computer systems. Works that use the codelet model include ETI's SWift Adaptive Runtime Machine (SWARM) [36] and TIDeFlow [37, 38, 39].

Chapter 3
PROBLEM FORMULATION

The new many-core era motivated by the recent efforts to build peta-scale and exa-scale machines have brought several challenges for exploiting the parallelism on new many-core architectures with hundreds, or even thousands, of independent processing elements. The scenario inside these chips is different to previous multi-core processors, some of the new characteristics are:

- Increasing amount of shared resources.
- Heterogeneity of resources.
- Diversity in coordination and arbitration mechanisms for shared resources.
- Constraints in energy consumption.

This new environment requires new techniques that seek finer granularity and a greater interplay in the sharing of resources. These work re-evaluate several elements of computer systems and algorithm design under these new scenarios, it includes runtime systems, scheduling schemes and compiler transformations.

Moore's law is still valid, the number of transistor in a single chip doubles every 18 months approximately, but single processor architectures are not able to take advantage of the increasing amount of transistors. Today, Computer Architecture has become extremely parallel at all levels. The many-core era has arisen: A large number of simple processing elements are preferred over few very complex but powerful processors.

This new era brings two main challenges in the algorithms implemented on these modern many-core architectures:

1. Shared resources have become the norm, ranging from the memory hierarchy and the interconnections between processing elements and memory to arithmetic blocks such as double floating point units, different mechanism at software and hardware levels are used for the arbitration of these shared resources and need to be consider on the scheduling and orchestration of tasks.
2. In order to take advantage of the increasing amount of parallelism available, the number of tasks has increased and tasks have become finer, imposing new challenges for a light and balanced scheduling subject to resource and energy constraints.

The research proposed here will provide an analysis of these new scenarios, proposing new methodologies and solutions that leverage these new challenges in order to increase the performance and energy efficiency of modern many-core architectures. During the pursue of these objectives, this research intends to answer the following question:

1. Which is the impact of low-level compiler transformations such as tiling and percolation to effectively produce high performance code for many-core architectures?
2. What are the trade-offs of static and dynamic scheduling techniques to efficiently schedule fine grain tasks with hundreds of threads sharing multiple resources under different conditions in a single chip?
3. Which hardware architecture features can contribute to better scalability and higher performance of scheduling techniques on many-core architectures on a single-chip?
4. How to effectively model high performance programs on many-core architectures under resource coordination conditions?
5. How to efficiently model energy consumption on many-cores managing trade offs between scalability and accuracy?
6. Which are feasible methodologies for designing power-aware tiling transformations on many-core architectures?

So far, this thesis establish a clear methodology, propose solutions and provide evidence in order to answer these questions.

Chapter 4

AN INNOVATIVE MANY-CORE ARCHITECTURE

Cyclops-64 (C64) is an innovative architecture developed by IBM, designed to serve as a dedicated petaflop computing engine for running high performance applications. A C64 chip is an 80-processor many-core-on-a-chip design, as can be seen in Figure 4.1. Each processor is equipped with two thread units (TUs), one 64-bit floating point unit (FP) and two SRAM memory banks of 30kB each. It can issue one double precision floating point "Multiply and Add" instruction per cycle, for a total performance of 80 GFLOPS per chip when running at 500MHz.

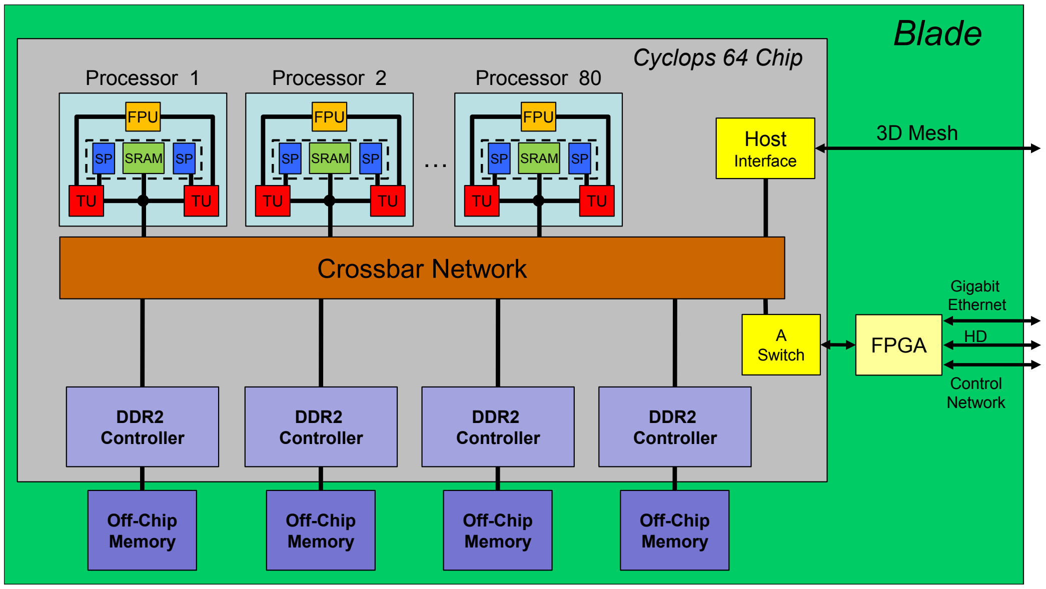


Figure 4.1: C64 Chip Architecture

A 96-port crossbar network with a bandwidth of 4GB/s per port connects all TUs and SRAM banks. The total crossbar network bandwidth of 384GB/s supports

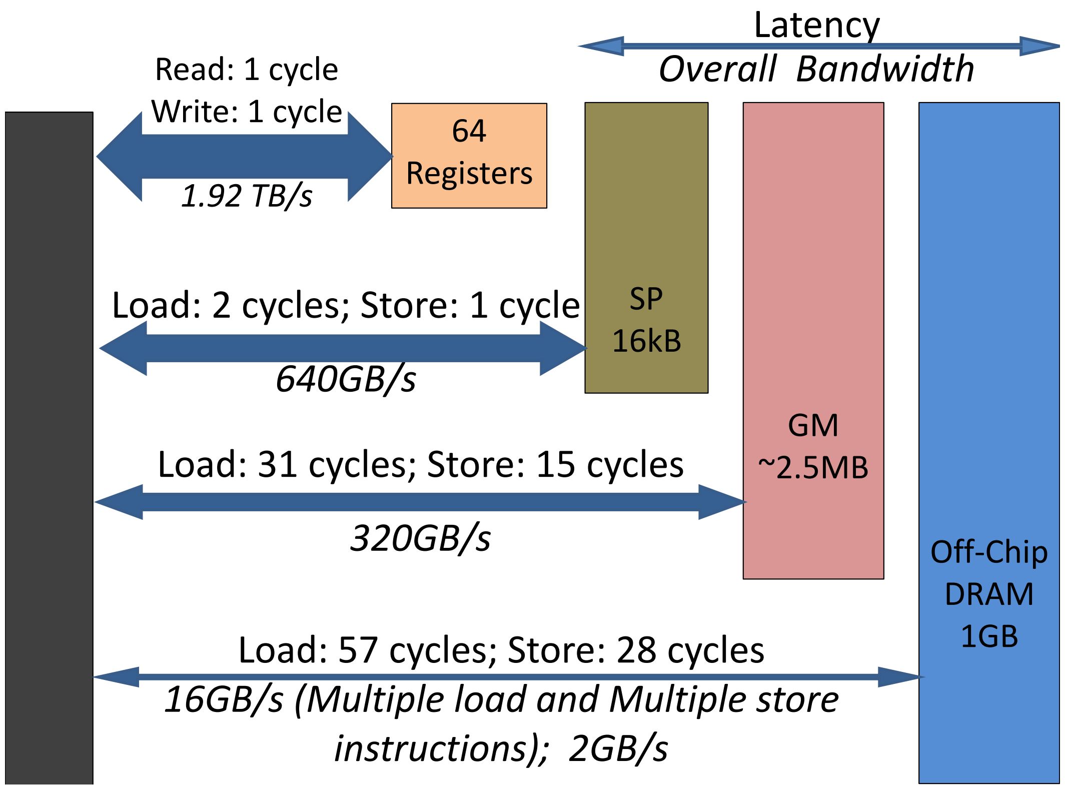

Latency
Overall Bandwidth
Read: 1 cycle
Write: 1 cycle
1.92 TB/s
64 Registers
Load: 2 cycles; Store: 1 cycle
640GB/s
SP 16kB
Load: 31 cycles; Store: 15 cycles
320GB/s
GM ~2.5MB
Off-Chip DRAM 1GB
Load: 57 cycles; Store: 28 cycles
16GB/s (Multiple load and Multiple store instructions); 2GB/s

Figure 4.2: Memory Hierarchy of C64

both the intra-chip communication, as well as the six routing ports that connect each C64 chip to its neighbours [40]. The complete C64 system is built out of tens of thousands of C64 processing nodes arranged in a 3-D mesh topology. Each processing node consists of a C64 chip, external DRAM, and a small amount of external interface logic.

4.1 Memory Hierarchy

A C64 chip has an explicit three-level memory hierarchy (scratchpad memory, on-chip SRAM, off-chip DRAM), 16 instruction caches of 32kB each (not shown in the figure) and no data cache. The scratchpad memory (SP) is a configured portion of each on-chip SRAM bank which can be accessed with very low latency by the TU it belongs to. The remaining sections of all on-chip SRAM banks consist the on-chip global memory (GM), which is uniformly addressable from all TUs. As a summary, Figure 4.2 reflects the current size, latency (when there is no contention) and bandwidth of each level of the memory hierarchy.

Execution on a C64 chip is non-preemptive and there is no hardware virtual memory manager. The former means that the C64 micro-kernel will not interrupt the execution of a user application unless an exception occurs. The latter means the three-level memory hierarchy of the C64 chip is visible to the programmer.

All memory controllers in C64 support *in-memory* atomic operations: Each memory controller has an ALU that allows it to execute atomic operations in 3 clock cycles directly inside the memory controller (both SRAM and DRAM), without help from a thread unit.

4.2 Energy Consumption

Because C64 is a general purpose many-core architecture it has not been designed for energy efficiency and it does not have special features for saving power. For example, it is not possible to turn off cores not used or to slow down the clock rate of a set of cores or for the whole chip.

Despite the fact that the C64 Instruction Set Architecture (ISA) does not include any additional instructions that help reduce energy consumption we can group the instructions according to the hardware units they use and the complexity of the operation (reflected indirectly on the execution time if there is not contention). Furthermore, we can use these groups to build our energy consumption model. According with that, the taxonomy proposed for the ISA is:

- Logical Operations: And, or, etc.
- Integer Arithmetic Operations:
  - Simple: Add, sub.
  - Medium: Multiply.
- Floating Point Operations:
  - Simple: Add, sub.

  - Medium: Multiply, multiply and add.

- Memory Operations:

  - On Registers: Move, load immediate.
  - On SPM: load, store.
  - On SRAM: load, store.
  - On DRAM: load, store.

Some instructions not mentioned here. For example, branches can be included in the logical operations category, given the hardware resources and amount of work they require.

Chapter 5

STATIC OPTIMIZATIONS IN THE CONTEXT OF MANY-CORE ARCHITECTURES

Traditional parallel programming methodologies for improving performance assume cache-based parallel systems. They exploit temporal locality making use of cache tiling techniques with tile size selection and padding [41, 42]. However, the data location and replacement in the cache is controlled by hardware making fine control of these parameters difficult. In addition, power consumption and chip die area constraints make increasing on-chip cache an untenable solution to the memory wall problem [43, 44].

As a result, new architectures like the IBM Cyclops-64 (C64) belong to a new set of many-core-on-a-chip systems with a software managed memory hierarchy. These new kinds of architectures hand the management of the memory hierarchy to the programmer and save the die area of hardware cache controllers and over-sized caches. Although this might complicate programming at their current stage, these systems provide more flexibility and opportunities to improve performance. Following this path, new alternatives for classical algorithmic problems, such as Dense Matrix Multiplication (MM), LU Factorization (LU) and Fast Fourier Transform (FFT) have been studied under these new many-core architectures [45, 46, 47]. The investigation of these new opportunities leads to two main conclusions: (1) The optimizations for improving performance on cache-based parallel system are not necessarily feasible or convenient on software managed memory hierarchy systems. (2) Memory access patterns reached by appropriate tiling substantially increase the performance of applications.

Based on these observations we can conclude that new programming and compiling methodologies are required to fully exploit the potential of these new classes of architectures. We believe that a good starting point for developing such methodologies are classical algorithms with known memory access and computation patterns. These applications provide realistic scenarios and have been studied thoroughly under cache-based parallel systems.

Following this idea, we present a general methodology that provides a mapping of applications to software managed memory hierarchies, using MM on C64 as a case of study. MM was chosen because it is simple to understand and analyze, but computationally and memory intensive. For the basic algorithm, the arithmetic complexity and the number of memory operations in multiplications of two matrices of sizes $m \times m$ are $O(m^3)$.

The methodology presented in this thesis is composed of three strategies that result in a substantial increase in performance, by optimizing different aspects of the algorithm. The first one is a balanced distribution of work among threads. Providing the same amount of work to each thread guarantees minimization of the idle time of processing units waiting for others to finish. If a perfect distribution is not feasible, a mechanism to minimize the differences is proposed. The second strategy is an optimal register tiling and sequence of traversing tiles. Our register tiling and implementation of the sequence of traversing tiles are designed to maximize the reuse of data in registers and minimize the number of memory accesses to slower levels, avoiding unnecessary stalls in the processing units while waiting for data. The last strategy involves more specific characteristics of C64. The use of special instructions, optimized instruction scheduling and other techniques further boost the performance reached by the previous two strategies. The impact on performance can change according to the particular characteristics of the many-core processor used.

The experimental evaluation was performed using a real C64 chip. After the implementation of the three strategies proposed, the performance reached by the C64 chip is 44.12 GFLOPS, which corresponds to 55.2% of the peak performance.

5.1 Classic Matrix Multiplication Algorithms

MM algorithms have been studied extensively. These studies focused mainly on two areas: (1) Algorithms that decreases the naïve complexity of $O(m^3)$. (2) Implementations that take advantage of advanced features of computer architectures to achieve higher performance. This study is oriented towards the second area.

In the first area, more efficient algorithms are developed. Strassen's algorithm [48] is based on the multiplication of two 2×2 matrices with 7 multiplications, instead of 8 that are required in the straightforward algorithm. The recursive application of this fact leads to a complexity of $O(m^{log 7})$ [49]. Disadvantages, such as numerical instability and memory space required for sub-matrices in the recursion, have been discussed extensively [50, 51]. The current best lower bound is $O(m^{2.376})$, given by the Coppersmith–Winograd algorithm [52]. However, this algorithm is not used in practice, due to its large constant term.

The second area focuses on efficient implementations. Although initially more emphasis was given towards implementations for single processors, parallel approaches quickly emerged. A common factor among most implementations is the decomposition of the computation into blocks. Blocking algorithms not only give opportunities for better use of specific architectural features (e.g., memory hierarchy) but also are a natural way of expressing parallelism. Parallel implementations have exploited the interconnection pattern of processors, like Cannon's matrix multiply algorithm [53, 54, 55], or the reduced number of operations like Strassen's algorithm [56, 57, 58]. These implementations have explored the design space along different directions, according to the targeted parallel architecture.

The many-core architecture design space has not yet been explored in detail, but existing studies already show their potential. A performance prediction model for Cannon's algorithm has shown a huge performance potential for an architecture similar to C64 [59]. Previous research of MM on C64 showed that is possible to increase performance substantially by applying well known optimizations methods and adapting them to specific features of the chip [46]. More recent results on LU Factorization conclude that some optimizations that performs well for classical cached-based parallel system are not the best alternative for improving performance on software managed memory hierarchy systems [47].

5.2 Proposed Matrix Multiplication Algorithm

In this section we analyze the proposed MM algorithm and highlight our design choices. The methodology used is oriented towards exploiting the maximum benefit of features that are common across many-core architectures. Our target operation is the multiplication of dense square matrices $A \times B = C$, each of size $m \times m$ using algorithms of running time $O(m^3)$. Throughout the design process, we will use some specific features of C64 to illustrate the advantages of the proposed algorithm over different choices used in other MM algorithms.

Our methodology alleviates three related sources identified to cause poor performance in many-core architectures: (1) Inefficient or unnecessary synchronization. (2) Unbalanced work between threads. (3) Latency due to memory operations. Relation and impact in performance of these sources are architecture dependent and modeling their interactions has been an active research topic.

In our particular case of interest, the analysis of MM is easier than other algorithms not only for the simple way it can be described but also for the existence of parallel algorithms that do not required synchronizations. It simplifies the complexity of our design process because we only need to carefully analyze in two instead of the three causes of poor performance we have identified as long as the algorithm proposed does not require synchronizations. These challenges will be analyzed in the following subsections.

5.2.1 Work Distribution

The first challenge in our MM algorithm is to distribute work among P processors avoiding synchronization. It is well known that each element $c_{i,j} \in C$ can be calculated independently. Therefore, serial algorithms can be parallelized without requiring any synchronization for the computation of each element $c_{i,j}$, which immediately solves this requirement.

The second step is to break the $m \times m$ matrix C into blocks such that we minimize the maximum block size pursuing optimal resource utilization and trying to avoid overloading a processor. This is optimally done by breaking the problem into blocks of $\frac{m^2}{P}$ elements, but the blocks must be rectangular and fit into C.

One way to break C in P rectangular blocks is dividing rows and columns of C into q_1 and q_2 sets respectively, with $q_1 \cdot q_2 = P$. The optimal way to minimize the maximum block size is to divide the m rows into q_1 sets of $\left\lfloor \frac{m}{q_1} \right\rfloor$ rows (with some having an extra row) and the same for columns. The maximum tile size is $\left\lceil \frac{m}{q_1} \right\rceil \cdot \left\lceil \frac{m}{q_2} \right\rceil$ and it is bounded by $\left(\frac{m}{q_1} + 1\right) \cdot \left(\frac{m}{q_2} + 1\right)$. The difference between this upper bound and the optimal tile size is $\frac{m}{q_1} + \frac{m}{q_2} + 1$ and this difference is minimized when $q_1 = q_2 = \sqrt{P}$. If P is not a square number, we find the q_1 that is a factor of P and closest but not larger than $\sqrt{P}$. To further optimize, we can turn off some processors if the maximum tile size could be decreased. In practice, this reduces to turning off processors if $q_2 - q_1$ is smaller and in general, this occurs if P is prime or one larger than a square number.

5.2.2 Minimization of High Cost Memory Operations

After addressing the synchronization and load-balancing problems for MM, the next major bottleneck is the impact of memory operations. Despite the high bandwidth of on-chip memory in many-core architectures (e.g. C64), bandwidth and size of memory are still bottlenecks for algorithms, producing stalls while processors are waiting for new data. As a result, implementations of MM, LU and FFT are still memory bound [45, 46, 47]. However, the flexibility of software-managed memory hierarchies provides new opportunities to the programmer for developing better techniques for tiling and data locality without the constraints imposed by cache parameters like line sizes or line associativity [60, 47]. It implies an analysis of the tile shapes, the tile size and the sequences in which tiles have to be traversed taking advantage of this new dimension in the design space.

While pursuing a better use of the memory hierarchy, our approach takes two levels of this hierarchy, one faster but smaller and the other slower but bigger. Our objective is to minimize the number of slow memory operations, loads (LD) and stores (ST), that may are function of the problem (Λ), the number of processors (P), the tile parameters (L) and the sequence of traversing tiles (S), subject to the data used in the current computation (R) cannot exceed the size of the small memory ($R_{\max}$). This can be expressed as the optimization problem:

$$\min_{L,S} \quad LD\left(\Lambda, P, L, S\right) + ST\left(\Lambda, P, L, S\right), \quad s.t. \quad R\left(\Lambda, P, L, S\right) \leq R_{\max} \tag{5.1}$$

In our case, internal registers are the fast memory and Λ is the MM with the partitioning described in subsection 5.2.1. Our analysis assumes a perfect load-balanced case where each block $C' \in C$ of size $n \times n$ $\left(n = \frac{m}{\sqrt{P}}\right)$ computed by one processor is subdivided in tiles $C'_{i,j} \in C'$ of size $L_2 \times L_2$. Based on the data dependencies, the required blocks $A' \in A$ and $B' \in B$ of sizes $n \times m$ and $m \times n$ are subdivided in tiles $A'_{i,j} \in A'$ and $B'_{i,j} \in B'$ of sizes $L_2 \times L_1$ and $L_1 \times L_2$ respectively.

Each processor requires 3 nested loops for computing all the tiles of its block. Using loop interchange analysis, an exhaustive study of the 6 possible schemes to traverse tiles was conducted and two prototype sequences S_1 and S_2 were found. The algorithms that describe these sequences are in Figure 5.1.

Based on the data dependencies of this implementations, the general optimization problem described in (5.1) can be expressed for our case by Eq. (5.2).

$$\min_{\substack{L \in \{L_1, L_2\}, \\ S \in \{S_1, S_2\}}} \quad f\left(m, P, L, S\right) = \begin{cases} \frac{2}{L_2} m^3 + m^2 & \text{if } S = S_1 \\ \left(\frac{2}{L_1} + \frac{1}{L_2}\right) m^3 + \left(\sqrt{P} - 1\right) m^2 & \text{if } S = S_2 \end{cases} \tag{5.2}$$
$$\text{s.t.} \quad 2L_1L_2 + L_2^2 \leq R_{\max}$$

Analyzing the piecewise function f, it can be easily shown that S_1 sequence has an smaller objective function than S_2 under the conditions $P \geq 4$ and $\frac{L_2}{L_1} \geq \frac{1}{2}$. The first

```
S1:  for $i = 1$ to $\frac{n}{L_2}$
S2:    for $j = 1$ to $\frac{n}{L_2}$
S3:      Initialize $C'_{i,j}$
S4:      for $k = 1$ to $\frac{m}{L_1}$
S5:        Load $A'_{i,k}$, $B'_{k,j}$
S6:        $C'_{i,j} += A'_{i,k} \cdot B'_{k,j}$
S :      end for
S7:      Store $C'_{i,j}$
S :    end for
S : end for
```

(a) Algorithm using sequence S_1

```
S1:  for $i = 1$ to $\frac{n}{L_2}$
S2:    for $k = 1$ to $\frac{m}{L_1}$
S3:      Load $A'_{i,k}$
S4:      for $j = 1$ to $\frac{n}{L_2}$
S5:        if $k = 1$ then Initialize $C'_{i,j}$
S6:        else Load $C'_{i,j}$
S7:        Load $B'_{k,j}$
S8:        $C'_{i,j} += A'_{i,k} \cdot B'_{k,j}$
S9:        Store $C'_{i,j}$
S :      end for
S :    end for
S : end for
```

(b) Algorithm using sequence S_2

Figure 5.1: Implementation of sequences for traversing tiles in one block of C

one is easily satisfied in many-cores, the second one can be satisfied when $2L_2 \geq L_1$ and it can be verified with the solution.

We will solve the integer optimization problem using the branch and bound technique. Since f only depends on L_2 (when $S = S_1$), we minimize the function f by maximizing L_2. Given the constraint, L_2 is maximized by minimizing L_1. Thus $L_1 = 1$, we solve the optimum L_2 in the boundary of the constraint and round off it. The solution of Eq. (5.2) is:

$$L_1 = 1,\ L_2 = \left\lfloor \sqrt{1 + R_{\max}} - 1 \right\rfloor \tag{5.3}$$

This result is not completely accurate, since we assumed that there are not remainders when we divide the matrices into blocks and subdivide the blocks in tiles. Despite this fact, they can be used as a good estimate.

For comparison purposes, C64 has 63 registers and we need to keep one register for the stack pointer, pointers to A, B, C matrices, m and stride parameters, then $R_{\max} = 63 - 6 = 57$ and the solution of Eq. (5.3) is $L_1 = 1$ and $L_2 = 6$. Table 5.1 summarizes the results in terms of the number of LD and ST for the tiling proposed and other 2 options that fully utilizes the registers and have been used in practical algorithms: inner product of vectors ($L_1 = 28$ and $L_2 = 1$) and square tiles ($L_1 = L_2 = 4$). As a consequence of using sequence S_1, the number of ST is equal in all tiling strategies. As expected, the tiling proposed has the minimum number of LD: 6 times less than the inner product tiling and 1.5 times less than the square tiling.

Table 5.1: Number of memory operation for different tiling strategies

Memory Operations	Inner Product	Square	Optimal
Loads	$2m^3$	$\frac{1}{2}m^3$	$\frac{1}{3}m^3$
Stores	m^2	m^2	m^2

5.2.3 Architecture Specific Optimizations

Although the general results of subsection 5.2.2 are of major importance, an implementation that properly exploits specific features of the architecture is also important for maximizing the performance. We will use our knowledge and experience for taking advantage of the specific features of C64 but the guidelines proposed here could be extended to similar architectures.

The first optimization is the use of special assembly functions for Load and Store. C64 provides the instructions multiple load (*ldm RT, RA, RB*) and multiple store (*stm RT, RA, RB*) that combine several memory operations into only one instruction. For the *ldm* instruction, starting from an address in memory contained in *RA*, consecutive 64-bit values in memory are loaded into consecutive registers, starting from *RT* through and including *RB*. Similarly, *stm* instruction stores 64-bit values in memory consecutively from *RT* through and including *RB* starting in the memory address contained in *RA*.

The advantage in the use of these instructions is that the normal load instruction issues one data transfer request per element while the special one issues one request each 64-byte boundary. Because our tiling is 6×1 in A and 1×6 in B, we need A in column-major order and B in row-major order as a requirement for exploiting this feature. If they are not in the required pattern, we transpose one matrix without affecting the complexity of the algorithms proposed because the running time of transposition is $O(m^2)$.

The second optimization applied is instruction scheduling: the correct interleaving of independent instructions to alleviate stalls. Data dependencies can stall the execution of the current instruction waiting for the result of one issued previously. We want to hide or amortize the cost of critical instructions that increase the total computation time executing other instructions that do not share variables or resources. The most common example involves interleaving memory instructions with data instructions but there are other cases: multiple integer operations can be executed while one floating point operation like multiplication is computed.

5.3 Experimental Evaluation

This section describes the experimental evaluation based on the analysis done in section 5.2 using the C64 architecture described in section 4. Our baseline parallel MM implementation works with square matrices $m \times m$ and it was written in C. The experiments were made up to $m = 488$ for placing matrices A and B in on-chip SRAM and matrix C in off-chip DRAM, the maximum number of TUs used is 144.

To analyze the impact of the partitioning schema described in subsection 5.2.1 we compare it with other two partition schemes. Figure 5.2 and Figure 5.3 show the performance reached for two different matrix sizes. In *Partitioning 1*, the m rows are divided into q_1 sets, the first $q_1 - 1$ containing $\left\lfloor \frac{m}{q_1} \right\rfloor$ and the last set containing the remainder rows. The same partitioning is followed for columns. It has the worst performance of the three partitions because it does not minimize the maximum tile size. *Partitioning 2* has optimum maximum tile size of $\left\lceil \frac{m}{q_1} \right\rceil \cdot \left\lceil \frac{m}{q_2} \right\rceil$ but does not distribute the number of rows and columns uniformly between sets q_1 and q_2 respectively. Its performance is very close to our algorithm *Partitioning 3*, which has optimum maximum tile size and better distribution of rows and columns between sets q_1 and q_2 respectively. A disadvantage of *Partitioning 2* over *Partitioning 3* is that for small matrices ($n \leq 100$) and large number of TUs *Partitioning 2* may produce a significant lower performance as can be observed in Figure 5.2. Our partitioning algorithm *Partitioning 3* performs always better, the maximum performance reached is 3.16 GFLOPS. The other one with optimum maximum tile size performs also well for large matrices, indicating that minimizing the maximum tile size is an appropriate target for optimizing the work load. In addition, our partition algorithm scales well with respect to the number of threads which is essential for many-core architectures.

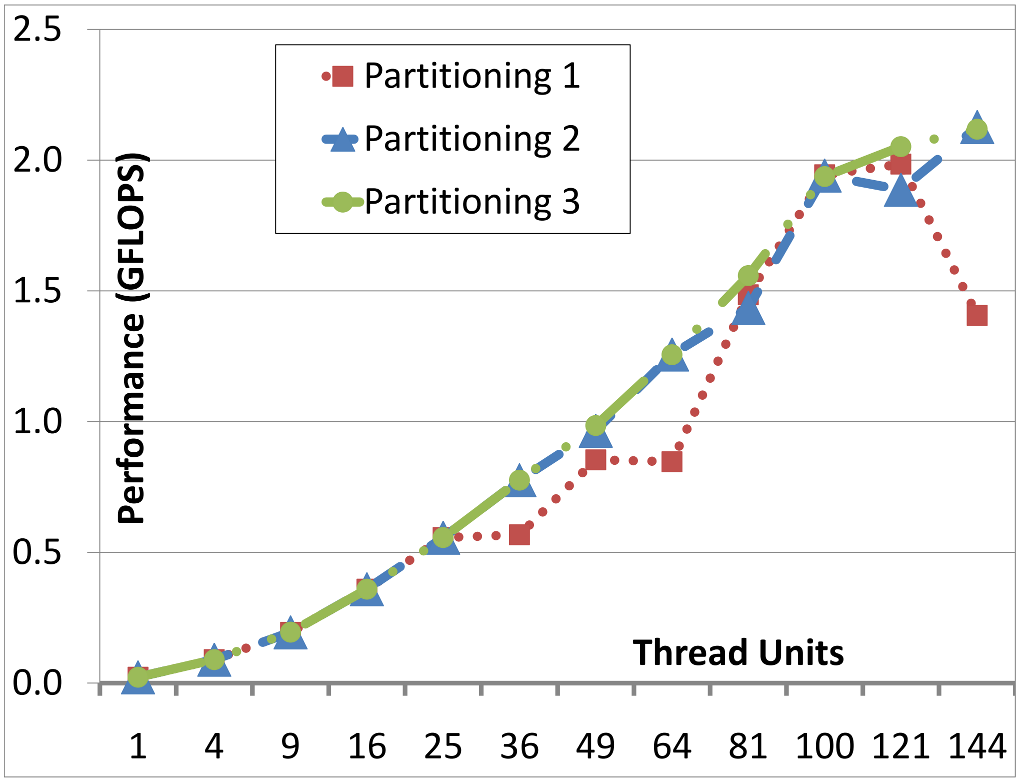


Figure 5.2: Different Partition Schemes vs. Number of Threads Units. Matrix Size 100×100

The results of the progressive improvements made to our MM algorithm are shown in Figure 5.4 for the maximum size of matrices that fits on SRAM. The implementation of the tiling strategy proposed in subsection 5.2.2 for minimizing the number of memory operations, was made in assembly code using tiles of 6×1, 1×6 and 6×6 for blocks in A, B and C respectively. Because the size of blocks in C

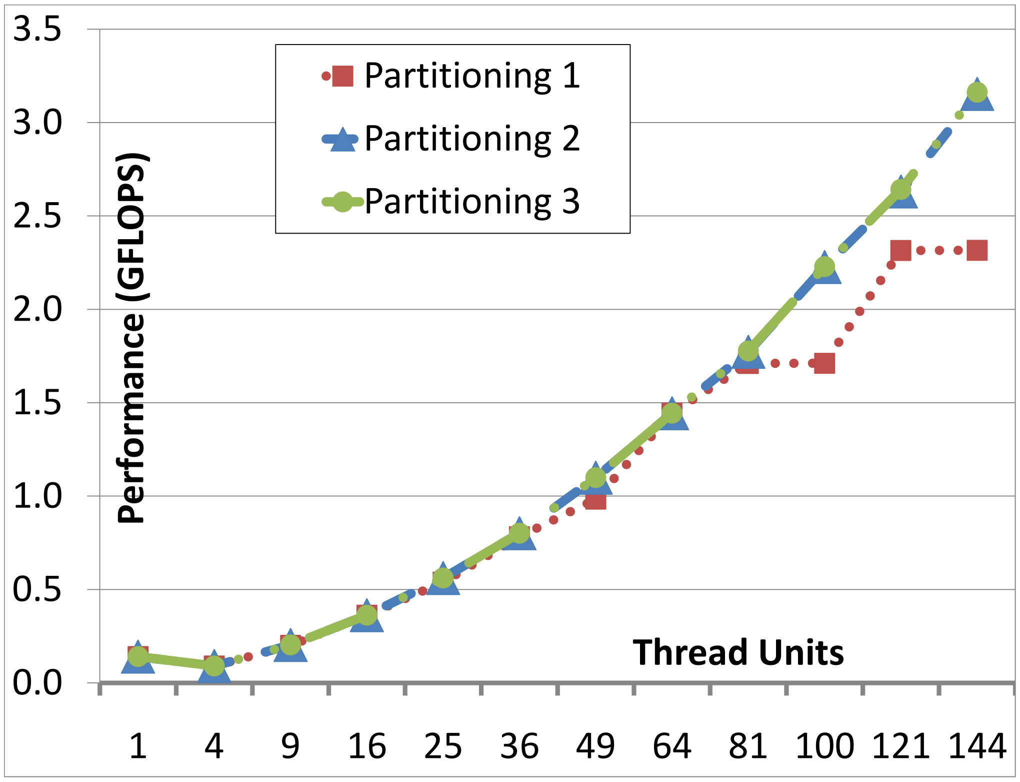


Figure 5.3: Different Partition Schemes vs. Number of Threads Units. Matrix Size 488×488

are not necessarily multiple of 6, all possible combinations of tiles with size less than 6×6 were implemented. The maximum performance reached was 30.42 GFLOPS, which is almost 10 times the maximum performance reached by the version that uses only the optimum partition. This big improvement shows the advantages of the correct tiling and sequence of traversing tiles that directly minimizes the time waiting for operands, substituting costly memory operations in SRAM with operations between registers. From another point of view, our tiling increases the reuse of data in registers minimizing number of access to memory for a fixed number of computations.

The following optimizations related more with specific features of C64 also increased the performance. The use of multiple load and multiple store instructions

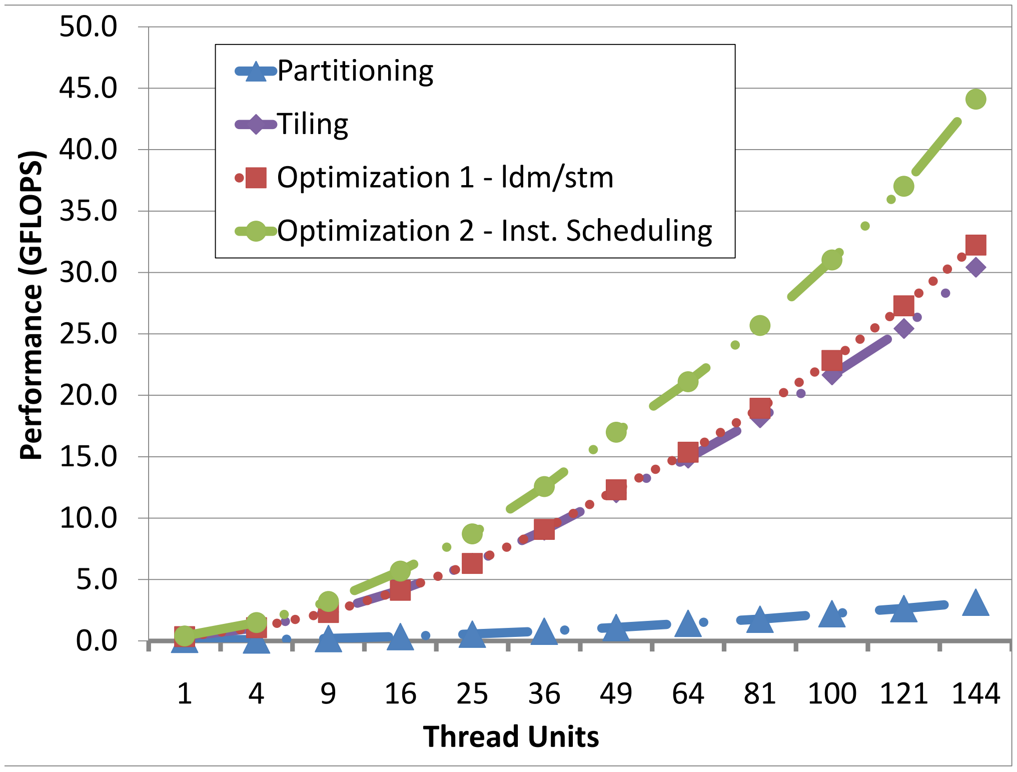


Figure 5.4: Impact of each optimization on the performance of MM using $m = 488$

(*ldm/stm*) diminishes the time spent transferring data addressed consecutively in memory. The new maximum performance is 32.22 GFLOPS: 6% better than the version without architecture specific optimizations. The potential of these features has not been completely explored because transactions that cross a 64-byte boundary are divided and transactions in groups of 6 do not provide an optimum pattern for minimizing this division. Finally, the instruction scheduling applied for hiding the cost of some instructions doing other computations in the middle increases performance by 38%. The maximum performance of our MM algorithm is 44.12 GFLOPS which corresponds to 55.2% of the peak performance of a C64 chip.

Chapter 6

THE PROBLEM OF STATIC TECHNIQUES AND THE RISING OF DYNAMIC OPTIMIZATIONS FOR MANY-CORE ARCHITECTURES

Recent trends in computer architecture, where many-core processors are routinely composed of hundreds of processing cores, have unleashed challenges in many aspects of computing technology. Task scheduling, in particular, is difficult in many-core architectures due to the quantity, availability, and diversity of resources: Static Techniques, including for example Static Scheduling (SS), were usually preferred over Dynamic Techniques, including for example Dynamic Scheduling (DS), for regular and embarrassingly parallel applications on general purpose architectures. However, techniques such as SS are not necessarily the best choice for many-core architectures, even for regular, embarrassingly parallel applications.

The two main factors that usually hurt the expected advantages of SS over DS in many-cores are:

1. The large number of processing elements in a many-core chip results in fewer tasks assigned to each processing element.
2. The behavior and interaction of shared resources are not necessarily uniform during execution.

These two new factors blunt the effectiveness of SS while greatly favoring DS, even under scenarios where SS has traditionally been the logical solution.

Keeping the abundant number of Processing Elements (PEs) inside of a chip busy, when resources are limited, results in few tasks per PE, often with comparatively small durations. Thus, a totally balanced distribution of tasks becomes a daunting challenge as problem sizes and application features make individual situations very different. Even within the design space of fixed problem sizes, not all tasks will be identical, because the problem size may or may not be a multiple of the expected task size, resulting in varying task sizes. These small variations in the sizes can contribute to the imbalance of the system, in particular when the granularity of the task is fine and the number of tasks per PE is decreased.

Shared resources are an important source of task imbalance because the arbitration of shared resources may produce unexpected stalls that could change the completion time of similar tasks. The most common shared resources on many-cores are the memory, the communication infrastructure (e.g. crossbar, access ports), and the functional units (e.g. Floating Point Units and other special purpose units).

All of these sources of imbalance make it difficult for SS to provide a strategy that fully utilizes the hardware. This produces results below those expected, even for classical regular applications like Matrix Multiply [61].

The nature of DS can manage and compensate for the unpredictable effects of resource sharing and imbalance introduced by the granularity of tasks. However, it remains a challenge to execute a low-overhead implementation of DS in architectures without adequate hardware support. In contrast, when hardware support is available, it is possible to deliver high throughput and low latency using a DS implementation with overall results superior to those of an SS approach. This superiority can be observed in situations that were traditionally favorable to SS (regular applications in homogeneous architectures) and in situations with fine-grained tasks with some degree of heterogeneity.

In addition, to solve the difficulties in percolation and scheduling, we can take advantage of the fine-grain synchronization primitives available in many-core architectures. Percolation and dynamic scheduling can be fused together in what we call *dynamic percolation* which dynamically schedules data prefetching at an appropriate time so that (1) data is available when the computation needs it and (2) the percolation operation is done when enough memory bandwidth is available.

Operand Placement	**Optimization**	**Performance (GFLOPS)**
SRAM	Static Partition	3.16
SRAM	+Register Tiling	30.42
SRAM	+Instruction Scheduling	44.20
DRAM	Percolation +Synch. Optimization +Opt. Data Movement	13.90

Table 6.1: Summary of Previous Results of MM on C64

6.1 Motivation

Several efforts have been made to study the optimization of applications [61, 45, 47, 60] for the IBM Cyclops-64 (C64) [40]. Those studies, however, show results that are still far from the theoretical maximum performance. To understand the problem, we have analyzed one of the simplest examples of Static Scheduling to find the issues that have prevented better results.

Two recent studies in the implementation of Matrix Multiplication on C64 [46, 61] have shown the effectiveness of several optimization techniques (Table 6.1). The initial implementations studied targeted SRAM and DRAM and they achieved a performance of 13.9 GFLOPS [46]. Further optimization of on-chip SRAM memory usage resulted in a performance of 44.12 GFLOPS [61].

As can be observed from Table 6.1, the maximum performance reported after several static optimizations barely surpassed one half of the peak performance with all operands in SRAM.

The implementation in our study (Matrix Multiply) was improved to 58.95 GFLOPS when Optimum Register Tiling [61] with Data-Prefetching was used. Surprisingly, this implementation is still far from the expected peak performance, even after months of optimizing carefully-written assembly code and after significant theoretical and experimental effort to find an optimal register-tiling strategy [61].

The comparatively low performance achieved – even after the carefully optimized assembly code implementation– prompted an investigation into the factors that prevented us from reaching a higher performance. To do so, we conducted an extensive and careful profiling of the application.

Two cases, both using SS, were studied in particular: A multiplication of the largest matrices that can fit in SRAM (Figure 6.1) and a multiplication of smaller matrices also in SRAM (Figure 6.2). Smaller matrices are required for implementation of matrix multiplication in DRAM doing overlapping of computation and data movement with SRAM.

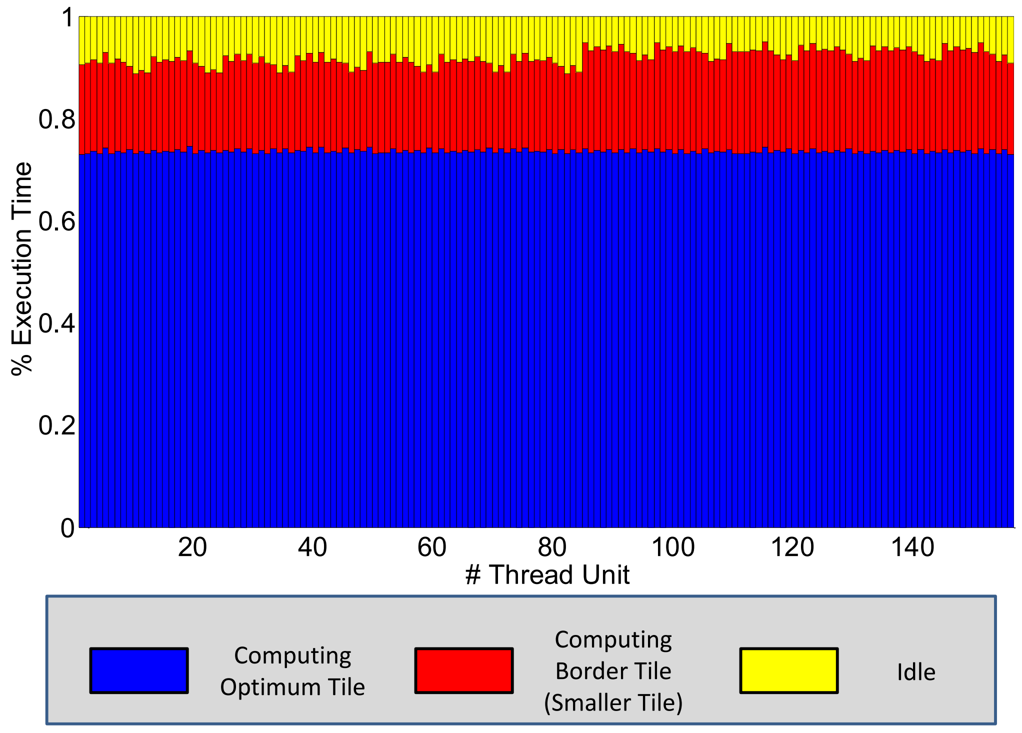


Figure 6.1: Workload Distribution for a MM of size 488×488

The following observations can be made:

- The amount of time computing tiles whose size was optimized for maximum performance does not surpass 70% of the execution time in any thread.

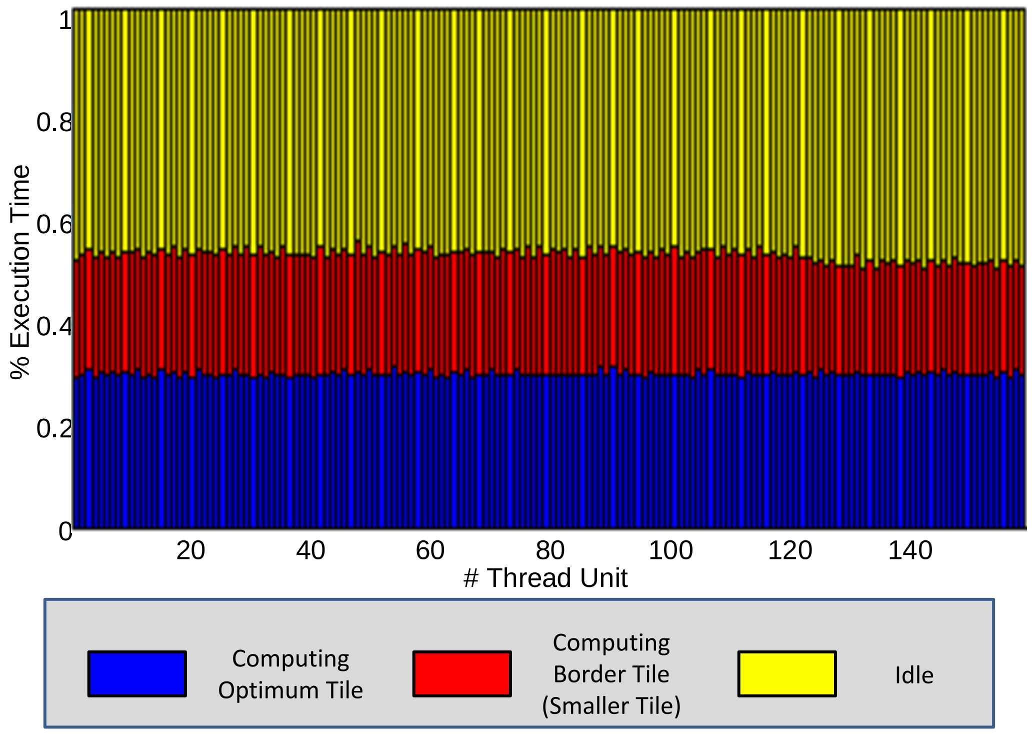


Figure 6.2: Workload Distribution for a MM of size 192 × 192

- The problem size is not always a multiple of the optimum tile size, so smaller tiles have to be included in the computation, adding to the imbalance of the system. This causes problems because either (1) the computation is partitioned into tiles, which may result in different number of tiles per thread or (2) the computation is partitioned evenly among threads, which may not be as efficient as partitioning the computation in carefully chosen tiles.
- In general, when the size of a problem decreases, the fraction of tiles that are not of the optimum size increases. This hurts the performance because smaller tasks may not fully take advantage of the available resources, even if they are optimized.
- The idle (wasted) time includes the time spent in synchronization, flow control, and scheduling. It is more than 10% in the best case and increases dramatically when the matrix size decreases.

We see that SS is not necessarily a good strategy in many-core processors, even for the case of highly regular parallel benchmarks.

6.2 Static Scheduling and Data Partitioning

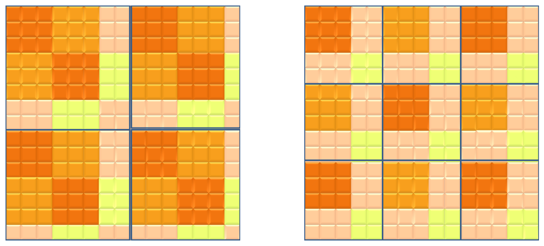
Scheduling is an important optimization for programs once the bottleneck of memory bandwidth has been removed through tiling.

Scheduling presents challenges in itself since it requires assignment of work to processors at the appropriate time, taking into account issues such as availability of resources and availability of data. The scheduling problem is complicated by the fact that the tasks scheduled to each processor are not necessarily identical. The problem seems simpler for regular and embarrassingly parallel applications, where the amount of data can be distributed uniformly between TUs, expecting similar execution times. Two main factors under the scenario imposed by many-core architectures decrease the expected performance of this static approach to the point of making it impractical even for regular applications. These two factors are: 1) shared resources and 2) size and shape of tiles.

Shared resources such as function units or bandwidth are a source of imbalance even with tasks that perform similar computations over the same amount of data. As a result, tasks may have different execution times due to competition for shared resources. This factor is critical on many-cores, where shared resources are abundant at different levels with diverse arbitration schemes.

The size and shape of the tiles greatly influence the performance of an application. Numerous publications [61, 62, 47, 60] have been devoted to the discussion of what is the optimal tile that must be used for a problem. Usually, criteria to select a good tile size is that which maximizes the ratio of computation to memory operations given some constraints such as available memory, the desired parallelism, or the number of processing units in a chip. Although tiling effectively improves the efficiency of the computation, it is not always possible to place all of a problem's data into tiles since it is frequent that the problem dimensions are not a multiple of the tile size. For that reason, in general, problems result in a combination of optimal-sized tiles and non-optimal-sized tiles.

For example, as explained in Section 5, the best strategy to compute a matrix



Partition for TU=4 Partition for TU=9

Figure 6.3: The figure illustrates the problem of partitioning. Tiles of 3×3 are optimum-sized and they result in the best performance. However, as the number of Thread Units (TUs) increase, the number of optimum-sized tiles decrease. In the Figure, a matrix of 15×15 results in 16 optimum-sized tiles when using 4 TUs, but only 9 optimum-sized tiles are available when using 9 TUs.

multiplication was to divide the computation uniformly between blocks according to the number of Thread Units (TUs) and partition these blocks into optimal-sized tiles if possible [61], even if such a partition left some non-optimal-sized tiles. Although the idea of partitioning a problem into equal work for all the TUs works well, it may still result in some non-optimal-sized tiles left because the problem size is not necessarily a multiple of the tile size used. These remaining non-optimal-sized tiles result in poor performance during execution. Two factors exacerbate the presence of slow non-optimal tiles: 1) an increased number of TUs working in parallel and 2) a limited amount of on-chip shared memory available to host the data. These two conditions are evident on a many-core environment and they will ultimately limit the ability of an application to reach peak performance. Figure 6.3 shows a simple example where the amount of data that belongs to non-optimum-sized tiles (in light colors) increases when the number of TUs increases and the amount of data shared is limited.

6.3 Percolation

Uninterrupted computation by the processing units in a many-core chip requires data to be available continuously. Percolation is the process by which data is moved across the levels of memory hierarchy to meet the necessities of locality for computation.

Percolation is related to data prefetching in that both achieve the same objective. As opposed to conventional data prefetching, percolation operations are expressed as tasks on their own, with precedence relationships with other computational tasks and with restrictions to available resources such as bandwidth or on-chip memory space.

One of the main challenges of percolation is the decision of when to move data. Data moved too early will occupy buffer space for an unnecessary amount of time while data moved too late will stall the computation that depends on it. It is difficult to know *a priori* when percolation should be done. As explained in the previous section, not all tiles are of the same size, and not all tiles take the same amount of time, even when they perform similar amounts of computations.

For example, Garcia's approach to optimize Matrix Multiply failed to achieve peak performance [61], in part, due to the difficulties of scheduling the percolation operations at an appropriate time.

An interesting research direction is the development of strategies to schedule percolation operations at the right time.

Taking as example the Matrix Multiply algorithm, the majority of the processing is computing tiles. Therefore, computation deserves special attention. The Instruction Scheduling made by Garcia et. al. [61], only partially hides the latency incurred while fetching the operands a and b from SRAM to registers.

To eliminate stalls due to latency, we used percolation of operands a and b into registers using loop unrolling in the calculation of the tile. Figure 6.4 shows the pseudo-code for calculating one tile C of size $L_1 \times L_2$ with and without Percolation. Uppercase variables are arrays in share memory and lowercase variables are arrays allocated in registers.

Unrolling the loop once and carefully using instruction scheduling, we can significantly increase the time between the issue of loads for operands a and b and computations where they are required through carefully scheduled percolation operations. In [61] we showed that $L_1 = L_2 = 6$ consumes a bandwidth below crossbar saturation. A basic analysis of the unrolled loop show us that the required number of registers

```
S1:  $c[1..L_1][1..L_2] = 0$
S2:  for $k = 1$ to $m$, $k++$
S3:     $a[1..L_1][1] = A[i..i + L_1][k]$
S4:     $b[1][1..L_2] = B[k][j..j + L_2]$
S5:     $c[1..L_1][1..L_2] += a[1..L_1][1] \times b[1][1..L_2]$
S :  end for
S6:  $C[i..i + L_1][j..j + L_2] = c[1..L_1][1..L_2]$
```

(a) Without Percolation

```
S1 :  $c[1..L_1][1..L_2] = 0$
S2 :  $a[1..L_1][1] = A[i..i + L_1][1]$
S3 :  $b[1][1..L_2] = B[1][j..j + L_2]$
S4 :  for $k = 1$ to $m$, $k++$
S5 :     $a[1..L_1][2] = A[i..i + L_1][k + 1]$
S6 :     $b[2][1..L_2] = B[k + 1][j..j + L_2]$
S7 :     $c[1..L_1][1..L_2] += a[1..L_1][1] \times b[1][1..L_2]$
S8 :     $k++$, if $k == m$ then break
S9 :     $a[1..L_1][1] = A[i..i + L_1][k + 1]$
S10:     $b[1][1..L_2] = B[k + 1][j..j + L_2]$
S11:     $c[1..L_1][1..L_2] += a[1..L_1][2] \times b[2][1..L_2]$
S : end for
S12:  $C[i..i + L_1][j..j + L_2] = c[1..L_1][1..L_2]$
```

(b) With Percolation

Figure 6.4: Algorithm for computing a tile of C with size $L_1 \times L_2$

increases from $L_1 + L_2 + L_1 \cdot L_2$ without Percolation to $2(L_1 + L_2) + L_1 \cdot L_2$ with Percolation.

Of the available 63 register of C64, 5 registers are used for pointer and indexes leaving 58 registers for computation. A careful live variable analysis shows that registers used to store a and b vectors in one iteration can be reused in another one. Therefore, we were able to retain a 6×6 tile without spilling register.

With respected to Instruction Caches (I-Caches). Instruction misses will produce costly stalls in the execution while instruction are accessed from main memory. The case for many-core architectures imposes additional constrains because I-caches are shared and executing the same code by the processors that share the I-cache is desirable. For the particular case of MM, the most used code is the scheduling and the code for computing a tile.

In order to minimize the I-misses, we can apply two strategies. The first one is to align the functions of the scheduling and Tile computation with the I-Cache block size, minimizing the number of cache blocks for that code. The second one is to apply Instruction Percolation (IP), it can be done executing the scheduling code and Tile computation code prior to the execution of the whole MM, allocating that code in the shared I-Caches reducing the excessive number of I-misses on the first iterations.

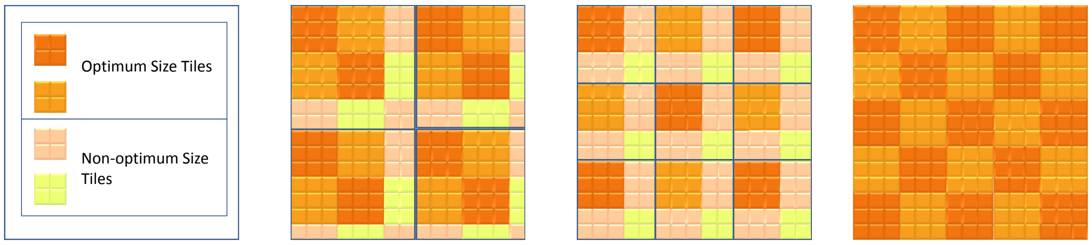
6.4 Dynamic Scheduling for Fine Grained Parallelism

Finer task granularity is one of the ways in which sufficient parallelism is provided for many-core processors. We strive to address these challenges by making observations that motivate a deeper understanding of the trade-offs that are normally left unconsidered by SS, even in the face of new many-core architectures. We will show how DS is a better alternative due to the disadvantages of SS, even in the presence of fine-grained tasks. First, we will discuss the impact of a fine-grained task partition on overall performance. Second, we will show the implications on load balancing and performance in a scenario where resources are shared between PEs. Finally, we will study the required characteristics for a low overhead DS and how to implement it efficiently for a set of similar tasks.

6.4.1 Fine-grained task partitioning

In this section, we will explain that, under certain conditions, DS can result in faster execution of programs because it can partition the work into better tasks. At this point we consider ideal conditions of no scheduling overhead, no shared resources, and tasks with very similar execution times. We will expand this reasoning to include more realistic scenarios in the following sections.

The problem faced by SS is the trade-off between load balance and the efficient processing of tasks given by the partitioning of data into blocks for threads, and the further partitioning of blocks into tasks for optimal execution. On one hand, an SS



(a) Legend (b) SS for $P = 4$ (c) SS for $P = 9$ (d) Without Blocks

Figure 6.5: Partition Schemes for a matrix C of 15×15 with tiles of 3×3

that maximizes load balancing will distribute equally sized blocks among all processors. Unfortunately, the partitioning of blocks may result in non-optimum tiles at the boundaries of blocks, decreasing performance. This is even worse for situations where the ratio between the block size and task size decreases due to a limited amount of on-chip memory or an increasing number of processing units results in smaller tasks. These two cases are particularly evident on modern many-core architectures. Figure 6.5 illustrates different scenarios for the amount of data in border tiles (highlighted). The worst scenario is when the number of processing elements (PEs) is increased and the best one is when blocks are not used. On the other hand, an SS that uses just tasks will decrease the penalty due to border tiles but will decrease load balancing in cases where the number of tasks is not a multiple of the number of processing elements.

In contrast, DS has the ability to deal with these unbalanced scenarios that decrease the size of border tiles. The partitioning of data into blocks is not required and each processing element request a new task as soon as it finishes the previous one. Assuming a similar overhead, DS will be able to produce equal or better scheduling than SS given the fact that the assignation of task is given at run time. Further analysis of how to deal with shared resources and how to reach a competitive overhead on DS will be discussed in the following sections.

6.4.2 Load Balancing in Scenarios with Shared Resources

We have analyzed an ideal scenario where tasks of the same size would take the same amount of time to complete. Unfortunately, this does not take into account the indirect interaction between tasks given by shared resources involving arbitration of third parties and starvation.

Some functional units are shared in order to diminish the required chip area and power consumption, saving room to include, for example, more PEs that can increase the overall performance. Examples of these functional units are Floating Point Units and special purpose units, such as specialized DSP blocks. Sharing these resources impose limitations, especially in SIMD programs. While one PE is using the shared resource, others are stalled waiting for that resource. A context switch or task migration can alleviate the impact, but this behavior introduces unexpected variations in the cost of tasks (e.g. time for completion).

Memory, the most commonly shared resource, acts as an efficient method of communication between PEs. Several techniques have been employed to improve the way memory resources are shared, such as multiple memory banks to allow simultaneous access to variables in different banks and caches to reduce the number of memory requests. It is common that both memory banks and caches are shared by several PEs to reduce the complexity of the architecture at the price of slightly reducing the benefit.

Interconnects provide the mechanism for accessing some of these resources. According to their type and size, interconnects have complex arbitration rules that control the use of these resources. Shared memory and its interconnections with the PEs are normally the biggest source of uncertainty with respect to a task's cost. The growing number of PEs has made these structures very complex and their modeling involves stochastic processes [63, 64], making the cost of a task a random variable that is also a function, among other things, of other parameters in the architecture.

Statistics about the average costs and variances of tasks can be found to adjust the parameters of the SS. However, there are several aspects that limit the effectiveness of this approach: 1) The number of tasks is limited, so the disparities made by the variations in cost may not be overcome. These limitations may be due to the nature of the problem itself. 2) The load on each port of the crossbar or memory bank may vary, particularly when the size of a block of memory is not a multiple of the memory line size. A critical case would be where one memory bank is accessed by all processors. 3) The scenario becomes even more complex when we model the process as non-stationary, taking into account that memory transaction patterns can change in the same application over time.

In the end, the simple model for SS is just an approximation that works very well for scenarios where the demand for shared resources is low because the stochastic component is negligible. For high performance scenarios, where these resources are required to be used at full capacity, congestion and arbitration require a more accurate model. However, a highly accurate model that considers a significant number of the variables that describe the behavior of the architecture, including the interactions of shared resources and their arbitration, is impractical for SS given the difficulty of producing such a model, the overhead of using the model to compute an optimal partition and the intrinsic limitations of the model.

Alternatively, DS performs the distribution of tasks at runtime allowing it to deal with all possible variations. With a competitive overhead, DS will be able to deliver better performance than SS, even for highly regular applications running under the previously described conditions.

6.4.3 Low Overhead Fine grained Dynamic Scheduling

Dynamic Scheduling has been explored extensively for Instruction Level Parallelism and its advantages are well known [3, 65]. However, its implementation has traditionally required special hardware support.

For Task Level Parallelism, software implementations are preferred. When all tasks to be executed are similar and the parallel architecture is homogeneous, SS has been the preferred choice because the overhead for scheduling is only paid once and is largely independent of the data size whereas the overhead of dynamic scheduling grows linearly with the data size.

The overhead of DS can negate any advantages over SS if the implementation is not well designed. Unfortunately, special hardware support at the functional unit level is also impractical for general purpose many-core architectures.

Dynamically scheduling multiple, similar, tasks can be achieved with a single integer variable that is sufficient to uniquely identify a task in its set. A piece of code that implements Dynamic Scheduling in this manner is showed in Figure 6.6.

```
// Globals
int TotalNumTasks;
int TaskIndex;
// Scheduling Function
int GetNewTask(int* Index){
  return (atomic_add(&Index, 1));
}
...
// Scheduler algorithm on each PE
  int i;
  i = GetNewTask(&TaskIndex);
  while ( i<TotalNumTasks ){
    Execute_Task(i);
    i = GetNewTask(&TaskIndex);
  }
...
```

Figure 6.6: Code Fragment for a DS implementation

The bottleneck of this algorithm is the function `GetNewTask(.)`. Specifically, it is the serialized access over the variable `TaskIndex`. We will use the throughput μ (maximum number of requests that can be serviced per unit of time) over the variable `TaskIndex` to determine the tradeoffs between task granularity and number of PEs. The lower bound for the size of a task can be obtained by considering that during the execution of a task, on average, all other $(P-1)$ processors will request one task. Since the duration of the execution of a task T is given by $f(T)$, the lower bound for the average size of a task is

$$f(T) \geq \frac{P-1}{\mu} \tag{6.1}$$

Equation 6.1 shows that, as the number of PEs increases, a matching increase in throughput is required to guarantee scalability. Also, fine granularity of optimized tasks on a many-core environment requires the highest maximum throughput for the variable `TaskIndex` in order to avoid contention and lost performance under DS.

Unfortunately, implementations of the function `GetNewTask` that use locks or "inquire-then-update" approaches have very low throughput [39, 66]. The main reason is that, for a lock implementation, the algorithm will 1) obtain a lock, 2) read and update the variable `TaskIndex` and 3) release the lock. A lock-based implementation of `GetNewTask` needs at least two complete roundtrips to memory, limiting its throughput to $\mu = \frac{1}{2q}$ where q is the minimum latency for a memory operation. Similarly, an "inquire-then-update" approach, such as Compare and Swap (CAS), requires `TaskIndex` to remain unchanged for at least 2 memory roundtrips, resulting in the same throughput as in the previous case.

We propose taking advantage of the support provided by *in-memory* atomic operations. In this case, each memory controller has an ALU that allows it to execute atomic operations directly inside the memory controller, without help from a thread unit, avoiding unnecessary roundtrip delays. In this case, the use of the *in-memory* atomic addition allows the throughput to be limited only by the time k taken by the memory controller to execute the operation, resulting in a throughput of

$$\mu = \frac{1}{k} \tag{6.2}$$

Cyclops-64 (C64) is an example of a many-core architecture that provides adequate hardware support for Dynamic Scheduling. In C64, implementations that use *in-memory* atomic additions enjoy a significant throughput increase because atomic operations in C64's memory controller take $k = 3$ cycles, whereas a roundtrip to on chip shared memory requires $q = 30$ cycles. It is a theoretical improvement of $20X$ over the throughput obtained using simple software implementations. In practice, the throughput will be lower because the memory controller is shared with the actual computation of the tasks. Nevertheless, this high throughput allows the overhead of DS to remain competitive with the traditional SS approach.

6.4.4 Example: On-chip SRAM Dense Matrix Multiplication

SS is suboptimal because it does not consider two main sources of imbalance in a many-core environment: 1) The amount of work is a function of how the block is tiled and what fraction of tiles does not have optimum size. 2) Possible stalls due to arbitration of shared resources.

The unpredictable effects of resource sharing are a formidable challenge for SS. A static block partition exacerbates problems, especially when the number of Processing Elements (PE) is increased. Despite the simplicity and regular behavior in computation and data access of MM, static techniques cannot overcome these problems. At that point, DS arises as a feasible solution able to alleviate the overhead and scalability problems of SS.

We propose a work-stealing approach for MM in on-chip (SRAM) where the computation of optimum size tiles in matrix C are scheduled dynamically using atomic in-memory operations. The proposed DS has the following advantages over SS:

1. A carefully designed DS can be managed with low overhead using atomic *in-memory* operations, specifically, atomic increment/decrement. In-memory operations can complete in very few cycles (e.g. 3 cycles on C64), allowing more requests to be completed per unit of time and avoiding unnecessary roundtrips to memory.

2. The dynamic approach load-balances optimally in the presence of stalls due to arbitration of shared resources, increasing the efficiency by keeping all threads working.

3. Since the work unit is the optimal size tile, the number of non-optimum size tiles is minimized and it does not depend on the number of PEs.

The first and second advantages suggest that DS will have a better performance than SS and the maximum performance will be reached when the amount of data is big enough to feed all processors in parallel.

The third advantage implies that DS will overcome SS especially when the size of matrix m is limited or the number of PEs increases. Thanks to the in-memory computation capabilities of C64 there is little contention and overhead due to the use of DS as described in Figure 6.6.

Due to the advantages explained before, it is feasible to expect a better scalability when using DS for a broad range of values of m and PE.

6.5 Dynamic Percolation

Our target operation is dense Matrix Multiplication ($C = A \times B$) for matrices with size $m \times m$. For bigger sizes, we propose a separation of the problem into two orthogonal subproblems: 1) optimizing Matrix Multiply in SRAM moving operands between SRAM and Registers and 2) moving data between DRAM and SRAM.

To extend the matrices to DRAM we simply partition matrices A, B and C into $n \times n$ blocks $A_{i,k}$, $B_{k,j}$ and $C_{i,j}$ that fit in SRAM. This is similar to the blocking performed in traditional cache hierarchies. Yet, C64 has no automation of data movement by caches or DMA engines but instead must use Thread Units (TUs) associated to Processing Elements (PEs). This means we have a trade off between compute and data movement.

Each block of C is calculated by

$$C_{i,j} = \sum_{k=0}^{\frac{m}{n}-1} A_{i,k} \cdot B_{k,j} \tag{6.3}$$

Considering the limitation of bandwidth in the crossbar and the unpredictable effects of resource sharing, we must devise a schedule that considers both computation and data movement efficiently.

For that reason, we must use TUs to transfer the data. These computational TUs must be orchestrated with data movement TUs to enforce the data dependencies:

work cannot be done before a matrix is loaded and a matrix cannot be unloaded until work using it is completed. Further, TUs performing data movement should help with computation if there is no data to move.

A straightforward static schedule for the MM algorithm detailed previously would synchronize tasks using barriers and would parallelize each task. To compute the whole matrix C, the tasks detailed in Figure 6.7 would be executed $\frac{m^2}{n^2}$ times.

```
1 :  Initialize $C_{i,j}$ to 0 on SRAM
2 :  Compute the block $C_{i,j} = \sum_{k=0}^{\frac{m}{n}-1} A_{i,k} \cdot B_{k,j}$.
2 :  This can be subdivided in 2 subtasks:
2a:     Copy $A_{i,k}$ and $B_{k,j}$ from DRAM to SRAM.
2b:     Compute a partial $C_{i,j}$ and accumulate.
3 :  Copy Back the block $C_{i,j}$ calculated.
```

Figure 6.7: Tasks for computing one block $C_{i,j} \in C$

Although task 2b is implemented efficiently, as described in section 5, a direct implementation of task 2, with barriers between tasks 2a and 2b, would waste resources while TUs are waiting on barriers. Further, it would be inefficient for all TUs to copy data at the same time given the limited DRAM bandwidth. A dynamic scheduling approach replaces the barriers with finer-grained signals while still enforcing data dependencies.

We introduced *Dynamic Percolation*, where data movement tasks and computation tasks were assigned dynamically. Helper Threads (HT) are in charge of the data movement tasks and Computation Threads (CT) are in charge of the computation tasks. Computation and data movement tasks are overlapped by a pipelined schema using a double buffer in SRAM (e.g. buffers $F1$ and $F2$). Moreover, the distribution of computation tasks and data movement tasks will vary in the course of Dynamic Percolation. A set of simple rules for creation (based on data dependencies) and issue of tasks helps the dynamic scheduler keep threads working efficiently on a computation task or a data movement task:

1. Task Creation rules:

   (a) A set of computation tasks on buffer $F1$ is created and ready to be fired when all the data movement tasks for buffer $F1$ are complete. The same is true of buffer $F2$.
   (b) A set of data movement tasks for buffer $F1$ are created and ready to be fired when computation is complete for the data buffer $F1$. The same is true of buffer $F2$.

2. Task Issue rules:
   (a) A set of tasks (computation or data movement) are scheduled dynamically between the threads that belong to a set of that type of task (CT or HT).
   (b) When a HT has finished and all data movement tasks of a buffer have been issued, it becomes a CT for the current actively computed buffer.
   (c) There is a maximum number of HT that can run in parallel to avoid contention on DRAM bandwidth.
   (d) When a CT has finished and all tasks of that set (e.g. on buffer $F1$) have been issued, it becomes a HT for the set of data movement tasks on that buffer when that set is created presuming that the maximum number of HT has not been reached. Otherwise it becomes a CT for the next set of computation tasks (e.g. on buffer $F2$).

The Dynamic Scheduler for each set of tasks is implemented by using atomic in-memory operations, specifically, in-memory atomic addition. The main advantage of this implementation is the low overhead given by the low latency of in-memory operations. They avoid unnecessary roundtrips to memory and they provide the necessary synchronization thanks to the atomicity supported by the hardware. Under normal conditions (e.g. no unexpected failures of any components in the chip) a possible scenario for stalls is given by rule 2d: a CT stalls when it becomes a CT of the next set of computation tasks (e.g. buffer $F2$) while the buffer's data movement tasks have not finished. This condition can be easy solved if the size of the HT set is large enough to guarantee that the data movement tasks finish before their associated computation tasks. This parameter is architecture dependent and it is related to the compute/bandwidth ratio and the size of on-chip buffers. A rough starting point estimation is given by eq. (6.4), where N_{HT} is the maximum number of helper threads:

$$\frac{DataMoved}{Bandwidth(N_{HT})} \leq \frac{FLOPS\ Computed}{Performance(P - N_{HT})} \tag{6.4}$$

The tasks in Figure 6.7 can be classified into two groups: 1) Computation tasks (2b) and 2) Data movement tasks (1, 2a, 3). Also, there is a hierarchy of tasks. At the highest level, tasks $1-3$ are related with blocks $C_{i,j}$ (Initialize, Compute, Copy Back) while at the next level down, tasks $2a$ and $2b$ are specific for computing one block $C_{i,j}$ using several blocks $A_{i,k}$ and $B_{k,j}$ (Copy, Compute). We will analyze each level separately, starting with the inner level: tasks for computing a block $C_{i,j}$, and continuing with the outer level: tasks for computing the whole matrix C.

6.5.1 Computation of one block $C_{i,j}$

Data is percolated as shown in figure Figure 6.8. Tasks 2a map to the data movement tasks and tasks 2b map to the computation tasks. In the initialization step, we create the first set of data movement tasks and create the second set when all data movement tasks in first set have been issued.

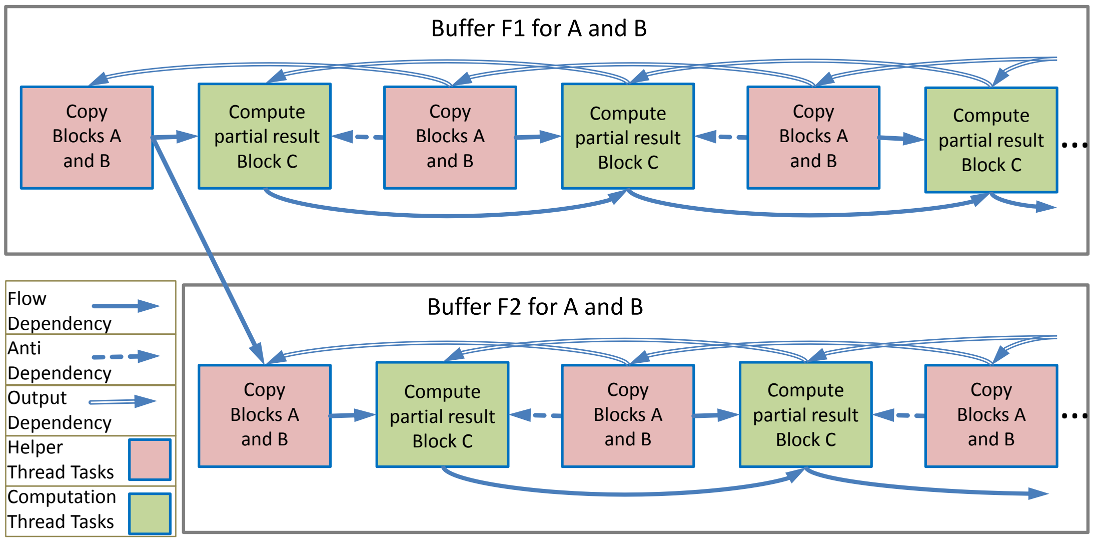


Figure 6.8: Dynamic Percolation for Computation of one block $C_{i,j}$

6.5.2 Computation of matrix C

The process of computing the whole matrix involves a *Hierarchical Dynamic Percolation*, where tasks 2 are a subset of the percolation model for tasks 1 - 3 as shown in Figure 6.9. However, at this level, all tasks in task 2 are considered computation tasks. There are two data movement tasks (1 and 3) where task 1 of the next outer loop iteration is dependent on task 3 of the current iteration. In the initialization step, we only initialize $C_{i,j}$ and do not copy it back until the first computation task is completed.

Under the assumption that the number of HTs at both levels have been chosen properly to do the data movement tasks in less time than the computation tasks, the Dynamic Percolation for MM not only allows runtime redistribution between helper threads and computational threads to achieve better utilization of TUs, but also its dynamic behavior can efficiently manage the unpredictable effects of resource sharing (e.g. arbitration of crossbar network ports and limited off-chip bandwidth). This is a challenging problem on many-core architectures that, as discussed previously, SS cannot overcome.

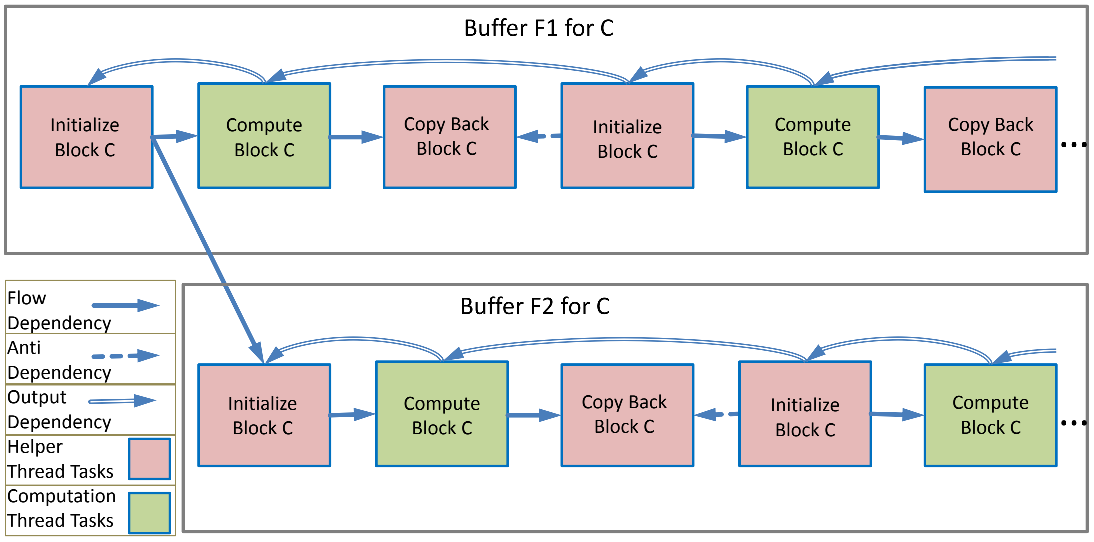


Figure 6.9: Dynamic Percolation for Computation of matrix C

The performance of the DRAM MM with respect to the SRAM MM is expected to be slightly lower because now some threads are not doing computation and the cost of data movement has to be included. This cost depends on the maximum number of HTs allowed at each task level, the bandwidth for memory transfers, and the size of blocks on SRAM.

6.6 Experimental Evaluation

In this section, we analyze the advantages of DS over SS for very regular workloads under the presence of shared resources and hundreds of Processing Elements (PEs). We have illustrated different scenarios with fine grain tasks in order to compare the traditional SS and a low overhead DS. Our results show that applications with many similar tasks scale better, and can take advantage of a low overhead DS, when the PEs are sharing resources and the amount of tasks is limited.

6.6.1 Experimental Testbed

We have chosen C64, a many-core processor architecture described in Section 4, as the testbed architecture because it has a large number of processors sharing many diverse resources including, but not limited to, an on-chip memory, a crossbar switch and shared FPUs. In addition, it supports in-memory atomic addition, an essential component for a low overhead DS implementation as described in section 6.4.3.

Our experiments were compiled with ET International's C64 C compiler, version 4.3.2, with compilation flag -O3. C64 processor chips are, as of the writing of this paper, available only to the US Government. For that reason, we ran our experiments with FAST [67], a very accurate simulator that has been shown to produce results that are within 10% of those produced by the real hardware. The simulator includes all the behaviors related to the arbitration of shared resources.

We ran three different tests. The first is a microbenchmark that performs a memory copy of a vector in shared memory and computes a checksum on the elements of the vector. The second is a highly optimized Dense Matrix Multiplication using both on-chip and off-chip memory. The third test is a Sparse Vector Matrix Multiplication with variable parameters such as sparsity and variance of number of elements between columns. All benchmarks were implemented with an SS strategy that distributes work uniformly and a low-overhead DS that uses in-memory atomic addition.

6.6.2 Memory Copy microbenchmark

The tasks in this microbechmark process 256 bytes of data from on-chip memory as follows: First, the PE copies a chunk of data from on-chip memory to local memory. Then, it computes the checksum of the bytes and the chunk is stored back to another location in on-chip memory. Note that all tasks perform the same amount of work but the arbitration of shared resources, like the crossbar switch, can result in varying performance as described in Section 6.4.2. We report the relative speed up of the DS approach with respect to its SS counterpart using the same number of PEs (Thread Units). We use different numbers of tasks to study and compare the behavior of SS and DS.

Figure 6.10 clearly shows the trade offs between DS and SS with respect to the number of PEs. As expected, when the number of PEs is small, DS cannot compete with SS and demonstrates a slowdown, with the worst case being 27% for 32 PEs. After 24 PEs, a smaller number of tasks start to give better performance with DS and after 48 PEs all the datasets favor DS. The maximum relative speed up is 137%, which is reached with the smallest problem size and the maximum number of PEs.

We further studied the scalability of SS and DS with the maximum number of PEs allowed. Figure 6.11 shows the number of tasks per microsecond using different numbers of tasks. We note how DS scales better for cases with a limited number of tasks, quickly reaching the limit imposed by crossbar congestion. At the same time, SS shows poor performance when compared to DS for every case.

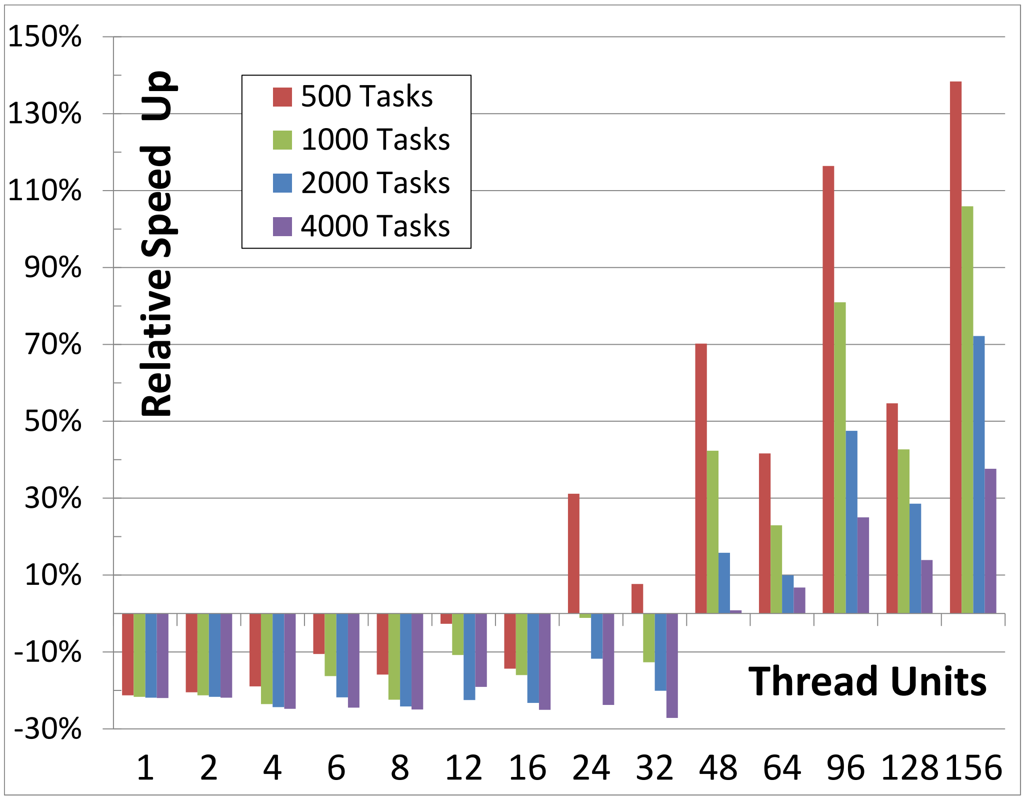


Figure 6.10: Relative Speed Up of DS vs. SS

6.6.3 Dense Matrix Multiplication

Dense Matrix Multiplication (DMM) exemplifies the type of highly regular and embarrassingly parallel application where SS seems to be the better choice over DS. We use, for our baseline, the Highly Optimized DMM for C64 using on-chip memory as described in Section 6.1 and detailed in [61]. We further increased the performance to 58.95 GFLOPS by using the Percolation explained in Section 6.3. Based on the observations made in Section 6.4, we implemented a DS for DMM using the same optimized register tiling described in [61]. With the implementation of DS, the maximum performance and scalability with respect to the number of PEs (Thread Units) increased significantly, as detailed in Figure 6.12.

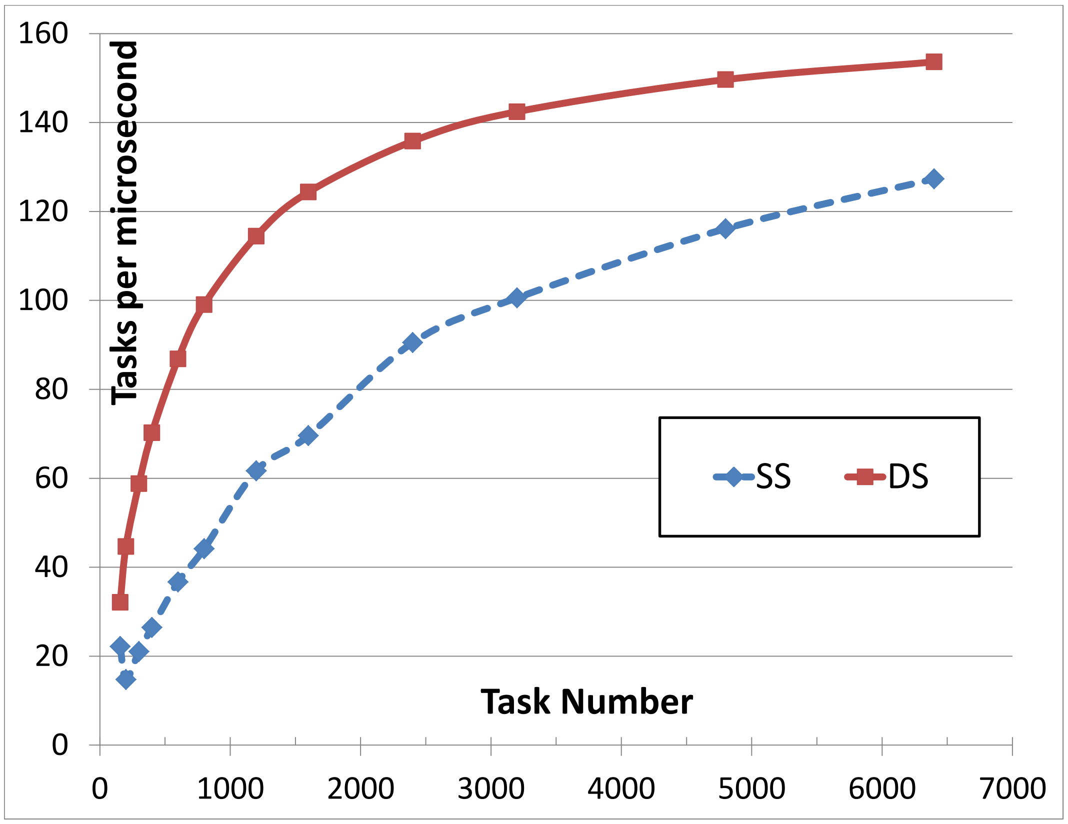


Figure 6.11: Scalability for 156 PEs

The maximum performance reached is 70.87 GFLOPS, which is 88.86% of the theoretical peak performance. It is important to note the highly linear scalability with the number of PEs whereas the SS implementations start to show problems after only a hundred PEs. We further studied the scalability with respect to the matrix sizes. Figure 6.13 shows that the performance of DS increases significantly for smaller sized problems, with near maximum performance being reached using matrices of sizes 200×200. Note that the optimized SS version would be able to reach a slightly better performance than the DS version, given a suitably large problem size because of the constant overhead of SS. However, on-chip memory places an upper bound on the problem size making DS preferable for all implementations that use on-chip memory

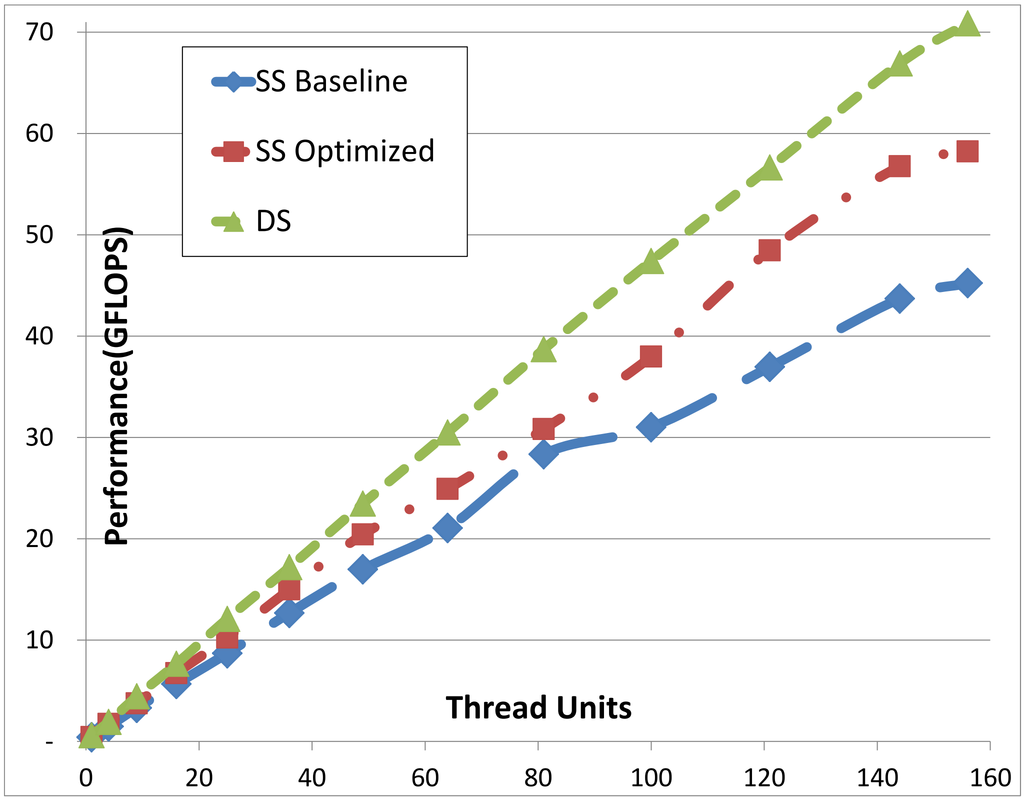


Figure 6.12: Performance for a DMM of size 486 × 486

only.

We also studied the impact of the scheduling with larger matrices using off-chip memory. Because C64 has a software managed memory hierarchy, the programmer is in charge of the data movement between off-chip and on-chip memory. In order to sustain the performance reached in on-chip memory, overlapping of computation and data movement was used by implementing a double buffering schema. We determined, experimentally, that 8 PEs dedicated to data movement was enough to keep the remaining PEs working on computation.

Two versions of the DMM were implemented. In the static version, all tasks (computation and data movement) were determined and assigned statically from the

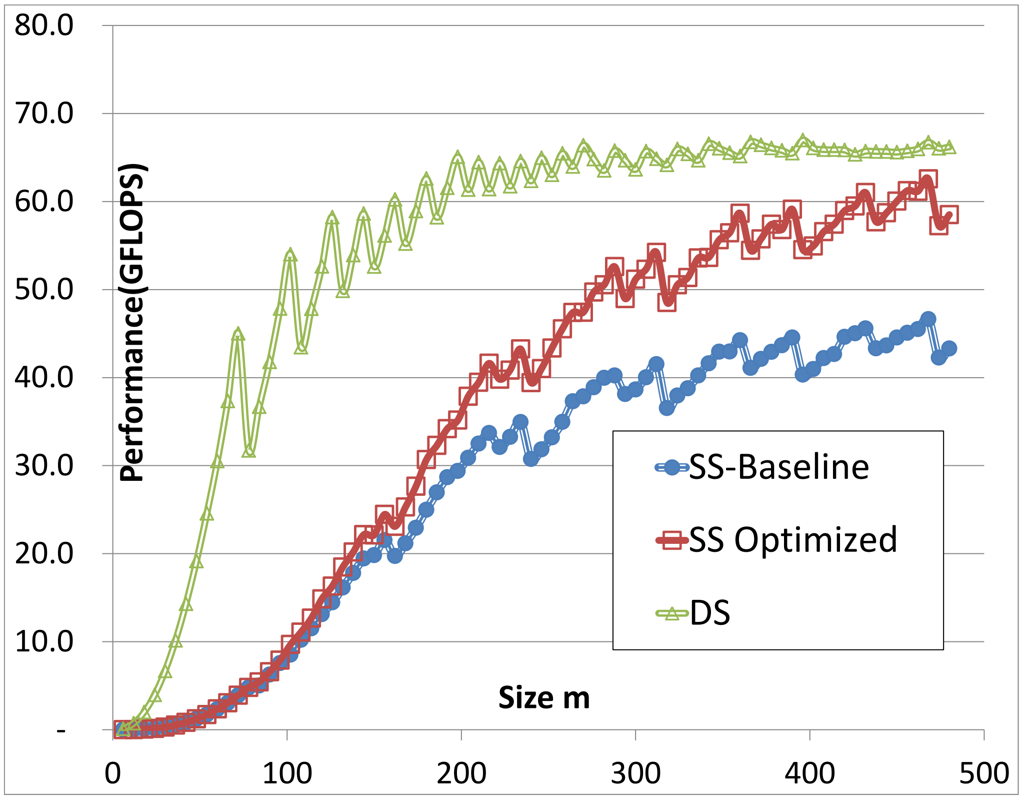


Figure 6.13: Scalability for a DMM with 144 PEs

beginning of execution. The necessary synchronization between tasks was performed using the low latency hardware barriers available on C64. In the dynamically scheduled version, tasks are available after satisfying their dependencies in a dataflow inspired manner [68] with a Dynamic Percolation that takes advantage of the in-memory atomic operations available in C64.

The results in figure 6.14 show the high scalability and excellent performance reached by the Dynamic Scheduling implementation, whereas the Static version is not able to surpass half the theoretical peak performance of C64. Furthermore, the scalability of the SS implementation decreases after 120 PEs.

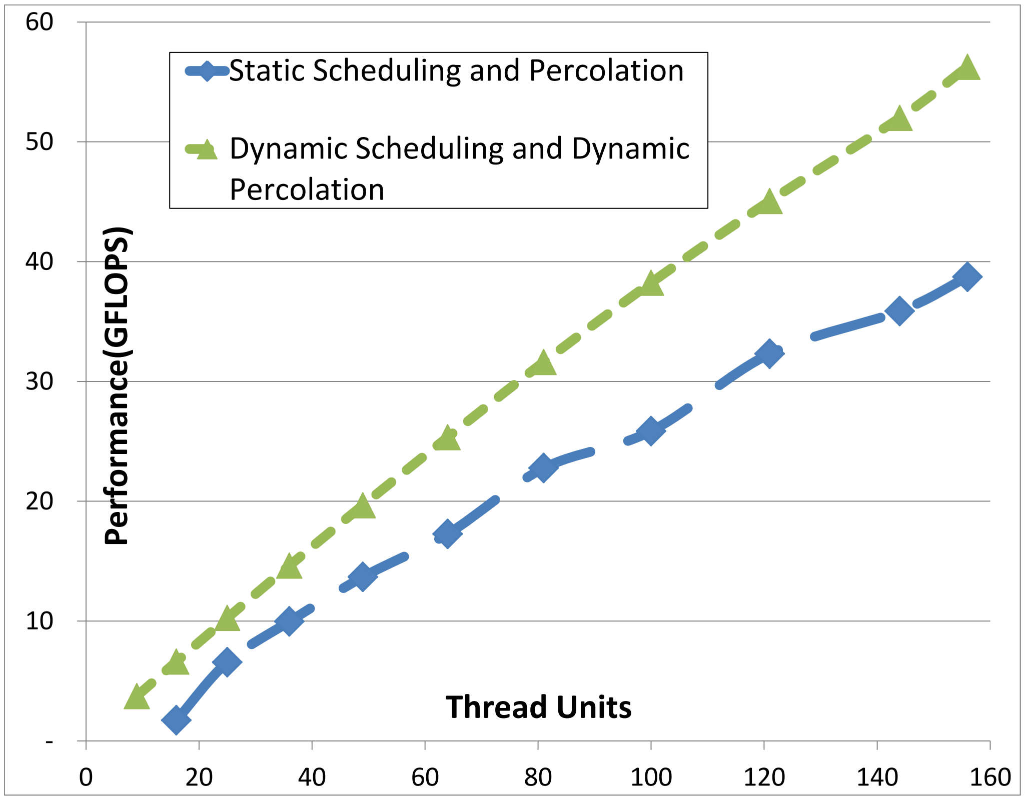


Figure 6.14: Scalability for a DMM of size 6480 $\times$ 6480

6.6.4 Sparse Vector Matrix Multiplication

Sparse linear algebra applications present additional challenges to their dense counterparts, including variable memory access patterns and other difficulties related to the particular structure of the sparse matrices. We use the Sparse Vector Matrix Multiplication (SpVMM) defined by equation 6.5 where A is a sparse matrix of size $m \times n$, and v and w are vectors of lengths m and n respectively.

$$w_j = \sum_{A_{i,j} \neq 0} v_i A_{i,j} \tag{6.5}$$

The sparse matrix A is stored using the Compressed Spare Column format (CSC). A task is defined as the computation of one element of w. Two parameters were varied to explore different behaviors of the SpVMM: The sparsity s varies in the range $[0-1]$ and defines the number of non-zero elements. The non-zero elements are distributed uniformly across columns with a normalized variance u in the range $[0-1]$. The matrix is generated randomly without any particular spatial locality

Figure 6.15 shows the relative speed up of DS with respect to its SS counterpart with the same characteristics. All the matrices have $n = 400$. The results are reported for different sizes of the tasks m and sparsity of the matrix s. In addition, the experiments were made using 3 possible values of the normalized variance $u = \{0.1, 0.5, 0.9\}$. The results illustrate how, with a high variance of task sizes, DS overcomes SS even with few PEs. If the variance between tasks is decreased, SS has better performance than DS when the number of PEs is small but SS cannot scale properly when the number of PEs increases. Even in the case of very similar tasks ($u = 0.1$), DS has higher performance than SS for 128 PEs.

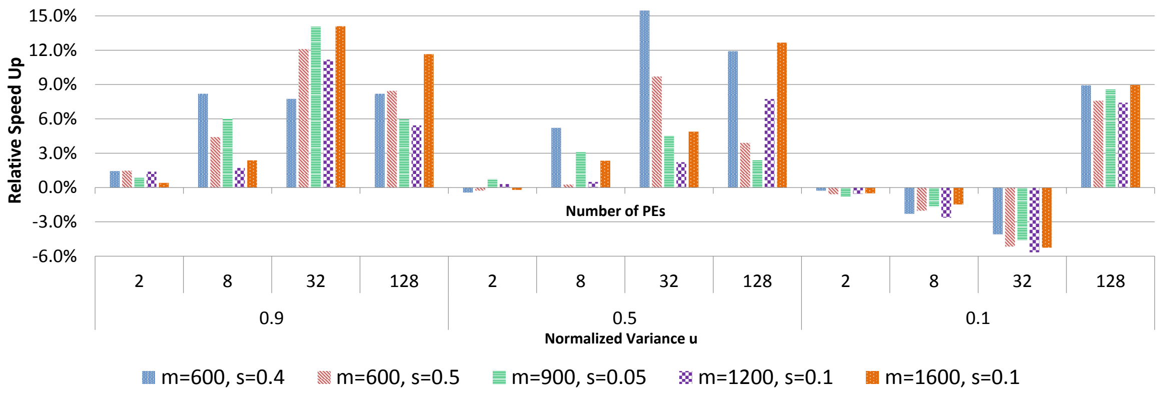


Figure 6.15: Relative Speed Up of DS vs. SS for SpVMM

Chapter 7

PERFORMANCE MODELING OF MANY-CORE ARCHITECTURES UNDER DYNAMIC SCHEDULING AND RESOURCE CONSTRAINTS

As explained previously in chapter 6, new trends in computer architecture have posed new paradigms in the area. The race to peta-scale and exa-scale computers requires a challenging increase to hundreds or even thousands of independent processing elements inside a single chip and these requirements have motivated several changes on modern many-core architectures at both the hardware and software levels such as simplification of individual elements in processor chips in current many-core processors and new programming models.

The scenarios found in current many-core processors have prompted new programming models that look for better opportunities to exploit parallelism. For example, the limited amount of on-chip memory, shared by an increasingly large number of processors, has motivated the use of finer-grained tasks. Synchronization mechanisms have become more important. Asynchronous and data-driven models have increased the variety of mechanisms employed. Program execution models are evolving from legacy models, such as OpenMP [69] and MPI [70], to data-flow oriented execution models with finer granularity, such as the codelet execution model [31], SWift Adaptive Runtime Machine (SWARM) [36] and the Time Iterated Dependency Flow Execution Model (TIDeFlow) [71, 72], to exploit parallelism for extreme-scale machines. These new execution models require performance modeling tools in order to leverage the design process of new applications and to determine better strategies to exploit task parallelism on this new type of architectures.

There are several approaches for performance models that achieve varying levels of accuracy at the expense of increased complexity. For example, Dwork's very powerful mathematical model [73] has limited applicability to large systems given its complexity. King's simpler model [74], on the other hand, uses Timed Petri Nets to model coarse programs but achieves limited accuracy. Yet other approaches, such as Anglano's [75], focus on modeling specific details, such as communication, and do not attempt to address a more general case.

The evolution to multi-core and many-core chips has imposed new challenges for the traditional methods used to model computer architectures. In particular, performance modeling under resource coordination conditions remains challenging, especially with fine-grain task parallelism under new execution models. To this point, we have explored timed Petri nets [76] as a way to model many-core programs with resource coordination conditions. Our target is the co-design of hardware and software, the modeling of algorithms and the impact of particular optimizations under an environment constrained by the architecture.

We propose applying the concept of timed Petri nets [76] to model the performance of parallel applications on new many-core architectures. The applications that we target are based on the execution of independent, fine-grained tasks called Codelets. We also consider resource constraints in our modeling. We show how to map different common code optimizations into Petri nets and how to model some specific architectural constrains in parallel algorithms. We compare our predictions of performance with performance measurements on a real many-core architecture, the average error of our model was 4.4%. We also use our performance model to analyze trade offs between the number of threads used, the available memory bandwidth, and the amount of available memory. In addition, we use timed Petri nets to select better algorithm strategies under particular architecture specifications.

This chapter is organized as follows: Section 7.1 introduces the main motivations for our approach to performance modeling, Section 7.2 provides relevant background on many-core architectures, scheduling and Petri nets, and Section 7.3 describes in detail how to model the performance of fine-grained parallel applications on many-core architecture environments with resource constrains using timed Petri nets. Finally, experimental results validating our model and analyzing different trade offs for several algorithmic approaches are presented on Section 7.4.

7.1 Motivation

Early experiences with the optimization of programs for many-core architectures have prompted the research community to direct their efforts toward finding the techniques required to optimize many-core programs under architectural constraints. As the field of parallel computing advances, architectures with over 100 independent hardware thread units are emerging. And due to the difficulty of programming for and optimizing these machines, studying the underlying algorithm via performance models will vastly increase the speed and efficiency with which algorithms can be implemented on, and optimized for, many-core architectures.

Performance models are a powerful tool to select which optimization techniques should be applied to an algorithm; they allow evaluation of specific optimizations without the cost of a full implementation.

We have performed a survey of existing performance modeling techniques, used for parallel algorithms, to identify their advantages and limitations.

Many performance modeling approaches, such as Anglano's work [75], focus on parallel programs designed to be executed on heterogeneous clusters. While Anglano also models the behavior of the parallel program with Timed Petri Nets, queuing theory is used to model the effects of resource contention on execution time. Because of a focus on communication, the petri net models used in Anglano's work generally consist of segments, which are portions of the computation in between communication statements. While accurate, this model focuses on the cost of communication and does not take into account other sources of resource contention. In addition, it is designed to work with coarse grain tasks. In contrast, the problem we consider is that of a shared memory system.

Dwork et al. [73] previously approached this problem from the perspective of a formal model in which individual memory operations, such as *read*, *write*, and *read-modify-write*, were considered. Dwork et al. present a very powerful mathematical model, that despite its accuracy, is impractical due to its complexity. Our work, instead, utilizes a layer of abstraction to maintain a high degree of accuracy while vastly simplifying the resulting model. With this layer of abstraction, we are potentially able to support much larger systems, such as many-cores, that require additional resources to be modeled.

At the opposite side of the spectrum is King and Pooley [74] who utilized techniques to transform a UML diagram into an architecture and implementation-independent timed Petri net. The resulting performance model achieves simplicity at the cost of degraded accuracy.

A compromise must also be reached between granularity and complexity. For example, Chen and Aamodt [77] model cache contention and throughput of multi-threaded systems through the use of an analytical cache contention model and Markov chains. While advanced, this model is only designed for applications where threads do not communicate, although the authors have indicated that this can be extended through a two-level model in which the upper-level model models synchronization overhead. Many-core architectures rely very heavily upon fine-grain synchronization, so any model must take this into account.

Hong and Kim's work on integrated power and performance models for GPU architectures [78, 79] is a recent and novel work that uses empirical data and powerful equations to model the power and performance of applications on GPU architectures. In their work, Hong and Kim model parallelism and contention at the warp level for nVidia GPU architectures. They have obtained a highly accurate model for execution time [78] aided by a detailed understanding of the underlying architecture and program execution model. By combining this model with power measurement techniques, the performance of an application with respect to execution time and power consumption can be measured [79]. However, Hong and Kim's model, while powerful, is designed with GPU architectures in mind and relies upon the specific scheduler's ability to schedule warps, avoiding stalls to a fair degree throughout preemption, and maintaining a high degree of accuracy. As such, Hong and Kim's results show that overall trends are modeled quite well, but fluctuations and other interesting behavior are not readily available. The impressive amount of threads that this model is able to handle is leveraged in the groupings of threads into warps and blocks, that follow fairly regular patterns in the execution, reaching a fair complexity in the analytical model. A more complex case where more than one hundred treads are running independent pieces of code (e.g. with different program counters) using very fine grain tasks is another different but very interesting trend not cover at all by this kind of analytical models.

From our survey of previous work, we are able to deduce the characteristics required by a performance model for applications on many-core architectures. To simplify the resultant model, we must start from the algorithm, not the architecture. This has the added bonus of creating an architecture-independent model, at least during the first stages of the model, as per King and Pooley's work. Furthermore, the model must be fine-grained, but not so fine-grained as to potentially become too complex, as with Dwork et al.'s work. Most importantly, our approach must be able to model all forms of resource contention.

We considered many approaches. A mathematical model, such as the one proposed by Dwork et al. [73], would allow for the performance to be computed rapidly. However, such models are largely dependent upon the profiling of existing implementations and applications. Another option considered was queuing theory. Queuing theory, while powerful, is not a perfect fit. To create an accurate queuing network, the parameters of the system must be understood with highly detail. For a specific architecture, those parameters can be thoroughly studied. However, our target is to model the co-design of hardware and software, and for that we need to perform a careful study of both the application and the architecture.

Based on this, we have found that Timed Petri Nets are a feasible solution to model the performance of applications in this new environment. Timed Petri nets are a good choice because they are able to represent the features in many-core architectures as well as the methods for execution. In particular, Petri nets are able to model large numbers of processing elements that execute tasks following a dynamic schedule in an environment with several kinds of resource coordination conditions. And, by using Petri nets to perform a detailed simulation, we are able to study the nuanced behavior of the code prior to implementation, as seen in Section 7.4.

It is worth noting that the optimization of parallel programs on many-core architectures has been extensively studied in the past. However, there are very few studies on architectures with more than 100 independent hardware thread units. Cyclops-64 (Chapter 4) is one of such architectures. With its 160 independent hardware thread units, it is an ideal architecture to explore the new trends in computer architecture.

The study of Cyclops-64 (C64) has helped to reveal some challenges for future generations of many-cores. During the last couple of years, several efforts have been made to study the optimization of a large amount of Cyclops-64 applications [80, 61, 45, 47, 60] to determine the most useful compiler techniques. Unsurprisingly, these studies showed that tiling at the register level [61, 81] in conjunction with techniques that hide the latency of memory operations [80] were required to achieve high performance. However, the results reached by these studies are still far from the theoretical maximum performance of the chip. Further studies showed the importance of scheduling fine grained tasks in the presence of shared resources [82] or the possibilities for scalability in future many-cores [83, 84].

Scheduling and granularity have become essential parameters for many-cores. Light Dynamic Scheduling of very fine grained tasks with hundreds of threads sharing large amounts of resources are new scenarios for optimization and modeling of parallel algorithms. Classical techniques that targeted coarse grained programs with static scheduling are not competitive in these environments. For example, even fine grained static scheduling fails when the amount of data to be processed is limited. One of the main reasons for these large differences is the variation in the completion time of tasks, even in the case where the amount of work per task was the same. This was shown in the case of a tiled matrix multiplication program, where the variation of the execution time of identical, small tasks, can be as large as ±22%. These differences are related to the influence of shared resources and their arbitration schemas [80].

As a solution, new program execution models [31, 36] with support for fine grain parallelism are built to express data-driven applications under resource coordination conditions. This has been one of the key motivations for the revolutionary designs proposed as part of the DARPA UHPC project [85].

Having in mind these new needs for modeling under the new scenarios explained, Timed Petri Nets provide a good balance between intuition and scalability. Transitions correspond to codelets, as described in Section 7.2.1. This allowed us to create our clear methodology by which to map applications to Timed Petri Nets, as described in Section 7.3.4. The complexity of the resulting model is solely dependent upon the complexity of the algorithm and the number of different shared resources, not the system itself.

A Petri net model has a high degree of scalability, for the purposes of many-core architectures. This is because the Petri net scales well with the architecture itself. For example, the differences between a system with five thread units and a system with fifty thread units will be the number of tokens in the Place corresponding to available thread units. In addition, efficient solutions of these timed Petri Nets are a well-researched topic [86, 87, 88], so their simulation is not a constraint.

7.2 Background

This section provides further detail on topics of interest discussed in the following sections. Section 7.2.1 defines the concept of a codelet and Section 7.2.2 discusses the foundations of Petri nets. In addition, architectural details about Cyclops-64 can be found on Chapter 4 while additional information about Dynamic Schedulers can read in Chapter 6

7.2.1 The Codelet Execution Model

The Codelet Execution Model (CXM) [31] is motivated by new trends in processor and system architecture, driven by power and complexity, that point toward very high-core-count designs and extreme software parallelism to solve exascale-class problems. This research explores a fine-grain, event-driven model in support of adaptive operation of these machines. The CXM breaks applications into codelets (small bits of functionality) and dependencies (control and data) between these objects. It then uses this decomposition to accomplish advanced scheduling, to accommodate code and data motion within the system, and to permit flexible exploitation of parallelism in support of goals for performance and power.

In this model, programs are decomposed into snippets of code (codelets) and their dependencies. The natural breaks between codelets provide several opportunities to observe and adapt the execution of a program. The Codelet Program Execution Environment provides a runtime system and system software in which the adaptability benefits of codelets can be realized. It relies on existing work in data flow theory [6] to provide strong theoretical results that guarantee forward progress in the program.

7.2.2 Petri Nets

Petri nets are a mathematical tool, proposed by Carl Adam Petri's dissertation [89], that use directed, weighted, bipartite graphs to model a variety of systems. Petri nets can be used to represent and analyze parallel, concurrent, asynchronous or stochastic systems.

Since its presentation in Petri's dissertation, Petri nets have enjoyed significant amount of attention. In 1989, Murata [90] compiled an excellent publication explaining Petri nets and presenting their most relevant applications.

A Petri net graph is composed of two kinds of nodes (*places* and *transitions*). When modeling systems, places usually represent conditions that must be met while transitions represent events that happen. In graphical representations, places are usually shown as circles and transitions as rectangles. Having two types of nodes allows greater expressiveness than traditional dataflow graphs. For example multiple arcs can go from a single place to different transitions; where the place represents a resource shared by many tasks (the transitions). This type of scenarios has been a limitation for dataflow graphs.

Arcs in a Petri net graph can only go from places to transitions or vice versa. The weight of an arc is used as a shorthand to represent several equivalent parallel arcs between two nodes in a graph. An arc with no labeled weight is equivalent to an arc of weight one.

The state of a Petri net graph is called its *marking*. The initial state of a Petri net is called its *initial marking,* M_0. The marking (state) of each place is a nonnegative integer that represents a number of tokens in the place. Usually, the marking of a place is represented by drawing tokens, as dots, inside the circle that represents the place. The marking of a program is defined by a vector that contains the marking of each place in the graph.

The operational semantics of a Petri net can be defined by three simple rules. First, a transition is enabled if each input arc is connected to a place with a token. Second, enabled transitions may or may not fire. Finally, when firing a transition, one token is removed from the associated place for each input arc, and the transition produces one token on each output arc. The excellent survey by Murata [90] contains a more comprehensive explanation of the operational semantics of Petri nets along with several examples.

Figure 7.1 shows an example of how to model the execution of $y = a + b + c$ in a system with only one adder.

Petri nets, though powerful in modeling programs and concurrent operations, are not complete enough to model synchronization and scheduling, because no assumptions are made about the duration of transitions. This restriction is removed by *timed Petri nets* [76], where an amount of time has been assigned to the transitions in the Petri net graph.

Timed Petri nets can be deterministic or stochastic. Deterministic timed Petri

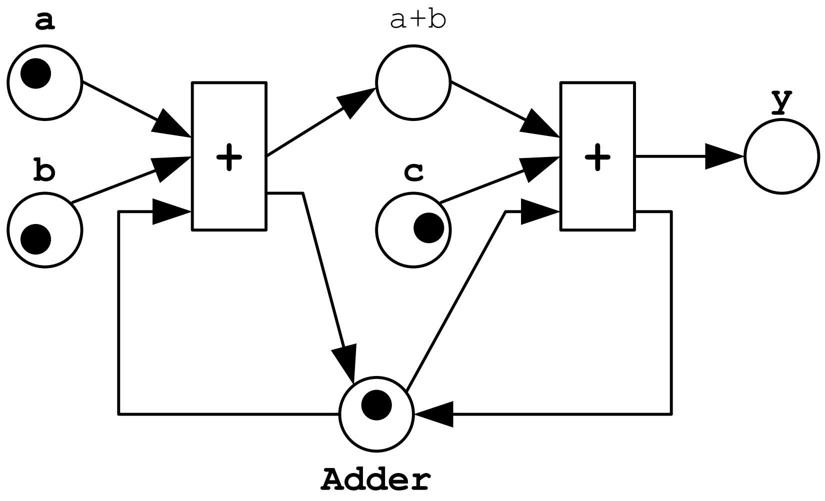


Figure 7.1: A Petri net model of the computation of $y = a + b + c$ in a system with only one adder.

nets have a deterministic time associated with each transition. In contrast, in stochastic timed Petri nets, each transition has a random firing time.

7.3 Solution Method

The rise of new program execution models that manage fine-grained tasks using dynamic scheduling show the need for a performance modeling tool for modern many-core architectures. We propose applying the concepts of timed Petri nets to develop an efficient and accurate alternative to model the behavior of High Performance Applications under these new environments. The flexibility of timed Petri nets can be adapted to several scenarios where shared resources are the main constraints.

In this section we will first introduce the description and behavior of basic blocks in timed Petri nets and then we will explain how to construct more complex programs using a Dense Matrix Multiplication algorithm with several levels of tiling as an example. Finally, we will explain a general methodology for generation of timed Petri nets based on pseudo-code or source code.

7.3.1 Basic actors in timed Petri nets

In general, tasks of a particular process are modeled as transitions in timed Petri nets. In our particular case, *Codelets*, as defined in section 7.2.1, are modeled as transitions with durations defined by their execution time. In addition, auxiliary transitions are required for the appropriate behavior and correct construction of programs under this model. The following are the different types of auxiliary transitions:

7.3.1.1 Init

This transition has a single input arc and one or more output arcs with a cumulative weight of W. It is used to instantiate the creation of several parallel Codelets ready to be fired, but it does not necessarily mean they will be executed in parallel. The transition duration is related to the details of the particular architecture and the specific software implementation of the runtime. It has been shown that, for particular cases, this can be done with high efficiency because several similar tasks can be scheduled with complexity $O(1)$ [37].

7.3.1.2 Clean

This transition has an input arc of weight W and a single output arc. Its purpose is to represent the synchronization process of a set of Codelets. The transition duration depends on the synchronization methods available in the architecture, ranging from high-overhead software barriers that use locks, to better mechanisms that use hardware barriers, or other lighter implementation of asynchronous constructs using native atomic operations. Previous studies [39, 82] have shown that *in-memory* atomic operations can be used to implement very lightweight synchronization under scenarios with hundreds of threads and fine grained tasks.

7.3.1.3 Done

The Done transition is used to handle `for` loops. It has a single input and two outputs that are mutually exclusive. At a functional level, it can be described as a decreasing counter. After each decrement, the transition will produce a token on the output arc labeled True (T) if the counter has reached 0. Otherwise, the transition will produce a token on the output arc labeled False (F). A possible implementation using the init and clean transitions is shown on Figure 7.2. The transition duration is related to the time that is spent checking the counter value.

7.3.1.4 Schedule

The Schedule transition is used to model the behavior by which the scheduling process assigns a limited amount of resources (e.g. threads) to Codelets. The Schedule transition has a single input and a single output and its transition duration is related to the overhead of the scheduler assigning these resources.

7.3.1.5 Other auxiliary constructs

There are other several useful constructs that can be modeled with Petri nets. For example, an `if-else` statement can be modeled by a Done transition. However, the `if-else` has an implementation that is slightly different due to the necessity to verify the conditional statement. Particular details about other synchronization constructs such as mutexes and semaphores can be found in [91].

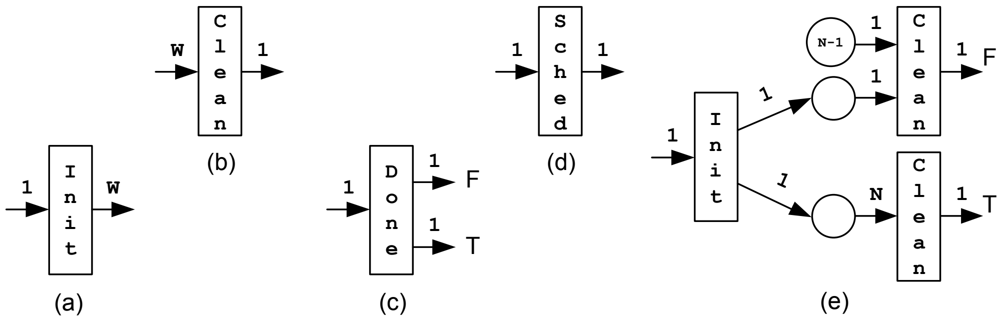


Figure 7.2: Basic Actors (a) Init Transition. (b) Clean Transition. (c) Done Transition. (d) Schedule Transition. (e) Detailed Petri net of a Done Transition in a for a loop with N iterations.

7.3.2 Expressing concurrency

The ability to expressing both serial and concurrent sequences of tasks is essential to HPC programs. We were able to express parallel programs using the basic Petri net actors described in Section 7.3.1. In our Petri net representation, we have modeled Codelets as transitions. In order to examine different scenarios, we use a two-level tiled Dense Matrix Multiplication $C = A \times B$ with matrices in off-chip memory of sizes $M \times M$. In order to provide locality, each matrix in the Dense Matrix Multiplication program has been divided into *Blocks* of size $N \times N$ that fit in on-chip memory. Furthermore, each *Block* has been divided into *Tiles* of size $L \times L$ while each Codelet is in charge of computing one *Tile* of C.

Figure 7.3 shows a pseudo algorithm for the Dense Matrix Multiplication. Notice that all the data movements (e.g. Copy and Copy Back), the initialization of $C_{i,j}$ and the computation of partial results of $C_{i,j}$ can be expressed as a set of parallel tasks using Codelets. We will use a bottom up approach to model the behavior of this algorithm using timed Petri nets.

```
DenseMatrixMultiply (A[M,M], B[M,M], C[M,M])
1 :  for $i = 0$ to $\frac{M}{N} - 1$ and $j = 0$ to $\frac{M}{N} - 1$
2 :     Initialize Block $C_{i,j} = 0$ on on-chip memory.
3 :     for $k = 0$ to $\frac{M}{N} - 1$
4 :        Copy Blocks $A_{i,k}$ and $B_{k,j}$ from off-chip memory to on-chip memory.
5 :        Compute partial result of Block $C_{i,j} += A_{i,k} \times B_{k,j}$
6 :     end for
7 :     Copy Back the Block $C_{i,j}$ from on-chip memory to off-chip memory
8 :  end for
```

Figure 7.3: Pseudo algorithm for Dense Matrix Multiplication

7.3.2.1 Parallel for loop - On Chip Matrix Multiplication

With blocks A, B and C in on-chip memory, computing a partial result of Block C is just another matrix multiplication. There are $W^2 = \left(\frac{N}{L}\right)^2$ concurrent tasks that may execute in parallel. A parallel `for` loop using Petri nets can be implemented using the Init and Clean transitions. Figure 7.4 shows the model of an on-chip matrix multiplication using Petri nets.

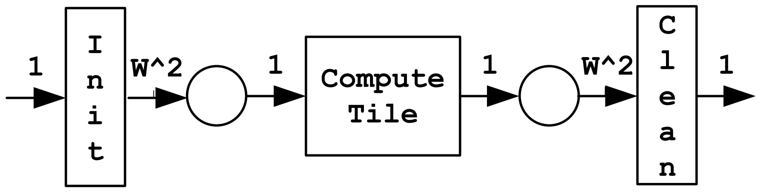


Figure 7.4: Petri net model for on-chip matrix multiplication

7.3.2.2 Serial for loop - Computing a whole block from off-chip memory

In order to compute a whole $C_{i,j}$ block, it is necessary to compute partial results by copying blocks from A and B and computing smaller matrix multiplications that fit in on-chip memory. This is the sequential process represented by lines $3 - 6$ of Figure 7.3. This will require parallel `for` loops for the compute and copy Codelets in addition to a serial `for` loop that can be expressed using the Done transition. Figure 7.5 shows how this can be implemented using Petri nets. Notice that the Copy task can be execute by R Codelets. The selection of an appropriate R depends on specific architecture characteristics but, in general, a higher number of parallel Codelets is desired to increase throughput. However, there are two main problems with selecting a large R: 1) The increased scheduling overhead when large number of tasks are used. 2) The limited amount of available resources (such as memory bandwidth) can cause a degradation in performance. We will discuss this problem with more detail in the following subsections.

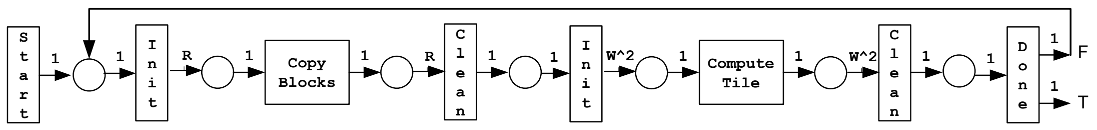


Figure 7.5: Petri net model for computation of one block of matrix multiplication with operands in off-chip memory

7.3.3 Implementation of Performance Optimizations and Modeling of Resource Constraints

Performance optimization is of paramount importance for HPC programs. It is required that the proposed modeling approach be able to express the most important and common optimizations. In addition, many-core architectures are characterized by the increasing number of shared resources. It is imperative that a performance model captures these architectural details in order to obtain high accuracy and to find possible bottlenecks in specific parallel implementations. Fortunately, Petri nets are a good candidate to model the different types of interactions of these hardware resources. Figure 7.6 shows a higher performance version of the matrix multiplication shown in Figure 7.5. This version also models some very common resource constraints (represented by the dotted arcs) found in many-core architectures. The following paragraphs will explain, in detail, the modifications introduced.

7.3.3.1 Double Buffering and Pipelining

The Copy Blocks task in Figure 7.5 is a bottleneck of the algorithm because no computations can be done until the data required is in on-chip memory. A solution to this problem is to implement a pipelined double buffering strategy where the two types of tasks are interleaved: While the first buffer's R threads are copying the blocks needed from off-chip memory, the remaining threads are executing the computation of tiles over the blocks in on-chip memory in the second buffer. Under this dynamic schema, data is available when the computation needs it and the Codelets for computation and data movement are redistributed dynamically at runtime, avoiding losses in performance because of stalls.

This can be modeled using petri nets by duplicating the net responsible for a single buffer, shown in Figure 7.5, so as to represent multiple buffers. This is shown in Figure 7.6, where the top and bottom sequential for loops each represent a single buffer.

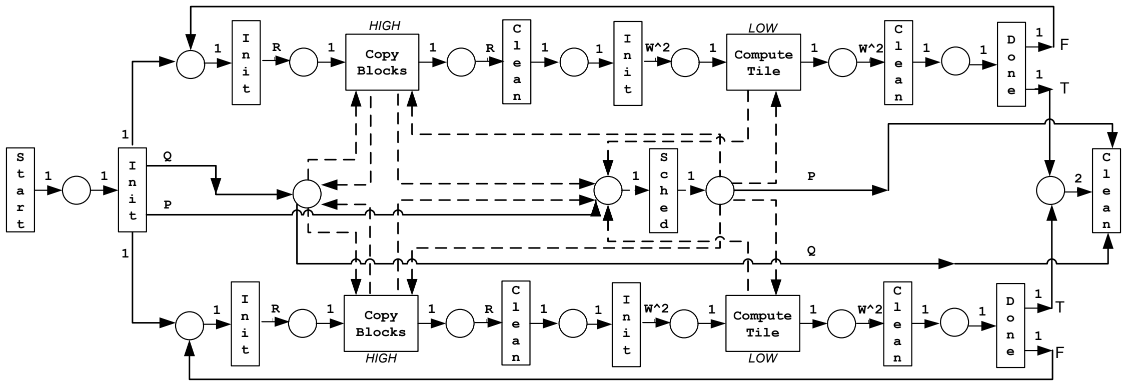


Figure 7.6: Petri net model for computation of one block of matrix multiplication with operands in off-chip memory using double buffering with pipelining for Blocks of A and B matrices. It also includes resource constraints on the number of PEs equal to P and the number of memory banks equal to Q. Resource constraints are highlighted using dotted arcs and the priorities for resource assignment are in *Italic*

7.3.3.2 Resource Constraints

Resource constraints always limit the intrinsic parallelism of applications. The most important one is the number, P, of Processing Elements (PEs) available in the whole system. Another significant constraint is the Memory Bandwidth, which is closely related to the number of memory banks. While the former is a very strict constraint in non-preemptive systems, the latter can be violated at the expense of a decrease in performance because of the increasing delay in memory transactions.

There are scenarios where different groups of Codelets are competing for a limited amount of PEs, this is the case of the optimized Matrix Multiplication with double buffering described in section 7.3.3.1. While the dependencies between computation and data movement in the same buffer make them mutually exclusive, computation Codelets on the first buffer are competing for PEs with data movement Codelets on the second buffer and vice versa. In addition, it is possible that the number of Codelets for Compute tiles W^2 is larger than the available number of PEs so all of them can not be fired in parallel, some have to wait until more PEs are available. By definition, each Codelet requires one PE in order to be fired. Figure 7.6 shows how the Schedule transition is used to model the assignment of PEs to each one of the four Codelets.

In addition, the copy block Codelets are competing for memory bandwidth. An upper bound on the number of memory banks used can be established because each copy block Codelet is a serial process that move data from off-chip memory to on-chip memory. It can be said that, under controlled conditions, the limit in the memory bandwidth is reached when the amount of copy Codelets is equal to the number of memory banks in the architecture. This ideal scenario supposes that each Codelet is accessing data on a different memory bank and that the copy Codelet is highly optimized. In practice, the limit on the number of parallel copy Codelets before the competition for memory bandwidth decreases the performance is lower because of potential bank conflicts between concurrent Codelets. Figure 7.6 assumes an architecture with Q memory banks. The schedule transition is not used because this is not a resource managed by the scheduler or the runtime system. We are merely preventing contention that will decrease the overall performance.

7.3.3.3 Priorities

While the dependencies between computation and data movement in the same buffer make them mutually exclusive, computation Codelets on the first buffer are competing for PEs with data movement Codelets on the second, and vice versa. A simple policy of first come first served will not be helpful because the large ratio between the number of computation Codelets and data movement Codelets favors computation. In practice, it is necessary to favor data movement, the bottleneck of the algorithm [92]. This is the motivation behind the introduction of priorities for the assignment of resources. Figure 7.6 shows that the Copy Blocks Codelets have *high* priority while the Compute Tile Codelets have *low* priority. The expected result is that, with a sufficient number of Codelets copying data (keeping in mind the memory bandwidth constraints), there will always be data available on the buffers for the compute Codelets. In addition, if there are no copy tasks ready to fire, all available PEs will be assigned to compute Codelets.

7.3.3.4 Composability - The Complete Off-Chip Memory Matrix Multiplication

Until now, Figure 7.6 only modeled the computation of one block of C. This corresponds to lines $3 - 6$ in Figure 7.3. This petri net is merely a building block for a larger model. Several approaches have been studied in the past for the structuring and composability of Petri nets [93]. For our purpose of performance modeling, we found that the concept of macro-transitions suits our needs. A macro-transition is a sub net where transitions constitute the boundary of the subnet.

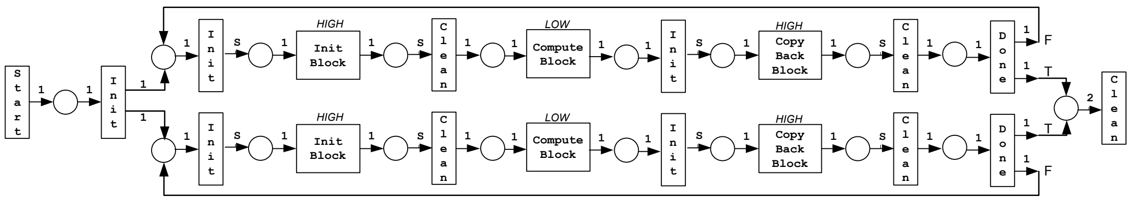


Figure 7.7: Petri net model for computation of matrix multiplication with operands in off-chip memory using double buffering with pipelining for C blocks. For simplicity, resource constraints are not included.

Figure 7.7 shows an optimized version of the Petri net used to model the Dense Matrix Multiplication in Figure 7.3. The initialization and copy back of block C

is executed by S concurrent Codelets. For performance purpose a double buffering strategy, such as the one described in Section 7.3.3.1, is required to avoid the bottleneck of memory bound Codelets. Although resource constraints are not shown in the figure, priorities are still required to avoid stalls due to a lack of data to process in the buffers.

7.3.4 Methodology for generation of timed Petri nets with resource coordination conditions

The generation of a Timed Petri net model for a particular parallel algorithm can be summarized in Figure 7.8. We will illustrate our methodology for the generation of a Timed Petri net using a kernel for simulation of an electromagnetic wave propagation using the Finite Difference Time Domain algorithm in 1 Dimension (FDTD1D), the pseudo-code is shown in Figure 7.9.

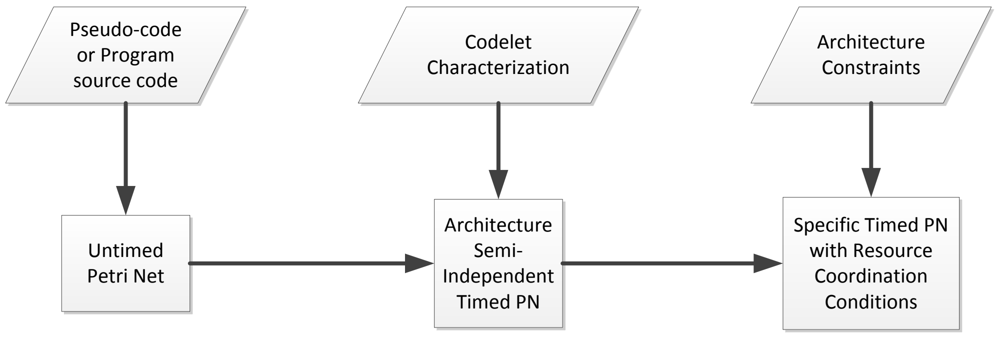

Pseudo-code or Program source code
Codelet Characterization
Architecture Constraints
Untimed Petri Net
Architecture Semi-Independent Timed PN
Specific Timed PN with Resource Coordination Conditions

Figure 7.8: Methodology for generation of a Timed Petri net model with resource coordination conditions using pseudo code or source code

The first step requires the source code program of the algorithm or a simple pseudo-code that shows the data and control flow of the codelets involved. The generation of the untimed Petri net need to identify some features of the algorithm: 1) The parallel constructs (e.g. parallel for). 2) The codelets (e.g. fine grain tasks) involved in each parallel construct. For our example, there is a total of 3 (Copy Tile, Compute Tile E and Compute Tile H). and 3) The control flow and data flow inside each parallel construct and between parallel constructs.

```
FDTD ( double *E, double *H, int N, int Timesteps, int NT,
int TileSizeconst double k1, const double k2) % NT=NumTiles
1 :  for $t = 0$ to Timesteps do
2 :     parallel for $i = 0$ to NT do
3 :        CopyTile (i, E, H, TileSize)
4 :        ComputeTileE (i, E, H, TileSize, k1, k2)
5 :     parallel for $i = 0$ to NT do
6 :        ComputeTileH (i, E, H, TileSize, k1, k2)
7 :  end for
```

Figure 7.9: Pseudo algorithm for Finite Difference Time Domain

According to Figure 7.9, our example involves 2 parallel for loops, each one with NT tasks, the first one runs serially two codelets: Copy Tile and Compute Tile E. The second one runs Compute Tile H. These two parallel loops are mapped as highlighted shapes in Figure 7.10. The two parallel for loops run serially with an external for loop that is controlled by the external feedback and the Done transition. At this point, we have an untimed Petri net that just includes the codelets with control and data flow of the program represented by continuous arcs (the untimed Petri net block in Figure 7.8).

The second step is to determine the timing of the transitions that represent the codelets. It will require some information about the architecture such as instruction timing. Several methods can be used to characterize the codelets, from code profiling and simulation to analytical models based on instruction timing. The result will be a timed Petri net semi-independent of the architecture. At this point, the Petri net does not include information about the architecture constraints (e.g. number of memory banks, number of threads).

The final step is to use the timed Petri net and add the constraints of the specific architecture in order to obtain a specific timed Petri net with resource coordination conditions. For our example, we are using two constraints: The architecture has Q shared memory banks and P PEs available. These constraints are represented by tokens and initialized by the first Init transition in the model. Figure 7.10 shows in red the place that model the memory banks, a pair of arcs from and to every Codelet that implies data movement is required because all of them are sharing a limited amount of these shared resource. In our example, the Copy Tile codelet is using extensively the memory. For the shared PEs, two places (highlighted in green on Figure 7.10) and the Schedule transition (See Sec. 7.3.1.4) are used to model this constraint: every Codelet can take one token from the last place while they return the token to the first place after finishing execution.

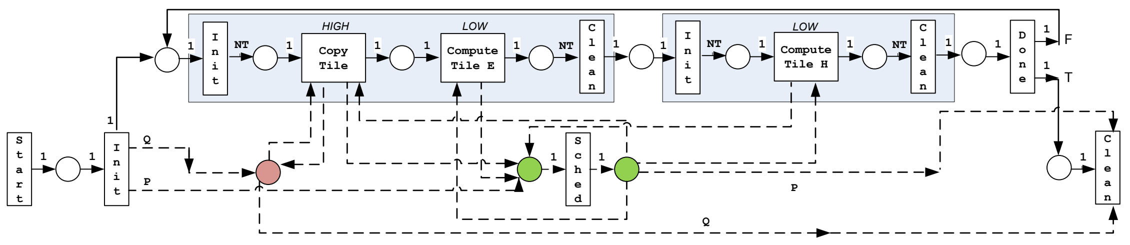


Figure 7.10: Petri net model for computation of Finite Difference Time Domain benchmark

7.4 Experiments

This section will present several experiments, with different characteristics and complexity, that were designed to validate our model and demonstrate our ability to study different trade-offs in many-core architectures. First, in Section 7.4.1 we show the accuracy of our approach by using Petri nets to compare our performance model with real measurements on a many-core architecture and with an analytical model proposed in the literature [78]. We used several applications: a highly optimized parallel implementation for Matrix Multiplication with hierarchical tiling across the memory hierarchy, a 1-Dimension, and a 2-Dimension algorithm for the solution of Maxwell's Equations using multiple tiling schemes, for the last two applications we evaluated two different tiling techniques. Second, we study the impact of some architectural modifications in Section 7.4.2. Finally, we test two LU Factorization algorithms in order to determine the best fit and trade-offs for the architecture in Section 7.4.3.

We have selected the Cyclops-64 architecture because it has a large amount of independent processing elements and due to its particular characteristics in terms of resource sharing. In addition, Cyclops-64 offers support to the codelet execution model using features such as in-memory atomic operations that allow efficient implementation of a light dynamic scheduling. More details about Cyclops-64 architecture have been explained in Section 4. For the purpose of rapidly obtaining data over a range of input parameters, such as transition delay, topology, and the interconnection of elements inside of the timed Petri nets, we utilized a sequential branch of PICASim [94].

7.4.1 Verification of Model and Evaluation of Performance Optimizations

To ensure the validity and accuracy of our approach, we modeled two different algorithms that have been previously implemented and executed on a modern many-core architecture. This provides us with a benchmark so that we can evaluate the accuracy of our model. We also conducted an evaluation-driven performance optimization for the second application.

7.4.1.1 Dense Matrix Multiplication

First, we modeled a highly sophisticated Dense Matrix Multiplication program, that had several levels of tiling, dynamic scheduling, and other optimizations, in Cyclops-64. In Section 7.3.2 we studied Dense Matrix Multiplication with support for off-chip memory and double buffering, as seen in Figures 7.6 and 7.7. As a first step in the development of this algorithm, a highly optimized version, designed solely for problems that fit in on-chip memory, was also developed, as seen in Figure 7.11. For the particular case of Cyclops-64, each Compute Tile codelet operates on a Tile of 6×6 for maximum locality, offering large amounts of parallelism with very fine grainularity, making its modeling a significant challenge. The codelets used for Matrix Multiply were manually optimized using assembly language with proper instruction scheduling and loop unrolling.

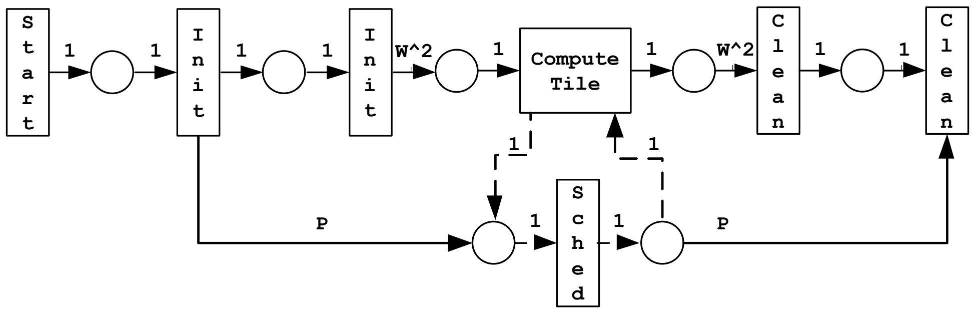


Figure 7.11: Dense Matrix Multiplication optimized for On-Chip Memory

To obtain the duration of the Compute and Copy Codelets, we profiled the execution of these algorithms and used mathematical models to determine their duration as a function of the problem size and other system parameters. In addition, we used architectural parameters to determine the duration of the basic actors explained in Section 7.3.1. For example, the dynamic scheduling was implemented with the in-memory atomic addition. The latency of this operation is 3 cycles, and it is supported by each one of the 8 memoy banks in Cyclops-64.

We simulated the execution of these algorithms on Cyclops-64 through the use of our previously described timed Petri net simulator, timing information collected during profiling, and our knowledge of the architectural parameters in Cyclops-64. With this, we were able to recreate our previously obtained data and compare our Petri net simulations with the results obtained via detailed architectural simulations. In addition, we used Hong and Kim's Analytical model as described in [78] in order to compare with other alternatives for modeling.

The results can be seen in Figure 7.12. It shows how the performance of our specialized on-chip memory Dense Matrix Multiply scales with the number of threads fixed at 156 and the problem size varied. Of particular note is that our model matches the "jagged" line seen in the experimental results. This shows that our petri net model closely matches the behavior of the algorithm, as opposed to just the overall trend. On the other side, the analytical model has a higher error and just follows the

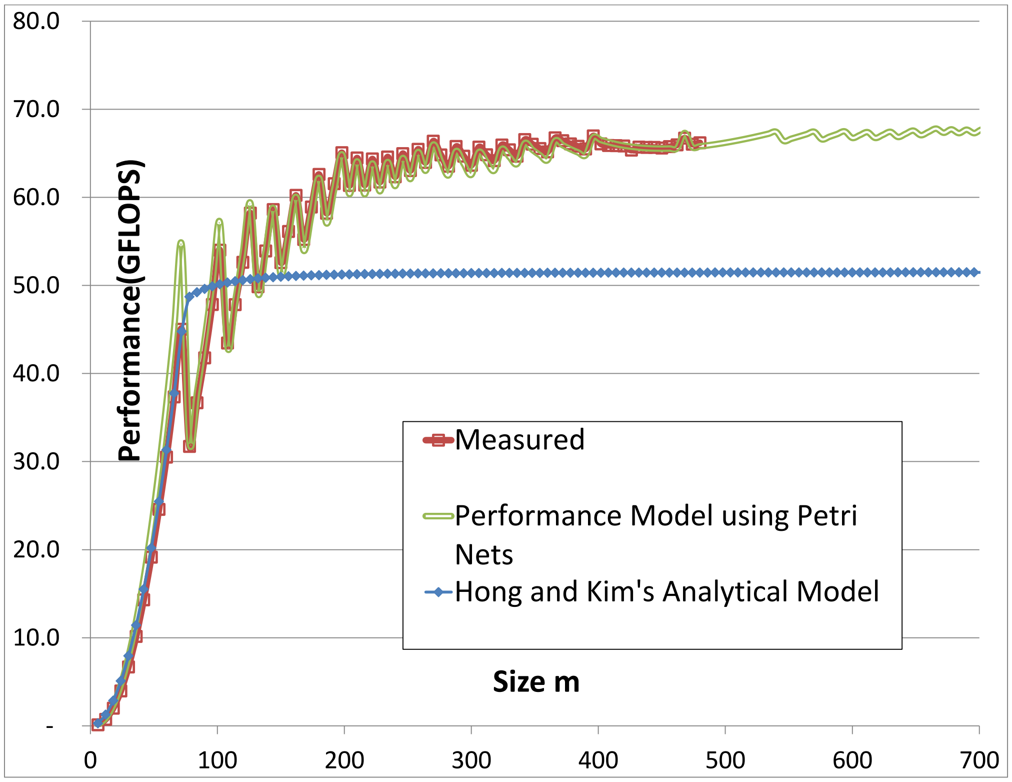


Figure 7.12: On-Chip Memory Dense Matrix Multiplication

overall trend. When small tiles are used, the analytical model is able to closely predict the behavior of the program. However, the maximum performance is subestimated, perhaps due to their method of counting instructions as a way to analyze a codelet. A more detailed study of the assembly code of our kernel revealed that instruction scheduling is extremely important to hide latencies of long latency instruction such as loads and stores from on-chip memory. This drawback of the analytical technique can not be noticed on other architectures such as GPUs because the scheduler takes advantage of preemption to hide these latencies when possible. The average error of our Petri net model is 2.5% vs. 19.0% of error using Hong and Kim's model. The behavior of the analytical model also evidences how challenging it is to model fine-grain programs under resource constrains.

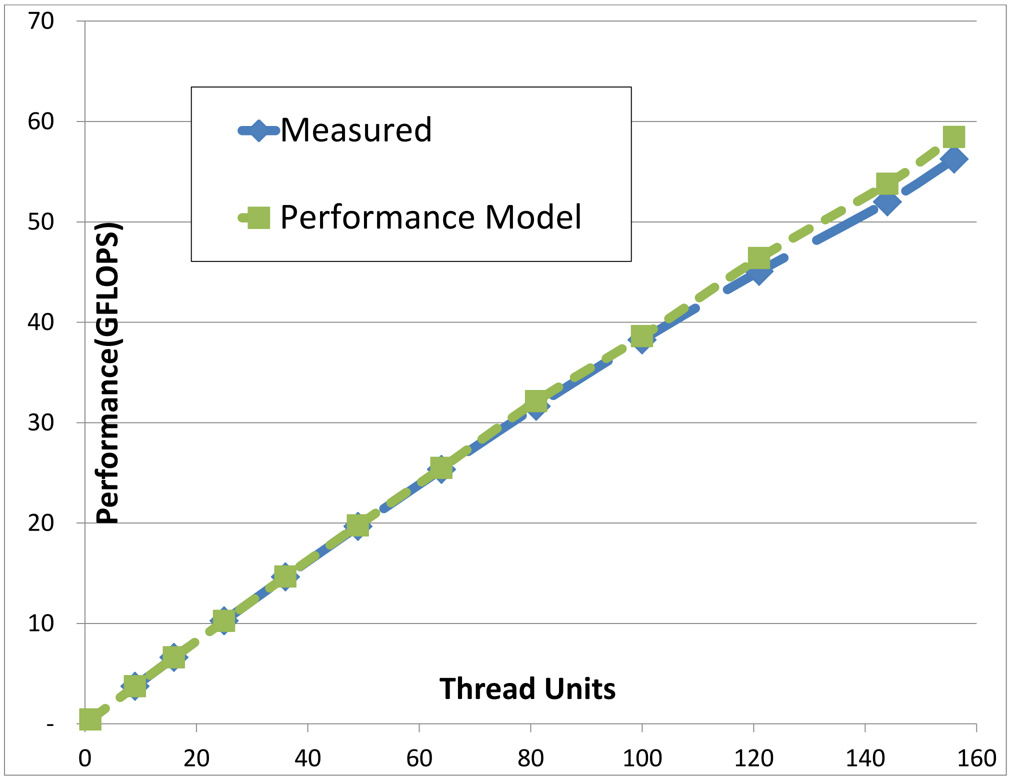


Figure 7.13: Off-Chip Memory Dense Matrix Multiplication with Double Buffering

We use these results to leverage the performance modeling of off-chip memory Dense Matrix Multiply, a more complex case. This case uses an on-chip memory Matrix Multiplication and data movement between off-chip memory and on-chip memory using Double Buffering and Dynamic Percolation. We compare the performance produced by our model with the results of simulation. Unfortunately, Hong and Kim's analytical model is designed for SIMT programs, while the optimizations implemented in our program require MIMD capabilities. Figure 7.13 demonstrates how, for a given problem size, the performance varies as an increasing number of threads are made available. The average error is 1.0% showing that our method is highly competitive.

For the on-chip memory version we extended the sizes of the matrix beyond the maximum capacity of Cyclops-64 (e.g. matrices of 486 × 486) in order to see the scaling of performance with respect to matrix sizes. As expected, the performance does not increase. This means that our on-chip implementation is limited by the serial performance of the Codelets.

7.4.1.2 Finite Difference Time Domain Solution of Maxwell's Equations

Next, we considered a more complex and challenging problem: a finite difference time-domain (FDTD) solution for the propagation of electromagnetic waves given by Maxwell's Equations, as described in previous work by Orozco and Gao [60] using the algorithm of Figure 7.9. More specifically, we modeled four different kernels. We modeled the FDTD in 1-Dimension and in 2-Dimensions and we tested two different tiling strategies: Overlapped tiling where extra computations are done along the time dimension and the spatial dimensions in order to decrease communication between tiles and increase locality, and Diamond tiling where memory operations are minimized.

The 1 dimensional implementation computes 4000 timesteps of a problem of size 10000 while the 2 dimensional application computes 500 timesteps of a problem of size 1000×1000. The tile sizes used were 256 and 24×24 for the implementations in 1 and 2 dimensions respectively. These tile sizes were chosen so that they used all the available on-chip memory.

Using the techniques described in Section 7.3.4, we generated a Petri Net model, as seen in Figure 7.10. It is worth noting that, due to the nature of the program, this one Petri Net represents the behavior of all four kernels. The only differences are the weights of the arcs, the number of tiles required to cover the overall space according to the tiling selected, and the duration of the computation and communication stages.

This problem is particularly interesting in that we were able to demonstrate how our model is capable of taking into account the complexities that arise when applying specific optimizations, such as new tiling schemes, to real scientific applications. Also, the amount of data required and the complexity of each codelet is larger than in other examples.

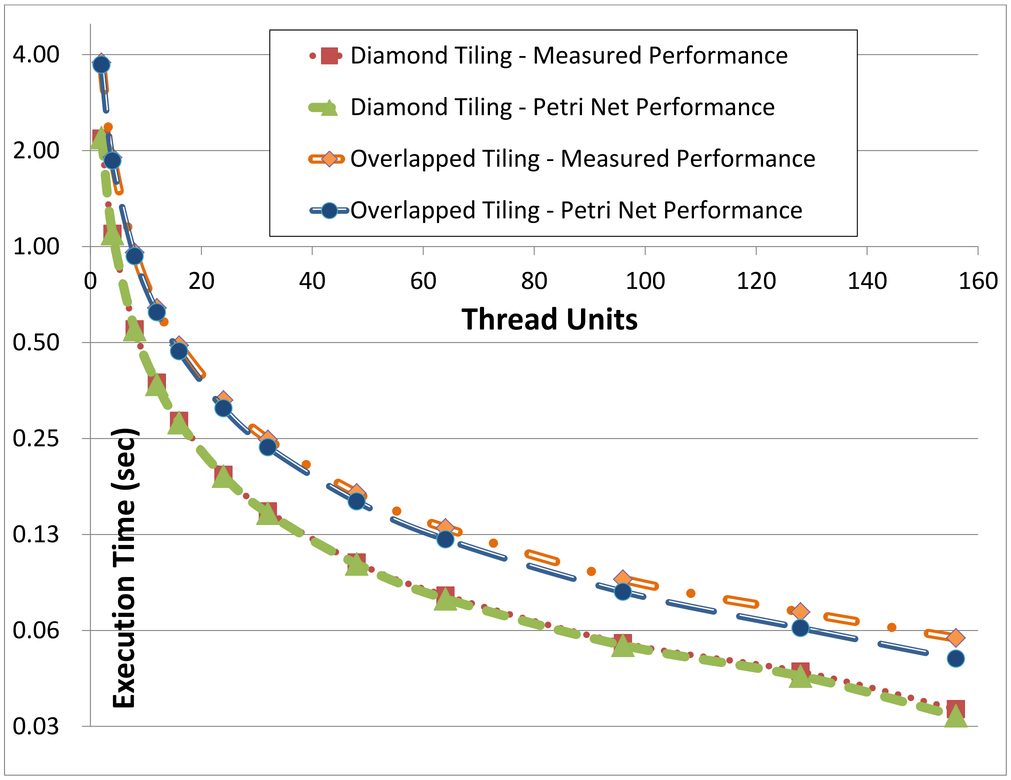


Figure 7.14: FDTD in 1 Dimension

Figures 7.14 and 7.15 show the comparison of our model's predicted performance with that of the measured performance. The model for diamond tiling follows the measured performance with a very high degree of accuracy, for both 1 Dimension and 2 Dimensions. For overlapped tiling there are some discrepancies, but the overall trend is followed quite closely. The average error for this benchmark is 9.6%. Also, we can corroborate the advantages of diamond tiling over overlapped tiling for 1 and 2 dimensions.

7.4.2 Extrapolation of Results on Similar Architectures

One possible use of our approach based on Petri net modeling is to study how different architectural features will influence an algorithm's performance and behavior. Our approach allows one to examine algorithm trade offs with respect to memory size,

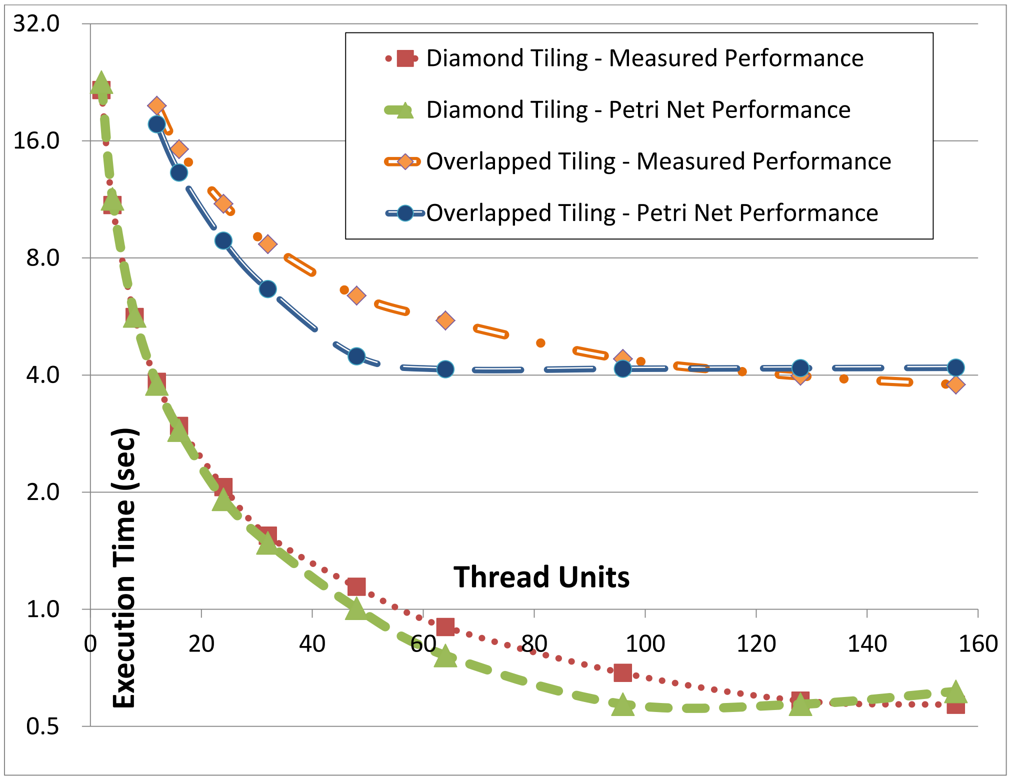


Figure 7.15: FDTD in 2 Dimensions

memory bandwidth, and scalability with respect to number of threads. Using our previous models for dense matrix multiplication, we considered how such trade offs would affect the algorithms.

To provide a baseline for comparison, we first extended our results for off-chip memory Matrix Multiply under the assumption that a Cyclops-64 chip contains up to 1000 independent thread units. This can be seen in Figure 7.16. While there are diminishing returns after 450 threads, it is worth noting that the performance continues to increase for the same problem sizes, even up to 800 threads. This is due to the benefits of our double buffering and dynamic scheduling.

To study the effects of a larger on-chip memory, we considered a case where the size of on-chip memory is doubled while the amount of off-chip memory remains the

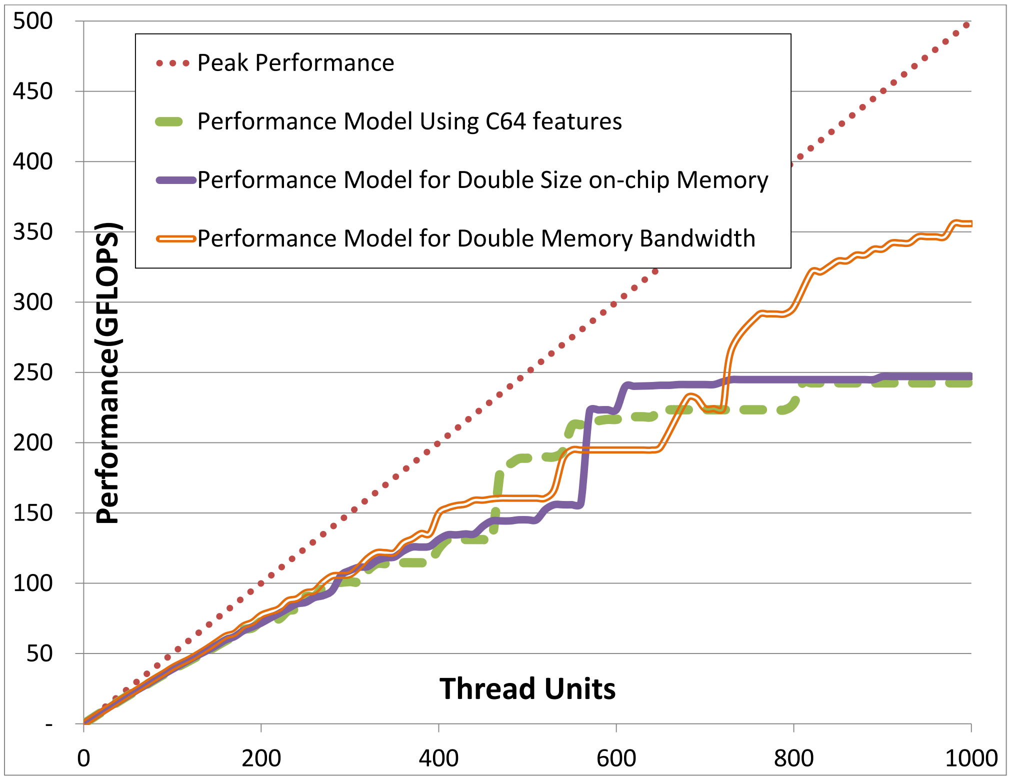


Figure 7.16: Study of New Features on Dense Matrix Multiplication

same. This results in larger block sizes, influencing the duration of the Compute and Copy Codelets as well as the values of W and R, while leaving the overall input size unmodified. This is demonstrated in Figure 7.16. While the asymptote at approximately 250 GFLOPS is reached sooner (at approximately 600 threads as opposed to 800), the maximum performance remains the same. This is because, with so many threads, the limiting factor is memory bandwidth. This is also coherent with the results of Figure 7.12 where we show that, for 156 threads, increasing the size of on-chip memory does not improve performance.

However, with our approach we can easily investigate the benefits of increased memory bandwidth. To do this, we considered the original problem size for off-chip Matrix Multiplication, modified to have twice as much memory bandwidth. We achieved this by doubling the R and Q parameters in Figure 7.6 and altering the Copy Codelet's duration accordingly. The results of this can be seen in Figure 7.16. As expected, an increase in memory bandwidth allows for more threads to be utilized concurrently in the Copy Codelets, resulting in increasing performance out to at least 1000 threads.

7.4.3 Preliminary Analysis of New Algorithms

An additional benefit of our approach is that we are able to model architectural trade offs with respect to new algorithms. We demonstrate this by modeling multiple versions of the tiled LU Factorization algorithm for factorization of a matrix A as the product of a lower triangular matrix L and an upper triangular matrix U, our implementation uses optimized synchronization using low latency hardware barriers [47].

First, we consider a naive LU Factorization, as seen in Figure 7.17. Then, we examine the benefit of a more complex algorithm that utilizes lookahead techniques in Figure 7.18. For readability's sake, we did not show the resource coordination conditions, but each compute codelet requires a single token corresponding to a thread unit, as per Figure 7.11. Each petri net corresponds to an LU Factorization of a Matrix of size $(N \cdot B) \times (N \cdot B)$. Where B is the size of the Block processed by a Codelet and N is the number of Blocks per dimension. For this particular case $B = 6$.

The LU Factorization is an iterative algorithm. Because of its nature, the amount of work varies with each iteration. Obviously, this presents a challenge for our Petri net model because it results in arcs with variable weights. This is represented in our graph as $N - j - 1$, with j representing the current iteration. An estimation of the transition duration for the Codelets was determined by counting the arithmetic operations and the data required, supposing that instructions were correctly scheduled to avoid stalls.

The results of the LU Factorization are shown in Figures 7.19 and 7.20. Figure 7.19 is a plot of the speed-up for LU modeled with 156 thread units for varying problem sizes. Figure 7.20 considers a fixed matrix size and varies the number of threads.

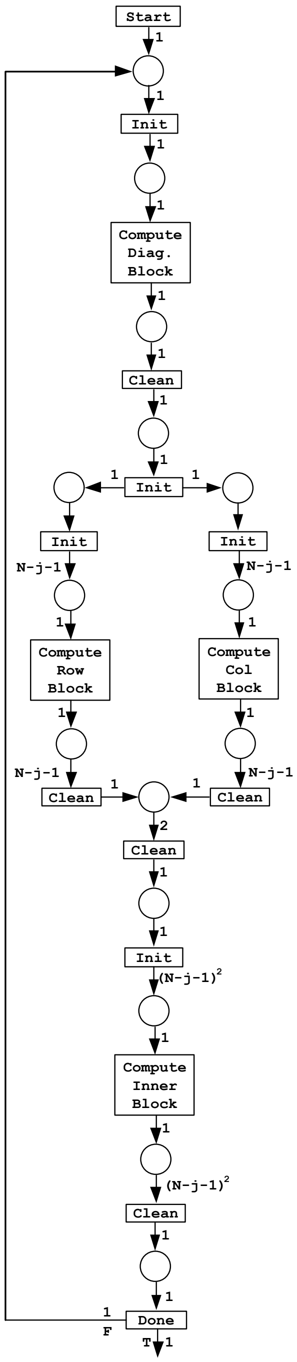


Figure 7.17: Petri Net Modeling of LU Factorization

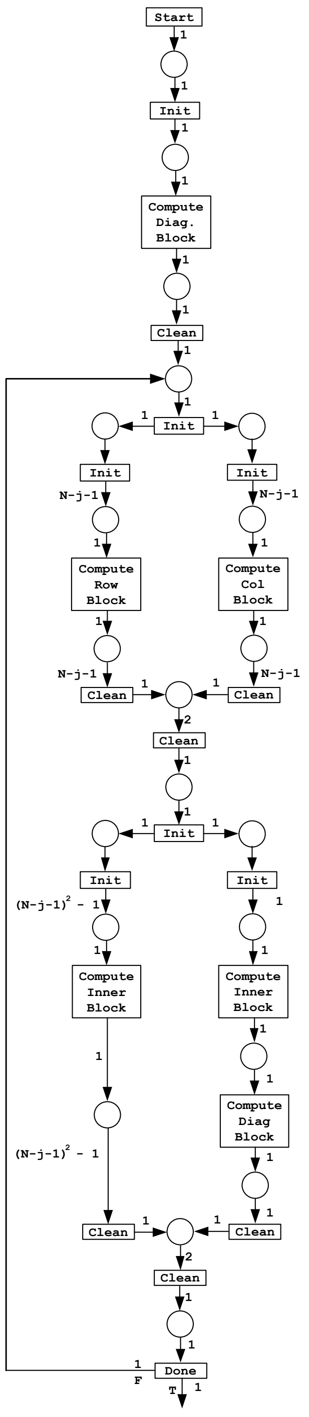


Figure 7.18: Petri Net Modeling of LU Factorization with Lookahead of 1

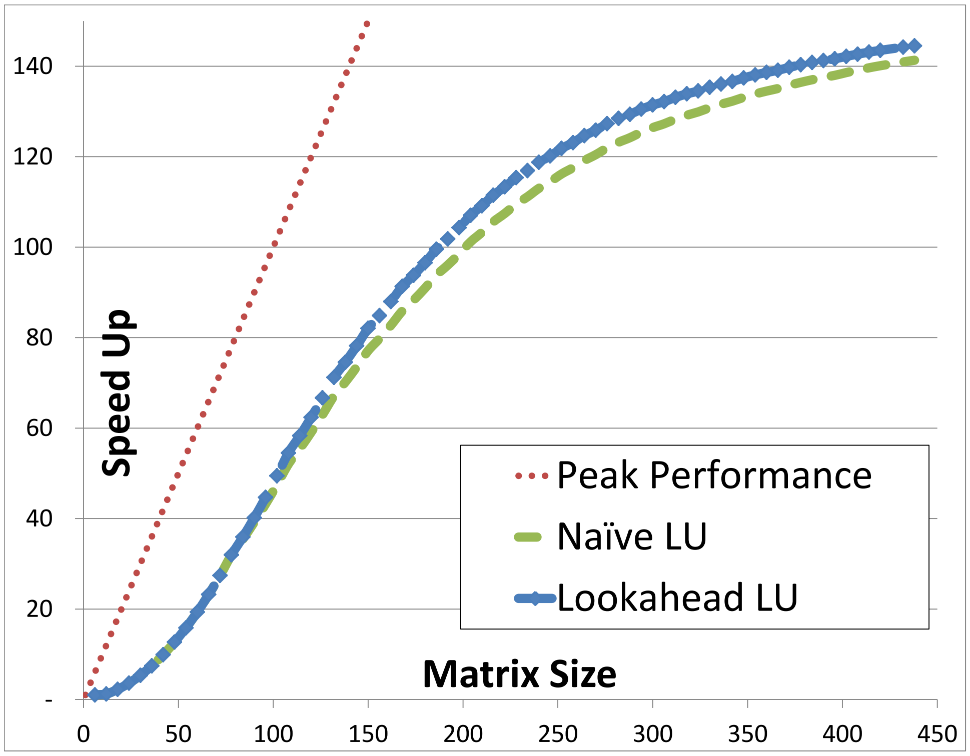


Figure 7.19: Predicted Results for LU Factorization - Speed-Up For 156 Threads Relative to Matrix Size

Figure 7.19 demonstrate that, even the fine grain tasking implemented, both algorithms share the same limitations on scalability with the matrix size for a fixed number of threads. On the other side, Figure 7.20 shows that while for few threads there are not significant differences between both algorithms, the lookahead of 1 provides benefits in performance when using more than 100 threads. In addition, the scalability of both algorithms with respect to the number of threads and the maximum performance increment with the matrix size.

With these results, we are able to make informed decisions regarding the design and optimization of the LU Factorization algorithm on Cyclops-64. While a lookahead of 1 still provides some benefits at cost of complexity in the implementation, it is an indication of how we can continue to improve the algorithm. With this information, we

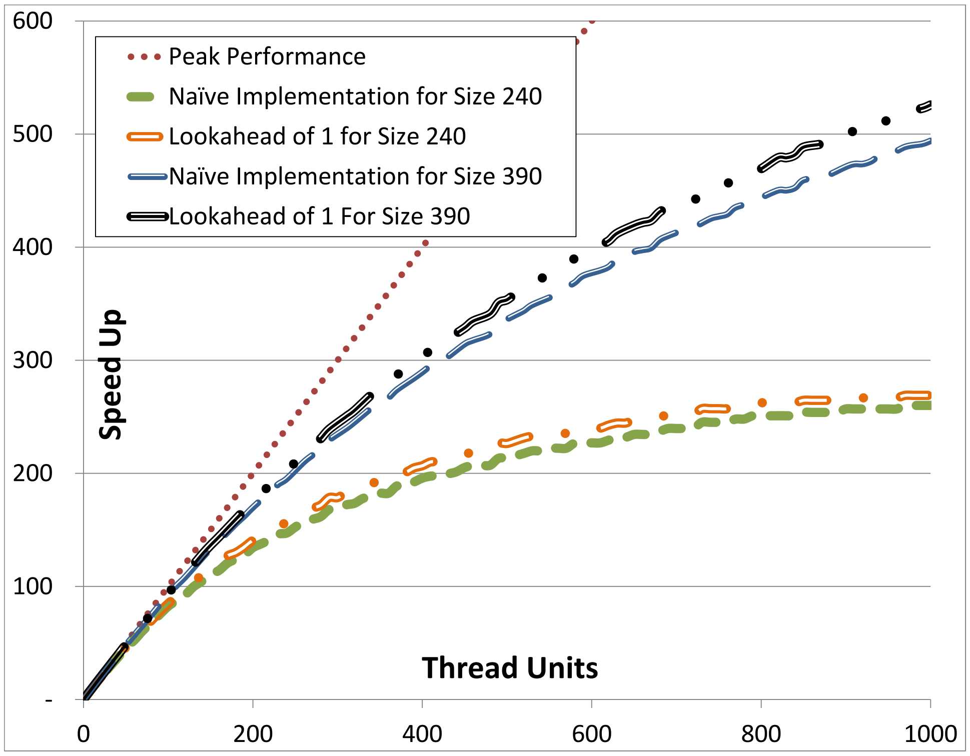


Figure 7.20: Predicted Results for LU Factorization - Speed-Up Relative to the Number of Threads

can examine the benefits of a larger lookahead as well as other optimization techniques. And, by examining the benefit of many of these optimizations with our model, we are able to save time by avoiding the costly process of coding and profiling our algorithms on the actual Cyclops-64 machine.

Chapter 8

POWER AWARE TILING TRANSFORMATIONS

The rapid progress of technology has made possible the integration of large number of processing cores on a single chip. As a consequence, parallel computing design has turned of special interest to the scientific community. Indeed, many-core and multi-core architectures have risen as the solution to most of the issues facing the field of high-performance computing. Energy efficiency and power consumption have become an imperative requirement, the design of new generation of exa-scale supercomputers is restricted to feasible power requirements [95, 96].

Integration of processors on a chip becomes challenging at different levels. From the point of view of semiconductor manufacturing process, new technologies and materials are needed for increasing the number of transistors per area. The integration of hundreds of processors on a single chip under area constraints and the significant increase on leakage current requires the redesign of traditional uniprocessor architectures with deep pipelines, complex branch prediction hardware and a cache-based memory hierarchy.

Particularly, traditional parallel programming methodologies have been focusing on improving performance and they assume cache-based parallel systems exploiting temporal locality. However, the data location and replacement in the cache is controlled by hardware making difficult a fine control and wasting energy [43, 44]. As a result, innovative architectures have arisen; one, unique on its type, is the IBM Cyclops-64 (C64) many-core-on-a-chip system. C64 contains 160 hardware Thread Units (TU) and it has a software-managed memory hierarchy where the data movement between different levels of the hierarchy is managed by the programmer. It saves the die area of hardware cache controllers and over-sized caches. Although this might complicate programming at their current stage, these systems provide more flexibility and opportunities to improve not only performance but also energy efficiency.

Several studies focusing on increasing the performance of a broad range of applications have been done on this architecture (e.g. Matrix Multiplication, LU Factorization, Fast Fourier Transform, etc) [61, 45, 46, 47], but none of these techniques has directly considered the energy efficiency as a goal. Despite of that, some of them have provided evidence of the power efficiency of C64 [61, 68].

In this chapter, we develop an energy consumption model for many-core architectures with software-managed memory hierarchy. The energy consumption model depends of the number and type of instructions executed and the total execution time of the application. We use the C64 many-core architecture to illustrate that our model is scalable with the number of hardware thread units and it considers stalls produced by data dependencies or arbitration of shared resources.

We also propose a general methodology for designing tiling techniques for energy efficient applications. The methodology proposed is based on an optimization problem that produces optimal tiling and sequence of traversing tiles minimizing the energy consumed and parameterized by the sizes of each level in the memory hierarchy. We show two different techniques for solving the optimization problem for two different applications: Matrix Multiplication (MM) and Finite Difference Time Domain (FDTD). Our experimental evaluation uses a real IBM Cyclops-64 chip (C64) that proves the accuracy of our energy consumption model and shows that the techniques proposed reduce the total energy consumption and also increase the power efficiency.

8.1 Energy Consumption Model on a Many-Core Architecture

In this section we will propose a model for energy consumption on general purpose many-core architectures with software-managed memory hierarchy. Given our special interest on scalability, C64 (described in Section 4) seems the only one that has more than one hundred hardware threads and it has already been built.

Our energy consumption model has two main components. The first one is called *static energy* E_s, it comes from the leakage currents and other units that work continuously such as the clock. This component is a function of time t.

The second one is called *dynamic energy* E_d, it is the energy consumed by each functional unit in the execution of some instruction without the leakage component. It is related with the power consumption of transistors on registers and logic during switching, also called dynamic power.

Based on that, given a program Λ with K instructions I_j, the energy consumed can be expressed by:

$$E_T(\Lambda) = E_s(t) + \sum_{j=1}^{K} E_d\left(I_j\right) \tag{8.1}$$

Clearly, the model can be detailed even more because the power dissipated by leakage current is constant (given the absence of mechanism for reducing voltage or turning off functional units in C64) and also other units are always working at the same frequency (given the absence for changing this parameter). In other words E_s is linear with time.

In a similar way, instructions that use the same resources doing a similar amount of work, like the hierarchy explained on section 4.2, consume the same amount of energy. This linearity helps us to express our energy consumption model by:

$$E_T(\Lambda) = e_0 \cdot t + \sum_{i=1}^{M} e_i \cdot N\left(C_i\right) \tag{8.2}$$

Where e_0 is the static power dissipated, and e_i for $i = 1, \ldots, M$ is the energy consumed by one instruction of class C_i. The function $N(\cdot)$ counts the number of instructions in the program Λ that belong to a given class. This class can have only one instruction (e.g. when the kind of processing and the functional units that it uses are unique like integer multiplication) or multiple instructions (e.g. when they are similar in terms of amount of work and use the same resources like all the logical operations)

This model also considers the case of shared resources and overlapping, extremely important on many-core. First, each instruction represents the use of some resources for some task and it would take similar time. In a scenario of contention (e.g. the crossbar network for accessing memory), the amount of work made by the functional units will be the same but the time will increase. This will be reflected on the increase in the term that correspond to static energy. In a similar way, in the same processor multiple units can work in parallel (e.g. Floating Point Unit and Integer Unit) taking less time to complete the tasks compared with the sequential execution, as a result the term for static energy will decrease but the dynamic energy will remain similar. Even more important, for a chip with more than a hundred of processors, the dynamic energy terms reflect the energy per instruction regardless of whether it was executed in parallel with others or serially.

In Addition, it is natural to think than some instructions (or group of them) consumes more energy than others, some cases are:

- An operation that requires more computations than another of the same type. (e.g. integer multiplication vs. integer addition).
- An operation that uses a more complex hardware than another one. (e.g. floating point addition vs integer addition, on-chip memory operations vs integer operations).
- An operation that uses off-chip resources compared with one that only uses on-chip resources (e.g. load from DRAM vs load from SRAM).

8.2 Tiling Techniques for Energy Efficient Applications

In this section we will analyze the problem of designing tiling techniques for energy efficiency. Although instruction scheduling techniques are able to hide latency of operations, this kind of techniques are not useful here because *dynamic energy* E_d can not be hidden. We propose to find a feasible tiling that minimizes the total energy cost by minimizing the energy contribution of the most energy hungry instructions.

The optimization problem proposed is based on two facts: (1) Memory operations on off-chip memory are the most expensive in terms of energy, followed by on-chip memory operations. (2) There is not a dependency between different latencies for the same operation (e.g. contention of memory operations) and the dynamic energy it consumes. These two facts will be proved on section 8.3.1.

Our objective is to find the tiling T described by its parameters L and the sequence of traversing tiles S that minimize the consumed Dynamic Energy E_d on Γ processors by the subset of most energy hungry instructions I_E subject to the data stored D_H at each level H of the memory hierarchy cannot exceed the maximum memory size available $\mathrm{Mem}_{H\max}$ and the tiling allows parallel computation without communication between tiles. According to our model described on eq. (8.2), this Dynamic Energy E_d for a problem Λ is function of the number of instructions $N(\Lambda, I_j)$ with $I_j \in I_E$ and its energy coefficients e_j. This can be expressed as the optimization problem:

$$\begin{aligned} \min_{T(L,S)} \quad & E_d\left(I_E\right) = \sum_{I_j \in I_E} \left(e_j \cdot N\left(I_j\right)\right) \\ s.t. \quad & D_H\left(\Lambda, \Gamma, T\right) \leq \mathrm{Mem}_{H\max} \\ & T \text{ is parallel} \end{aligned} \tag{8.3}$$

Given the fact that memory operations are the most energy hungry instructions on most architectures and particularly on the C64 many-core architecture. The particular optimization problem using the Load LD and Store ST instructions for *off-chip* memory (DRAM) and *on-chip* memory (SRAM) is:

$$\begin{aligned} \min_{T(L,S)} \quad & e_1 N(LD_{\text{dram}}) + e_2 N(ST_{\text{dram}}) + e_3 N(LD_{\text{sram}}) + e_4 N(ST_{\text{sram}}) \\ s.t. \quad & D_H\left(\Lambda, \Gamma, T\right) \leq \mathrm{Mem}_{H\max} \\ & T \text{ is parallel} \end{aligned} \tag{8.4}$$

Where $N(LD)$ and $N(ST)$ are also function of Λ, Γ, T.

The optimization problem described by 8.3 and 8.4 cannot be easily solved. Even more, there is not guarantee of analytical solution. The following subsections will show two approaches for solving these kind of optimization problems for two kind of applications: Matrix Multiplication (MM) and Finite Difference Time Domain (FDTD).

8.2.1 Matrix Multiplication

Despite Matrix Multiplication (MM) algorithms have been studied extensively, the many-core architecture design space has not yet been explored in detail. MM is extremely important on scientific applications that use linear algebra. Our target operation is the multiplication of dense square matrices $A \times B = C$, each of size $m \times m$ using algorithms of running time $O(m^3)$. We will focus on matrices that fit in on-chip memory SRAM and the memory operations will be load and store from SRAM to registers. For this case, the optimization problem on 8.4 becomes:

$$\begin{aligned} \min_{T(L,S)} \quad & e_3 N(LD_{\text{sram}}) + e_4 N(ST_{\text{sram}}) \\ s.t. \quad & R\left(\Lambda, \Gamma, T\right) \leq R_{\max} \\ & T \text{ is parallel} \end{aligned} \tag{8.5}$$

An optimal partition for a load-balanced distribution between processors P assumes blocks $C' \in C$ of size $n \times n$ $\left(n = \frac{m}{\sqrt{\Gamma}}\right)$. Each block is subdivided in tiles $C'_{i,j} \in C'$ of size $L_2 \times L_2$. Based on the data dependencies, the required blocks $A' \in A$ and $B' \in B$ of sizes $n \times m$ and $m \times n$ are subdivided in tiles $A'_{i,j} \in A'$ and $B'_{i,j} \in B'$ of sizes $L_2 \times L_1$ and $L_1 \times L_2$ respectively. Each tile can be calculate using $C'_{i,j} = \sum_{k=1}^{m/L_2} A'_{i,k} \cdot B'_{k,j}$.

The number of loads and stores can be calculated analytically for each one of the 6 alternatives for traversing tiles that can be summarize on two sequences S_1, S_2. The specific optimization problem now becomes:

$$\min_{\substack{L \in \{L_1, L_2\}, \\ S \in \{S_1, S_2\}}} \quad f\left(m, \Gamma, L, S\right) = \begin{cases} \frac{2e_3}{L_2} m^3 + e_4 m^2 & \text{if } S = S_1 \\ \left(\frac{e_3 + e_4}{L_1} + \frac{e_3}{L_2}\right) m^3 + e_3 \left(\sqrt{\Gamma} - 1\right) m^2 & \text{if } S = S_2 \end{cases} \tag{8.6}$$

$$\text{s.t.} \quad 2L_1 L_2 + L_2^2 \leq R_{\max}, \quad L_1, L_2 \in \mathbb{Z}^+$$

Analyzing the piecewise function f, it can be easily shown that S_1 sequence has an smaller objective function than S_2 under the conditions $\frac{e_4}{e_3} \leq \sqrt{\Gamma} - 1$ and $\frac{L_2}{L_1} \geq \frac{e_3}{e_3+e_4}$. The first one is easily satisfied if Γ is big enough, the second one can be satisfied when $L_2 \geq L_1$ and it can be verified with the solution.

We will solve the integer optimization problem using the branch and bound technique. Since f only depends on L_2, we minimize the function f by maximizing L_2. Given the constraint, L_2 is maximized by minimizing L_1. Thus $L_1 = 1$, we solve the optimum L_2 in the boundary of the constraint and round off it. The solution of Eq. (8.6) is:

$$L_1 = 1,\ L_2 = \left\lfloor \sqrt{1 + R_{\max}} - 1 \right\rfloor \tag{8.7}$$

The solution satisfies the constraints and also proves the hypothesis $L_2 \geq L_1$, finishing the branch and bound process. This result is not completely accurate, since we assumed that there are not remainders when we divide the matrices into blocks and subdivide the blocks in tiles. Despite this fact, they can be used as a good estimate.

For comparison purposes, C64 has 63 registers and we need to keep one register for the stack pointer, pointers to A, B, C matrices, m and stride parameters, then $R_{\max} = 63 - 6 = 57$ and the solution of Eq. (8.7) is $L_1 = 1$ and $L_2 = 6$. Table 8.1 summarizes the results in terms of dynamic energy consumed by LDs and STs for the tiling proposed and other 2 options that fully utilizes the registers and have been used in practical algorithms: inner product of vectors ($L_1 = 28$ and $L_2 = 1$) and square tiles ($L_1 = L_2 = 4$). As a consequence of using sequence S_1, the dynamic energy of STs is equal in all tiling strategies. As expected, the tiling proposed consumes minimum energy: approximately 6 times less than the inner product tiling and 1.5 times less than the square tiling.

8.2.2 Finite Difference Time Domain

The Finite Difference Time Domain (FDTD) [97] technique is a common algorithm to simulate the propagation of electromagnetic waves through direct solution

Table 8.1: E_d consumed by memory operations for MM

Memory Operations	Inner Product	Square	Optimal
Loads	$2e_3m^3$	$\frac{e_3}{2}m^3$	$\frac{e_3}{3}m^3$
Stores	e_4m^2	e_4m^2	e_4m^2

of Maxwell's Equations. FDTD was chosen to illustrate the techniques presented here since it is easy to understand, it is widely used, and it can be easily written for multiple dimensions. Specifically, we will study FDTD in one dimension i of size m and q time steps. The data is read directly from off-chip memory with tiles on on-chip memory. For this case, the optimization problem on eq. 8.4 becomes:

$$\begin{aligned} \min_{T(L,S)} \quad & e_1N(LD_{\text{dram}}) + e_2N(ST_{\text{dram}}) \\ s.t. \quad & \text{Mem}_{\text{sram}}\,(\Lambda, \Gamma, T) \leq \text{Mem}_{\text{max}} \\ & T \text{ is parallel} \end{aligned} \tag{8.8}$$

The solution of this problem is based on the analysis of its Data Dependency Graph (DDG) that can be detailed on Figure 8.1. Our solution is inspired by [81] where they find the tiling that maximize the data reuse. Because the number of useful computations can not be decreased by the tiling. For a FDTD problem of size fixed size, maximize the data reused is equivalent to minimize the number of memory operations $N(LD_{\text{dram}}) + N(ST_{\text{dram}})$. In addition, given the regularity of the DDG, a tiling that saves energy will not load extra data for doing extra computations. It means that the number of loads and stores will be the same. In that order the ideas, the *diamond tiling* showed on Figure 8.2 solves the optimization problem given by eq. 8.8

Table 8.2 summarizes the results in terms of dynamic energy consumed by LDs and STs for the tiling proposed and other 3 well-known techniques [98]. The unit for the tile size L is the node $E[i], H[i]$. Clearly, Diamond tiling for FDTD has the smallest coefficients.

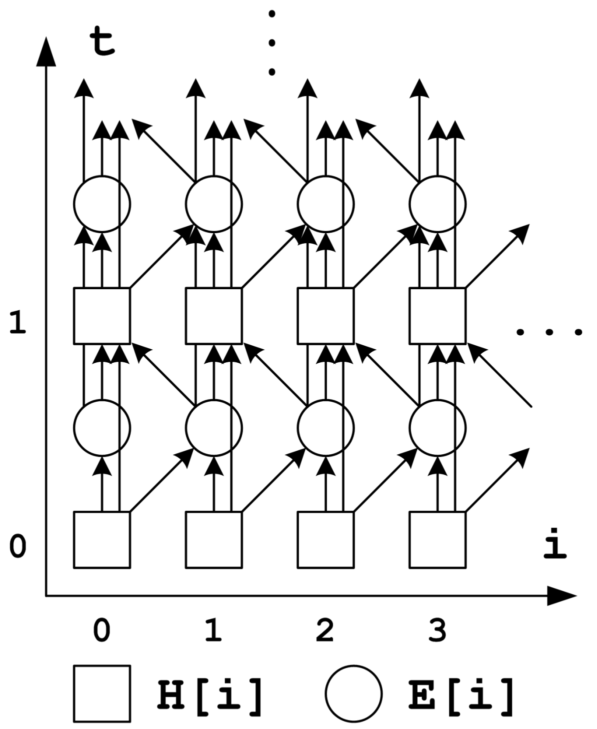


Figure 8.1: DDG for FDTD 1D

8.3 Experimental Evaluation

This section describes the experimental evaluation of the proposed energy consumption model given in section 8.1 and the tiling techniques for energy efficiency analyzed in section 8.2.

8.3.1 Evaluation of the Energy Consumption Model

The energy coefficients e_i where obtained using measurements of current and voltage from the power supplies in a real chip. The instantaneous power $P[t]$ at time t

Table 8.2: E_d consumed by memory operations for FDTD

Memory Operations	Naive	Split	Overlapped	Diamond
Loads	e_1qm	$\frac{9e_1}{2L}qm$	$\frac{9e_1}{L}qm$	$\frac{2e_1}{L}qm$
Stores	e_2qm	$\frac{9e_2}{2L}qm$	$\frac{3e_2}{L}qm$	$\frac{2e_2}{L}qm$

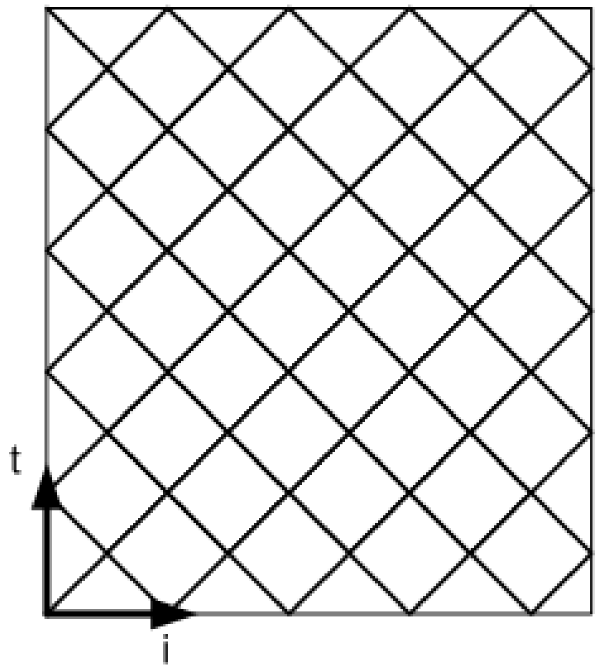


Figure 8.2: Diamond Tiling

can be calculated using $P[t] = v_1[t] \cdot i_1[t] + v_2[t] \cdot i_2[t]$, the average power $\bar{P}$ is estimated by the mean of several samples of $P[t]$ and the total energy consumed is $E_T = \bar{P} \cdot t$.

A test bed for the ISA of C64 was created for the estimation of the energy coefficients e_i of (8.2). The test bed include multiple programs, each one with a known number of instructions for a subset of the ISA. The estimation of $e_0 = 63.11W$ was straight forward calculated only measuring the consumption of the system on standby. Notice that while e_0 is estimated in Watts, e_i for $i > 0$ is estimated in Joules/Instruction.

The dynamic energy E_d for a program Λ running in parallel on Γ processors with a fixed number of instructions of class I_j per processor can be estimated by eq. 8.9

$$E_d(\Lambda, I_j, \Gamma) = \left(\bar{P} - e_0\right) \cdot t \tag{8.9}$$

The results for a representative subset of the ISA are shown on Figure 8.3. As shown on Figure 8.3, load and store on DRAM (ldddram, stddram) are the most

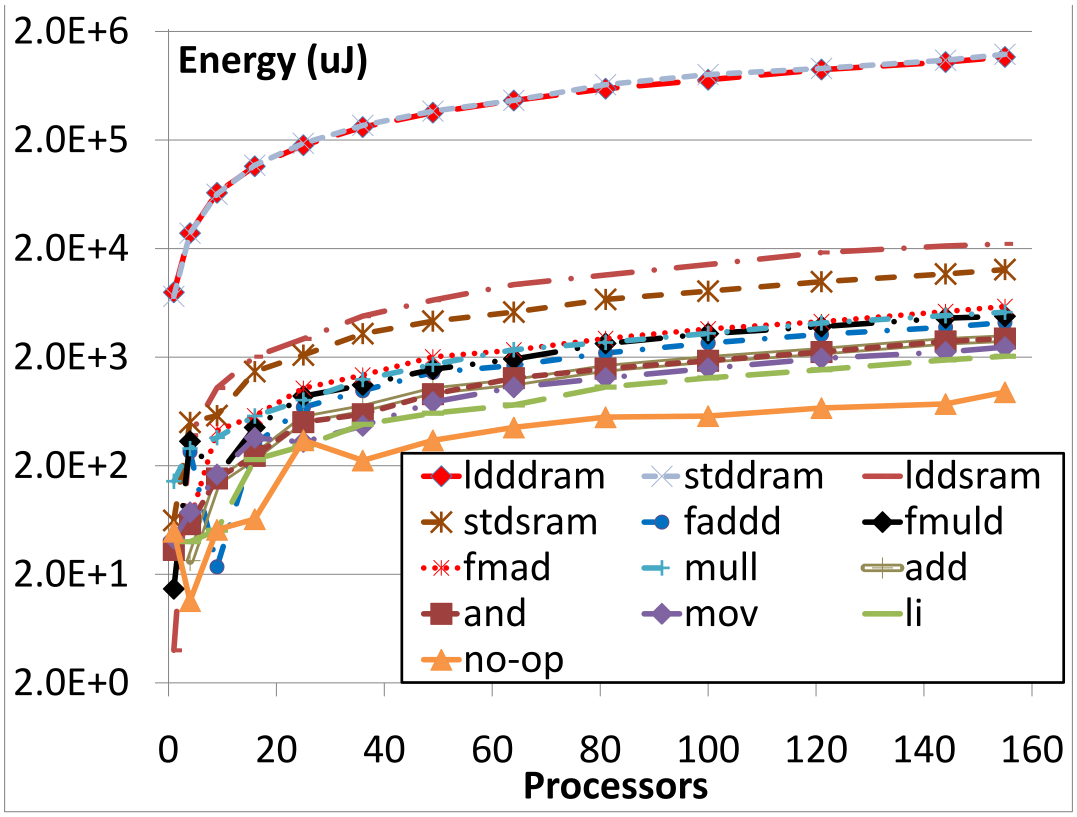


Figure 8.3: Overall comparison of selected ISA

energy hungry, followed by load and store on SRAM (lddsram, stdsram), the difference of energy consumption between DRAM and SRAM operations is almost 2 orders of magnitude. Figure 8.4 proves the linearity of energy consumption with Γ. It details that after memory operations, floating point operations (fmaddd, fmuld and fmad) and difficult integer operations (mull) consumes similar energy. Integer, logical and register movement operations (add, and, mov, li) are on the bottom of the list. The instruction that consumes less is no-op as expected.

The remainder energy coefficients e can be extrapolated using a linear regression from the E_d estimated for each instruction. We used a model with intercept at origin given the assumption that no dynamic energy is consumed on standby. The resultant

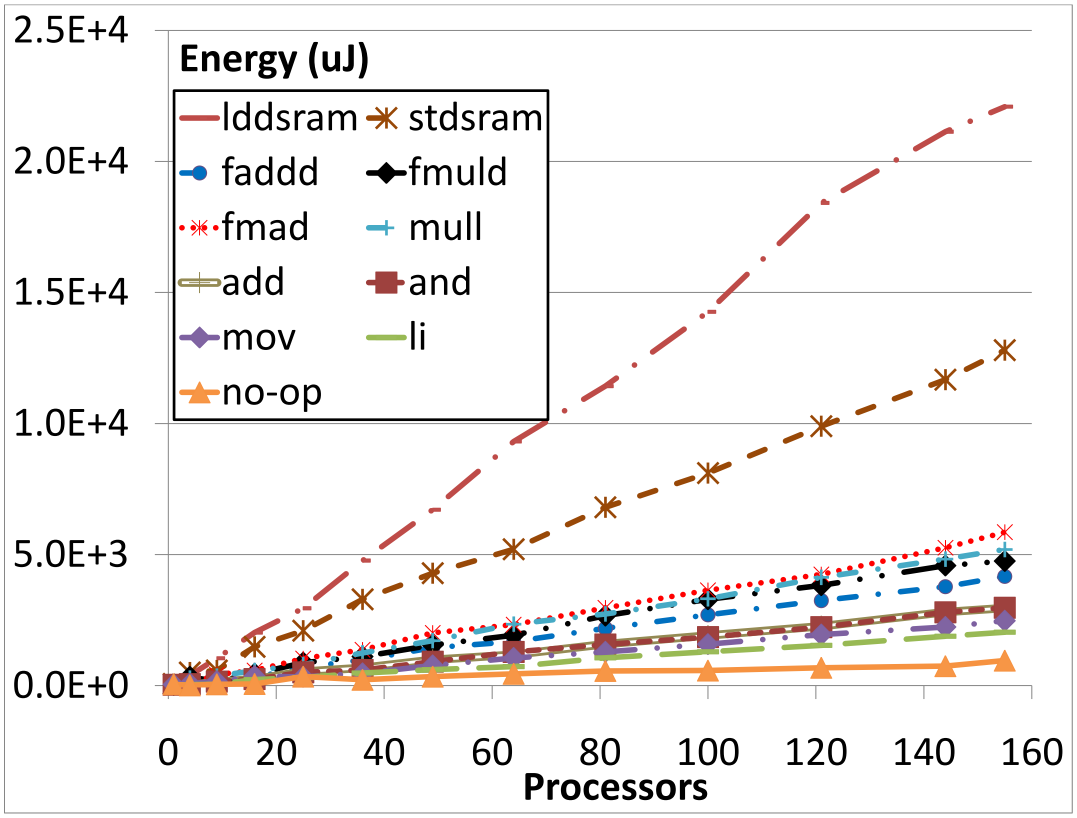


Figure 8.4: Comparison for On-chip Mem. Op., FPU Op. and Integer/Logical Op.

coefficients e for a subset of the ISA is shown on Table 8.3. The table also includes the coefficients of determination R^2 for measuring the variability between the data and the model proposed. As expected, a linear approximation with the number of processors models accurately E_d, its coefficients R^2 are really close to 1, it corroborates that there is not dependency between the latency of the operation and the dynamic energy consumed. Some additional aspects to highlight are: (1) Instead DRAM operations consume similar energy, a load from SRAM consumes almost twice the energy of an store to SRAM. (2) Despite the floating point fused-multiply-add (fmad) consumes a little bit more energy than a simple floating point multiply (fmuld) or floating point add (faddd), notice that one fmad executes a multiply and an addition. At the end, an

Table 8.3: Energy Coefficients e and R^2

Instruction	e[pJ/Operation]	R^2
ldddram	48924.10	0.999
stddram	51488.99	0.998
lddsram	964.65	0.997
stdsram	548.31	0.999
fmad	245.27	0.997
faddd	178.30	0.995
fmuld	210.15	0.996
mull	225.43	0.998
add	127.65	0.998
and	126.69	0.998
mov	105.48	0.996
li	86.01	0.997
no-op	39.66	0.936

fmad saves around 63% of energy compare with separates fmuld and faddd. (3) Integer and floating point multiplication cost similar, the same is true for logical and simple integer operations. The last two observations confirms the high correlation between the energy consumption of an instruction and the related hardware and functional units the instruction requires.

8.3.2 Evaluation of the Energy Efficient Tiling

We will use the Matrix Multiplication (MM) explained before for showing the advantages of the tilings that solve the optimization problems of section 8.2. First, we will compare the estimated energy consumption using the coefficients of section 8.3.1 with the measured energy based on voltage and current on the real chip. Second, we will compare energy consumption of the tiling proposed with other well known tiling techniques.

For MM we use a matrix size that fits on SRAM, we compare our approach *(OptT)* with the register tiling based on dot product *(DPT)*. Both methods uses assembly for taking advantage of the complete register file. For FDTD, the tile size is the maximum possible that fits on SPM, we compare our diamond tiling *(DmT)* with 3 well-known techniques: A rectangular tiling (naive) *(NT)*, the overlapped tiling *(OT)* that uses redundant computations in order to tile time and space dimensions and split tiling *(ST)* that uses multiple shapes for fully partitioning the iteration space [98].

Figure 8.5 compares the energy consumption measured with the energy predicted by our model for the MM application. We can see how the predictions are highly close to the measured value for the dynamic and static components. The average error of our model for E_d and E_T is 26.6% and 0.82% respectively. We also noticed how the tiling proposed decreases substantially the dynamic and total energy consumption in 56.52% and 61.21% on average. An interesting result that can be extrapolated from the measurements of performance and power is that the power efficiency [MFLOPS/W] increases between 2.62 and 4.13 times for this test example. For the FDTD application, figure 8.6 shows the effectiveness of diamond tiling for decreasing the total and dynamic energy with respects to the other tiling techniques. The total average energy reduction was 81.26%, 57.27% and 15.69% compared with split tiling, overlapped tiling and naive tiling respectively. Also our energy consumption model is accurate to the real behavior of the application, the average error is 7.3% for E_T.

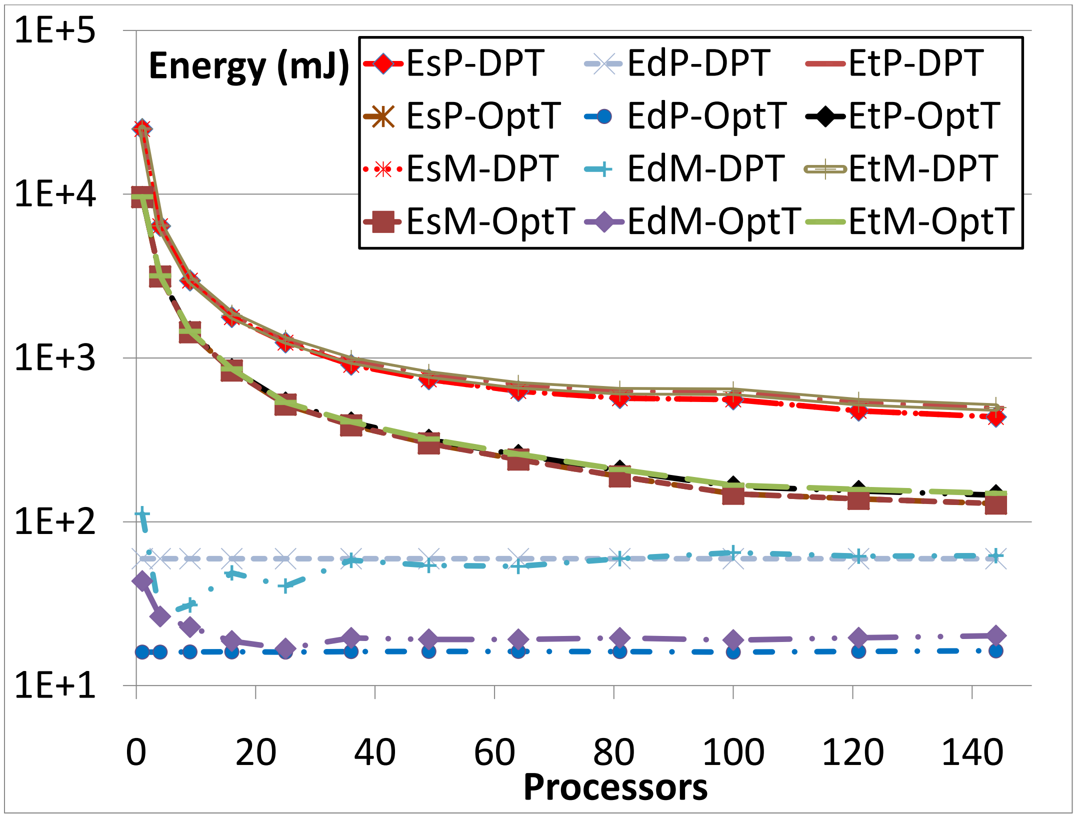


Figure 8.5: Energy consumption (Static Es, Dynamic Ed and Total Et) vs Predicted model P and Measured M using different tilings for MM with $m = 300$

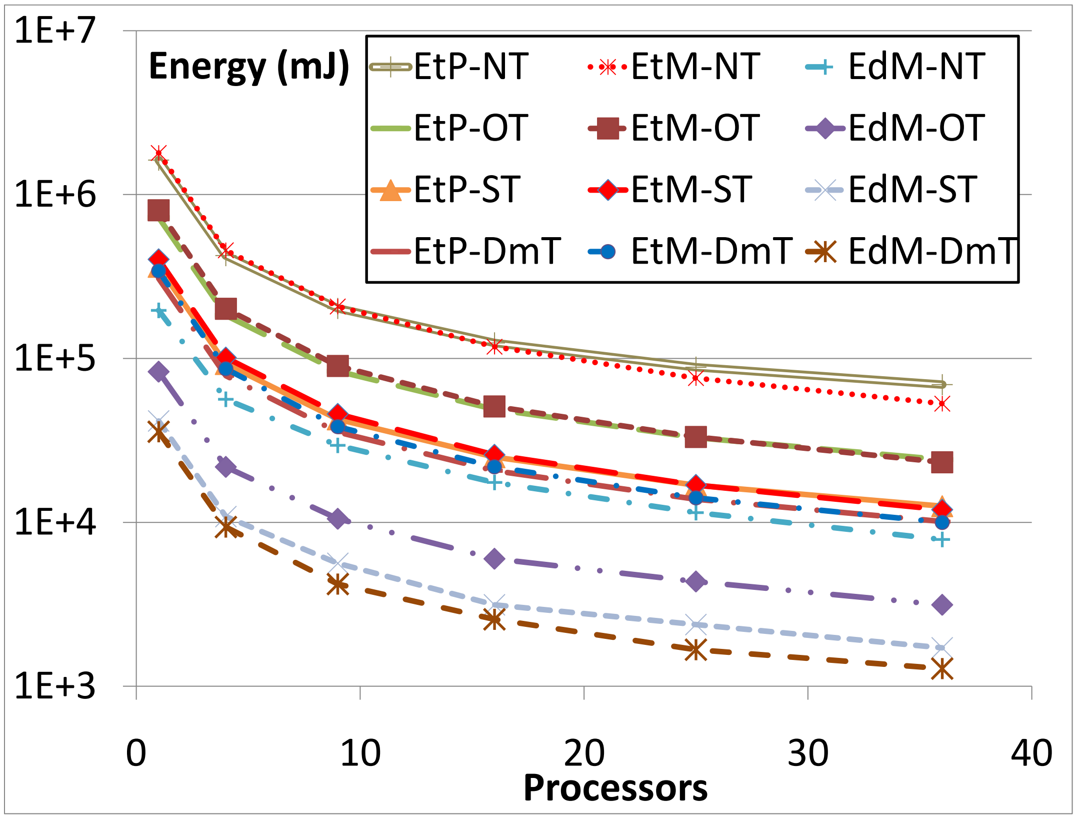


Figure 8.6: Energy consumption (Static *Es*, Dynamic *Ed* and Total *Et*) vs Predicted model *P* and Measured *M* using different tilings for FDTD with $m = 100k$ and $q = 500$

Chapter 9

ENERGY OPTIMIZATIONS IN THE CONTEXT OF MANY-CORE ARCHITECTURES

Recently, the many-core revolution brought forward by advances in computer architecture has reached another wall. In the past, major efforts and progress have been made in order to achieve high performance on many-core chips. In particular, optimizations have been developed to improve the number of Floating Point Operations per Second. However, recent developments have shifted the focus to other constraints [80] such as energy consumption. The design of the new generation of exa-scale supercomputers is restricted by power requirements [95, 96]. As a result, Energy efficiency and power consumption have become an imperative.

Energy efficiency is limited by many factors. From the point of view of semiconductor manufacturing processes, the integration of hundreds of independent processors on a single chip within a given area results in an increase in temperature and leakage current. This, in turn, results in more energy and transistors dedicated toward cooling and a deep rethinking of traditional architectures.

An interesting case study is the IBM Cyclops-64 many-core architecture [99] (described in Chapter 4). Many-core with a software-managed memory hierarchy where the programmer controls data movement are a feasible alternative to improve energy efficiency at the cost of a higher complexity with respect to programmability.

Extensive studies on performance for the Cyclops-64 have been performed in the past (e.g. Matrix Multiplication, FFT, LU Factorization, etc) [61, 62, 47, 81] and the impact of particular compiler optimization techniques have been described in detail [80, 82, 39, 100, 92]. energy efficiency has only recently been studied with early efforts resulting in a scalable energy consumption model for Cyclops-64 [101]. A deep understanding of this model can allow for the design of specific optimizations to decrease energy consumption. This particular model has been described in Chapter 8

In this Chapter, we study and implement several techniques to target energy efficiency on many-core architectures with software managed memory hierarchies. We use the LU factorization as a case of study for techniques proposed. We study the impact of these techniques on the Static Energy and the Dynamic Energy using a scalable energy consumption model described by Garcia et. al. [101]. The main contributions of this chapter are: First, the modeling and analysis of energy consumption and energy efficiency for LU factorization; second, the study and design of instruction-level and task-level optimizations for the reduction of Static and Dynamic energy; third, the design and implementation of an energy aware tiling for the LU factorization benchmark; and fourth, the experimental evaluation of the scalability and improvement in energy consumption and energy efficiency of the proposed optimizations using the IBM Cyclops-64 many-core. The proposed optimizations for energy efficiency increase the power efficiency of the LU factorization benchmark by 1.68X to 4.87X, depending on the problem size, with respect to a highly optimized version designed for performance.

The rest of this chapter is organized as follows. In Section 9.1, we discuss the basics of a parallel LU factorization algorithm. In Section 9.2, we study the impact of several optimizations in the Static and Dynamic Energy. Finally, in Section 9.3, we present the experimental evaluation of the proposed optimizations.

9.1 LU Factorization

The LU factorization is a matrix factorization which represents the product of two matrices; a lower triangular matrix, L, and an upper triangular matrix, U. This algorithm is often used in linear systems in order to solve linear equations. Assuming A to be a square matrix, it can be represented as $A = L \times U$. This type of LU factorization is called *without pivoting* and is the one presented in this document. An LU factorization with pivoting performs a permutation of the rows or columns of the matrix A using one of several strategies such as Partial Pivoting, Partial Scaled

Pivoting, Total Pivoting, or Total Scaled Pivoting. A comprehensive study of different pivoting strategies for LU factorization can be found in [102].

Because the LU factorization is a well studied algorithm, there are many variations such as the Linpack benchmark [103], High Performance Linpack (a parallel version of Linpack) [104], and the SPLASH-2 suite [105].

The classical approach for parallel LU factorization in cache-based systems uses fixed-size blocks that fit into cache to distribute the workload among threads. As shown in Figure 9.1, in the first step of the algorithm the matrix A is divided into one *Diagonal* block and several *Column*, *Row*, and *Inner* blocks. Each block is assigned to one processing element, which further divides the block into tiles in order to improve data reuse and locality. At this point, the *Diagonal* block is computed individually by one processing element, followed by a concurrent computation of the *Column* and *Row* blocks. Once all the *Column* and *Row* blocks have been computed, the *Inner* blocks are processed. In the second step of the algorithm, the *Inner* blocks of the previous step are grouped again into one *Diagonal* block and several *Column*, *Row*, and *Inner* blocks, which are computed following the rules previously mentioned. This is repeated until there is only one *Inner* block, which is processed as a *Diagonal* block in the last step. The progression of steps following this classical approach is illustrated at the top of Figure 9.1. As can be seen, the number of blocks (i.e. the number of tasks assigned to the processing elements) decreases as the algorithm moves forward. This is translated into an increasing number of processing elements becoming idle, which lowers the performance of the application.

The *Dynamic Repartitioning* technique proposed by Venetis and Gao [47] uses varying-size blocks in each step of the algorithm in order to optimize the distribution of work among processing elements. As shown at the bottom of Figure 9.1, the size of the blocks is calculated at the beginning of each iteration of the LU factorization. This size is calculated as a function of the number of processing elements, so each processing element has at least one assigned task (i.e. one block to process). This optimization has been proved to increase the overall performance up to $2.8X$ in systems with a

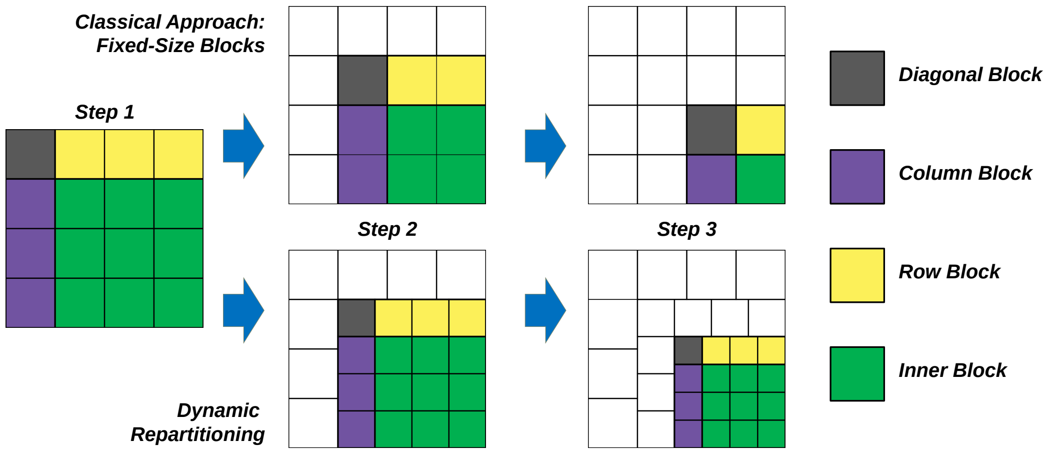


Figure 9.1: Progress in each step of LU Factorization

software managed memory hierarchy [47].

9.2 Energy Optimizations

In this section we will study the impact of several optimizations on the energy consumption of the LU factorization algorithm targeting systems with software managed memory hierarchy such as C64. The impact of these optimizations can affect the two sources of energy consumption described in Section 8.1: Static Energy E_s and Dynamic Energy E_d. Our baseline implementation is the LU factorization without pivoting by Venetis and Gao [47]. They used the *Dynamic Repartitioning* technique described in Section 9.1 and implemented a carefully designed register tiling. All their optimizations were targeting high performance.

While the increase in performance obtained by Venetis and Gao is reflected in savings of Static Energy, this high performance LU implementation has some drawbacks from the Energy consumption point of view: First, its register tiling focuses on increasing locality and it is not aware of the energy consumption of each instruction. Second, the static distribution of work does not consider the variance in completion time of processing similar tasks in presence of shared resources such as memory, crossbar interconnections, and FPUs. And finally, the hierarchical division into blocks and further into tiles, produces an increasing amount of smaller tiles in the borders of each block, which can hurt not just the performance but also the energy consumption.

9.2.1 Energy Aware Tiling design

To reduce the Dynamic Energy consumption of the LU factorization, we will focus on the instructions that contribute the most to it. Using the Energy consumption model described in Section 8.1, we characterized the Dynamic Energy of the LU Factorization implementation optimized for performance by Venetis and Gao [47] using the traces generated during the simulation of the application on a C64 architecture and a matrix of 840×840 allocated in on-chip memory.

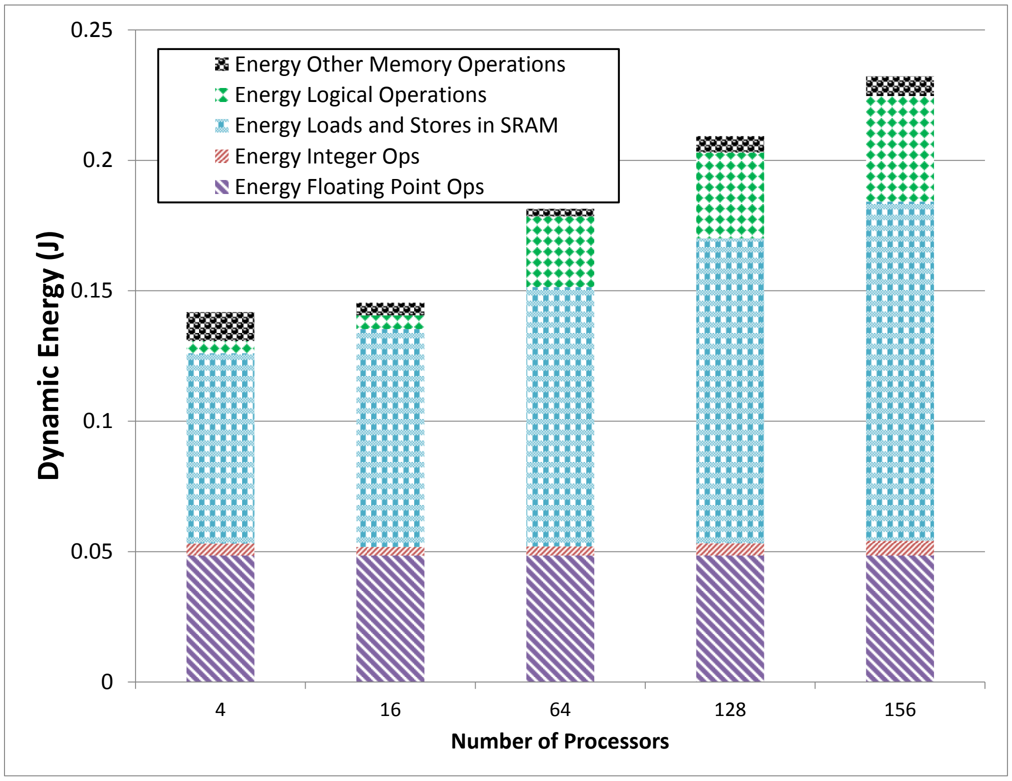


Figure 9.2: Dynamic Energy Distribution for LU factorization of 840×840

Figure 9.2 shows how the Dynamic Energy of the LU factorization increases with the number of processors. As can be seen, Loads and Stores on the on-chip memory

(SRAM) are the instructions with the largest contribution to the Dynamic Energy; this contribution also increases with the number of processors. On the other hand, the Energy of Floating point operations remains constant and the contribution of integer, logical, and other memory operations is not significant.

In order to minimize the Dynamic Energy E_d for a particular algorithm Λ, we propose to minimize the energy contribution of the most power hungry operations, in this case Loads LD and Stores ST with energy coefficients e_1 and e_2. The minimization is done on a set of possible tilings T with parameters S and L (e.g. shape and tile size). The optimization problem is shown in Eq.(9.1) [106].

$$\begin{aligned} \min_{T(L,S)} E_d\left(\Lambda, T\right) \approx \quad & e_1\left|\mathrm{LD}\right| + e_2\left|\mathrm{ST}\right| \\ \text{subject to} \quad & R\left(\Lambda, T\right) \leq R_{\max}, \quad T \text{ is parallel} \end{aligned} \tag{9.1}$$

There are two constraints in the optimization problem: The registers used by the tiling ($R(\Lambda, T)$) need to fit in the available registers R_{max} and the tiling has to allow parallel execution. The former avoids unnecessary energy consumption produced by register spilling and the later prevents solutions with low performance due to increasing execution time produced by inability to exploit task parallelism.

In order to solve this problem for LU factorization, we analyze the energy consumption of each type of block (*Diagonal*, *Row*, *Column* and *Inner*) with sizes $M_0 \times M_0$, $M_0 \times M_1$, $M_2 \times M_0$ and $M_2 \times M_1$ respectively. Each block is assigned to a processor and further divided into tiles. There are 3 cases of sequences to traverse the tiles (e.g. S_0, S_1 and S_2) for each type of block [106]. A detailed explanation of the procedure to find the optimum tiling for the *Inner* block and a summary of the results for the other type of blocks are presented in the next paragraphs.

Inner Blocks: For the computation of an *Inner* block, a *Row* block and a *Column* block are required. *Row*, *Column* and *Inner* blocks are divided into tiles of $L_0 \times L_1$, $L_2 \times L_0$ and $L_2 \times L_1$ respectively. The three possible sequences of traversing tiles reuse tiles on a different operand: The *Row* block (case S_0), the *Column* block (case S_1) and the *Inner* block (case S_2). The problem formulation for the Dynamic Energy is shown in Eq. 9.2.

$$\min_{\substack{L\in\{L_0,L_1,L_2\},\\ S\in\{S_0,S_1,S_2\}}} \quad f\left(L,S\right) = \begin{cases} e_1 M_0 M_1 \left(\frac{M_2}{L_0} + \frac{M_2}{L_1} + 1\right) + \frac{e_2 M_0 M_1 M_2}{L_0} & \text{if } S = S_0 \\ e_1 M_0 M_2 \left(\frac{M_1}{L_0} + \frac{M_1}{L_2} + 1\right) + \frac{e_2 M_0 M_1 M_2}{L_0} & \text{if } S = S_1 \\ e_1 M_1 M_2 \left(\frac{M_0}{L_1} + \frac{M_0}{L_2} + 1\right) + e_2 M_1 M_2 & \text{if } S = S_2 \end{cases} \tag{9.2}$$
$$\text{s.t.} \quad L_0L_1 + L_0L_2 + L_1L_2 \leq R_{\max}, \quad L_0, L_1, L_2 \in \mathbb{Z}^+$$

The the non-linear optimization problem was solved using the Karush Kuhn Tucker conditions. We assumed all the variables being positive and M_0, M_1 and M_2 being bigger or equal than L_0, L_1 and L_2. In addition, we used the fact that M_1 and M_2 are equal to M_0 or M_0+1. We found that the best solution was to reuse the *Inner* tile (case S_2) with parameters $L_0 = 1$, $L_1 = N$ and $L_2 = N$, with $N^2 + 2N \leq R_{\max}$. In this case, an *Inner* block is computed by dividing it into tiles of $N \times N$ elements and loading each *Inner* tile into the registers, which act as accumulators for the partial results. Each partial result is calculated from a pair composed of one tile of $N \times 1$ elements of the corresponding *Column* block and one tile of $1 \times N$ elements of the corresponding *Row* block. The registers used as accumulators are stored back into memory only when there are no more pairs of *Column* and *Row* tiles to process. An example of this process is shown in Figure 9.3

Row Blocks: To compute a *Row* block, this is divided into tiles of $N \times N$ elements (with N being the same as for the *Inner* block). The process followed to compute each *Row* tile is similar to the one used for an *Inner* tile. The main difference is that the computation of a *Row* tile requires tiles of $N \times 1$ elements of the corresponding *Diagonal* block and tiles of $1 \times N$ elements that have been previously processed in the current *Row* block. Each *Row* tile to be processed is loaded into the registers, which are used as accumulators for the partial results of the computation of each pair of *Diagonal* and *Row* tiles. These registers are stored back into memory when there are no more pairs to process.

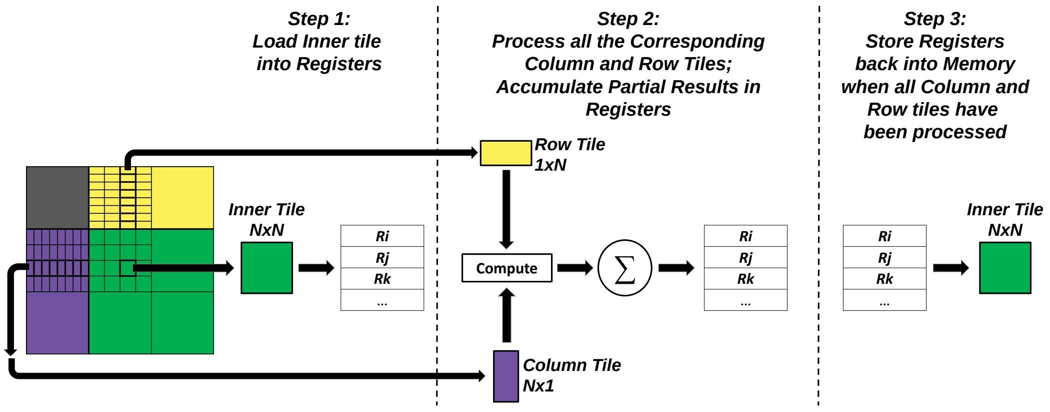

Step 1:
Load Inner tile into Registers
Step 2:
Process all the Corresponding Column and Row Tiles; Accumulate Partial Results in Registers
Step 3:
Store Registers back into Memory when all Column and Row tiles have been processed
Row Tile 1xN
Inner Tile NxN
Ri
Rj
Rk
...
Compute
Σ
Column Tile Nx1
Inner Tile NxN

Figure 9.3: Optimum Energy-Aware Tiling for an Inner Block

Column Blocks: To compute a *Column* block, this is also divided into tiles of $N \times N$ elements. Each *Column* tile is computed using tiles of $1 \times N$ elements of the corresponding *Diagonal* block and tiles of $N \times 1$ elements that have been previously processed in the current *Column* block. In order to minimize the Dynamic Energy of loads and stores, each *Column* tile to be processed is firstly loaded into registers. Then, these registers are used as accumulators for the partial results computed for each pair of *Diagonal* and *Column* tiles. When there are no more pairs to process, the content of the registers used as accumulators is stored back into memory.

Diagonal Block: A *Diagonal* block can be seen as another matrix A' that needs to be LU-factorized. Consequently, the *Diagonal* block can be divided into tiles of $N \times N$ elements, labeled as *Diagonal*, *Column*, *Row*, and *Inner* tiles. They can be latter processed following the same rules used in the computation of the matrix A and the same traversing of tiles previously described for the *Column*, *Row*, and *Inner* blocks.

9.2.2 Minimizing Static Energy using Pipelining

The design of specific tilings for energy consumption already targets Dynamic Energy. However, the long latency of memory operations with respect to the latency of arithmetic operations can produce stalls, where each processor is waiting for data required for computation. This scenario becomes worse if hundreds of threads, starvation of shared resources and bandwidth limitations are considered [106]. This behavior can increase the Static Energy consumption due to increasing latency produced by contention.

In order to successfully minimize the impact of Static Energy, further optimizations were done to the implementation of the tilings described in Section 9.2.1. Each *for* loop was software-pipelined and unrolled twice, using different registers for each unrolled iteration if possible and sharing registers when necessary.

Following Figure 9.3, a *for* loop iteration computes a partial result for an *Inner* tile of $N \times N$ elements using a *Row* tile of $1 \times N$ elements and a *Column* tile of $N \times 1$ elements; the next iteration uses a different *Row* tile and a different *Column* tile to compute the next cumulative partial result of the same *Inner* tile. Consequently, a *for* loop that has been unrolled twice requires at least $N^2 + 4N$ registers. Since additional registers are required in the loop iterations for loop control and pointers (a pointer for the *Row* tiles and a pointer for the *Column* tiles; no pointer is necessary inside the loop for the *Inner* tile since this tile is the same for all the iterations), some registers were shared between iterations in order to decrease the requirement in the number of registers.

To diminish the impact of this register-sharing, the instructions of the loop were later properly interleaved to ensure that memory-related instructions (i.e. *loads* and *stores*) were already completed at the moment the registers involved in such operations were used in a arithmetic instruction, decreasing the execution time to directly impact the static energy.

9.2.3 Dynamic Task Scheduling for Energy Reduction

At this point, the fine-grain tasks have been optimized in order to decrease energy consumption while using the performance-oriented Static scheduling proposed by Venetis and Gao [47]. Even though the *Dynamic Repartition* technique is meant to perform an optimized distribution of work among processing elements, it does not take into account the undesirable delays produced by the competition of access to shared resources (e.g. competition for memory bandwidth on shared memory). This results in variations in the completion time between tasks of the same size. As a consequence, the energy consumption per task will not be uniform. This variation will be most significant with fine-grained tasks, such as the tiles described for LU factorization. In the end, a static distribution of limited work, even for cases of very regular tasks, will result in scenarios where the unbalanced distribution of work will have a negative impact on the Static Energy consumption. In addition, division of blocks into tiles produces a set of smaller border tiles per block that are suboptimal in terms of energy consumption.

In order to overcome these problems, a Dynamic Scheduling of tasks was used in the LU factorization, using the tile as a unit of work assigned to each processing element, instead of a block. First, the matrix is divided into tiles of $N \times N$ elements, which are processed following the LU factorization algorithm, that is, first the *Diagonal* tile, then all the *Column* and *Row* tiles, and finally all the *Inner* tiles. However, in this case, the assignment of tiles is not made statically (as in Venetis and Gao [47]) but in a first-come first-served basis: A tile is assigned to a processing element as soon as the processing element becomes available (i.e. as soon as the processing element finishes the computation of the previous assigned tile) and the tile dependencies are satisfied.

Dividing the matrix in tiles of $N \times N$ leads to a significant amount of tasks, which could increase the overhead of the implementation and reduce the data reuse. Nevertheless, the Dynamic Scheduling of tasks has ultimately a positive impact in the Static Energy consumption of the application since it ensures a better workload balance by keeping the number of idle processors low. This is ultimately translated in a reduction of the execution time of the application. In addition to this, the overhead associated with Dynamic Scheduling is diminished thanks to the support of in-memory atomic operations in the C64 [82]. Using an in-memory atomic operation such as *L_ADD*, a Dynamic Scheduler can be easily implemented with a counter for the number of tasks. Every time a processor is available, it asks for a new task and increments the counter. Since this increment is performed atomically in memory, additional round trips are avoided increasing the throughput of this counter.

To increase the data reuse with Dynamic Scheduling and to avoid that a *Diagonal* tile of $N \times N$ becomes a bottleneck for the whole algorithm (since no tile can be processed until that tile is computed), the size of the *Diagonal* tile can be increased to $bN \times bN$ with $b \in \mathbb{N}$ and $b \geq 2$, while the sizes of other tiles remain as $N \times N$. This reduces by b the number of steps required to compute the LU factorization. The use of a tile as a unit of work for the Dynamic Scheduling, instead of a block, decreases significantly the number of suboptimal border tiles, decreasing the *Dynamic Energy* too.

9.3 Experimental Evaluation

This section describes the experimental evaluation of the proposed optimizations targeting energy consumption and power efficiency described in Section 9.2. We have used the IBM C64 platform described in Chapter 4 and the energy estimations using the model described in Section 8.1. All benchmarks were written in C with hand-tuned assembly for the register tiling. Benchmarks were compiled with ET International's C64 C compiler with compilation flags `-O3`.

We implemented several versions of LU factorization using on-chip shared memory. The power-aware tiling proposed in Section 9.2.1 uses $N = 6$ given the 64 registers per Thread Unit (TU) available in Cyclops-64. Also, for the Dynamic Task Scheduling described in Section 9.2.3, we used $b = 2$ so the *Diagonal* tile is 12×12. The Static Energy coefficient e_0 was computed using measurements on a real chip and the number of TUs used, having in mind that 4 additional TUs are reserved: 1 for executing the runtime system and other 3 for managing the communication with other chips using a 3D mesh.

Our first set of experiments uses a matrix of 840×840, the maximum size that fit in on-chip memory. We study the scalability of Dynamic Energy (Figure 9.4) and Total

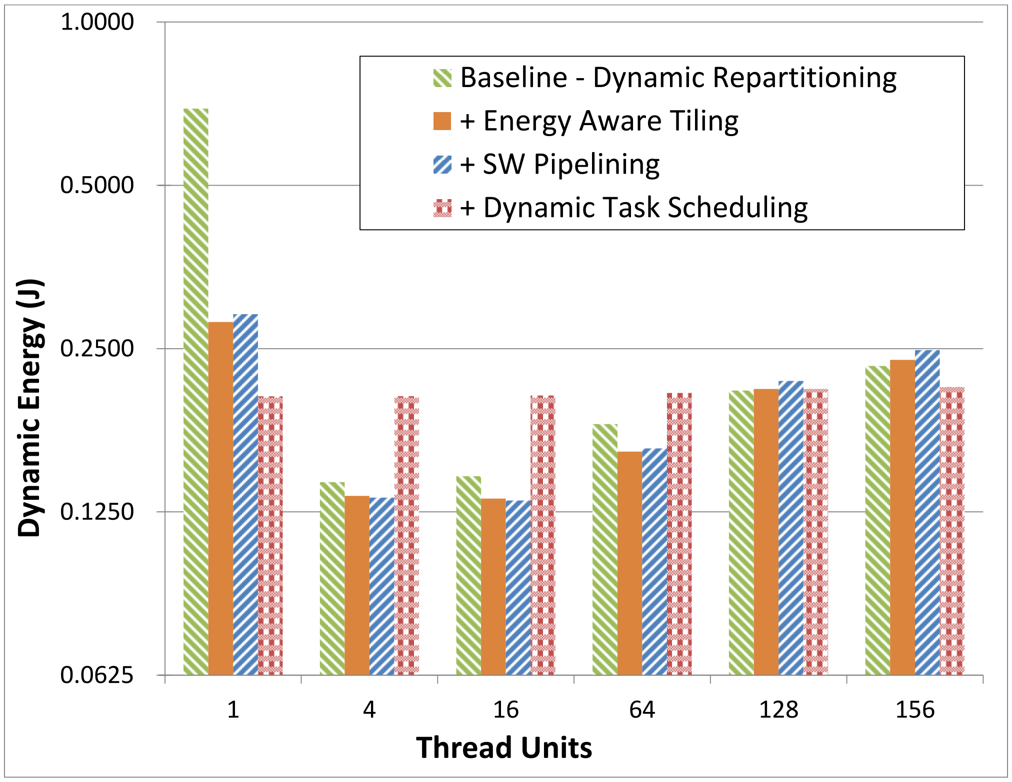


Figure 9.4: Dynamic Energy vs. Thread Units for a matrix of 840 × 840

Energy (Figure 9.5) using different number of TUs. As expected, our Energy Aware tiling decreases the Total Energy with respect to the baseline version that uses Dynamic Repartitioning. This is also true for the Dynamic Energy up to 128 TUs. The software pipelining do not significantly impact the Dynamic Energy because the instructions executed are practically the same but this technique decreases Total Energy because the total execution time and the Static Energy decreases. In addition, we noticed that the Dynamic Energy consumption of our Dynamic Task Scheduling does not vary with the number of TUs. The reason is that the size of the basic unit of work, the tile, is function of architectural parameters such as the number of registers but it is not function of the number of TUs like the blocks used in Dynamic Repartitioning. Our approach using Dynamic Scheduling seems useful for decreasing dynamic energy and

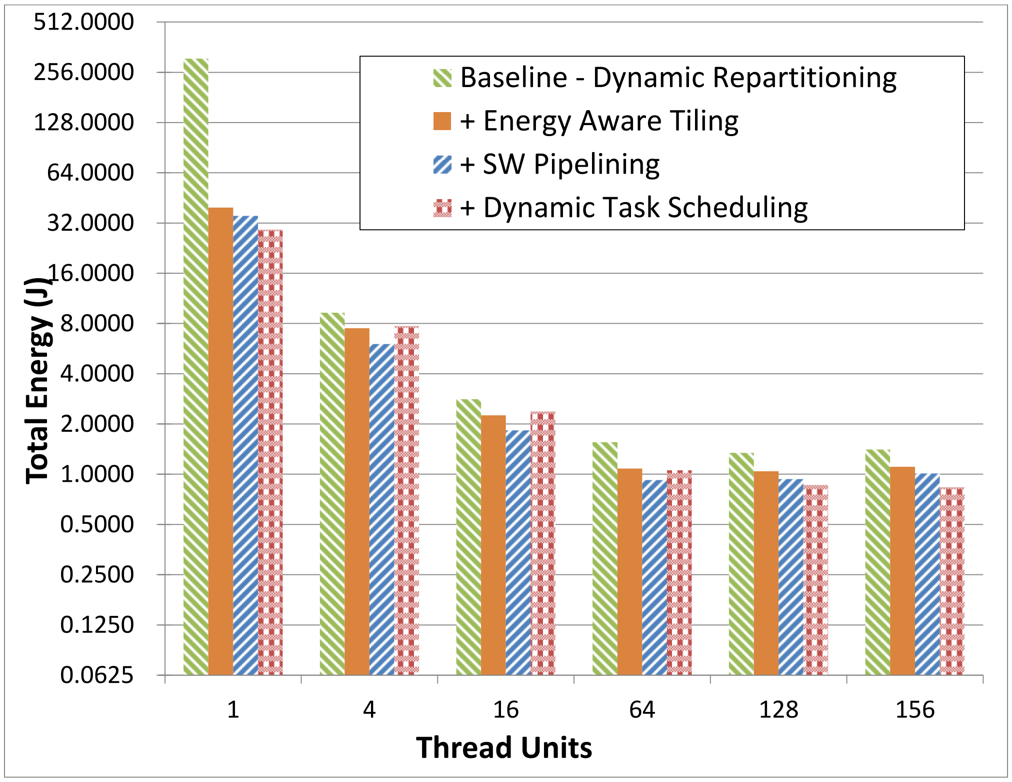


Figure 9.5: Total Energy vs. Thread Units for a matrix of 840 × 840

total energy when the number of TUs surpasses 128. In addition, we noticed that total energy and dynamic energy of the baseline implementation using 1 TU are particularly high, compared with higher number of threads. The reason is that the *Diagonal* register tiling used in the *Diagonal* block calculation is highly inefficient compared with the other tilings; a serial execution computes an LU Factorization as a single *Diagonal* block and exposing this fact.

We also study the impact of the optimizations proposed in terms of Power Efficiency (the ratio between performance and power consumption) in order to examine the trade offs between performance and power consumption. Figure 9.6 shows the scalability of the Power Efficiency with respect to the matrix size using the maximum number of TUs available, while Figure 9.7 shows the scalability of the Power Efficiency

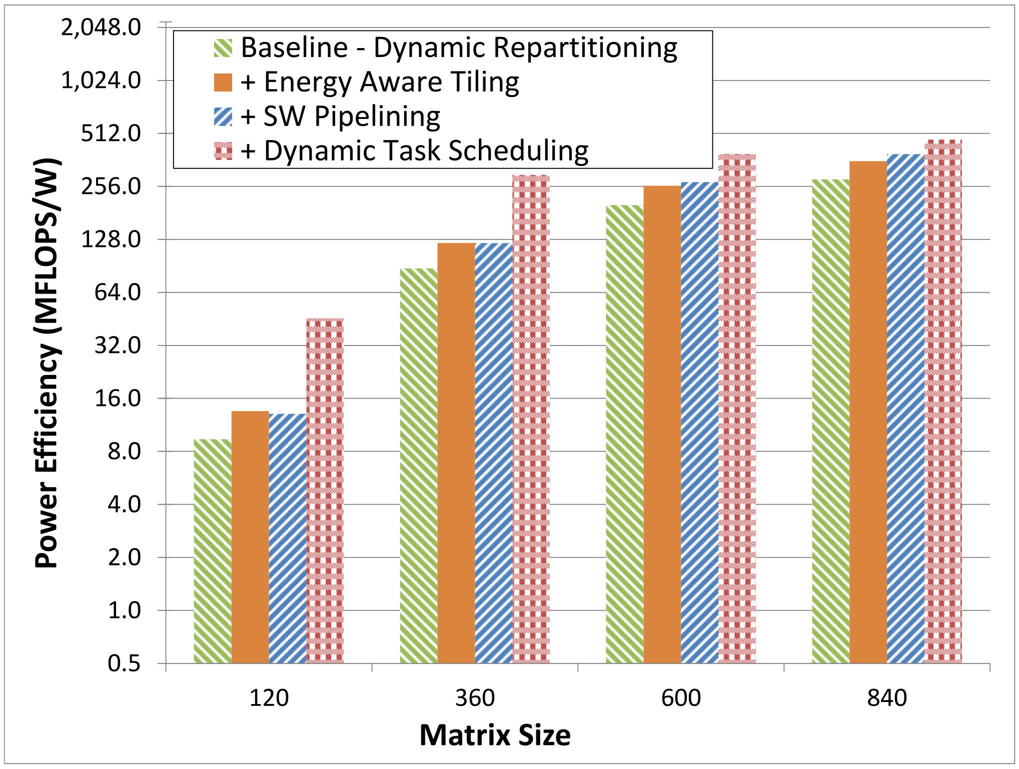


Figure 9.6: Power Eff. vs. Matrix Size for $TU = 156$

with respect to the number of TUs for the biggest matrix that fits on SRAM.

For different matrix sizes on Figure 9.6, all the proposed optimizations increase the power efficiency. The increase in power efficiency for the LU factorization varies between 1.68X and 4.87X with respect to a highly optimized version that targets performance (Our baseline that uses Dynamic Repartitioning). The major returns of the techniques proposed are reached with small matrices. The optimization with the higher impact is the Dynamic Task Scheduling: between 1.2X and 3.5X to the power efficiency.

A careful comparison of the behavior between Power efficiency (Figure 9.7) and Performance (Figure 9.8) shows similarities when few threads are used. For the baseline implementation, as well as for the Energy-aware tiling and the Software Pipelining

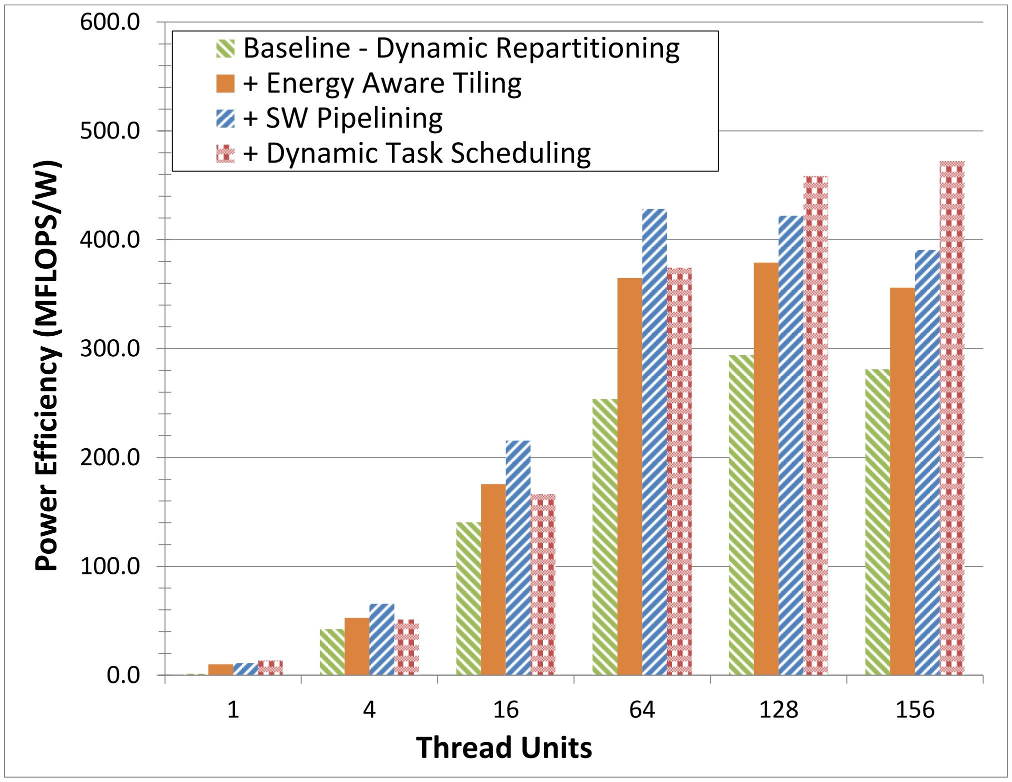


Figure 9.7: Power Eff. vs. TUs for Matrix Size 840 × 840

optimizations, the power efficiency drops after 128 TUs. This is related to the fact that even though the execution time and Static Energy decreases for an increasing number of TUs in all three implementations, the Dynamic Energy increases because these optimizations schedule tasks based on blocks. In contrast, the Power Efficiency of the Dynamic Task Scheduling optimization increases properly with the number of TUs because this type of scheduling does not only scales in terms of performance and Static Energy but also because it keeps the Dynamic Energy constant with the number of TUs.

For the C64 architecture there is a big correlation between the performance and the energy efficiency using few TUs given the high contribution of the static energy to the total energy budget. However, this scenario changes when more TUs are used.

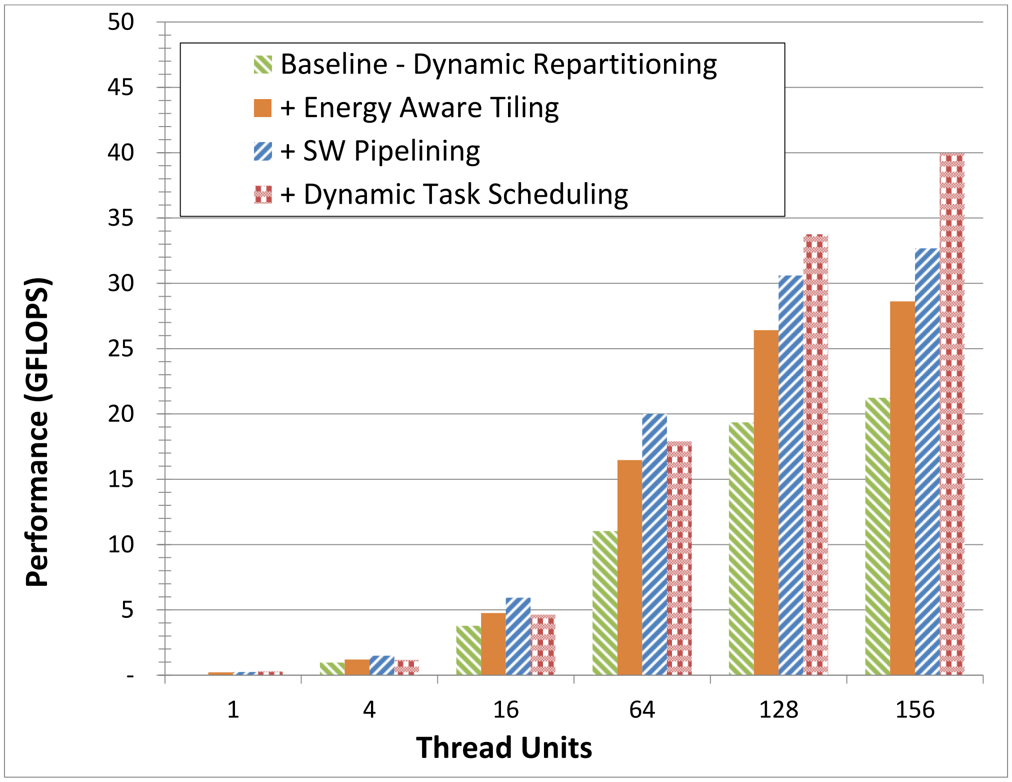


Figure 9.8: Performance vs. TUs for Matrix Size 840 × 840

While all the techniques proposed improve the performance (as seen in Figure 9.8), the power efficiency decreases after 64 TUs or 128 TUS for the Static scheduling techniques (as seen in Figure 9.7). On the other hand, the Dynamic Task scheduling scales in Performance and Power Efficiency.

Chapter 10

TRADEOFFS BETWEEN PERFORMANCE AND ENERGY OPTIMIZATIONS FOR MANY-CORE ARCHITECTURES

While Chapters 5, 6 and 7 focused on Performance optimizations and modeling and Chapters 8 and 9 focused on Energy Consumption, we have seen that both optimizations are related. However, optimizing for performance is not necessarily the same as optimizing for energy consumption. This Chapter studies some scenarios that reflect the trade offs between performance and energy optimizations for many-core architectures.

10.1 Optimizing for Energy is More Difficult than Optimizing for Performance

There is a clear relation between Performance and Energy optimizations through time as shown in equation 8.1. This equation can be simplified as:

$$E_T = E_s(t) + E_d \tag{10.1}$$

Equation 10.1 shows that the total energy consumption E_T of an application has two major components. The Static Energy E_s is related to leakage currents and, in general, it is proportional to execution time t. The Dynamic Energy E_d is related to the type of tasks performed (e.g. instructions executed).

As we can see, any performance optimization is indirectly modifying the total energy consumption through a decrease of Static Energy [107]. However, Improvements in total energy consumption for optimizations targeting performance and without impact in the Dynamic Energy will be lower than improvements in Performance. For example, assuming the energy consumption of an application can be split half for static energy and half for dynamic energy, a potential performance optimization that decreases the execution time by 50% (performance speed up of $2X$), will decrease the energy consumption just by 25% (asumming not significant change in the dynamic energy consumption).

Many performance optimizations fall in this scenario, where the instructions, computations or tasks are just reordered to provide a better utilization of resources, while the amount of total work remains the same. Some examples are Instruction Scheduling, Loop Unrolling, Software Pipelining and Percolation.

All these performance optimizations reflect the fact that Latency can be hidden but Energy cannot. Reordering work for better utilization will not decrease the Energy associate to the work performed, just the energy associate to the time spent.

In some other cases, performance optimizations can hurt the Dynamic Energy consumption, diminishing the returns in energy savings. An example is the Dynamic Repartitioning presented by Venetis and Gao in [47]; as shown in the motivating example in Chapter 9 and detailed in Figure 9.2. It can be seen that this optimization targets a better load balancing and parallelism for better performance by keeping the amount of computational work equal (e.g. Floating Point Operations) but when the number of thread units is increased, the number of memory operations required also increases, affecting the Dynamic Energy consumption significantly. It increases $\tilde{6}5\%$ going from 4 TU to 156 TU. This negative effect reflects more or less in the total energy consumption based on the proportion between Static and Dynamic Energy. While the amount of parallelism increases on new systems and applications performance get close to theoretical peak performance, the dynamic energy becomes significant with respect to the static energy.

The importance of optimizations that also target Dynamic Energy such as Power Aware Tiling was explained in Chapters 8 and 9. This type of tiling keeps the amount of computational work similar (and the dynamic energy associated to this work) while decreasing the energy consumption of memory operations (commonly the most power hungry operations), having also a positive impact on Static Energy and allowing further optimizations targeting Performance.

10.2 Trade offs between Performance and Energy Optimizations

While Energy efficiency and power consumption became more and more important in all the spectrum of computing ranging from mobile and embedded systems to future exascale computing, performance drops are not an option, current design constraints are looking for maximum performance and minimum energy consumption. Unfortunately this two requirements are not necessarily pointing in the same direction but also they are not orthogonal.

For example, taking the Matrix Multiplication of square matrices under a three level memory hierarchy (Registers, on-chip SRAM and off-chip DRAM), designing a tiling in registers and SRAM that is optimum for performance requires: First, minimize the number of memory operations in SRAM and DRAM. And second, hide the latency of this optimum number of memory operations using the strategies described in Chapters 5 and 6.

The solution for the tiling in registers has been described in detail in section 5.2.2 and it just depends of a parameter L where $2L + L^2 \leq R$ and R is the number of registers in each thread unit.

Assuming the matrix multiplication $A \times B = C$ of matrix sizes $m \times m$ and tiling in on-chip memory SRAM of sizes $Z \times Y$, $Y \times Z$ and $Z \times Z$ for matrices A, B and C. The proposed optimization problem is:

$$\begin{aligned} \min_{Y,Z} \quad & LD_{DRAM}(Y,Z,L,m) + ST_{DRAM}(Y,Z,L,m) \\ & + LD_{SRAM}(Y,Z,L,m) + ST_{SRAM}(Y,Z,L,m) \\ & s.t. \quad SRAM(Y,Z) \leq SRAM_{\max} \end{aligned} \tag{10.2}$$

An analytical calculation of the number of loads and stores per memory level will simplify equation 10.2 as follows:

$$\min_{Y,Z} \quad \left(\frac{4}{Z}+\frac{1}{LY}+\frac{2}{L}\right)m^3+5m^2 \quad s.t. \quad 2YZ+Z^2 \leq SRAM_{\max} \tag{10.3}$$

Because L and m are known in equation 10.3, the final optimization problem is:

$$\min_{Y,Z} \quad \frac{1}{Z}+\frac{1}{4LY} \quad s.t. \quad 2YZ+Z^2 \leq SRAM_{\max} \tag{10.4}$$

On the other side, designing an energy aware tiling for the same problem, minimizing the energy consumption of most hungry instructions using the techniques and models described in Chapters 8 and 9, can be formulated by this optimization problem:

$$\begin{aligned}\min_{Y,Z} \quad & e_1 LD_{DRAM}\left(Y,Z,L,m\right)+e_2 ST_{DRAM}\left(Y,Z,L,m\right)\\ & +e_3 LD_{SRAM}\left(Y,Z,L,m\right)+e_4 ST_{SRAM}\left(Y,Z,L,m\right)\\ & \quad s.t. \quad SRAM\left(Y,Z\right) \leq SRAM_{\max}\end{aligned} \tag{10.5}$$

Where e_1, e_2, e_3 and e_4 are the dynamic energy consumption coefficients for Loads from DRAM, Stores to DRAM, Loads from SRAM and Stores to SRAM respectively.

$$\begin{aligned}\min_{Y,Z} \quad & \left(\frac{2(e_1+e_4)}{Z}+\frac{e_3}{LY}+\frac{2e_3}{L}\right)m^3+(e_1+e_2+e_3+2e_4)m^2\\ & s.t. \quad 2YZ+Z^2 \leq SRAM_{\max}\end{aligned} \tag{10.6}$$

Because L and m are known in equation 10.6, the final optimization problem is:

$$\min_{Y,Z} \quad \frac{1}{Z}+\frac{e_3}{2(e_1+e_4)LY} \quad s.t. \quad 2YZ+Z^2 \leq SRAM_{\max} \tag{10.7}$$

A numerical analysis of equation 10.7 can show that using tall and skinny SRAM blocks in matrix A with short and wide SRAM blocks in matrix B (small Y and large Z) will reduce the contribution of dynamic energy of these operations. However, it is important to notice that this rectangular shapes on the blocks may affect the performance due a decrease in the task size and extra overhead of the dynamic scheduler. These type of effects cannot be easily captured on the optimization problems proposed.

Beyond to optimizing for just performance or energy, it is of interest to know, for a particular optimization, the trade offs between performance and energy consumption. A more relaxed requirement than maximum performance or minimum energy consumption, is to explore how much energy savings can be obtained by allowing an small decrease in performance. A qualitative and quantitative analysis is performed in the next section.

10.3 A Case of Study for Performance and Energy Consumption Trade offs

This section will show the trade offs between the performance and the energy consumption of a rectangular block applied to on-chip SRAM for the parallel matrix multiplication algorithm. The platform used is the IBM Cyclops-64 described in sec 4, a real chip was used for the performance and energy measurements while the break down between static and dynamic energy was done using the model explained in Chapter 8.

It is assumed that the blocks are of size $n \times m$, $m \times n$ and $n \times n$ and fit in on-chip SRAM.

Figure 10.1 shows the normalized expected performance (blue line). The x axis starts with very wide and short block in matrix A and tall and skinny block in matrix B (small n and large m). The x axis ends with very tall and skinny block in matrix A and wide and short block in matrix B (large n and small m). In the middle of the x axis, all the blocks of matrices A, B and C are square.

The performance expected has been divided in 4 regions P, Q, S and T. First, region P has very low performance because an small n and large m produce very few and large parallel tasks, the number of tasks is not enough to feed all the processors; the performance increases as n increases. Region Q is characterized because there are enough fine-grain parallel tasks and the maximum performance is reached in this

Figure 10.1: Projection of Performance and Dynamic Energy vs. size of the rectangular tiling for Matrix Multiplication

region. The number of parallel tasks keeps growing in region S while tasks size is decreasing, it starts affecting the performance due to the overhead of the scheduler. The overhead of the dynamic scheduler in region T is too big and it affects significantly the performance.

Due to the expected smooth change in performance in region S and the variations of dynamic energy produced by the changes in the size of the rectangular tiling according to section 10.2, this region shows a potential for a good trade off between performance and energy consumption.

Normalized measurements of Performance and Energy Consumption (Dynamic and Total) for a C64 chip are shown in Figure 10.2. Measurements of performance follow the expected behavior described in Figure 10.1. The maximum performance is reached by an square tiling ($n = 264$ and $m = 264$) but also that is the point of maximum dynamic energy consumption. It is important to notice that as the parameter n

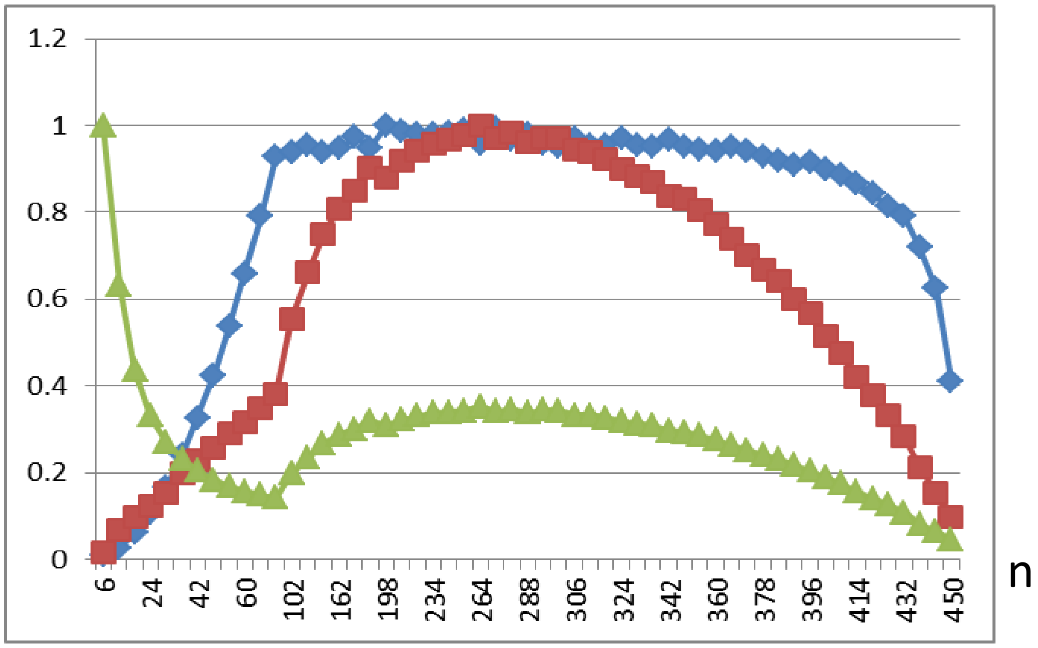


Figure 10.2: Normalized Performance, Dynamic Energy and Total Energy vs. size of the rectangular tiling for Matrix Multiplication

in the rectangular tiling is increased, the performance and dynamic energy decrease but on a different rate. It gives the opportunity to exploit a trade off between performance and energy consumption around the region S described previously.

The normalized total energy is also shown in Figure 10.1. The trade off between static and dynamic energy can be noticed here. A very small n constraints the amount of parallelism and affects the performance, making dominant the static energy component and increasing the energy consumption. In addition, the contribution of dynamic energy to the total energy is significant on the regions where the performance is high (e.g. regions Q and S), making feasible to make significant energy saving sacrificing a minimum in performance. For example, sacrificing a 10% in performance using $n = 396$

and $m = 66$ produces a decrease in total energy of 42%

These type of trade offs are profitable because the energy savings are higher than the drops in performance, a parameterized tiling in conjunction with a profiling in terms of energy and performance can help similar applications to obtain this type of benefits. Also, a self-aware system able to monitor energy consumption and performance can allow a real time control of this type of trade off.

Chapter 11

RELATED WORK AND EXTENSIONS

This section explains some of the most related research that has influenced the development of this thesis and possible extensions to this work. Some of these works are very close and they arisen as collaborations around common interests while others are independent past or parallel efforts in the area.

The optimization of parallel applications is widely spread in the community. Most of the strategies are focused to GPGPUs; while the list of related work is very extensive, some useful general references are [108, 109, 110, 111] and some specific works for particular type of algorithms are [112, 113, 114, 115]. Also an increasing amount of work in optimization on the Intel Xeon Phi Co-processor [116, 117, 118], also known as MIC, faces most of the challenges analyzed in this thesis. The solutions proposed in this work can benefit a wide range of present and future many-core architectures where large amount of parallelism is exposed and there are plenty of shared resources.

This research used heavily the IBM Cyclops-64 many-core architecture and the software tool-chain and simulator designed for this system [67, 119, 120]. Previous to this effort, several applications were targeted for been optimized for performance [45, 46, 47, 121, 122], all of them used static techniques and reached speedup compared to naive versions but they failed to reach performance comparable to the theoretical peak performance of the architecture. All this previous work motivated the use of dynamic techniques, the improving of previous static optimizations and better use of shared resources. The performance optimizations and lightweight fine-grained dynamic scheduling proposed will benefit a broad spectrum of applications beyond the ones studied in this work.

Dynamic Scheduling has been studied for other many-core architectures. The Intel Larabee many-core architecture implements task scheduling entirely with software, allowing for a lightweight Dynamic Scheduling [123]. In conjunction with two other research projects at Intel (The The 80-core Tera-scale research chip program [124, 125] and the Single-Chip Cloud Computer initiative [126, 127]), Larabee has evolved into the Many Integrated Core (MIC). However, this thesis provides evidence indicating the importance and complexity of task scheduling on many-cores, even under homogeneous workloads, and how hardware support can be leveraged to drastically reduce the overhead of more complex scheduling policies under finer grain scenarios. In general, runtime implementations are focused on scheduling loads that are frequently heterogeneous, based on queues, and focused on locality. One of the most popular examples is Cilk [128]. Other approaches include EARTH [28] and Habanero [129]. The contribution of this thesis is more focused on explaining the challenges of scheduling fine-grained homogeneous tasks on many-core architectures and, in the process, showing some limitations of Static Scheduling with regard to scalability and shared resources, and how those limitations are overcome by Dynamic Scheduling.

Several approaches consider the task scheduling problem as a Bin-Packing problem. Different scheduling techniques have been proposed according to the desired optimization function. Good summaries can be found in [130, 131]. Most are not architecture aware and do not consider the overhead and arbitration of shared resources, which is especially important for finer granularity. They have been useful for coarse grained tasks and distributed systems providing boundaries for optimum scheduling strategies under these scenarios.

The Dynamic optimization techniques proposed and the particular implementation and benefits of lightweight fine-grained dynamic scheduling have been used in related research on dataflow inspired runtime systems [72, 37, 100, 39]. Also, other many-core runtime systems have similar roots and have been close to this work such as DARTS and SWARM [132, 133, 36]. There are other multiple parallel efforts that have pointed to Dynamic Schedulers under many-core as the base for scalable fine-grained runtime systems such as Habanero, Concurrent Collections (CnC) and the Open Community Runtime (OCR) [134, 135, 136, 137, 138, 139, 140] and also for specialized libraries for HPC such as the Parallel Linear Algebra Software for Multi-core Architectures (PLASMA) and the Matrix Algebra on GPU and Multi-core Architectures (MAGMA) [141, 142, 143, 144].

Many approaches to performance modeling of multi-threaded programs are based on statistical models of the system and application. Examples of this include the work of Lee [145] and Marin [146]. While they differ in their approach, both works use predictive models to make performance predictions across the entire parameter space. Jacquet et al. also create statistical models of the performance of parallel programs through the use of queuing simulation tools [147]. This work provides detailed modeling of the performance of multiprocessor architectures and key insights into how to reduce contention. However, this thesis is primarily focused on the performance of specific algorithms on parallel architectures. Tarvo and Reiss's work [148] also utilizes simulation to model the performance of a program with a high degree of accuracy. Tarvo and Reiss's work, much like our own, does not require a large degree of training data and can provide accurate results with a comparatively small amount of data. The key difference between our work and Tarvo and Reiss's are the applications being targeted. Our work focuses on performance modeling of the algorithms used in scientific computing whereas their work focuses on applications that make heavy use of I/O and other features of the Linux OS.

Petri nets have long been used to model concurrency with resource dependencies. Govindarajan et al. previously utilized time Petri nets to model multithreaded multiprocessor architectures [149]. Govind et al. would later build upon this work to model a commercial network multithreaded multiprocessor with Petri nets [150]. While similar, the key difference between these works and our own is that Govindarajan models the system whereas we model the algorithm. While Govindarajan's approach has the potential for highly accurate results, each model is inherently architecture specific.

Our approach instead makes the model algorithm specific with the transition durations based upon the underlying architecture. Nguyen and Apon used Petri nets to model the dependencies and latencies of the Linux file system [151] whereas Gilmore et al. used PEPA nets, Petri nets combined with stochastic process algebra, to model secure web services. Furthermore, Kavi and Chen utilized petri nets to model parallel programming structures, such as mutexes and semaphores, to identify race conditions and deadlocks [91].

Fine-grained dataflow inspired systems are also a feasible solution for scalable parallel and modular simulator. A promising approach is the PCA Inspired Computer Architecture Simulation framework (PICASim) [94]

Energy consumption on traditional architectures has been extensively studied [152]. Most of the research has focused on systems with caches [153]. Accurate but highly complex models and techniques for reducing energy consumption has been proposed for uniprocessor architectures. They uses precise information about the hardware and are based on elaborated instruction scheduling [154, 152]. As a consequence the extrapolation to many-core architectures is highly difficult and not scalable with the number of hardware threads. Energy efficiency on multiprocessors has been focused on the hardware design, including hardware features like power saving off-chip memory or dynamic voltage selection [155].

With respect to power efficiency, this thesis made particular emphasis on optimization of dynamic energy; it complements all the efforts in energy optimizations through performance. Several efforts have focused on dynamic voltage and frequency scaling [156, 157, 158], this type of techniques offers a very good trade off between power and performance. The techniques explored here can work on top of these for additional energy savings.

As previously mentioned, the modeling of and optimization for energy consumption is a well researched topic. Many models focus on scheduling and are based on the overall amount of work per unit time [152] or energy [159]. These approaches yield a simplified model that is comparatively easy to use. However, the options and optimizations are limited by the coarse-grained approach.

In contrast, fine-grain approaches [160], like our own, exchange complexity for the potential optimizations that can be applied. Previous works utilized highly accurate, but highly complex, techniques to reduce energy consumption on uniprocessor architectures. This focus on the individual core worked well for uniprocessor architectures but it is unclear how well it will scale for multi-cores. Additionally, these models do not fit with the comparatively recent worldwide pursuit of energy efficiency on multiprocessors.

Large distributed systems face similar challenges for energy efficiency. Due to high cost of communication between nodes, impact in performance and energy of communication avoiding algorithms (including an important subset known as 2.5D algorithms) has become an important research area for distributed systems [161, 162, 163, 164]. The techniques presented in this thesis are complementary to these optimizations because our studies are focused on shared memory architectures and they are targeting optimizations at node level. It is still an open problem the integration of these two levels of parallelism on a single mathematical framework that allows the simultaneous optimization of both.

Hardware mechanism for monitoring and controlling power consumption are still too coarse for many-core architectures and they are more oriented to Thermal Management [165, 166, 167]. In addition, power and energy estimations were done through performance monitoring of events [168, 169]. Recently, finer granularity of performance counters and specific hardware counters for power and energy monitoring with additional power related features are available on commercial CPUs; an example is the Intel RAPL (Running Average Power Limit). This interface provides platform software with the ability to monitor, control, and get notifications on SOC power consumptions. Since its first appearance on Sandy Bridge, more features have being added to extend its usage. In RAPL, platforms are divided into domains for fine grained control. These domains include package, DRAM controller, CPU core (Power Plane 0), graphics uncore (power plane 1), etc. [170, 171, 172].

While most studies have focused on showing potential tradeoffs between Performance and Energy consumption, there are few of them that emphasize the importance of finer granularity in the control variables [157, 173]. In addition, most studies require inconvenient careful profiling or other offline techniques. Accurate real-time managing of tradeoffs between Performance and Energy Consumption is still and open question. This thesis opens the door to this area through a qualitative and quantitative analysis of some tradeoffs and gives valuable elements to continue this path; promising scalable fine-grained strategies for managing these tradeoffs in future parallel technologies are suggested in the work of Landwehr et. al. [174].

This thesis has focused on two major challenges of parallel computing and their tradeoffs: Performance and Energy Consumption. Fault tolerance and Resiliency arise as a third component that need to be modeled, optimized and balanced with performance and energy requirements on present and future many-core architectures. Extensive literature that describe the challenges and feasible solutions in the area of Fault Tolerance has been published recently [175, 176, 177, 178, 179, 180, 181]. Lately there is a particular interest on scalable fine-grained strategies [182, 183].

Chapter 12

SUMMARY AND CONCLUSIONS

The development of new processors with hundreds, or even thousands, of independent processing units through the path to exa-scale systems have brought new challenges to several areas of computer science and computer engineering. Evaluation and reformulation of traditional methodologies used for traditional serial architectures is required given the new scenarios of present and future parallel architectures.

This thesis provided an analysis of new scenarios found in many-core architectures, proposing new methodologies and solutions that leverage the increasing amount of shared resources and the requirement of fine grain tasks to exploit the parallelism available in hardware in order to increase the performance and energy efficiency of these architectures.

An introduction to the general challenges of Computer Architecture and the motivation for the study of parallel computing are presented in Chapter 1. Background particularly related with Dataflow is detailed in Chapter 2.

The Problem Formulation of this thesis is addressed in Chapter 3. The following questions summarize the objectives of these research:

1. Which is the impact of low-level compiler transformations such as tiling and percolation to effectively produce high performance code for many-core architectures?
2. What are the tradeoffs of static and dynamic scheduling techniques to efficiently schedule fine grain tasks with hundreds of threads sharing multiple resources under different conditions in a single chip?
3. Which hardware architecture features can contribute to better scalability and higher performance of scheduling techniques on many-core architectures on a single-chip?

4. How to effectively model high performance programs on many-core architectures under resource coordination conditions?
5. How to efficiently model energy consumption on many-cores managing tradeoffs between scalability and accuracy?
6. Which are feasible methodologies for designing power-aware tiling transformations on many-core architectures?

A representative architecture that reflects the trends in parallel computing is the IBM Cyclops-64 (C64). An overview of this platform is presented in Chapter 4. C64 architecture features have been used through this thesis for the experimental evaluation as a representative of modern many-core architectures.

Chapter 5 studied the impact of low-level compiler transformations such as tiling among others to effectively produce high performance code.

This Chapter presented an static methodology to design algorithms for many-core architectures with a software managed memory hierarchy taking advantage of the flexibility these systems provide. The techniques presented were applied to design and implement a Dense Matrix Multiplication (MM) for the C64 Architecture. Three strategies for increasing performance and show their advantages under this kind of architecture were proposed.

The first strategy is a balanced distribution of work amount threads: the partitioning strategy not only distributes the amount of computation as uniform as possible but also minimizes the maximum block size that belongs to each thread. Experimental results show that the partitioning proposed scales well with respect to the number of threads for different sizes of square matrices and performs better than other similar schemes.

The second strategy alleviates the total cost of memory accesses. We propose an optimal register tiling with an optimal sequence of traversing tiles that minimizes the number of memory operations and maximizes the reuse of data in registers. The implementation of the proposed tiling reached a maximum performance of 30.42 GFLOPS which is almost 10 times larger than the maximum performance reached by the optimum partition alone.

Finally, specific architecture optimizations were implemented. The use of multiple load and multiple store instructions (*ldm/stm*) diminishes the time spent transferring data that are consecutive stored/loaded in memory. It was combined with instruction scheduling, hiding or amortizing the cost of some memory operations and high cost floating point instructions doing other computations in the middle. After these optimizations, the maximum performance of the MM algorithm is 44.12 GFLOPS which corresponds to 55.2% of the peak performance of a C64 chip.

Chapter 6 shows that despite the careful implementation of additional static techniques such as Percolation, it is not enough for reaching high performance in many-cores. This Chapter examined the limitations of static techniques and the suitability of dynamic optimizations for many-core architectures. First, it complements the set of static optimizations studied in Chapter 5 using a Dense Matrix Multiplication (DMM) as example. An exhaustive study and careful implementation of static optimizations are not enough for exploiting the amount of parallelism on many-core architectures. For example, the DMM achieves a performance under %80 of the teoretical peak performance of the machine. Taken this experience as a motivating example, it is shown that for highly regular and embarrassingly parallel applications, Dynamic Scheduling (DS) is preferred over Static Scheduling (SS) in scenarios commonly found in many-core architectures. These scenarios involve the presence of shared resources under different arbitration policies, hundreds of processing units, and a limited amount of work.

It is also explained how these factors degrade the expected performance of SS and how DS behaves better under these conditions. The presence of shared resources, such as a crossbar switch, produces unexpected and stochastic variations on the duration of tasks that SS is unable to manage. In addition, a uniform mapping of work to processors without considering the granularity of the tasks is not necessarily scalable under limited amounts of work.

In addition, it is explained how the advantages of DS are further improved by a low-overhead implementation using mechanisms provided by the architecture, particularly *in-memory* atomic operations, diminishing the overall overhead of DS. As a result, DS can remain efficient for finer task granularities.

These factors allow DS to scale better than SS as the number of processors increase. We demonstrated how Dynamic Scheduling can overcome Static Scheduling with regard to performance. We did this with a synthetic microbenchmark and two applications. Using a Memory Copy microbenchmark, the tradeoffs of SS vs. DS are exposed. Under scenarios with small amount of Hardware threads (e.g. less that 48), SS overcome DS because SS is able to produce a balanced workload with minimum overhead. However, increasing the number Thread Units makes SS schedule highly unbalanced, loosing performance. DS is a feasible solution to manage these complex scenarios and produces balanced workloads under more than a hundred Thread Units with light overhead that allows to double the performance in some cases.

Sparse Vector Matrix Multiplication (SpVMM) was used to show the tradeoffs of SS vs DS under heterogeneity of task controlling the variance of the sparsity distribution for the matrix.

It is important to note that we were able to reach 70.87 GFLOPS for a Dense Matrix Multiplication using a fine-grained DS under C64. This result approaches the 80 GFLOPS of theoretical peak performance, and is far greater than previous published results for this architecture.

Chapter 7 addresses the problem of modeling high performance programs on many-core architectures under resource coordination conditions.

In this chapter, we have demonstrated a technique to model the performance of parallel applications on many-core architectures with resource coordination conditions. Our approach, based on timed Petri nets, results in algorithm specific models that allow us to account for the resource constraints of the system and the needs of the algorithm itself.

With our approach, we were able to model the performance of a dense matrix multiplication algorithm and a finite difference time-domain (FDTD) solution for the propagation of electromagnetic waves given by Maxwell's Equations in 1-Dimension and 2-Dimensions on the IBM Cyclops-64 with a very high degree of accuracy, an average error of 4.4% with respect to the actual performance of the algorithms. Furthermore, because of the nature of our model, we were easily able to investigate how our algorithms would perform on similar architectures.

Finally, we demonstrated how to use our approach to performance modeling to investigate, develop, and tune algorithms for modern many-core architectures, we compared two different tiling strategies for the FDTD kernel and we tested two different algorithms for LU Factorization.

In Chapter 8, we developed an energy consumption model for many-core architectures with software-managed memory hierarchy. We validated the accuracy of this model with the C64 many-core architecture and we showed the model depends of the number and type of instructions executed and the total execution time of the application. An advantage is that this model is scalable with the number of hardware thread units and consider stalls produced by data dependencies or arbitration of shared resources.

We also proposed a general methodology for designing tiling techniques for energy efficient applications. The methodology proposed is based on an optimization problem that produces optimal tiling and sequence of traversing tiles minimizing the energy consumed and parametrized by the sizes of each level in the memory hierarchy. We also showed two different techniques for solving the optimization problem for two different applications: Matrix Multiplication (MM) and Finite Difference Time Domain (FDTD). Our experimental evaluation shows that the techniques proposed reduce the total energy consumption effectively, decreasing the static and dynamic component. The average energy saving for MM is 61.21%, this energy saving is 81.26% for FDTD compared with the naive tiling.

In Chapter 9, we studied and implemented several optimizations to target energy efficiency on many-core architectures with software managed memory hierarchies using LU factorization as a case of study. Our starting point was a highly optimized LU factorization designed for high performance [47]. We analyzed the impact of these optimizations on the Static Energy E_s, Dynamic Energy E_d, Total Energy E_T and Power Efficiency. To facilitate this, we used the scalable energy consumption model described in Chapter 8. We designed and applied further optimizations strategies at the instruction-level and task-level to directly target the reduction of Static and Dynamic Energy and indirectly increase the Power Efficiency. We designed and implemented an energy aware tiling to decrease the Dynamic Energy. The tiling proposed minimizes the energy contribution of the most power hungry instructions.

The proposed optimizations for energy efficiency increase the power efficiency of the LU factorization benchmark by 1.68X to 4.87X, depending on the problem size, with respect to a highly optimized version designed for performance. In addition, we point out examples of optimizations that scale in performance but not necessarily in power efficiency.

In Chapter 10, we showed tradeoffs between performance and energy optimizations for Many-core architectures. We explained how performance and energy consumption are partially related through the Static Energy that depends on execution time. However the Dynamic Energy component becomes significant while the amount of parallelism increases and applications make efficient use of resources.

We also explained how energy optimizations are more challenging because a) Performance optimizations just target directly the Static Energy component, with diminishing benefits for the total energy consumption. b) Some performance optimization can affect negatively the Dynamic Energy component diminishing even more the benefits for total energy. And c) While multiple performance optimizations target a better use of resources by reordering instructions, computations or tasks in order to hide latency, the amount of work performed and the energy associated to them keeps the same. All these reasons, motivate a deeper look at strategies that optimize Dynamic Energy such as the Power Aware Tiling explained in Chapter 8.

Finally, we showed how to exploit tradeoffs between performance and energy using a parametric power aware tiling on a parallel matrix multiplication. We reached

42% energy saving allowing a 10% decrease in performance using a rectangular tiling instead of an square tiling.

BIBLIOGRAPHY

[1] J. Rabaey, A. Chandrakasan, and B. Nikolic, *Digital integrated circuits.* Prentice-Hall, 1996.

[2] J. Farrell and T. Fischer, "Issue logic for a 600-mhz out-of-order execution microprocessor," *Solid-State Circuits, IEEE Journal of*, vol. 33, pp. 707 –712, may 1998.

[3] R. M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," *IBM Journal of Research and Development*, vol. 11, pp. 25 –33, jan. 1967.

[4] S. Palacharla, N. P. Jouppi, and J. E. Smith, "Complexity-effective superscalar processors," in *Proceedings of the 24th annual international symposium on Computer architecture*, ISCA '97, (New York, NY, USA), pp. 206–218, ACM, 1997.

[5] G. Moore, "Cramming more components onto integrated circuits," *Proceedings of the IEEE*, vol. 86, pp. 82 –85, jan 1998.

[6] J. B. Dennis, "First version of a data flow procedure language," in *Programming Symposium, Proceedings Colloque sur la Programmation*, (London, UK), pp. 362–376, Springer-Verlag, 1974.

[7] J. Rumbaugh, "A data flow multiprocessor," *IEEE Trans. Comput.*, vol. 26, pp. 138–146, February 1977.

[8] A. Arvind and K. P. Gostelow, "The u-interpreter," *Computer*, vol. 15, pp. 42–49, February 1982.

[9] A. H. Veen, "Dataflow machine architecture," *ACM Comput. Surv.*, vol. 18, pp. 365–396, December 1986.

[10] W. A. Najjar, E. A. Lee, and G. R. Gao, "Advances in the dataflow computational model," *Parallel Computing*, vol. 25, pp. 1907–1929, Dec. 1999.

[11] J. B. Dennis and D. Misunas, "A preliminary architecture for a basic data flow processor," *in Proceedings of the 2nd Annual Symposium on Computer Architecture (ISCA1974)*, pp. 126–132, December, 1974.

[12] Arvind and K. P. Gostelow, "The U-interpreter," *IEEE Computer*, vol. 15, no. 2, pp. 42–49, 1982.

[13] J. Dennis, G. Gao, and K. Todd, "Modeling the weather with a data flow supercomputer," *IEEE Transactions in Computers*, 1984.

[14] G. R. Gao, "Maximum pipelining linear recurrence on static data flow computers," *International Journal of Parallel Programming*, vol. 15, pp. 127–149, 1986. 10.1007/BF01414442.

[15] K. B. Theobald, "Adding fault-tolerance to a static data flow supercomputer," *Tech. Rep. MIT/LCS/TR-499, MIT Lab. for Comp. Sci.*, Apr. 1991.

[16] D. Comte, N. Hifdi, and J.-C. Syre, "The data driven LAU multiprocessor system: Results and perspectives," in *IFIP Congress*, pp. 175–180, 1980.

[17] J.-L. Gaudiot, R. W. Vedder, G. K. Tucker, D. Finn, and M. L. Campbell, "A distributed VLSI architecture for efficient signal and data processing," *IEEE Trans. Computers*, vol. 34, no. 12, pp. 1072–1087, 1985.

[18] J. E. Hicks, D. Chiou, B. S. Ang, and Arvind, "Performance studies of Id on the monsoon dataflow system," *J. Parallel Distrib. Comput.*, vol. 18, no. 3, pp. 273–300, 1993.

[19] Arvind and R. S. Nikhil, "Executing a program on the MIT tagged-token dataflow architecture," *IEEE Trans. Computers*, vol. 39, no. 3, pp. 300–318, 1990.

[20] Arvind, A. T. Dahbura, and A. Caro, "From Monsoon to StarT-Voyager: University-Industry Collaboration," *IEEE Micro*, vol. 20, no. 3, pp. 75–84, 2000.

[21] J. B. Dennis, "A parallel program execution model supporting modular software construction," in *Proceedings of the Conference on Massively Parallel Programming Models*, MPPM '97, (Washington, DC, USA), pp. 50–, IEEE Computer Society, 1997.

[22] J. B. Dennis, "General parallel computation can be performed with a cycle-free heap," in *Parallel Architectures and Compilation Techniques, 1998. Proceedings. 1998 International Conference on*, pp. 96 –103, oct 1998.

[23] G. Gao, H. Hum, and J.-M. Monti, "Towards an efficient hybrid dataflow architecture model," in *PARLE '91 Parallel Architectures and Languages Europe* (E. Aarts, J. van Leeuwen, and M. Rem, eds.), vol. 505 of *Lecture Notes in Computer Science*, pp. 355–371, Springer Berlin / Heidelberg, 1991. 10.1007/BFb0035115.

[24] G. R. Gao, "An efficient hybrid dataflow architecture model," *J. Parallel Distrib. Comput.*, vol. 19, pp. 293–307, Dec. 1993.

[25] H. H.-J. Hum, *The super-actor machine: a hybrid dataflow/Von Neumann architecture*. PhD thesis, Montreal, Que., Canada, Canada, 1992. UMI Order No. GAXNN-74897 (Canadian dissertation).

[26] R. A. Iannucci, "Toward a dataflow/von neumann hybrid architecture," in *Proceedings of the 15th Annual International Symposium on Computer architecture*, ISCA '88, (Los Alamitos, CA, USA), pp. 131–140, IEEE Computer Society Press, 1988.

[27] R. A. Iannucci, G. R. Gao, R. H. Halstead Jr., and B. Smith, *Multithread Computer Architecture: A Summary of the State of the Art*. The Kluwer International Series in Engineering and Computer Science, Springer-Verlag GmbH, 1994.

[28] K. B. Theobald, *EARTH: An Efficient Architecture for Running Threads*. PhD thesis, May 1999.

[29] H. Hum, X. Tang, Y. Zhu, G. Gao, X. Xue, H. Cai, and P. Ouellet, "Compiling C for the EARTH multithreaded architecture," in *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques*, pp. 12–23, Oct 1996.

[30] H. Hum, O. Maquelin, K. Theobald, X. Tian, G. Gao, and L. Hendren, "A study of the EARTH-MANNA multithreaded system," *International Journal of Parallel Programming*, vol. 24, no. 4, pp. 319–348, 1996.

[31] G. Gao, J. Suetterlein, and S. Zuckerman, "Toward an execution model for extreme-scale systems -runnemede and beyond," *CAPSL Technical Memo 104*.

[32] A. Kulkarni, M. Lang, and A. Lumsdaine, "GoDEL: a multidirectional dataflow execution model for large-scale computing," *Data-Flow Execution Models for Extreme Scale Computing (DFM 2011)*, 2011.

[33] L. Hendren, X. Tang, Y. Zhu, S. Ghobrial, G. R. Gao, X. Xue, H. Cai, and P. Ouellet, "Compiling c for the EARTH multithreaded architecture," *International Journal of Parallel Programming*, vol. 25, no. 4, pp. 305–338, 1997.

[34] J. B. Dennis, G. R. Gao, and X. X. Meng, "Experiments with the fresh breeze tree-based memory model," *Computer Science - Research and Development*, vol. 26, pp. 325–337, Apr. 2011.

[35] A. Davis and R. Keller, "Data flow program graphs," *Computer*, vol. 15, pp. 26 – 41, feb 1982.

[36] C. Lauderdale and R. Khan, "Towards a codelet-based runtime for exascale computing: position paper," in *Proceedings of the 2nd International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, pp. 21–26, ACM, 2012.

[37] D. Orozco, E. Garcia, R. Pavel, R. Khan, and G. Gao, "Polytasks: A Compressed Task Representation for HPC Runtimes," in *Proceedings of the 24th International Workshop on Languages and Compilers for Parallel Computing (LCPC*

2011), vol. 7146 of *Lecture Notes in Computer Science*, (Fort Collins, CO, USA), pp. 268–282, Springer-Verlag, Sep. 2011.

[38] D. Orozco, E. Garcia, R. Pavel, R. Khan, and G. Gao, "Polytasks: A Compressed Task Representation for HPC Runtimes," in *Proceedings of the 24th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2011)*, Lecture Notes in Computer Science, (Fort Collins, CO, USA), Springer-Verlag, September 2011.

[39] D. Orozco, E. Garcia, R. Khan, K. Livingston, and G. Gao, "Toward high-throughput algorithms on many-core architectures," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 8, pp. 49:1–21, Jan. 2012.

[40] M. Denneau and H. S. Warren Jr., "64-bit Cyclops: Principles of Operation," tech. rep., IBM Watson Research Center, Yorktown Heights, NY, April 2005.

[41] S. Coleman and K. S. McKinley, "Tile size selection using cache organization and data layout," in *PLDI '95: Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, (New York, NY, USA), pp. 279–290, ACM, 1995.

[42] M. D. Lam, E. E. Rothberg, and M. E. Wolf, "The cache performance and optimizations of blocked algorithms," in *ASPLOS-IV: Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, (New York, NY, USA), pp. 63–74, ACM, 1991.

[43] D. Callahan and A. Porterfield, "Data cache performance of supercomputer applications," in *Supercomputing '90: Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, (Los Alamitos, CA, USA), pp. 564–572, IEEE Computer Society Press, 1990.

[44] M. Kondo, H. Okawara, H. Nakamura, T. Boku, and S. Sakai, "Scima: a novel processor architecture for high performance computing," in *High Performance Computing in the Asia-Pacific Region, 2000. Proceedings. The Fourth International Conference/Exhibition on*, vol. 1, pp. 355–360 vol.1, 2000.

[45] L. Chen, Z. Hu, J. Lin, and G. R. Gao, "Optimizing the Fast Fourier Transform on a Multi-core Architecture," in *IEEE 2007 International Parallel and Distributed Processing Symposium (IPDPS '07)*, pp. 1–8, Mar. 2007.

[46] Z. Hu, J. del Cuvillo, W. Zhu, and G. R. Gao, "Optimization of Dense Matrix Multiplication on IBM Cyclops-64: Challenges and Experiences," in *12th International European Conference on Parallel Processing (Euro-Par 2006)*, (Dresden, Germany), pp. 134–144, Aug. 2006.

[47] I. E. Venetis and G. R. Gao, "Mapping the LU Decomposition on a Many-Core Architecture: Challenges and Solutions," in *Proceedings of the 6th ACM Conference on Computing Frontiers (CF '09)*, (Ischia, Italy), pp. 71–80, May 2009.

[48] V. Strassen, "Gaussian Elimination is not Optimal," *Numerische Mathematik*, vol. 14, no. 3, pp. 354–356, 1969.

[49] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms.* The MIT Press, 2nd ed., 2001.

[50] N. J. Higham, "Exploiting Fast Matrix Multiplication Within the Level 3 BLAS," *ACM Transactions on Mathematical Software*, vol. 16, no. 4, pp. 352–368, 1990.

[51] D. H. Bailey and H. R. P. Gerguson, "A strassen-newton algorithm for high-speed parallelizable matrix inversion," in *Supercomputing '88: Proceedings of the 1988 ACM/IEEE Conference on Supercomputing*, (Orlando, Florida, United States), pp. 419–424, 1988.

[52] D. Coppersmith and S. Winograd, "Matrix Multiplication via Arithmetic Progressions," in *Proceedings of the 19th Annual ACM symposium on Theory of Computing (STOC '87)*, (New York, NY, USA), pp. 1–6, 1987.

[53] L. E. Cannon, *A Cellular Computer to Implement the Kalman Filter Algorithm.* PhD thesis, Montana State University, Bozeman, MT, USA, 1969.

[54] C.-T. Ho, S. L. Johnsson, and A. Edelman, "Matrix Multiplication on Hypercubes Using Full Bandwidth and Constant Storage," in *Proceeding of the 6th Distributed Memory Computing Conference*, pp. 447–451, IEEE Computer Society Press, 1991.

[55] Hyuk-Jae Lee and James P. Robertson and José A. B. Fortes, "Generalized Cannon's algorithm for parallel matrix multiplication," in *Proceedings of the 11th International Conference on Supercomputing (ICS '97)*, (Vienna, Austria), pp. 44–51, ACM, 1997.

[56] D. H. Bailey, K. Lee, and H. D. Simon, "Using Strassen's Algorithm to Accelerate the Solution of Linear Systems," *Journal of Supercomputing*, vol. 4, pp. 357–371, 1991.

[57] C. C. Douglas, M. Heroux, G. Slishman, and R. M. Smith, "GEMMW: A Portable Level 3 Blas Winograd Variant Of Strassen's Matrix-Matrix Multiply Algorithm," 1994.

[58] Sascha Hunold and Thomas Rauber and Gudula Rünger, "Multilevel Hierarchical Matrix Multiplication on Clusters," in *Proceedings of the 18th Annual International Conference on Supercomputing (ICS '04)*, (Malo, France), pp. 136–145, 2004.

[59] J. N. Amaral, G. R. Gao, P. Merkey, T. Sterling, Z. Ruiz, and S. Ryan, "Performance Prediction for the HTMT: A Programming Example," in *Proceedings of the Third PETAFLOP Workshop*, 1999.

[60] D. A. Orozco and G. R. Gao, "Mapping the fdtd application to many-core chip architectures," in *ICPP '09: Proceedings of the 2009 International Conference on Parallel Processing*, (Washington, DC, USA), pp. 309–316, IEEE Computer Society, 2009.

[61] E. Garcia, I. E. Venetis, R. Khan, and G. Gao, "Optimized dense matrix multiplication on a many-core architecture," in *Proceedings of the Sixteenth International Conference on Parallel Computing (Euro-Par 2010)*, (Ischia, Italy), August 2010.

[62] L. Chen and G. R. Gao, "Performance analysis of cooley-tukey fft algorithms for a many-core architecture," in *Proceedings of the 2010 Spring Simulation Multiconference*, SpringSim '10, (San Diego, CA, USA), pp. 81:1–81:8, Society for Computer Simulation International, 2010.

[63] I. Kaj, *Stochastic Modeling in Broadband Communications Systems.* Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2002.

[64] M. Lin and N. McKeown, "The throughput of a buffered crossbar switch," *Communications Letters, IEEE*, vol. 9, pp. 465 – 467, may 2005.

[65] J. Stark, M. D. Brown, and Y. N. Patt, "On pipelining dynamic instruction scheduling logic," *Microarchitecture, IEEE/ACM International Symposium on*, vol. 0, p. 57, 2000.

[66] D. Orozco, E. Garcia, R. Khan, K. Livingston, and G. R. Gao, "High throughput queue algorithms," *CAPSL Technical Memo 103*, January, 2011.

[67] J. del Cuvillo, W. Zhu, Z. Hu, and G. R. Gao, "FAST: A Functionally Accurate Simulation Toolset for the Cyclops-64 Cellular Architecture," in *Workshop on Modeling, Benchmarking, and Simulation (MoBS '05). In conjunction with the 32nd Annual International Symposium on Computer Architecture (ISCA '05)*, pp. 11–20, 2005.

[68] E. Garcia, D. Orozco, R. Khan, I. Venetis, K. Livingston, and G. Gao, "A Dynamic Schema to increase performance in Many-core Architectures through Percolation operations," in *Proceedings of the 2013 IEEE International Conference on High Performance Computing (HiPC 2013)*, (Bangalore, India), IEEE Computer Society, Dec. 2013.

[69] L. Meadows, "Openmp 3.0 — a preview of the upcoming standard," in *Proceedings of the 3rd international conference on High Performance Computing and Communications*, HPCC '07, (Berlin, Heidelberg), pp. 4–4, Springer-Verlag, 2007.

[70] R. L. Graham, "The mpi 2.2 standard and the emerging mpi 3 standard," in *Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, (Berlin, Heidelberg), pp. 2–2, Springer-Verlag, 2009.

[71] D. Orozco, *TIDeFlow: A dataflow-inspired execution model for High Performance Computing Programs.* PhD thesis, 2012.

[72] D. Orozco, E. Garcia, R. Pavel, and G. Gao, "TIDeFlow: The Time Iterated Dependency Flow Execution Model," in *Proceedings of Workshop on Data-Flow Execution Models for Extreme Scale Computing (DFM 2011); 20th International Conference on Parallel Architectures and Compilation Techniques (PACT 2011)*, (Galveston Island, TX, USA), pp. 1–9, IEEE Computer Society, Oct. 2011.

[73] C. Dwork, M. Herlihy, and O. Waarts, "Contention in shared memory algorithms," *J. ACM*, vol. 44, pp. 779–805, Nov. 1997.

[74] P. King and R. Pooley, "Derivation of petri net performance models from uml specifications of communications software," in *Specifications of Communication Software, Proceedings of XV UK Performance Engineering Workshop*, pp. 262–276, Springer, 2000.

[75] C. Anglano, "Predicting parallel applications performance on non-dedicated cluster platforms," in *Proceedings of the 12th international conference on Supercomputing*, ICS '98, (New York, NY, USA), pp. 172–179, ACM, 1998.

[76] C. Ramchandani, "Analysis of asynchronous concurrent systems by timed petri nets," 1974.

[77] X. Chen and T. Aamodt, "A first-order fine-grained multithreaded throughput model," in *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, pp. 329 –340, feb. 2009.

[78] S. Hong and H. Kim, "An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness," in *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, (New York, NY, USA), pp. 152–163, ACM, 2009.

[79] S. Hong and H. Kim, "An integrated gpu power and performance model," *SIGARCH Comput. Archit. News*, vol. 38, pp. 280–289, June 2010.

[80] E. Garcia, D. Orozco, R. Khan, I. Venetis, K. Livingston, and G. R. Gao, "Dynamic Percolation: A case of study on the shortcomings of traditional optimization in Many-core Architectures," in *Proceedings of 2012 ACM International Conference on Computer Frontiers (CF 2012)*, (Cagliari, Italy), ACM, May 2012.

[81] D. Orozco, E. Garcia, and G. Gao, “Locality Optimization of Stencil Applications using Data Dependency Graphs,” in *Proceedings of the 23rd International Workshop on Languages and Compilers for Parallel Computing (LCPC2010)*, vol. 6548 of *Lecture Notes in Computer Science*, (Houston, TX, USA), pp. 77–91, Springer-Verlag, October 2010.

[82] E. Garcia, D. Orozco, R. Pavel, and G. R. Gao, “A discussion in favor of Dynamic Scheduling for regular applications in Many-core Architectures,” in *Proceedings of 2012 Workshop on Multithreaded Architectures and Applications (MTAAP 2012); 26th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2012)*, (Shanghai, China), IEEE, May 2012.

[83] M. Chu, R. Ravindran, and S. Mahlke, “Data access partitioning for fine-grain parallelism on multicore architectures,” in *In Proceedings of the 40th Annual IEEE/ACM Symposium on Microarchitecture*, 2007.

[84] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, “A class of parallel tiled linear algebra algorithms for multicore architectures,” *Parallel Comput.*, vol. 35, pp. 38–53, Jan. 2009.

[85] DARPA, “Ubiquitous high performance computing (UHPC) - DARPA-BAA-10-37,” 2010. http://tinyurl.com/7lkeedl.

[86] R. German, C. Kelling, A. Zimmermann, and G. Hommel, “Timenet: a toolkit for evaluating non-markovian stochastic petri nets,” *Performance Evaluation*, vol. 24, no. 1, pp. 69 – 87, 1995. Performance Modeling Tools.

[87] J. Lilius, “Efficient state space search for time petri nets,” *Electronic Notes in Theoretical Computer Science*, vol. 18, no. 0, pp. 113 – 133, 1998. MFCS’98 Workshop on Concurrency.

[88] G. Chiola and A. Ferscha, “Distributed simulation of timed petri nets: Exploiting the net structure to obtain efficiency,” in *In 14 th International Conference on Application and Theory of Petri Nets*, pp. 14–6, Springer Verlag, 1993.

[89] C. A. Petri, *Kommunikation mit Automaten.* PhD thesis, Universit́at Hamburg, 1962.

[90] T. Murata, “Petri nets: Properties, analysis and applications,” *Proceedings of the IEEE*, vol. 77, pp. 541 –580, apr 1989.

[91] K. M. Kavi, A. Moshtaghi, and D.-J. Chen, “Modeling multithreaded applications using petri nets,” *Int. J. Parallel Program.*, vol. 30, pp. 353–371, Oct. 2002.

[92] E. Garcia, R. Khan, K. Livingston, I. E. Venetis, and G. R. Gao, “Dynamic percolation - mapping dense matrix multiplication on a many-core architecture,” *CAPSL Technical Memo 098*, June, 2010.

[93] L. Gomes and J. P. Barros, "Structuring and composability issues in petri nets modeling," *Industrial Informatics, IEEE Transactions on*, vol. 1, pp. 112 – 123, may 2005.

[94] R. Pavel, E. Garcia, D. Orozco, and G. R. Gao, "Toward a highly parallel framework for discrete-event simulation," *CAPSL Technical Memo 113*, April, 2012.

[95] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, and K. Yelick, "Exascale computing study: Technology challenges in achieving exascale systems," *DARPA Information Processing Techniques Office (IPTO) sponsored study*, 2008.

[96] J. Torrellas, "Architectures for extreme-scale computing," *Computer*, vol. 42, pp. 28 –35, Nov. 2009.

[97] K. Yee, "Numerical solution of inital boundary value problems involving maxwell's equations in isotropic media," *Antennas and Propagation, IEEE Transactions on*, vol. 14, pp. 302–307, May 1966.

[98] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan, "Effective automatic parallelization of stencil computations," *SIGPLAN Not.*, vol. 42, no. 6, pp. 235–244, 2007.

[99] M. Denneau, "Cyclops," in *Encyclopedia of Parallel Computing: SpringerReference (www.springerreference.com)* (D. Padua, ed.), Springer-Verlag Berlin Heidelberg, 2011.

[100] Y. Yan, S. Chatterjee, D. Orozco, E. Garcia, Z. Budimlic, R. Pavel, G. Gao, and V. Sarkar, "Hardware and Software Tradeoffs for Task Synchronization Using Phasers on Manycore Architectures," in *Proceedings of the 17th International European Conference on Parallel Computing (Euro-Par 2011), Part II*, vol. 6853 of *Lecture Notes in Computer Science*, (Bordeaux, France), pp. 112–123, Springer-Verlag, Aug. 2011.

[101] E. Garcia, D. Orozco, and G. Gao, "Energy efficient tiling on a Many-Core Architecture," in *Proceedings of 4th Workshop on Programmability Issues for Heterogeneous Multicores (MULTIPROG-2011); 6th International Conference on High-Performance and Embedded Architectures and Compilers (HiPEAC)*, (Heraklion, Greece), pp. 53–66, January 2011.

[102] O. Y. Chen, "A comparison of pivoting strategies for the direct lu factorization," *Electronic Proceedings of the Eighth Annual International Conference on Technology in Collegiate Mathematics Houston, Texas, November 16-19, 1995*, Nov. 1995.

[103] J. J. Dongarra and D. W. Walker, "Software libraries for linear algebra computations on high performance computers," *SIAM Rev.*, vol. 37, pp. 151–180, June 1995.

[104] J. Dongarra, P. Luszczek, and A. Petitet, "The linpack benchmark: past, present and future," *Concurrency and Computation: Practice and Experience*, pp. 803–820, 2003.

[105] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The splash-2 programs: characterization and methodological considerations," *SIGARCH Comput. Archit. News*, vol. 23, pp. 24–36, May 1995.

[106] E. Garcia, J. Arteaga, R. Pavel, and G. Gao, "Optimizing the LU Factorization for Energy Efficiency on a Many-Core Architecture," in *Proceedings of the 26th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2013)*, Lecture Notes in Computer Science, (Santa Clara, CA, USA), Springer-Verlag, Sep. 2013.

[107] E. Garcia and G. R. Gao, "Strategies for improving Performance and Energy Efficiency on a Many-core," in *Proceedings of 2013 ACM International Conference on Computer Frontiers (CF 2013)*, (Ischia, Italy), pp. 9:1–4, ACM, May 2013.

[108] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu, "Optimization principles and application performance evaluation of a multithreaded gpu using cuda," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pp. 73–82, ACM, 2008.

[109] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "Gpu computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008.

[110] M. Pharr and R. Fernando, *Gpu gems 2: programming techniques for high-performance graphics and general-purpose computation.* Addison-Wesley Professional, 2005.

[111] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover, "Gpu cluster for high performance computing," in *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, p. 47, IEEE Computer Society, 2004.

[112] G. Tan, L. Li, S. Triechle, E. Phillips, Y. Bao, and N. Sun, "Fast implementation of dgemm on fermi gpu," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, p. 35, ACM, 2011.

[113] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S.-Z. Ueng, J. A. Stratton, and W.-m. W. Hwu, "Program optimization space pruning for a multithreaded

gpu," in *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, pp. 195–204, ACM, 2008.

[114] Y. Hung and W. Wang, "Accelerating parallel particle swarm optimization via gpu," *Optimization Methods and Software*, vol. 27, no. 1, pp. 33–51, 2012.

[115] D. Unat, X. Cai, and S. B. Baden, "Mint: realizing cuda performance in 3d stencil methods with annotated c," in *Proceedings of the international conference on Supercomputing*, pp. 214–224, ACM, 2011.

[116] J. Jeffers and J. Reinders, *Intel Xeon Phi coprocessor high performance programming*. Newnes, 2013.

[117] G. Chrysos and S. P. Engineer, "Intel xeon phi coprocessor (codename knights corner)," in *Proceedings of the 24th Hot Chips Symposium, HC*, 2012.

[118] A. Heinecke, K. Vaidyanathan, M. Smelyanskiy, A. Kobotov, R. Dubtsov, G. Henry, A. G. Shet, G. Chrysos, and P. Dubey, "Design and implementation of the linpack benchmark for single and multi-node systems based on intel® xeon phi coprocessor," in *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pp. 126–137, IEEE, 2013.

[119] J. del Cuvillo, W. Zhu, Z. Hu, and G. R. Gao, "Toward a software infrastructure for the cyclops-64 cellular architecture," in *High-Performance Computing in an Advanced Collaborative Environment, 2006. HPCS 2006. 20th International Symposium on*, pp. 9–9, IEEE, 2006.

[120] Y. P. Zhang, T. Jeong, F. Chen, H. Wu, R. Nitzsche, and G. R. Gao, "A study of the on-chip interconnection network for the ibm cyclops64 multi-core architecture," in *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pp. 10–pp, IEEE, 2006.

[121] D. Orozco and G. Gao, "Mapping the fdtd application to many-core chip architectures," in *Parallel Processing, 2009. ICPP'09. International Conference on*, pp. 309–316, IEEE, 2009.

[122] W. Zhu, P. Thulasiraman, R. K. Thulasiram, and G. R. Gao, "Exploring financial applications on many-core-on-a-chip architecture: A first experiment," in *Frontiers of High Performance Computing and Networking–ISPA 2006 Workshops*, pp. 221–230, Springer, 2006.

[123] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan, "Larrabee: a many-core x86 architecture for visual computing," *ACM Trans. Graph.*, vol. 27, pp. 18:1–18:15, August 2008.

[124] T. G. Mattson, R. Van der Wijngaart, and M. Frumkin, "Programming the intel 80-core network-on-a-chip terascale processor," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, (Piscataway, NJ, USA), pp. 38:1–38:11, IEEE Press, 2008.

[125] S. Dighe, S. Vangal, N. Borkar, and S. Borkar, "Lessons learned from the 80-core tera-scale research processor," *Intel technology journal*, vol. 13, no. 4, 2009.

[126] T. G. Mattson, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, and S. Dighe, "The 48-core scc processor: the programmer's view," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, (Washington, DC, USA), pp. 1–11, IEEE Computer Society, 2010.

[127] C. Clauss, S. Lankes, P. Reble, and T. Bemmerl, "Evaluation and improvements of programming models for the intel scc many-core processor," in *High Performance Computing and Simulation (HPCS), 2011 International Conference on*, pp. 525 –532, july 2011.

[128] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: an efficient multithreaded runtime system," *SIGPLAN Not.*, vol. 30, pp. 207–216, August 1995.

[129] Y. Guo, R. Barik, R. Raman, and V. Sarkar, "Work-first and help-first scheduling policies for async-finish task parallelism," in *Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing*, (Washington, DC, USA), pp. 1–12, IEEE Computer Society, 2009.

[130] E. G. Coffman, *Bounds on Performance of Scheduling Algorithms. In Computer and Job Shop Scheduling Theory.* New York: Wiley, 1976.

[131] E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson, *Approximation algorithms for bin packing: a survey*, pp. 46–93. Boston, MA, USA: PWS Publishing Co., 1997.

[132] S. Zuckerman, J. Suetterlein, R. Knauerhase, and G. R. Gao, "Using a codelet program execution model for exascale machines: position paper," in *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, pp. 64–69, ACM, 2011.

[133] J. Suettlerlein, S. Zuckerman, and G. R. Gao, "An implementation of the codelet model," in *Euro-Par 2013 Parallel Processing*, pp. 633–644, Springer, 2013.

[134] K. Knobe, "Ease of use with concurrent collections (cnc)," *Hot Topics in Parallelism*, 2009.

[135] R. Newton, F. Schlimbach, M. Hampton, and K. Knobe, "Capturing and composing parallel patterns with intel cnc," in *Proc. of 2nd USENIX Workshop on Hot Topics in Parallelism (HotPar 2010), Berkley, CA, USA (June 2010)*, 2010.

[136] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar, "Habanero-java: the new adventures of old x10," in *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, pp. 51–61, ACM, 2011.

[137] R. Barik, Z. Budimlic, V. Cavè, S. Chatterjee, Y. Guo, D. Peixotto, R. Raman, J. Shirako, S. Taşırlar, Y. Yan, *et al.*, "The habanero multicore software research project," in *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pp. 735–736, ACM, 2009.

[138] "Open Community Runtime (OCR)." https://01.org/open-community-runtime.

[139] Intel, "Traleika Glacier X-Stack Project." http://goo.gl/L69GfL.

[140] Department of Energy (DOE), "X-stack software." http://www.xstack.org/.

[141] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov, "Numerical linear algebra on emerging architectures: The plasma and magma projects," in *Journal of Physics: Conference Series*, vol. 180, p. 012037, IOP Publishing, 2009.

[142] A. Buttari, J. Dongarra, J. Kurzak, J. Langou, P. Luszczek, and S. Tomov, "The impact of multicore on math software," in *Applied Parallel Computing. State of the Art in Scientific Computing*, pp. 1–10, Springer, 2007.

[143] F. Song, A. YarKhan, and J. Dongarra, "Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems," in *High Performance Computing Networking, Storage and Analysis, Proceedings of the Conference on*, pp. 1–11, IEEE, 2009.

[144] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra, "Dense linear algebra solvers for multicore with gpu accelerators," in *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pp. 1–8, IEEE, 2010.

[145] B. C. Lee, D. M. Brooks, B. R. de Supinski, M. Schulz, K. Singh, and S. A. McKee, "Methods of inference and learning for performance modeling of parallel applications," in *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '07, (New York, NY, USA), pp. 249–258, ACM, 2007.

[146] G. Marin and J. Mellor-Crummey, “Cross-architecture performance predictions for scientific applications using parameterized models,” in *Proceedings of the joint international conference on Measurement and modeling of computer systems*, SIGMETRICS ’04/Performance ’04, (New York, NY, USA), pp. 2–13, ACM, 2004.

[147] A. Jacquet, V. Janot, C. Leung, G. R. Gao, R. Govindarajan, and T. L. Sterling, “An executable analytical performance evaluation approach for early performance prediction,” in *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, IPDPS ’03, (Washington, DC, USA), pp. 268.1–, IEEE Computer Society, 2003.

[148] A. Tarvo and S. P. Reiss, “Using computer simulation to predict the performance of multithreaded programs,” in *Proceedings of the third joint WOSP/SIPEW international conference on Performance Engineering*, ICPE ’12, (New York, NY, USA), pp. 217–228, ACM, 2012.

[149] R. Govindarajan, F. Suciu, and W. Zuberek, “Timed petri net models of multithreaded multiprocessor architectures,” in *Petri Nets and Performance Models, 1997., Proceedings of the Seventh International Workshop on*, pp. 153 –162, jun 1997.

[150] S. Govind and R. Govindarajan, “Performance modeling and architecture exploration of network processors,” in *Quantitative Evaluation of Systems, 2005. Second International Conference on the*, pp. 189 – 198, sept. 2005.

[151] H. Nguyen and A. Apon, “Hierarchical performance measurement and modeling of the linux file system,” *SIGSOFT Softw. Eng. Notes*, vol. 36, pp. 73–84, Sept. 2011.

[152] F. Yao, A. Demers, and S. Shenker, “A scheduling model for reduced cpu energy,” in *Foundations of Computer Science, 1995. Proceedings., 36th Annual Symposium on*, pp. 374 –382, Oct. 1995.

[153] H. Hanson, M. Hrishikesh, V. Agarwal, S. Keckler, and D. Burger, “Static energy reduction techniques for microprocessor caches,” *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 11, pp. 303 – 313, June 2003.

[154] S. Lee, A. Ermedahl, and S. L. Min, “An accurate instruction-level energy consumption model for embedded risc processors,” in *LCTES ’01: Proceedings of the ACM SIGPLAN workshop on Languages, Compilers and Tools for Embedded Systems*, (New York, NY, USA), pp. 1–10, ACM, 2001.

[155] A. Andrei, P. Eles, Z. Peng, M. Schmitz, and B. Hashimi, “Energy optimization of multiprocessor systems on chip by voltage selection,” *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 15, pp. 262 –275, Mar. 2007.

[156] G. Semeraro, G. Magklis, R. Balasubramonian, D. H. Albonesi, S. Dwarkadas, and M. L. Scott, "Energy-efficient processor design using multiple clock domains with dynamic voltage and frequency scaling," in *High-Performance Computer Architecture, 2002. Proceedings. Eighth International Symposium on*, pp. 29–40, IEEE, 2002.

[157] K. Choi, R. Soma, and M. Pedram, "Fine-grained dynamic voltage and frequency scaling for precise energy and performance tradeoff based on the ratio of off-chip access to on-chip computation times," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 24, no. 1, pp. 18–28, 2005.

[158] S. Herbert and D. Marculescu, "Analysis of dynamic voltage/frequency scaling in chip-multiprocessors," in *Low Power Electronics and Design (ISLPED), 2007 ACM/IEEE International Symposium on*, pp. 38–43, IEEE, 2007.

[159] M. Weiser, B. Welch, A. Demers, and S. Shenker, "Scheduling for reduced cpu energy," *Mobile Computing*, pp. 449–471, 1996.

[160] S. Steinke, M. Knauer, L. Wehmeyer, and P. Marwedel, "An accurate and fine grain instruction-level energy model supporting software optimizations," in *Proc. of PATMOS*, Citeseer, 2001.

[161] E. Solomonik and J. Demmel, "Communication-optimal parallel 2.5 d matrix multiplication and lu factorization algorithms," in *Euro-Par 2011 Parallel Processing*, pp. 90–109, Springer, 2011.

[162] J. Demmel, M. Hoemmen, M. Mohiyuddin, and K. Yelick, "Avoiding communication in sparse matrix computations," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pp. 1–12, IEEE, 2008.

[163] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou, "Communication-optimal parallel and sequential QR and LU factorizations," *SIAM Journal on Scientific Computing*, vol. 34, no. 1, pp. A206–A239, 2012.

[164] M. Mohiyuddin, M. Hoemmen, J. Demmel, and K. Yelick, "Minimizing communication in sparse matrix solvers," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, p. 36, ACM, 2009.

[165] D. Processor, "Power and thermal management in the intel® core tm," *Intel® Centrino® Duo Mobile Technology*, vol. 10, no. 2, p. 109, 2006.

[166] V. George, S. Jahagirdar, C. Tong, K. Smits, S. Damaraju, S. Siers, V. Naydenov, T. Khondker, S. Sarkar, and P. Singh, "Penryn: 45-nm next generation intel® core 2 processor," in *Solid-State Circuits Conference, 2007. ASSCC'07. IEEE Asian*, pp. 14–17, IEEE, 2007.

[167] H. Hanson, S. W. Keckler, S. Ghiasi, K. Rajamani, F. Rawson, and J. Rubio, "Thermal response to dvfs: Analysis with an intel pentium m," in *Proceedings of the 2007 international symposium on Low power electronics and design*, pp. 219–224, ACM, 2007.

[168] G. Contreras and M. Martonosi, "Power prediction for intel xscale® processors using performance monitoring unit events," in *Low Power Electronics and Design, 2005. ISLPED'05. Proceedings of the 2005 International Symposium on*, pp. 221–226, IEEE, 2005.

[169] C. Isci and M. Martonosi, "Runtime power monitoring in high-end processors: Methodology and empirical data," in *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, p. 93, IEEE Computer Society, 2003.

[170] V. M. Weaver, D. Terpstra, H. McCraw, M. Johnson, K. Kasichayanula, J. Ralph, J. Nelson, P. Mucci, T. Mohan, and S. Moore, "Papi 5: Measuring power, energy, and the cloud," in *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*, pp. 124–125, IEEE, 2013.

[171] D. Hackenberg, T. Ilsche, R. Schone, D. Molka, M. Schmidt, and W. E. Nagel, "Power measurement techniques on standard compute nodes: A quantitative comparison," in *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*, pp. 194–204, IEEE, 2013.

[172] A. Cabrera, F. Almeida, J. Arteaga, and V. Blanco, "Measuring energy consumption using eml (energy measurement library)," *Computer Science-Research and Development*, pp. 1–9, 2014.

[173] A. Kansal and F. Zhao, "Fine-grained energy profiling for power-aware application design," *ACM SIGMETRICS Performance Evaluation Review*, vol. 36, no. 2, pp. 26–31, 2008.

[174] A. Landwehr, S. Zuckerman, and G. R. Gao, "Toward a self-aware system for exascale architectures.," in *Euro-Par Workshops*, pp. 812–822, 2013.

[175] H. Yong, C. Oh, H. Choo, J. Chung, and D. Lee, "An efficient algorithm-based fault tolerance design using the weighted data-check relationship," *IEEE Transactions on Computers*, vol. 50, no. 4, pp. 371–383, 2001.

[176] Z. Chen, "Extending algorithm-based fault tolerance to tolerate fail-stop failures in high performance distributed environments," pp. 1–8, Mar. 2008.

[177] Z. Chen and J. Dongarra, "Algorithm-based checkpoint-free fault tolerance for parallel matrix computations on volatile resources," in *20th*, Apr. 2006.

[178] R. Ferreira, A. Moreira, and L. Carro, "Matrix control-flow algorithm-based fault tolerance," in *17th Intl. On-Line Testing Symposium (IOLTS)*, pp. 37–42, July 2011.

[179] S. Weis, A. Garbade, J. Wolf, B. Fechner, A. Mendelson, R. Giorgi, and T. Ungerer, "A fault detection and recovery architecture for a teradevice dataflow system," in *First Workshop on Data-Flow Execution Models for Extreme Scale Computing (DFM)*, pp. 38–44, 2011.

[180] T. Lanfang, T. Qingping, and X. Jianjun, "Automatic instruction-level recovery by duplicated instructions and checkpointing," in *The 5th International Conference on Biomedical Engineering and Informatics (BMEI)*, pp. 1304–1307, 2012.

[181] M. Imai and T. Yoneda, "Duplicated execution method for noc-based multiple processor systems with restricted private memories," in *IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pp. 463–471, 2011.

[182] X. Lei and T. QingPing, "Data flow error recovery with checkpointing and instruction-level fault tolerance," in *12th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, pp. 79–85, 2011.

[183] L. Rozo, J. Monsalve, and C. Yang, "Adapting fault resilience granularity to overcome varying failure rates in cps," *NSF Early Career Professionals Workshop on exploring new frontiers in Cyber-Physical Systems*, 2014.

[184] Y. Yan, S. Chatterjee, D. Orozco, E. Garcia, J. Shirako, Z. Budimlic, V. Sarkar, and G. R. Gao, "Synchronization for dynamic task parallelism on manycore architectures," *CAPSL Technical Memo 094*, February, 2010.

[185] E. Garcia, I. E. Venetis, R. Khan, and G. R. Gao, "Optimized dense matrix multiplication on a many-core architecture," *CAPSL Technical Memo 095*, February, 2010.

[186] D. Orozco, E. Garcia, and G. R. Gao, "Locality optimization of stencil applications using data dependency graphs," *CAPSL Technical Memo 101*, October, 2010.

[187] E. Garcia, D. Orozco, and G. R. Gao, "Energy efficient tiling on a many-core architecture," *CAPSL Technical Memo 102*, October, 2010.

[188] D. Orozco, E. Garcia, R. Pavel, R. Khan, and G. R. Gao, "Polytasks: A compressed task representation for hpc runtimes," *CAPSL Technical Memo 105*, June, 2011.

[189] E. Garcia, D. Orozco, R. Pavel, and G. R. Gao, "Toward efficient fine-grained dynamic scheduling on many-core architectures," *CAPSL Technical Memo 111*, February, 2012.

[190] E. Garcia, R. Pavel, D. Orozco, and G. R. Gao, "Performance modeling of fine grain task execution models with resource constraints on many-core architectures," *CAPSL Technical Memo 118*, June, 2012.

[191] E. Garcia, J. Arteaga, R. Pavel, and G. R. Gao, "Optimizing the lu factorization for energy efficiency on a many-core architecture," *CAPSL Technical Memo 124*, July, 2013.

[192] J. Arteaga, E. Garcia, S. Zuckerman, and G. R. Gao, "Locality-driven scheduling of tasks for data-dependent multithreading," *CAPSL Technical Memo 125*, January, 2014.

Appendix A
COPYRIGHT INFORMATION

This thesis contains, in part, results, figures, tables and text written by me and published in scientific journals, conference proceedings and technical memos.

In some cases, the copyright for the figures, tables and text belongs to the publisher of a particular paper. Because those parts have been used in this thesis, I have obtained permission to reproduce parts of it.

This appendix contains the relevant details of the copy permissions obtained.

As a first author, I own the copyright for the paper presented in Multiprog-Hipeac Workshop in 2011 [101] used in this thesis. Also for several CAPSL Technical Memos (TM) 094 [184], 095 [185], 098 [92] 101 [186], 102 [187], 103 [66], 105 [188], 111 [189], 118 [190], 124 [191] and 125 [192].

A.1 Copy of the Licensing Agreements

The following pages contain a copy of the licensing agreements for work published in IEEE and Springer, I obtained the copyright as first author of this publications [61, 82, 68].

SPRINGER LICENSE
TERMS AND CONDITIONS

Jul 08, 2014

This is a License Agreement between Elkin Garcia ("You") and Springer ("Springer") provided by Copyright Clearance Center ("CCC"). The license consists of your order details, the terms and conditions provided by Springer, and the payment terms and conditions.

All payments must be made in full to CCC. For payment instructions, please see information listed at the bottom of this form.

License Number	3424490492413
License date	Jul 08, 2014
Licensed content publisher	Springer
Licensed content publication	Springer eBook
Licensed content title	Optimized Dense Matrix Multiplication on a Many-Core Architecture
Licensed content author	Elkin Garcia
Licensed content date	Jan 1, 2010
Type of Use	Thesis/Dissertation
Portion	Full text
Number of copies	1
Author of this Springer article	Yes and you are the sole author of the new work
Order reference number	None
Title of your thesis / dissertation	TOWARD HIGH PERFORMANCE AND ENERGY EFFICIENCY ON MANY-CORE ARCHITECTURES
Expected completion date	Aug 2014
Estimated size(pages)	150
Total	0.00 USD

Terms and Conditions

Introduction
The publisher for this copyrighted material is Springer Science + Business Media. By