PYDAC: A DISTRIBUTED RUNTIME SYSTEM AND PROGRAMMING MODEL FOR A HETEROGENEOUS MANY-CORE ARCHITECTURE

by

Bin Huang

A dissertation submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in
Electrical Engineering

Charlotte

2014

Approved by:

____
Dr. Ronald R. Sass

____
Dr. James M. Conrad

____
Dr. Bharat Joshi

____
Dr. Jennifer W. Weller

ABSTRACT

BIN HUANG. PyDac: a distributed runtime system and programming model for a heterogeneous many-core architecture.
(Under the direction of DR. RONALD R. SASS)

Heterogeneous many-core architectures that consist of big, fast cores and small, energy-efficient cores are very promising for future high-performance computing (HPC) systems. These architectures offer a good balance between single-threaded performance and multithreaded throughput. Such systems impose challenges on the design of programming model and runtime system. Specifically, these challenges include (a) how to fully utilize the chip's performance, (b) how to manage heterogeneous, unreliable hardware resources, and (c) how to generate and manage a large amount of parallel tasks.

This dissertation proposes and evaluates a Python-based programming framework called *PyDac*. PyDac supports a two-level programming model. At the high level, a programmer creates a very large number of tasks, using the divide-and-conquer strategy. At the low level, tasks are written in imperative programming style. The runtime system seamlessly manages the parallel tasks, system resilience, and inter-task communication with architecture support. PyDac has been implemented on both an field-programmable gate array (FPGA) emulation of an unconventional heterogeneous architecture and a conventional multicore microprocessor. To evaluate the performance, resilience, and programmability of the proposed system, several micro-benchmarks were developed. We found that (a) the PyDac abstracts away task communication and achieves programmability, (b) the micro-benchmarks are scalable on the hardware prototype, but (predictably) serial operation limits some micro-benchmarks, and (c) the degree of protection versus speed could be varied in redundant threading that is transparent to programmers.

ACKNOWLEDGMENTS

I would like to thank my parent, whose deep love fueled my courage to pursue this degree.

TABLE OF CONTENTS

LIST OF TABLES vii

LIST OF FIGURES viii

CHAPTER 1: INTRODUCTION 1

1.1 CMOS Technology 2

1.2 Programming Paradigms 4

1.3 Proposed Approach 5

1.4 Thesis Question 6

CHAPTER 2: BACKGROUND 11

2.1 Basics of CMOS 11

2.2 Architectural Trends 13

2.2.1 Homogeneous Architecture vs. Heterogeneous Architecture 13

2.2.2 Memory Wall 15

2.3 Hardware-Software Codesign at OS Level 16

2.4 Programming Model 18

CHAPTER 3: GREEN-WHITE ARCHITECTURE 24

3.1 Modeling Techniques 24

3.2 Theoretical Model of Green-White Architecture 26

3.3 An Implementation of Green-White Architecture 30

3.3.1 Hardware Simulation 30

3.3.2 Hardware Prototype 31

CHAPTER 4: DESIGN OF PYDAC PROGRAMMING FRAMEWORK 34

4.1 Programming Model 35

4.1.1 Divide-and-Conquer Strategy 36

4.1.2 Two-Level Programming Model 37

4.1.3 Fibonacci Algorithm Coding Example 39

4.2 Design of PyDac Runtime 40

4.3 Concluding Remarks 43

CHAPTER 5: EVALUATION 45

5.1 Performance Evaluation 45

5.1.1 Benchmark Suite 45

5.1.2 Scalability 46

5.1.3 Resilience 47

5.2 Evaluating Programming Productivity by Case Study 55

5.2.1 Strassn's Algorithm 55

5.2.2 Symmetrical Tridiagonal Eigenvalue Decomposition 56

5.2.3 Recursive FFT 58

5.2.4 Analysis 60

CHAPTER 6: CONCLUSION 67

CHAPTER 7: FUTURE WORK 70

REFERENCES 72

LIST OF TABLES

TABLE 3.1: Summary of the hardware prototype. 32

LIST OF FIGURES

FIGURE 1.1: High-level block diagram of conventional runtime system. 8

FIGURE 1.2: High-level block diagram of distributed runtime system. 8

FIGURE 3.1: High-level block diagram of the green-white architecture. 27

FIGURE 3.2: High-level block diagram of a green core. 28

FIGURE 3.3: High-level diagram of green-white prototype on an FPGA device. 32

FIGURE 4.1: PyDac algorithm template 38

FIGURE 4.2: Fibonacci algorithm implemented in PyDac. 40

FIGURE 4.3: Proposed resilient runtime system on *green-white* architecture. 41

FIGURE 4.4: Control flow of user-level thread library 42

FIGURE 5.1: Speedup on *green-white* architecture. 48

FIGURE 5.2: Performance impact of adding second SPM: one green core 49

FIGURE 5.3: Performance impact of adding second SPM: six green cores 50

FIGURE 5.4: Normalized runtime on *green-white* architecture. 53

FIGURE 5.5: Strassen's algorithm 56

FIGURE 5.6: Strassen's algorithm implemented in PyDac 57

FIGURE 5.7: Symmetric Tridiagonal Eigenvalue Algorithm 58

FIGURE 5.8: Symmetric Tridiagonal Eigenvalue PyDac Implementation 59

FIGURE 5.9: A Divide-and-Conquer Algorithm for Radix-2 FFT 60

FIGURE 5.10: Recursive FFT implemented in PyDac 61

CHAPTER 1: INTRODUCTION

The technologies used to implement integrated circuits have profound impacts on computer system design and programming paradigm design. For many decades, the dominant technology—complementary metal oxide semiconductor (CMOS)—has enabled frequency scaling and doubling transistor density. This has kept computer architecture research narrowly focused on making single core faster, year after year, which also reinforced a sequential programming paradigm.

Although CMOS technology is slowing in terms of clock frequency growth rate, it is still expected to double the number transistors per chip every two years for many generations [1]. Moore's Law, a trend observed by Gordon Moore [2], states that the density of transistors on a chip doubles every two years. Moore's Law has fundamentally fueled the advancement of computing technology in the past.

Three notable changes in the fundamental assumption with CMOS technology have occurred. First, the Dennard Scaling [3], which states that the reduction in the transistor feature size is accompanied by improvement in power efficiency, had reached its end in the 2000's. Secondly, the reliability of CMOS has become more difficult to sustain, primarily due to process variation and single-event upset. Thirdly, the energy cost of moving data has become comparably more expensive than computation. Because of these changes, the characteristics of future computing devices are likely to be very different, leading to dramatically different chip architectures. Moreover, the new chip architectures have already required new programming paradigms, or at least a renewed emphasis on parallel programming paradigm. We will take a closer look at the impacts next.

1.1 CMOS Technology

Due to the end of Dennard Scaling, power efficiency has emerged as a first-class design constraint. The chip industry has attacked this problem at different levels, such as voltage scaling and architectural innovation. For example, research has shown that aggressive supply voltage scaling greatly improves the energy efficiency of a single processing unit [4]. For another example, the chip industry introduced multicore microprocessor chips [5, 6] to work around the power efficiency issue. However, the introduction of multicore technology was not completely satisfactory. Esmaeilzadeh *et. al* [7] have predicted transistor under-utilization on future chips due to a stringent power budget. They suggest radical micro-architectural innovations beyond CPU-like or GPU-like multicore designs. Examples of such unconventional chip architecture have been proposed with a heterogeneous mix of complex and simple cores [8, 9, 10]. The heterogeneous many-core architecture promises a good balance between single-threaded performance and multithreaded throughput. More importantly, it utilizes transistors in an energy-efficient manner by dedicating resources to an individual application or a group of applications sharing common compute kernels.

In addition to power, reliability will be more difficult to sustain in the future. First, advantages of low-power techniques (e.g., aggressive supply voltage scaling) are not free. There is a tradeoff between power and reliability; specifically, power efficiency from voltage scaling is achieved at a cost of an increasing soft error rate [11]. Secondly, parameter variation will pose a major challenge for the design of future high performance microprocessors [12]. One serious consideration that must be addressed is the ability of applications of interest to run through a variety of failures. Current high-performance computing (HPC) systems rely on checkpoint/restart (C/R) to recover from faults. As system size continues to grow, the overhead of global C/R will likely become a significant percentage of an application's run time [13, 14]. In order to save the overhead, local fault confinement and recovery mechanisms have been proposed [15]. Not only does C/R suffer from high overhead, but it also deals with a subset of possible faults. Silent data corruption (SDC) is a specific class of fault that the C/R technique could not mitigate [16]. Failing to detect SDC could significantly undermine the fidelity of simulation results, because invalid results may still be reported to end users. Replication techniques can detect and mitigate SDC but have been prohibitive for adoption by HPC systems due to high overhead. As the system-level size of HPC systems continue to grow, replication techniques have re-ignited research interest. Ferreira et al. [14] simulated an HPC system with more than 20,000 sockets and proved that replication is a viable alternative to the traditional C/R approach.

Thirdly, data movement will overtake—if it has not already—the floating-point operation as the major contributor to power usage [17]. The energy cost of data movement will limit the usage of hardware design techniques, such as out-of-order execution. These techniques have been successfully used in high-performance microprocessors in the past. To further save energy cost of data movement, a more aggressive memory management scheme may be needed, e.g., only moving data when it contributes to the solution. One mechanism that has been proposed is to use named memory segments and make the movement of data more explicit to the programmer [18]. These authors [18] and others [19] also advocate the use of a form of scratch-pad memory (i.e., software-controlled memory). Scratch-pad memory does not implement a tag RAM and complex comparator logic that are found in conventional direct or set-associative caches, which saves transistors and power. (Low-power embedded systems have long been leveraging scratch-pad memory [20, 21].) In addition, it reduces cache contention, further minimizing unnecessary data movement. Adopting scratch-pad memory was reported to significantly improve energy efficiency in one case of high-end computing [19].

Assuming there is no miraculous technological breakthrough, these trends suggest that future devices and architectures will be astonishingly different. Most likely, future CMOS chips will (a) consist of a large mix of heterogeneous computing cores, (b) use a radically different memory subsystem, and (c) experience higher rates of faults.

1.2 Programming Paradigms

All architectural changes will have a profound impact on how human programmers interact with future computing machines. For a good example, explicitly parallel programming paradigms have become more mainstream with the industry introduction of multicore microprocessor chips [5, 6]. We will take a closer look at the impacts on programming paradigm design next.

First, parallel programming paradigms have replaced the sequential programming paradigm. Before the multicore era, the sequential programming paradigm was so central to every computing system that every piece of software—from applications to libraries to operating systems—could assume compatibility with future devices. In addition, the frequency scaling granted a "free lunch" that software programmers have enjoyed for many years [22]. Due to the free performance lunch, software programmers added layers of abstractions into their software. This free ride mitigated or hid the performance overhead involved with adding layers of abstractions. More importantly, the resulting portability and programming productivity paid off the performance overhead many times. For example, high productivity programming languages, such as Java and Python, are built on top of extra software layers—virtual machines that provide portability to cross-platform programming. The end of free performance lunch now makes the use of extra layers of abstractions in software less viable. More importantly, while removing abstractions from the software stack may benefit performance, such actions will reduce programming productivity.

Moreover, software has become more fragile than before. Generally, software programmers could assume infallible hardware, which greatly simplifies software development. However, it is becoming more difficult to assume infallible hardware. The overhead of maintaining infallibility will become a significant percentage of an application's run time [13, 14]. Therefore, recent programming framework provides application programming interfaces (APIs) to programmers for expressing resilience concerns explicitly [15]. Such APIs clearly impose more burdens on programmers.

1.3 Proposed Approach

Combining these facts and trends together, one may wonder how to design more effective computer systems and programming paradigms. Consequently, the invention of a novel programming paradigm and an associated runtime system is critical. Complicating this investigation is the fact that the exact nature of these future computing machines is far from clear, which means that human understanding of the hardware and software will have to co-evolve. We believe that the best way to address these issues is to think of them "organically". In other words, we would like to look at this problem from a hardware-software codesign perspective. This dissertation focuses on the software aspect of this evolution.

We chose one of the most well-known parallel design patterns—divide-and-conquer—to begin this research. Design patterns are useful concepts for programmers, strongly encouraging and enabling the reuse of successful designs and proven techniques. Each design pattern describes a problem that repeatedly occurs and then describes the core of the solution [23]. Not only are design patterns established in sequential programming paradigm, there are also design patterns for the development of parallel applications [24, 25].

Specifically, we designed a two-level programming model and a runtime system based on the divide-and-conquer strategy. Divide-and-conquer is a well-known strategy for designing algorithms in the computer science community. Three steps are usually involved in this technique: *divide*, *conquer*, and *combine*. The divide-and-conquer strategy recursively decomposes a problem into smaller sub-problems, which in turn are decomposed into sub-sub-problems, and so on. Generating a very large number of parallel tasks without a great deal of programming effort is possible with this strategy. The two-level programming model is implemented in Python language as a library extension. It hides task communication, load-balancing, and resilience from programmers. A runtime system built on distributed Python virtual machines maps the two-level programming model to heterogeneous many-core architecture. Due to independent and parallel tasks, the runtime system is able to gracefully degrade when the hardware is hit by a fault. In particular, the runtime system monitors hardware for potential soft errors. If necessary, it resets a faulty core and reissues the task. However, this is not always enough. For example, output data may be corrupted, requiring a different technique for the runtime system to detect. In that case, the runtime system runs multiple copies of identical tasks and checks the results of those tasks for output data corruption. If data corruption is detected, the runtime system can either reissue a task or use a voting mechanism to determine the correct result if enough copies are available.

1.4 Thesis Question

To evaluate the proposed programming model and runtime system, we present a novel architecture called *green-white* architecture meant to stand in for some future "unconventional" chip architecture. It features a mix of simple and complex cores and a much flatter memory subsystem. Specifically, our model consists of two types of processor cores. It has many simple, energy efficient—and inherently less reliable—processor cores (called "green cores") and a few more robust, protected processor cores (called "white cores"). The white cores have a conventional memory hierarchy while the green cores treat the main memory as a collection of write-once memory segments. The white cores have a conventional symmetric multiprocessing (SMP) operating system and runtime while the green cores run independently with a "close to metal" runtime. To test the spectrum, we also have an implementation for conventional SMP architecture.

The key thesis question can be phrased as follows: *Is a programming model and a runtime system built upon distributed virtual machines superior to monolithic runtime on an architecture under the cross-cutting constraints of performance, resilience, and productivity?* The comparison is illustrated in Figure 1.1 and Figure 1.2. In Figure 1.1, the conventional runtime system is built entirely upon a monolithic operating system and SMP hardware. Adding layers of abstractions to the middleware in a vertically integrated system has been successful. Figure 1.2 presents the proposed programming model and runtime system built upon a set of decoupled distributed virtual machines. The choice of virtual machine is motivated by the fact that it is the foundation for many high productivity languages. In addition, adopting a virtual machine in the design of a runtime system provides software compatibility across various architectures.

The thesis question is broken down into three subordinate questions as follows.

1. *Does a runtime system built upon a set of decoupled distributed virtual machines deliver good performance?* Performance is measured by time-to-completion for a given task. In the context of this work, we measure performance under two different scenarios: (a) the absolute performance of adding more resources to the system (a.k.a. strong scaling), and (b) the relative performance under faults. Achieving a linear speedup is usually very difficult due to the incurred overhead of resource management. Our reasoning is that if there is no speedup in the first scenario or there is a significant overhead in the second scenario, then the proposed design is not viable. To answer this question, we will run multiple micro-benchmarks to measure performance numbers in both a fault-free environment and a faulty environment.

2. *Does a runtime system built upon a set of decoupled distributed virtual machines sustain transient faults?* A conventional monolithic runtime system itself is a

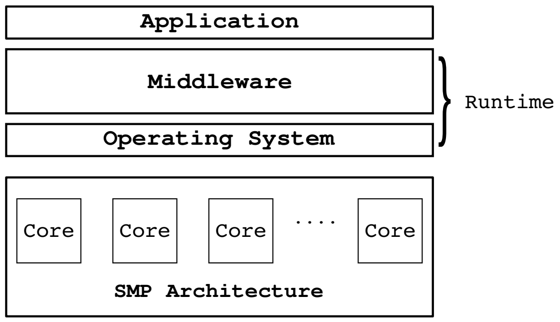

Figure 1.1: High-level block diagram of conventional runtime system. The runtime system is entirely built upon a monolithic operating system and SMP hardware. The runtime system may be “bloated” because of too many abstractions in the software stack.

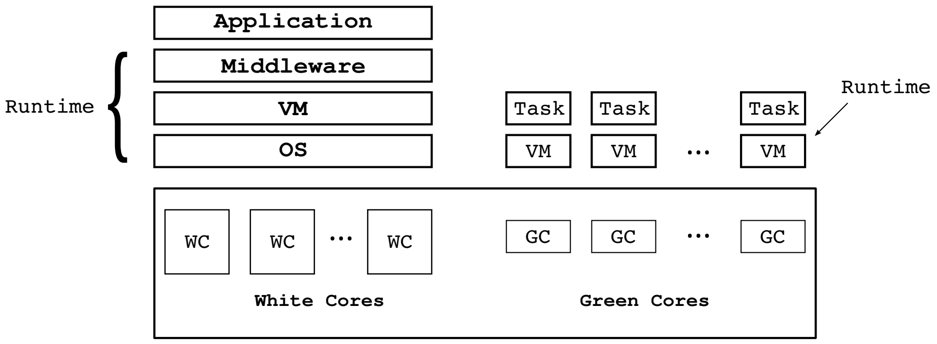

Figure 1.2: High-level block diagram of distributed runtime system. The runtime system is built upon distributed virtual machines. White cores have a conventional SMP operating system and runtime while green cores run independently with a “close to metal” runtime. Virtual machines provide support to high-level programming language and software compatibility across different architecture.

single point of failure. A transient fault could easily cause the monolithic runtime system to crash. Therefore, the monolithic runtime system heavily relies on the C/R technique to recover from a fault. However, as the system size and the degree of parallelism continue to grow, the C/R technique is expected to be inefficient. A runtime system that can sustain transient faults gives the system an opportunity to gracefully degrade while increasing resilience. To answer this question, we will use a fault injection mechanism to emulate transient faults on hardware and observe how the runtime system behaves.

3. *Is the programming paradigm supported by the runtime system productive?* Programming productivity is equally important to system performance. The conventional programming paradigm in high-performance computing features low-level language, such as C. While it is easier for programmers to control the behavior of a machine, it is also counter-productive to express algorithms. High-level languages have been observed to be preferred when quick implementation is required. Low-level languages are involved in rewriting the code only when additional performance is required. Since programming productivity is less measurable than other system indices (e.g., power and performance), case studies will be conducted to indirectly evaluate the effectiveness of the proposed programming model. We will present the source code of multiple micro-benchmarks and rely on the reader's judgment to assert our point.

To demonstrate the feasibility and quantify the behavior of the proposed approach, we implement a hardware prototype of green-white architecture on an field-programmable gate array (FPGA) device. This hardware is used to represent a mix of energy-efficient, simple cores and fast, conventional processor cores. We evaluated the proposed programming model by running multiple micro-benchmarks on the hardware prototype. In addition, we used a fault injection mechanism to emulate transient faults on the hardware and measured the recovery cost.

In addition to the central purpose of this thesis, there are several practical contributions:

- A runtime system based on a set of decoupled distributed virtual machines sustains transient faults on unreliable heterogeneous hardware through local fault recovery.
- Designing a runtime system that combines full-fledged virtual machine and "bare-metal" virtual machine for heterogeneous hardware is feasible.
- Programming a distributed runtime system with a two-level programming paradigm does not undermine programming productivity.

The rest of the dissertation is organized as follows. Chapter 2 presents technology trends that have fundamental impacts on chip architecture and reviews the related works. Chapter 3 details the design of green-white architecture and the hardware prototype. Chapter 4 shows the design of the PyDac programming model and the runtime system. Chapter 5 evaluates the proposed system from the perspective of performance, resilience, and programmability. Chapter 6 summarizes this dissertation and Chapter 7 discusses future works.

CHAPTER 2: BACKGROUND

This chapter first reviews the technological trends of CMOS in recent years. Then, it examines the advent of many-core technology and related design issues. Thirdly, this chapter inspects some hardware-software codesigns from the perspective of the operating system. Lastly, it reviews parallel programming models and compares them to the proposed PyDac programming model.

2.1 Basics of CMOS

CMOS will likely remain the dominant technology for integrated circuit design for another ten years [1]. For the advancement of CMOS technology, Gordon Moore observed that the transistor density doubles every two years [2]. Engineers have been able to reduce the transistor feature size and improve power efficiency for many generations of CMOS technology. Power efficiency was improved by reducing the total capacitance—as seen by the gates' outputs—and lowering the supply voltage, which is also known as Dennard Scaling [3]. This scaling resulted in a constant power density—a key factor in the design of computer architecture. The return of Dennard Scaling began to diminish in the early 2000's. Specifically, supply voltage is close to the threshold voltage and has leveled off [26]. Further reduction of supply voltage is costly; it requires a reduction in the threshold voltage, which has a negative impact on lost power from leakage current [27]. In addition, supply voltage that levels off has a negative impact on maximum operation frequency. The frequency is roughly linear to the supply voltage. Given a constant threshold voltage, the lower the supply voltage is, the longer it takes a signal to propagate. The maximum operation frequency must decrease accordingly to avoid timing violation. Consequently, the end of Dennard Scaling forced the chip industry to shift into a new design paradigm. The old paradigm that focused on improvement of single-threaded, sequential programs' performance was replaced by a new one that focuses on multithreaded, parallel programs.

The reliability of CMOS transistors is more difficult to sustain as the feature size continues to shrink. Parameter variation poses a major challenge to the design of future high performance microprocessors [12]. For example, random dopant fluctuation (RDF), which means that the dopant atoms implanted in the transistor are unevenly distributed, is a significant variation source [28]. Such variation may lead to an increase in intermittent or permanent faults, which may result in unexpected timing violations. The other major cause of the reliability issue is single event upsets (SEUs). An SEU taking place in dynamic random-access memory (DRAM) may be caused by high-energy particles (e.g., cosmic rays), which penetrates the die surface and creates a bit-flip [29]. Logic also becomes more susceptible to SEUs, because of the reduction in critical charge of logic circuits and the decrease in the feature size [30].

Data movement, which is significantly distance-dependent, continues to become more expensive than compute in terms of energy consumption. For instance, for 40-nanometer technology, moving 64 bits of data just from off-chip to on-chip would cost a few nanojoules; or moving 64 bits of data for a few millimeters on-chip would cost hundreds of picojoules. However, performing a double-precision fused multiply add (FMA) would cost only tens of picojoules [31]. An alternative technology, optical interconnect, shows that energy consumption is not dependent on distance. However, optical technology is not ready for production [17], and distance-dependent data movement poses a challenge to energy-efficient computing.

We have discussed four major technology trends in this section. As Moore's Law predicts, CMOS feature size will continue to decrease. For pragmatic reasons, transistors are not getting any faster. In addition, CMOS will become less reliable because of parameter variation and SEUs. Lastly, data movement will cost more energy than compute. The impacts of these trends on chip architecture will be discussed in next section.

2.2 Architectural Trends

Chip architectures synthesize what technology grants into compute power. Before the end of Dennard Scaling, uniprocessor architecture dominated the mass market, and a software ecosystem had been built on it. The end of Dennard Scaling forced the industry to shift to the multicore design paradigm. Now parallel architectures are ubiquitous, from low-end mobile processors to high-end server-class processors. It is not yet clear whether homogeneous architecture or heterogeneous architecture is better. We discuss major architectural trends in this section.

2.2.1 Homogeneous Architecture vs. Heterogeneous Architecture

When chip vendors introduced their first multicore microprocessors [5, 6], adding an identical copy of processor core to the design was a natural choice. Homogeneous architecture usually contains tens of processor cores. These cores are connected by a network-on-chip (NOC) with a cache-coherent memory hierarchy. The IBM POWER7 processor [32] is an eight-core design with a large on-chip embedded dynamic random-access memory (eDRAM) caches. Each core supports a four-way simultaneously multithreaded operation, and the eight cores provide 32 concurrent threads in total. To reduce power, the Power7 operates at a modest frequency and focuses on microarchitecture innovation for high performance. The Tilera's TILEPro64$^{\text{TM}}$processor [33] features 64 simple, three-way very long instruction word (VLIW) processor cores (tiles) connected by an on-chip mesh interconnect. Each tile can run a full Linux independently, or a group of tiles can run a full SMP Linux. Intel Core$^{\text{TM}}$i7 processor [34] is a high-performance general-purpose processor featuring out-of-order speculative microarchitecture.

While the homogeneous architectures have made significant impacts, many researchers and experts suggest heterogeneous many-core architectures as the future direction [8, 35]. Such a statement is based on an empirical observation called Pollack's rule [36, 37], which states that performance increase is roughly proportional to the square root of the increase in chip area. Due to the stringent power constraint and the energy cost of data movement, the return of architectural techniques (e.g., deep pipelining) diminishes. Therefore, computer architects leveraged Pollack's rule in a way that designs are smaller in terms of area and simpler in terms of data path. Such smaller and simpler designs consequently deliver less performance per unit. To further limit power consumption, chips have to run at relatively low clock speeds, which are expected to remain close to a few GHz, not utilizing the full potential. (Running at the full speed makes it uneconomical to cool [1].) Hence, there will be many small, simple, and slow processor cores on the chip [38].

In addition, there are conventional processor cores that are designed for high single-thread performance. The processor cores are coupled with the small, simple, and slow processors cores. These may leverage speculative execution and out-of-order techniques, which bring several benefits: (a) they compensates single-threaded performance and (b) they may provide a familiar software ecosystem to programmers (e.g., x86).

Taking these factors into consideration, future architecture is expected to combine both big, complex cores and small, simple cores to achieve good design trade-offs. Obviously, such architecture is heterogeneous. In fact, it has been shown that, for many workloads, heterogeneous hardware achieves better performance and power efficiency than conventional, general-purpose hardware [39].

System-on-chip (SoC) designs, which target mobile phones and embedded systems, are already heterogeneous and usually have a very tight power budget and expect a good performance. For example, the TI OMAP 5432 [40] uses two ARM Cortex-A15 processors for general-purpose applications and ARM Cortex-M4 processors for real-time applications. To process graphics and video applications, it has one dedicated graphics accelerator and video accelerators. ARM's big.LITTLE^{TM}architecture [41] combines high-performance Cortex-A15 processors and energy efficient Cortex-A7 processors. Cortex-A7 is an in-order processor that has a pipeline length between eight stages and ten stages. Cortex-A15 is an out-of-order processor with a pipeline length between 15-stages and 24-stages. ARM's report shows that the average performance of Cortex-A15 is two times as much as that of Cortex-A7, while the average energy efficiency of Cortex-A7 is three times as much as that of Cortex-A15. As a result, the big.LITTLE system enables threads to be executed on the processing resource that is most appropriate. From a programmer's perspective, the difference between Cortex-A15 and Cortex-A7 is hidden by the operating system.

For high-end computers, the power budget is still tight but less stringent. Accelerators are usually implemented as discrete components to provide higher performance [42, 43, 44]. These accelerators usually require a host computer for a software environment that is familiar to users. The Cell processor [45] combines processors optimized for performance per transistor on compute-intensive applications, with a more conventional processor architecture. The Cell processor also introduces software-controlled memory to allow overlapping computation with memory transfer. The SARC architecture [46] reuses Cell Synergistic Processing Elements but adds application-specific instructions.

2.2.2 Memory Wall

The exponential improvement of CMOS transistor and architectural innovation results in a tension between processor and main memory (i.e., "memory wall"). For economic reasons, the manufacturers of main memory have been focusing on the density instead of the performance [1, 47]. For a long time, memory latency has improved slower than the chip clock rate. Although the flattening of clock rate has a positive impact on "memory wall," the increasing number of cores continues to generate more concurrent memory requests and thus intensify this issue. The sheer number of parallel cores in future chips will continue to overwhelm current memory hierarchies, creating a situation where memory subsystems limit the rate of computation, but not the availability of parallelism or clock rate. To address this, some have suggested software-controlled data movement (rather than hardware-controlled) in the memory hierarchy [48]. Other recent developments suggested a programming model based on message passing through non-cache-coherent shared memory [49]. Others have proposed radically new memory hierarchies. In the Fresh Breeze project, Dennis *et al.* propose a view of main memory as a collection of write-once memory chunks [18]. The write-once principle frees programmers from maintaining the consistency of shared objects and leads to a functional view of memory, because one parallel task will not overwrite the internal memory of another task. It also enables active Checkpoint/Restart—the ability to concurrently checkpoint, while the application continues to progress.

2.3 Hardware-Software Codesign at OS Level

Operating system (OS) is arguably the most important software layer in runtime system. Heterogeneous architectures poses design challenges to the OS. Reconfigurable Computing community has long been successfully leveraging FPGA technology to deploy architecture that combines conventional processor cores and reconfigurable accelerators, using hardware-software codesign methodologies [50, 51, 52]. One of the most important codesign process aspects is to determine the boundary between the hardware and the software. A proper boundary normally tries to meet certain requirements. First, the boundary needs to be well understood by both software and hardware engineers to reduce non-recurring engineering cost. Secondly, the reconfigurable accelerators need to be treated as first-class citizens for efficiently utilization. Codesign methodology often takes advantage of the standardized semantics, such as UNIX semantics [51] and Pthreads [52], which greatly ease the interaction between the software and the hardware.

Hthread [52] proposed the design of *hardware thread* that complies with Pthreads APIs. In the hthread model, programmers specify their applications as a set of concurrent threads using the Pthreads semantics. A hardware thread shares memory with other threads and uses Pthreads synchronization primitives for communication.

ReconOS [53] also exploited thread-level parallelism as hthread does. In particular, the hardware threads are written in hardware description language (HDL) instead of being generated from a sequential language. When a hardware thread is created at runtime, a dedicated software thread is also created to represent its hardware counterpart. The dedicated software thread can communicate with other software threads through OS primitives.

BORPH [51] proposed the design of *hardware process* that conforms to the standard UNIX process semantics. A hardware process has a peer-to-peer relationship with software processes or other hardware processes and may communicate with its peers through UNIX file pipe which provides a one-way flow of data.

However, Moore's Law will grant more processing units per chip, which will force programmers to invest in parallelization techniques to increase the performance of their algorithms. In a multithreaded environment, programmers retrofit fine-grain locking to parallelize applications. Linux is the de facto OS in high performance computing. Since the chip industry shifted the paradigm from multicore architecture to many-core architecture, Linux has undergone many improvements addressing scalability issue. Big Kernel Lock (BKL) was first introduced into Linux to ease the transition to SMP systems. Essentially, the BKL is a global lock that only one thread in the kernel space can hold it. The BKL was later replaced by fine-grained locking mechanisms, such as mutex, spin-lock, and Read-Copy Update (RCU) [54]. More recently, an example of scalability efforts includes an analysis of Linux scalability to a 48-core machine [55]. In a high core count system, finding an optimal lock granularity for threads can be very challenging. The scalability issue in Linux has motivated several new operating system designs. Factored operating systems (FOS) [56] factors OS services into a set of communicating servers that are bound to distinct processing cores. An application sends messages to a server, which then executes the OS code and returns the result. Such design completely avoids global cache-coherent shared memory and the use of hardware locks. Multikernel [57] treats the machine as a network of independent cores and assumes no inter-core sharing at the lowest level. Each core holds a replication of the machine state. These new OS designs highlight the pressure a conventional monolithic kernel suffers. In addition, future hardware will be more difficult to sustain its reliability due to the parameter variation [12]. It is not clear if conventional monolithic kernel, which is a single point of failure, will be able to handle faults efficiently.

Our approach differs from these works in that we choose semantics of Python byte code as the core of our codesign. In the green-white architecture, we synthesize reconfigurable resources into soft processors to run distributed, lightweight Python virtual machines. Each lightweight Python virtual machine is capable of executing a parallel task represented by Python byte codes. Parallel tasks are independent in a way that a faulty green core is recovered locally without interrupting the application.

2.4 Programming Model

Programming models are roughly divided into three categories [58]: pseudo-comment directives approaches, language-based approaches, and library extension approaches. PyDac falls into the library extension category by providing programmers with a Python library to map tasks to the distributed Python virtual machines. The library approach ensures portability of the PyDac framework and eases the adoption of the framework by domain scientists.

Message passing interface (MPI) is currently the dominant programming model in HPC arena. With the emergence of the chip multiprocessors, a hybrid model called "MPI+X" is expected to better utilize hierarchical features of the hardware. For example, “MPI+OpenMP” [59], which combines the library extension approach and the pseudo-comment directive approach, builds a distributed memory programming model on top of a shared-memory programming model. To leverage increasingly popular heterogeneous hardware, OpenACC is proposed as an OpenMP-like directive set that supports accelerators, which is based on the concept of separate host and accelerator memory but emphasizes implicit memory management, which reduces programming burden on programmers. However, this hybrid model still requires programmers to invest a great amount of coding effort to utilize cores effectively [60]. In addition, MPI requires the number of processes to be specified when an application is launched. In the proposed programming model, the number of processes can be determined through the runtime system by dynamically adjusting the base case size.

OpenMP [61] is a popular programming paradigm for multicore SMP architectures. It inspires many similar programming models, such as CellSs, StarSs, and OmpSs. CellSs [62] is a programming model specifically designed for the Cell/BE processor. Similar to OpenMP, CellSs uses the pseudo-comment directives approach to create parallel tasks. However, when programmers annotate functions that need to be offloaded to accelerator cores, the annotation does not necessarily indicate parallel execution of a code section but a candidate for parallel execution. CellSs also features a source-to-source compiler by which applications are composed of two types of binaries. In fact, CellSs is contained in StarSs [63] as one of the instantiations. The StarSs programming model supports a wider range of architectures including multicore processor, GPU, Cell/BE, and cluster. Therefore, it provides a more natural support for heterogeneity than OpenMP, while the portability is not compromised. StarSs views architectures that feature separate memory spaces (i.e., host and device memories) as a two-level memory hierarchy and provides a software layer that implements memory coherence policies. The runtime system of StarSs automatically handles the data movement in its two-level memory hierarchy.

Sequoia [64] and Merge [65] introduce new language constructs to support map-reduce patterns. Sequoia abstracts a memory hierarchy as a tree of distributed memory modules and constrains the compute kernels to operate on leaf nodes. Task variants are generated statically by the compiler to be portable across levels of memory hierarchy. In contrast to using a task variant to suit different levels of memory hierarchy, PyDac focuses on a flatter memory hierarchy and generates tasks that specifically run on the scratch-pad memory. Merge, on the other hand, maps an application to a library of function-intrinsics that encapsulate accelerator-specific code. Merge's runtime automatically distributes computation to accelerators. In addition, Merge's framework removes OS and driver layers for accelerators. Similar to Merge, PyDac removes OS and driver layers on green cores to eliminate software bloat. However, PyDac allows tasks to migrate between cores by leveraging virtual machine byte codes.

Intel Thread Building Blocks (TBB) [24] is a C++ template library that is based on a work-stealing scheduler and provides control on low-level parallelism. It supports many popular design patterns, such as pipeline and divide-and-conquer. It abstracts away the complexity of using native threading packages (e.g., Pthreads). However, TBB only aims at shared memory architecture. In contrast, PyDac targets not only shared memory but also a novel memory subsystem that supports write-once memory model.

Intel Concurrent Collections (CnC) [25] is a programming model that provides higher level abstraction than TBB. CnC separates the development of parallel applications into two distinct stages. The first stage requires a domain expert, who understands data dependency and control dependency in an application but may not be an experienced parallel programmer, to write program in terms of high-level application-specific operations. The second stage relies upon tuning experts, who have expertise in extracting maximum performance from the computer, to tune the program for a specific architecture. In fact, CnC could be built on top of TBB as an approach to leverage shared memory computers. CnC imposes several important rules on domain experts. For example, computation (called “step collection” in CnC) may not reference any global values. Data (called “item collection”) is referenced by value instead of by its location. In addition, dependency should be explicitly stated. Such rules are not existent in serial languages, such as C/C++. These rules together eliminate race condition at the domain expert level and deliver explicit and useful constraints to tuning experts. Similarly, PyDac requires a base case to be referentially transparent. Each green core may reference data by its value instead of by location. While CnC provides a higher level abstraction to programmers that may require additional software support, PyDac intends to bring high-productivity programming closer to the hardware. In addition, PyDac focuses on the divide-and-conquer design pattern and provides a direct hardware support.

Microsoft’s Accelerator [66] hides the GPUs details from programmers by providing C# APIs (each associated to one array operation) and uses just-in-time compilation. PyDac does not use just-in-time compilation technique but requires some C libraries running on green cores to be statically compiled.

Parallex [67] is a programming model specifically designed for extreme-scale computing systems. Parallex has a view of global address space where objects (e.g., data and code) are identified by globally immutable names. Parallel threads in Parallex are first class objects with immutable names. As such, it is possible to move computation to data, which may reduce data movement and save energy. Instead of statically allocating threads, ParalleX dynamically schedules multiple threads using message-driven mechanisms for moving the work to the data.

More recently, there is a renewed interest in task-based parallel programming models [68]. Programmer are responsible for identifying which parts of the application can be computed in parallel. A runtime environment maps these parallel runnable computations to the available processors in the system. Cilk [69] is a widely available extension of C, which is a popular example of task-based parallel programming. Cilk uses keywords, such as *spawn* and *sync*, to identify safely runnable parallel computations. It does not specify any limitation on the size or the simplicity of these parallel tasks. Atlas [70] is a Java-based runtime system that adapts the Cilk programming model, extends work-stealing of Cilk scheduler with a hierarchy, and borrows fault tolerance mechanism from Cilk-NOW [71]. Satin [72] is also a Java-based runtime system that extended work-stealing with a cluster-aware capability. To the best of our knowledge, these runtime systems have not yet supported for heterogeneous many-core architecture with software-controlled memory.

Another important aspect of programming models is productivity. A group of parallel languages based on Partitioned Global Address Space (PGAS) include UPC [73], Titanium [73], Co-array Fortran [74], X10 [58], and Chapel [75]. PGAS enables writing codes in the global view style in which programmers express their algorithms and data structures as a whole. These languages tend to provide much more fine-grain control support.

X10 [58] is Java-based language with new language constructs for high-productivity high-performance parallel programming. Designed for concurrent and distributed programming, X10 supports notions of non-uniform data access across nodes, partitioning its global address into a set of *places*. A *place* contains a collection of data and activities that operate on the data. Mapping between *places* and physical locations is separate from the X10 program. Regarding data access, each activity reads and writes a shared-memory location synchronously within a *place*. To read or write remote data (i.e., another *place*), an activity may spawn new activity at a remote *place* to perform data access. Specifically, asynchronous activity is created and synchronized by language constructs *async* and *finish*. Nested *async* and *finish* allow more than one level of a *divide-and-conquer* phase. It also provides more fine-grain control on the activity with construct *future*.

Chapel [75] is a productivity-oriented programming language. Instead of giving programmers access to the threads via low-level fork/join mechanisms and naming, it provides high-level abstractions for parallelism using anonymous threads. It relies upon the programmers instead of the compiler to identify parallelism. To make parallel programming friendly to programmers who are more familiar with sequential languages, Chapel provides a rich set of built-in data structures and broad-market features. To manage data distribution and locality, it provides locality-specific construct *locale* for tasks that have uniform access to the machine's memory.

PyDac also emphasizes programming productivity by coding in a global view style. In PyDac, each green core has it own address space, but these spaces do not form a global address space. Another approach is to combine productivity-level languages and efficiency-level languages. SEJITS [76] leverages just-in-time technique to dynamically generate efficiency-level code from productive-level code. PyDac does not use just-in-time compilation technique but requires some C libraries running on green cores to be statically compiled. PyCUDA and PyOpenCL [77] are toolkits that improve GPU programming productivity by GPU runtime code generation within Python language. PyDac also intends to leverage Python programming language for programming heterogeneous hardware (e.g., the green-white architecture).

CHAPTER 3: GREEN-WHITE ARCHITECTURE

This chapter introduces a novel heterogeneous many-core chip architecture—"green-white" architecture. PyDac programming framework and green-white architecture are two aspects of a novel hardware-software codesign. The green-white architecture intends to ride technological trends into the era of heterogeneous many-core computing. PyDac focuses on solving consequent technological issues, including performance, resilience, and productivity.

To construct a model of the green-white architecture, we take advantage of two approaches—simulation and hardware emulation. This chapter discusses the advantages and disadvantages of each approach, presents the design of green-white architecture, and shows a hardware prototype of green-white architecture, which serves as an experimental setup for evaluating PyDac.

3.1 Modeling Techniques

Computer architects rely on modeling techniques to gain insights about how well their design may work. Modeling techniques could be divided into three main categories, depending on cost and accuracy: analytical modeling, simulation, and emulation.

Analytical modeling is usually applied in the earliest stages of design and focuses on one or more essential mathematical computer system design formulas. In addition, this technique intentionally ignores most of the design details, making it both faster and more inaccurate than other techniques. However, the inherent inaccuracy does not undermine the importance of this technique. Analytical modeling helps designers to make high-level design decisions and often leads to insights. For example, Hill et al. extend Amdahl's law to many-core processor design [35]. Based on a simple hardware cost model, they explore three different many-core designs (i.e., homogeneous, heterogeneous, and dynamic). Despite its simplicity, they came to an insightful conclusion that the heterogeneous architecture results in better performance than the homogeneous architecture. In fact, Hill et al.'s analytical modeling motivates this research.

Simulation generates more accurate results than analytical modeling by taking many design parameters into the model for consideration. It is also relatively cheaper compared to building hardware prototypes. In addition, some simulators provide software developers with a fully controlled environment. The developers may stop code execution and examine machine states freely. Such a feature is very helpful for debugging code; therefore, computer architects extensively apply simulation. For example, the gem5 simulator [78] allows complete software stacks, including unmodified commercial OS to run on the simulator.

In general, emulation can be divided into two groups: (a) emulation through software and (b) emulation through hardware. While the difference between simulation and emulation through software might seem obscure, the latter approach closely resembles the behavior of real systems (i.e., target designs). For example, QEMU [79] is a machine emulator that dynamically translates target CPU instructions into host instructions. Also, computer architects commonly use emulation through hardware (or hardware emulation). Both industry and academia use many different hardware emulation approaches. Here, we follow a taxonomy presented in Lieven Eeckhout's lecture [80]. A *functional emulator* is a circuit that is functionally equivalent to a target design, but does not provide any insight on specific design metrics. Its advantages include faster emulation speed than software simulation, because it can execute code at hardware speed. A *model* is a representation that is functionally equivalent and logically isomorphic with the target design. It allows for some abstraction, which simplifies model development. A *prototype* is also a functionally equivalent and logically isomorphic representation of the target design. However, it implements the same structure (i.e., the same hardware description language code) as in the target design. Because a prototype can be used to project performance, it is a useful vehicle for studying the scalability of software. In particular, many computer architects implement their prototypes through FPGA devices.

An FPGA device is an integrated circuit in which hardware configuration can be done after manufacturing process. A user may use hardware description language to program an FPGA device and implement desired hardware functions. Because an FPGA device can be re-programmed many times, its non-recurring engineering cost is relatively lower than an application-specific integrated circuit (ASIC) design. In addition, FPGA devices also benefit from Moore's Law. Therefore, the density of FPGA devices is able to grow with newer generations, which allows designers to emulate more sophisticated designs. Moreover, FPGA emulation is often hundreds of times faster than simulation, especially when application software and system software need to run against hardware design.

3.2 Theoretical Model of Green-White Architecture

Based on technology advancements and trends, this work is motivated to prepare for a hardware design that is: (a) heterogeneous many-core, (b) combined with scratch-pad memory, (c) likely to experience higher rates of faults, and (d) supported by a flat memory hierarchy. One such chip architecture is called green-white architecture, as illustrated in Figure 3.1.

This architecture assumes a view of main memory that is similar to Fresh Breeze [18]. Specifically, it assumes a flatter memory hierarchy coupled with a set of special compute cores that are denoted as *green core* (GC) in Figure 3.1. An on-chip network connects the active memory subsystem, which actively manages chunks of memory, to the green cores. Each green core consists of a simple processor and multiple, multiplexed banks of scratch-pad memories (locally byte-addressable blocks of memory)

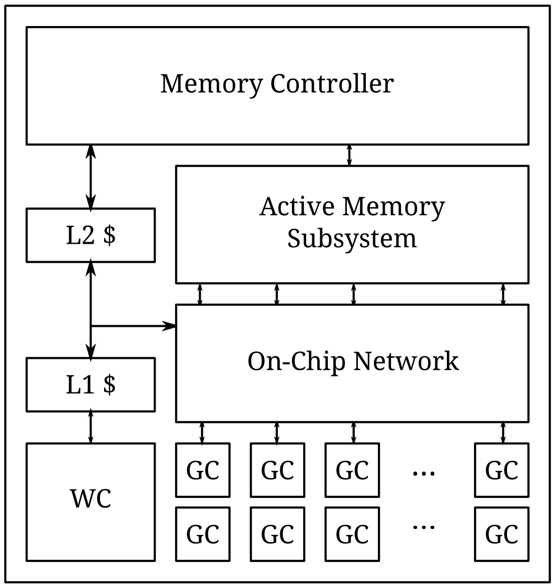

Figure 3.1: High-level block diagram of the green-white architecture.

that are actively managed. This is in stark contrast to the conventional memory subsystem, which consists of multiple layers of reactive caches. If the cores are oversubscribed with tasks, the proposed arrangement allows the memory subsystem to actively manage data transfer to one bank while a task is executing out of another bank, effectively overlapping memory movement and computation [48]. This allows for better utilization of off-chip memory bandwidth, helps hide latency, and reduces energy consumption.

The management of data transfer is illustrated in Figure 3.2. The processor inside a green core accesses one of the banks of scratch-pad memories in byte-addressable transaction. Such transaction is the same as the transaction between the level-one cache and a processor core in conventional designs. The cost of switching between banks is usually negligible, and latency for accessing a bank is very low. Therefore, enough banks of scratch-pad memories keep the processor busy continuously, because the processor is never starved for data. From the processor's perspective, it never has a cache miss and need not go fetch data from a level-2 cache (which is why the memory is flat). The data transfer between the scratch-pad memories

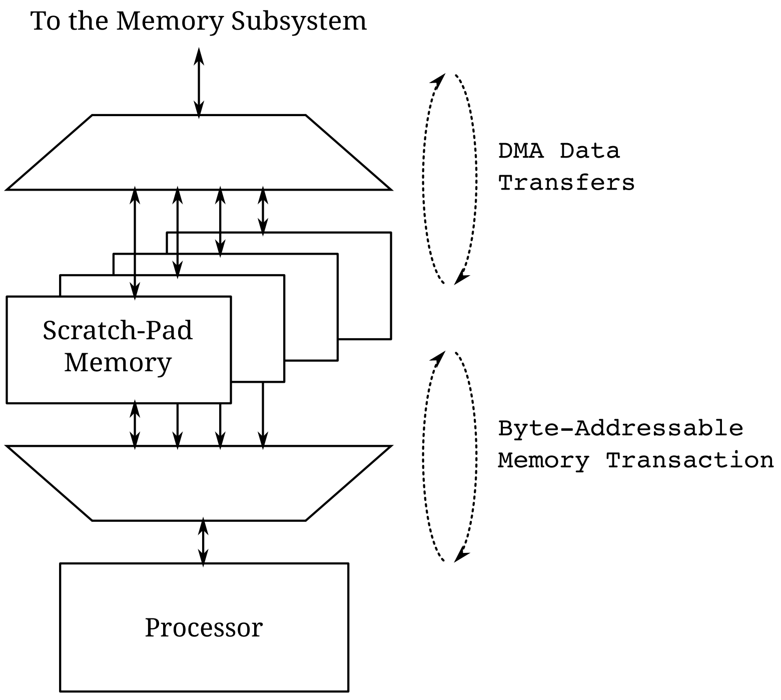

Figure 3.2: High-level block diagram of a green core.

and the memory-subsystem is through direct memory access (DMA). Typically, a few kilobytes of data can be moved within one DMA transfer. From an active memory subsystem's perspective, one DMA transfer may contain one memory chunk or multiple memory chunks.

The active memory subsystem is co-designed with a programming model from the beginning. Briefly, the programming model allows programmers to decompose a problem into sub-problems. The process ends when a sub-problem is small enough that a fast direct solution (called a *base case* in the divide-and-conquer strategy) is possible. The major criterion for a basic case is that it "fits" into the scratch-pad memory of a green core. High degree of parallelism fundamentally enables latency hiding through multiple scratch-pad memories. In addition, with a large number of parallel tasks that fit into the scratch-pad memory, enough memory transactions will be available for the memory subsystem to efficiently use the memory bandwidth and keep green cores busy. These parallel tasks are independent from each other and run on the green cores in the green-white architecture. The working data set for each task essentially is one or multiple memory chunks managed by the active memory subsystem. Restricted by the programming model, tasks are allowed to read a memory chunk many times but to write only once. Since multiple tasks may "subscribe" to the same memory chunk simultaneously, the active memory subsystem uses reference counting technique to track the number of subscribers. This number is important, because it allows the active memory subsystem to move a memory chunk up and down in the memory hierarchy and keeps the most needed memory chunk always on-chip. In addition, coherence issue is eliminated through a write-once policy. If one green core subscribes to a memory chunk, then any attempt of writing to the memory chunk by other green cores creates a new memory chunk instead of overwriting on the old one.

The processor core inside the green core is slow, small, and simple, presenting itself to provide increased system throughput (tasks completed per second). Conceptually, these processor cores incorporate low-power techniques (low clock rate, no protection) and feature a simpler design (e.g., reduced pipeline depth, no branch-prediction) to save silicon footprint and reduce power. To achieve the power efficiency (performance per watt), these processor cores are designed to be more application-specific and less reliable.

The chip architecture also includes one or more very fast, complex cores—denoted as *white core* (WC). The white cores are present to reduce the latency of sequential tasks and might be hardened to protect against faults. Conceptually, these cores incorporate the latest advances in single-thread performance and incorporate techniques (higher power, protection, hardened) to increase reliability. The on-chip network connects the green cores to a memory subsystem, providing direct access to the blocks of write-once memory. The white cores have a conventional memory hierarchy. The two memory subsystems share (off-chip) DRAM memory resources through a multi-ported memory controller.

In summary, the main assumption is that future devices will be a mix of simple and complex cores. There will be many simple cores (because they are smaller) and they will use less energy. However, these advantages come at the expense of reliability. In contrast, the complex cores (which are very expensive in terms of energy and resources) will be essential for sequential code and as a safe haven for critical operations. The active memory subsystem manages data transfer for multiple, multiplexed banks of scratch-pad memories in green cores so that the processor in the green core never starves.

3.3 An Implementation of Green-White Architecture

This section describes how to construct a model of the green-white architecture through a combination of two approaches — simulation and hardware emulation. The green core is first simulated as a subsystem for debugging and software development. Then, a hardware prototype of green-white architecture is implemented on an FPGA device.

3.3.1 Hardware Simulation

ARMv2a soft processor (called *Amber* [81]) is chosen to represent the processor in the green core because its source code is freely available. The ARMv2a processor has a three-stage pipeline, a unified instruction and data cache, and is capable of 0.75 DMIPS per MHz. The green core simulator incorporates one ARMv2a processor core and several peripheral cores, such as a timer, an interrupt controller, and a UART. The ARMv2a processor core's HDL code and peripheral cores can be synthesized into an FPGA device. The green core simulator also incorporates some modules that can not be synthesized into an FPGA device. These modules include scratch-pad memories and a clock generator. The green core simulator uses ModelSim^{TM}and VCS^{TM}.

The green core simulator also provides a sophisticated interface to software developers. A software developer with the knowledge about the hardware configuration of the green core simulator may write a wide range of applications and quickly verify applications. These applications may include a test that consists of tens of lines of assembly code or a Linux OS. When the simulation is launched, the green core simulator first invokes an ARM cross-compiler. The ARM cross-compiler compiles the application into an executable. The information that could not be executed, such as comments and debugging information, are then stripped to save memory space for the simulator. After that, a custom tool converts the reduced executable into a memory image, with which the testbench of the green core simulator is initialized. Once the ARMv2a processor core is reset, it fetches the first instruction from the memory and starts execution. With the green core simulator, software development could start very early, which reduces the risk of debugging a very complex software system on a hardware prototype. In fact, the virtual machine for green core is developed and debugged on the green core simulator before it is tested on the hardware prototype.

Even though the green core simulator only simulates a portion of the envisioned green-white architecture, the software developed on this simulator is easily reusable. This is because the proposed programming model decomposes a problem into many *stateless* tasks. In other words, the output of each task only depends on its inputs. When such task is developed on the green core simulator, the main goal is to verify that the output of the task is correct. Once it passes verification, it becomes a “black box” to the final runtime system running across both green cores and white cores. Later, if a software bug is suspected on the task running on the green core, the programmer only needs to examine the input of this task.

3.3.2 Hardware Prototype

In order to evaluate the proposed programming model and the runtime system, a *prototype* of the green-white architecture was emulated on a Xilinx Virtex 5 FPGA device on an ML-510 developer board, as illustrated in Figure 3.3. An overview of

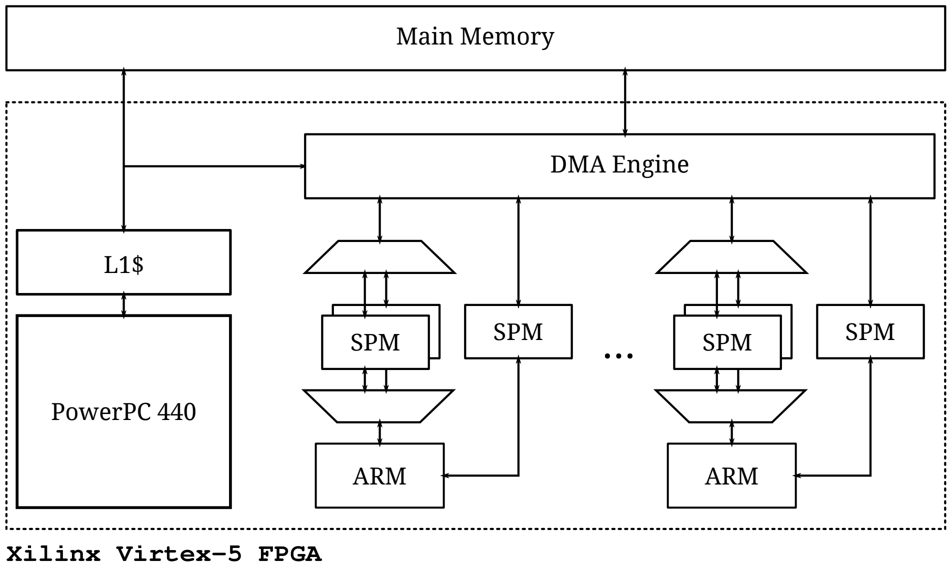

Figure 3.3: High-level diagram of green-white prototype on an FPGA device.

Table 3.1: Summary of the hardware prototype.

FPGA Board	Xilinx ML510
White Core	1 PowerPC440 at 400 MHz
Green Cores	6 ARM (v2a) cores at 50 MHz Each with 160 KB scratch-pad memory
Memory System	DMA assisted by software
Interconnect	Bus

this prototype is presented in Table 3.1. This prototype combines two type of cores (one of which owns multiple, multiplexed scratch-pad memories) and features a flat memory hierarchy.

Each green core is equipped with an ARM processor core and a 160 KB scratch-pad memory. The ARM processor is clocked at 50 MHz. The prototype has six ARM processor cores. The available on-chip resources of the FPGA limits the number of cores. The scratch-pad memories are single-cycle latency on-chip memories. The 160 KB scratch-pad memory is further divided into three banks: one 128 KB bank and two 16 KB banks. The three banks are all dual-ported: one port interfaces to the DMA engine and the other interfaces to the ARM processor core. From the perspective of the runtime system, the 128 KB bank and 16 KB bank are designed for different purposes, and therefore, they show different memory access patterns. The 128 KB bank holds virtual machine executable and ephemeral contents (e.g., heap and stack) for the ARM processor core. Therefore, the DMA engine only accesses the 128 KB bank for initialization and fault recovery. Unlike the 128 KB bank, the two 16 KB banks that hold the content of parallel tasks are multiplexed on both ports. The DMA engine frequently accesses the 16 KB banks for moving tasks and data around without interfering with execution on the ARM processor core. These 16 KB scratch-pad memories allow overlapping communication with computation.

The white core is a PowerPC 440 core clocked at 400 MHz, with 2.0 DMIPS/MHz performance [82]. The PowerPC 440 core integrates a superscalar seven-stage pipeline, separate instruction and data caches, and a memory management unit (MMU). The Xilinx Virtex 5 FPGA device on the ML-510 developer board provides two PowerPC 440 hard cores. However, only one PowerPC 440 core is utilized in this work, due to the capability of the OS. A bus and a DMA engine are implemented as the system interconnect, because it was more expedient than a network-on-chip.

The active memory subsystem discussed in the previous chapter is greatly simplified in the prototype. The DMA is a bidirectional streaming engine transferring data between the scratch-pad memories and the main memory. This engine takes the "starting address" and "data length" as input, and streams the data without involving the white core or the green core. A memory chunk anticipated by the active memory subsystem is emulated through data segments specified by the "starting address" and "data length." A software module in the runtime system actually manages data transfer between main memory and scratch-pad memories without a fully active memory subsystem.

CHAPTER 4: DESIGN OF PYDAC PROGRAMMING FRAMEWORK

A programming framework normally includes application programming interfaces (APIs), necessary libraries, compilers, and a runtime system. It provides an abstraction layer to users who develop application-specific software. Such abstraction visible to programmers is also known as a programming model. The runtime system also plays an important role. A runtime system, which is not visible to users, interfaces to programming model and hardware and provides several responsibilities: (a) it abstracts the underlying hardware, (b) it implements the core behavior of programming model and programming language, and (c) it maps the core behaviors of programming model and language to hardware and manages resources to meet requirements, such as power and performance.

Heterogeneous many-core architectures (e.g., green-white architecture) offer a good balance between single-threaded performance and multithreaded throughput. Such systems impose many challenges on the design of a programming model and a runtime system. Specifically, these include: (a) how to fully utilize the chip's performance, (b) how to manage heterogeneous, unreliable hardware resources, and (c) how to generate and manage a large amount of parallel tasks.

In this chapter, details are first given about a Python-based programming model called PyDac, which supports a two-level programming model based on the divide-and-conquer strategy. This programming model supports green-white architecture. To test the spectrum, PyDac also runs on conventional SMP architecture. We then present the design of a runtime system that is specifically co-designed with green-white architecture. The runtime system seamlessly manages the parallel tasks, system resilience, and all inter-task communication with architecture support.

4.1 Programming Model

The primary goal of the programming model is to make it possible to write programs that generate a very large number of parallel tasks without a great deal of programming effort. A functional style of programming [83] is very good at this but is generally viewed as difficult for computational scientists to use. In addition, there is a popular belief that the functional programming style leads to a mediocre performance. Pankratius et al. countered this belief through an empirical study evaluating Scala—a multi-paradigm programming language—and Java [84]. Their controlled study showed that programmers whose programs result in superior performance wrote about half their programs in a functional style and the other half in an imperative style. The result indicates the promise of the combination of the functional and the imperative programming styles. This is because using the imperative style may compensate the functional style for the potential performance loss.

The main idea in this programming model is to implement a two-level programming paradigm. It borrows the concept of the divide-and-conquer strategy from the functional programming style to decompose data and create tasks. The two-level programming paradigm uses the imperative style for individual tasks. The PyDac programming model is implemented with the Python programming language. Python is considered an easy language to learn, it supports both the functional and imperative styles, and it has popular modules to support scientific applications. (However, there is no reason that other high-level programming languages could not be used for this model.) Specifically, a programmer who wants to use this programming model needs to learn two concepts. The first is the divide-and-conquer strategy, and the second is how to express it in Python. In this section, a two-level programming model that suits both the SMP platform and a heterogeneous many-core platform is presented. Cases are studied to illustrate how applications are developed under this model.

4.1.1 Divide-and-Conquer Strategy

Divide-and-conquer is a well-known technique for designing algorithms in the computer science community. Three steps are usually involved in this technique: *divide*, *conquer*, and *combine*. In other words, the divide-and-conquer strategy recursively decomposes a problem into smaller sub-problems, which in turn are decomposed into sub-sub-problems, and so on. The process ends when a sub-problem is small enough that a fast direct solution is possible. Many algorithms based on this strategy have a clear performance model described by the Master theorem [85] when base case sizes are equal. For many algorithms based on this strategy, the number of base cases grows exponentially with input size, which helps to uncover a significant large amount of parallel tasks through a finite number of statements in the program.

The divide-and-conquer strategy is also widely applicable. The applications based upon this strategy include fast Fourier transform (FFT) [86], sorting [87], many linear algebra problems [88, 89], data visualization [90], biological sequence alignment [91], pattern recognition [92], neural network [93], image processing [94], graph algorithm [95], search algorithms, and geometry functions. There are other important algorithm design paradigms, such as dynamic programming. We focused on the divide-and-conquer strategy in this work, and other algorithm design paradigms are beyond the scope of this dissertation.

The divide-and-conquer strategy provides an opportunity to design algorithms without knowing hardware parameters, such as cache size and cache-line length. These algorithms are also known as *cache-oblivious* algorithms [96]. Such algorithms have many unique features [97]: (a) algorithm designers could design and analyze their algorithms in a much simpler two-level memory model, (b) the algorithm designed for the two-level memory model works well on an arbitrary many-level memory hierarchy, and (c) the designers could port code to machines with a different memory hierarchy easily. Future machine architecture may exhibit a deeper hierarchy to programmers. It is because the system-level size will grow and processor cores will likely be organized into a hierarchy. Designing an algorithm for a machine with a deep hierarchy will be more difficult. In addition, manual management of data movement to achieve good performance and good power efficiency is a challenging task. The cache-oblivious algorithm based on the divide-and-conquer strategy can alleviate such burdens on programmers.

Lastly, the divide-and-conquer strategy favors an asynchronous and local communication pattern as opposed to a synchronous and global one. In a highly parallel system, the synchronous behavior is very sensitive to variance. For example, Petrini et al. found that substantial performance loss occurred when an application resonates with non-orchestrated system activities on the 8,192-processor ASCI Q machine [98]. The asynchronous communication pattern is less sensitive to such variance.

4.1.2 Two-Level Programming Model

The programming model in the PyDac programming framework embodies the divide-and-conquer strategy in a two-level style. At the higher level of this model, the recursion follows the functional programming style and decomposes the data. At the lower level, the base case is solved in an imperative style, which is strongly embraced by the computational science community. Thus, the programming model follows a historically successful approach of using productivity-enhancing techniques—such as object-oriented programming with C++/Java and communicating sequential processing programming with MPI—at the high-level and imperative-style programming at the low-level.

Figure 4.1 illustrates a common code template that assists programmers to produce code under the programming framework. The code template starts with a function that solves base cases that need to be referentially transparent. In other words, programmers are not allowed to make references to global variables in the base case. For example, for a low-level language such as C, global variable is not allowed. For the Python language, the keyword "global" is not allowed. Also, the programmer is not allowed to pass mutable object to the function.

```
Function base_case( sub-problem ) is
    Solve the sub-problem directly;
    return result;
end

Function divide_and_conquer( problem, base case size ) is
    Data: Data to be decomposed or data to be processed in base case.
          User-specified base case size.
    Result: Merged result or direct result from a base case.
    if problem size is small enough then
        Invoke the function base_case();
    else
        Break the problem into smaller sub-problems;
        Invoke the function divide_and_conquer() and pass sub-problems ;
        Merge results from sub-problems;
    end
    return result;
end
```

Figure 4.1: PyDac algorithm template

The idea of breaking a large problem into smaller sub-problems, which eventually leads to a basic problem, is commonly used by functional programming languages. Despite its advantage in eliminating memory coherence problems, functional language has been largely rejected by the scientific programming community. In addition, the computational science community strongly embrace the imperative-style programming. Therefore, when implementing this programming model, we avoid a pure functional programming language. Instead, we are in favor of a multi-paradigm language that supports both the functional style and the imperative style. In addition to multi-paradigm, language popularity, especially in the scientific community, is one big concern.

This programming model is implemented in the Python programming language, which is known for clear syntax, ease of programming and multi-paradigm language (e.g., object-oriented (OO), imperative, and functional programming styles). Python also provides flexibility in that its functionality can be extended by attaching libraries of C functions (or even C with extensions for hardware accelerators [99]) into Python executable. Python has been a desirable language for quick prototyping applications in the high-performance computing field. Libraries, such as NumPy [100] and SciPy [101], further allow effective usage of Python in scientific computing. Recent developments, such as MPI [102] and Cuda [99], have demonstrated the interoperability of Python with other languages and programming models.

4.1.3 Fibonacci Algorithm Coding Example

To illustrate how this programming model works, we use Fibonacci algorithm implementation as an example. Specifically, the Fibonacci algorithm divides a problem F_n into two sub-problems F_{n-1} and F_{n-2} until it reaches the base case F_1 and F_0. Figure 4.2 illustrates the implementation of Fibonacci algorithm under this programming model. Our APIs hide the Python implementation from the programmer. In fact, this coding example runs on the standard Python implementation (i.e., CPython) and a variant called Stackless Python [103].

Specifically, line 16 shows how a problem is decomposed into two sub-problems. Each sub-problem is represented by a Python tuple, which is enclosed by parentheses. All sub-problems are contained in a Python list, which is enclosed by square brackets (i.e., *subpb*). At line 17, the sub-problems and the decomposition function (i.e., *fib_op()*) are passed into one of the APIs—*divide()*. The *divide()* interfaces into the runtime system that makes the decision on spawning parallel tasks. Unlike the communicating sequential processes (CSP) programming model, the communication is hidden from application developers, and the runtime system carries out synchronization. Line 18 presents the "merge" phase in Fibonacci algorithm. In particular, the results of sub-problems are stored in a Python list (i.e., *results*) which has one-to-one correlation with the Python list containing sub-problems.

```
1  def fib_base(n):
2      """Base case in imperative style
3      """
4      a, b = 0, 1
5      for i in range(n):
6          a, b = b, a+b
7      return a
8
9
10 def fib_op(n, bc_size):
11     """Divide-and-conquer in functional style
12     """
13     if (n <= bc_size):
14         result = ship(fib_base, [n])
15     else:
16         subpb = [(n-1, bc_size), (n-2, bc_size)]
17         results = divide(fib_op, subpb)
18         result = results[0] + results[1]
19     return result
```

Figure 4.2: Fibonacci algorithm implemented in PyDac.

A pivotal parameter *bc_size* (short for base case size) sets the boundary between the high level and the low level (line 13). Programmers determine the base case size and may exploit it for performance and resilience purposes. For instance, on an architecture that supports local recovery of a faulty processor core, this parameter allows dynamical adjustment of the workload size on the faulty processor core for least recovery overhead [104]. As shown at line 14, programmers may use the other API—*ship()*—to tell the runtime system when the sub-problem becomes small enough. The runtime system makes the decision whether or not to solve the sub-problem on co-processors (i.e., green cores).

4.2 Design of PyDac Runtime

The PyDac runtime abstracts the underlying hardware into distributed virtual machines. Therefore, a complete PyDac runtime consists of the following software components: (a) distributed virtual machines, (b) Python modules (which includes

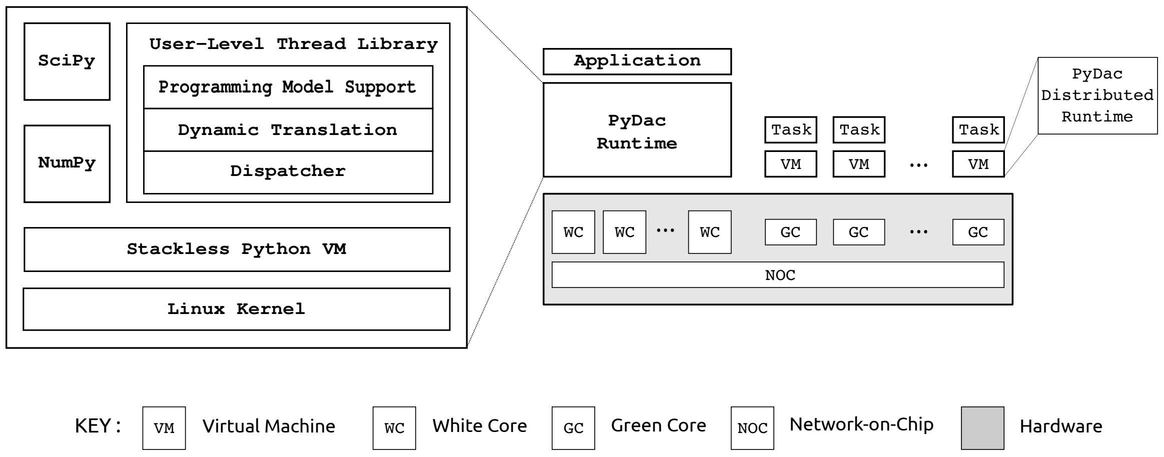

Figure 4.3: Proposed resilient runtime system on *green-white* architecture.

a user-level thread library and popular Python packages, such as NumPy [100]), and (c) independent and parallel tasks. Figure 4.3 presents a very high-level view of this design. The key features of this runtime system are:

- The runtime system is distributed as multiple virtual machines (VMs)
- OS layer is removed from the software stack on the green cores
- A user-level thread library provides the programming model support
- It reuses Python modules that are popular in the computational science community

PyDac consists of two types virtual machines: a full-featured Python interpreter that supports many standard libraries and a lightweight Python interpreter (code-named *PyMite* [105]) designed for micro-controllers. The lightweight Python interpreter was chosen for the current implementation for two reasons. First, its memory footprint is small, which makes it fit into an on-chip scratch-pad memory. Therefore, it does not generate memory requests to the off-chip memory. Secondly, it is built on the same set of bytecodes (version 2.6) as the full-fledged Python interpreter. Due to the same bytecode version, a task can be migrated between VMs.

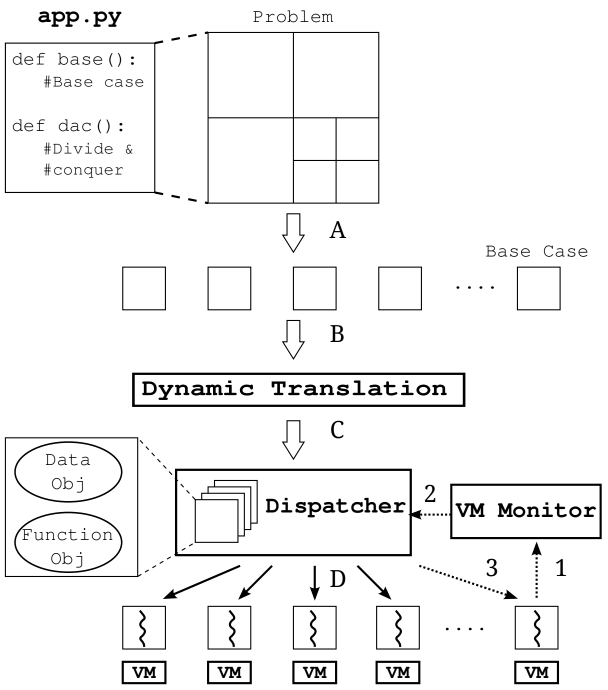

Figure 4.4: Control flow from dividing a problem to base case computation. Notation A-B-C-D shows mapping tasks to distributed virtual machines. Notation 1-2-3 shows monitoring and local recovery.

The user-level thread library in Figure 4.3 is the glue layer between the full-featured Python interpreter and the lightweight interpreter. Specifically, it consists of three sub-modules: an interface to programming model, a dynamic translation layer, and a dispatcher.

This programming model uses a Python implementation called Stackless Python, which has tasklets. (We implemented this programming model with CPython. That implementation uses Linux processes instead threads. Our experiments show that Stackless Python is about 100× faster.) Python has not yet been proven as a scalable solution due to Global Interpreter Lock (GIL) in the default Python implementation — CPython [106]. CPython uses a more heavy-weight mechanism than threads for concurrency. GIL serializes Python's own execution but does not affect the execution of non-Python threads. In this divide-and-conquer computation model, Stackless Python [103] replaces CPython for the benefits of thread-based programming.

In the divide-and-conquer programming paradigm, programmers specify a function name and input data for a compute kernel that solves a basic problem. The dynamic translation layer converts the function object (which is treated as a first-class object in Python) and input data into two C language byte arrays. The lightweight interpreter treats a function/data pair as an independent parallel task. A function byte array can be distributed to multiple VM instances in single program multiple data (SPMD) fashion. As soon as the basic problem is solved, the dynamic translation layer re-constructs the computation results into objects recognizable to full-fledged Python.

In the PyDac runtime framework, resilient execution mainly comes from isolation of function objects and data objects. These objects are scheduled to run on green cores. If a process running on a green core is hit by a fault, then the dispatcher can restart it. Since the OS has been removed from the software stack on green cores, restarting a green core does not affect the OS running on the white core.

Segregation of input parameters for parallel base case tasks enables a functional view of scratch-pad memory, where input parameters of one task cannot be modified by other tasks. Also, the internal memory of one task is hidden, and no part of a task can depend on values outside of internal memory or input memory of the task. Therefore, the resulting tasks are completely independent.

As is presented in Figure 4.4, the programmer recursively divides the problem until it reaches the base case. Then, the base cases are translated by the dynamic translation layer into a byte sequence, the format of which is recognizable to Pymite. A dispatcher running as a daemon on the complex core schedules base case tasks and drives multiple VM instances.

4.3 Concluding Remarks

The PyDac programming model is a two-level programming model based on the divide-and-conquer strategy. This programming model allows generating a large amount of parallel and asynchronous tasks through recursion. A high degree of parallelism is a key to fully utilizing future high-performance computing systems. Two-level design reduces incurred programming complexity. The high-level allows programmers to focus on decomposing problem. Through such a decomposition process, the complexity of a problem is reduced to a degree that problem solving can be done in an imperative style and can be optimized by a close mapping to hardware. The proposed PyDac programming model is implemented in Python programming language.

CHAPTER 5: EVALUATION

The hardware prototype of the green-white architecture was used to evaluate the proposed PyDac programming framework. Several micro-benchmarks were developed under the PyDac programming model. The performance of running micro-benchmarks on the hardware prototype was then analyzed. Lastly, the programming productivity was carefully reviewed through case studies.

5.1 Performance Evaluation

5.1.1 Benchmark Suite

In order to exercise the prototype hardware of green-white architecture, a synthetic benchmark suite containing compute *kernels* was developed. These compute kernels are common seen in high-performance computing area, and they are developed under the PyDac two-level programming model.

The *Strassen* micro-benchmark [107] partitions a square matrix into four equally sized block matrices, which are then further divided in a recursive manner until they become small enough that the sub-matrix and local variables fit into the scratch-pad memory of a green core. The time complexity of the Strassen's algorithm is better than conventional triple-loop matrix multiplication for large matrices; however, it is the large degree of parallelism that makes this operation attractive.

The *Block Matrix* micro-benchmark partitions a multiplicand matrix and a multiplier matrix into two equally sized block matrices, by row and by column respectively. These block matrices are further divided recursively in the same manner (by row or by column). The resultant matrix from a base case is concatenated with other sub-matrices into the final product.

This *Merge Sort* micro-benchmark sorts a set of integer keys, dividing the array into two equally sized sub-arrays. The sub-array is recursively divided until it fits into the scratch-pad memory. When a base case returns a sorted array, the runtime program fetches the sorted sub-arrays and merges them into the final sorted array.

The *Closest Pair* micro-benchmark finds the pair of points with the smallest Euclidean distance between them, recursively dividing the point set into two subsets. The base case finds the closest pair points in each smallest subset. When combining the result, the application also determines if there is any pairing across the two different subsets.

The *K-means* micro-benchmark classifies a given data set through a number of clusters. Specifically, this application handles two-dimensional data set. The computation of Euclidean distance between a point and centroids can be distributed to green cores. White cores are in charge of finding the new centroids by taking the mean of all the data points in each cluster.

5.1.2 Scalability

The key goal of the scalability study was to observe the overheads. If dynamic translation and dispatching of the tasks dominated the execution time, then the proposed approach would need to be revisited.

We ran the five micro-benchmarks on the green-white architecture hardware prototype and measured the wall clock time for each micro-benchmark. Figure 5.1 shows the scalability of each micro-benchmark as compared to an ideal speedup. All five micro-benchmarks demonstrate performance improvement over the range of available green cores. In particular, Strassen and block matrix multiplication benefit the most from six green cores and achieved more than $4\times$ speedup. Strassen spawns $2,401$ parallel tasks onto the green core while block matrix spawns 256 parallel tasks. When all six green cores are used, the Strassen and block matrix spent 1.5% of the runtime on combining computation results into final arrays. The percentage of sequential execution in runtime obviously had an impact on speedup. Compared to Strassen and block matrix multiplication, the performance of merge-sort grows slower and appears to level off around 3× speedup. Although merge-sort spawns a large number of tasks (2,048), a detailed look at the algorithm reveals that when using all six green cores, 41.2% of the run time is spent on the white core, assembling solutions from sorted sub-arrays. This suggests that it would be advantageous if some of the assembly process could be implemented with parallel tasks and kept on the green cores. The closest-pair micro-benchmark benefits the least from adding more green cores since its sequential execution takes 97.0% of total runtime. This micro-benchmark appears to be a poor candidate for this architecture, or it may need more work to exploit parallelism.

Figure 5.2 and Figure 5.3 show the performance impact of adding the second bank of scratch-pad memory to the system respectively with one green core utilized and six green cores utilized. When only one green core is utilized, all five micro-benchmarks demonstrate performance improvement after the second bank of scratch-pad memory is added. In particular, merge-sort and k-means gain about a 40% performance improvement. When six green cores are utilized, all micro-benchmarks except closest-pair micro-benchmark benefit from the second bank of scratch-pad memory.

5.1.3 Resilience

5.1.3.1 Fault Injection Mechanism

We have implemented five micro-benchmarks on the prototype green-white architecture and used fault injection mechanism to test fault recovery mechanism provided by the runtime system. Our fault model includes two scenarios: (a) an unreliable execution affects the output of instructions, and (b) the unreliable execution affects the instructions themselves. We use different approaches to emulate faults for the two scenarios.

For the first scenario, we flipped a bit in the input arguments of a task. The fault injector was actually implemented in the dynamic translation layer. When a task is

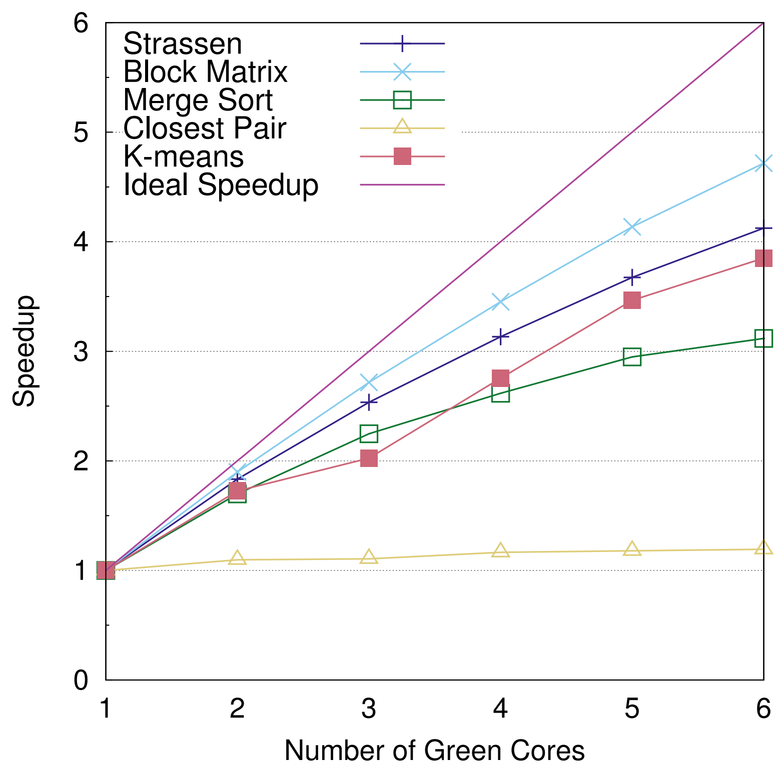

Figure 5.1: Speedup on *green-white* architecture. Strassen multiplies two 64×64 integer matrices with the base case set to 4×4. Block matrix micro-benchmark multiplies two 32×32 integer matrices with a 32×2 base case size. Merge-sort sorts a 8192-element integer array with a base case set to 4. Closest-pair finds the pair of points from a set of 1024 two-dimensional points with a base case set to 4 points. K-means clusters 1024 two-dimensional points into 4 groups with a base case set to 4 points.

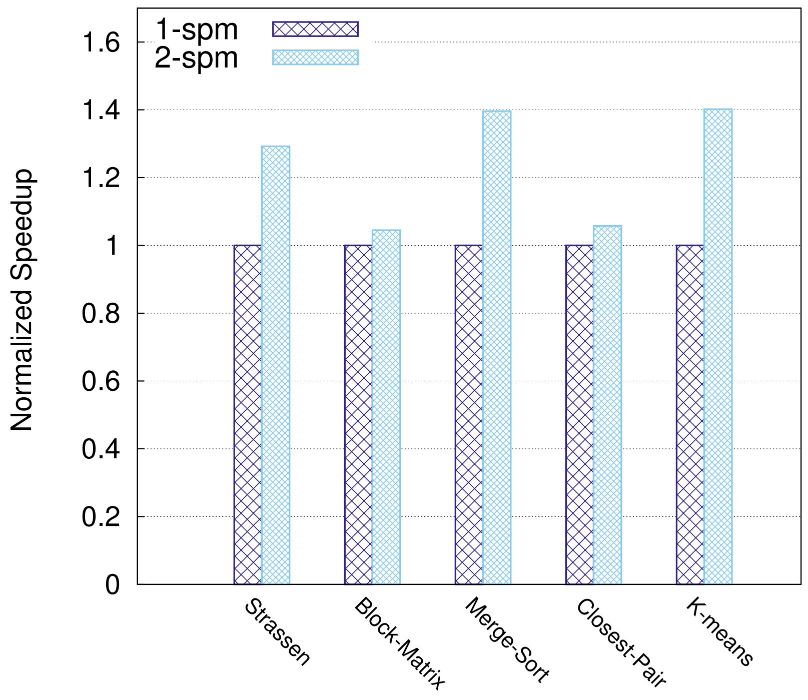

Figure 5.2: The performance impact of scratch-pad memory on *green-white* architecture. Only one green core is utilized. Strassen multiplies two 64×64 integer matrices with the base case set to 4×4. Block matrix micro-benchmarks multiplies two 32×32 integer matrices with 32×2 base case size. Merge-sort sorts a 8192-element integer array with a base case set to 4. Closest-pair finds the pair of points from a set of 1024 two-dimensional points with a base case set to 4 points. K-means clusters 1024 two-dimensional points into 4 groups with a base case set to 4 points.

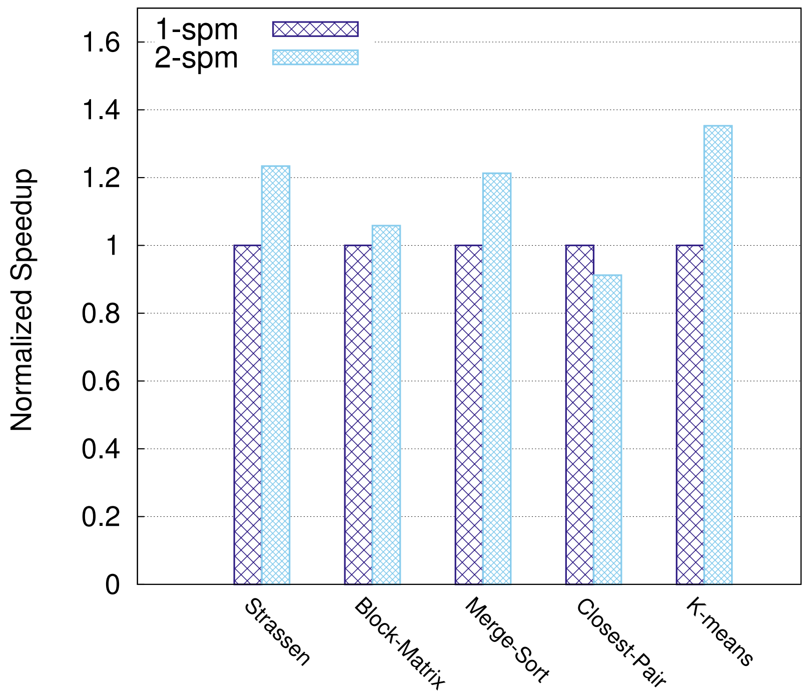

Figure 5.3: The performance impact of scratch-pad memory on *green-white* architecture. Six green cores are utilized. Strassen multiplies two 64×64 integer matrices with the base case set to 4×4. Block matrix micro-benchmarks multiplies two 32×32 integer matrices with 32×2 base case size. Merge-sort sorts a 8192-element integer array with a base case set to 4. Closest-pair finds the pair of points from a set of 1024 two-dimensional points with a base case set to 4 points. K-means clusters 1024 two-dimensional points into 4 groups with a base case set to 4 points.

migrated from the white core to the green core, the input arguments are represented by a C byte array. A bit-flip in the C byte array associated to input argument emulates *data corruption*. Such faults are silent unless a voting mechanism is used to detect the injected fault or correct the error in the dispatcher, and the runtime system generates an error message when it detected faulty results through voting.

The second scenario is more complicated, because such fault can be benign, masked by the hardware protection, or crash the program. We assume that our green core does not have hardware protection and such faults always result in program crashing. In other words, a virtual machine running on the green core crashes due to this type of fault. Here the focus is on recovery mechanisms; a complete fault detection scheme for this type of fault is beyond the scope of this paper. When the fault is injected, we assume that the executable of the virtual machine is corrupted and requires a reload. The function and input arguments are not reloaded since the cost is around 10% of loading the virtual machine executable. The hardware is reset to a fresh state. Following a re-computation, the results are then sent to the voters.

5.1.3.2 Results

The goal of the proposed runtime is to add system resilience, even if the underlying hardware is unreliable. To demonstrate this, two key performance questions were investigated. Scalability of the micro-benchmarks was examined to see the impact of the runtime system overhead. Resilience characteristics were then explored to see how redundancy choices impact the time to find the correct solution of each benchmark. A typical execution for these micro-benchmarks runs for about $30-60$ seconds. In many large cluster environments, this is far too short of a run; however, with a hardware emulator, it is very easy to get precise measurements where execution times represent billions of clock cycles. There is no focus on absolute execution times. While the hardware emulator is very fast when compared to a software simulation, one cannot compare a 50 MHz processor core to a multi-GHz CMOS processor. To understand the role of these experiments, it is important to refer to the architectural assumptions explained in Chapter 2.

PyDac offers the ability to implement resilient computation of tasks scheduled on the green cores. This is user-defined and entirely transparent, meaning that the application does not need to be modified in any way to compute resiliently. This greatly reduces the programmer overhead for developing resilient applications and, particularly on lower-reliability hardware, can be beneficial.

This is implemented in PyDac with two different degrees of Redundant Multi-Threading (RMT): Dual Modular Redundancy (DMR) or Triple Modular Redundancy (TMR). TMR is perhaps the more familiar concept where tasks are triplicated and processed independently by different green cores. Results are compared by the white core and voted on such that any two that produce the same answer "win." In the extremely rare case when none of the three results are the same, the entire set of three tasks are recomputed (rolled-back) and tried again.

DMR is a simpler form of the above voting technique where each task is duplicated and voted on. In this case, if the voting does not match then both are recomputed. DMR generally performs better in situations where systems are somewhat unreliable. TMR works well in systems with real-time sensitivities (e.g., a roll-back might cause too much delay in a result) as well as systems that are highly unreliable.

To demonstrate this capability in PyDac, micro-benchmarks were executed in DMR, TMR, and without any redundancy. These results are presented in Figure 5.4. Then, a single bit-flip fault was injected into each application during a DMR and TMR run. In Figure 5.4, DMR-0 and TMR-0 indicate runs without any faults and demonstrate the overhead imposed by this feature in PyDac. The DMR-1 and TMR-1 bars depict the runtime when a single fault was injected. It is important to realize that in all of these runs, the micro-benchmarks produced the correct answer; even in the presence of a soft error.

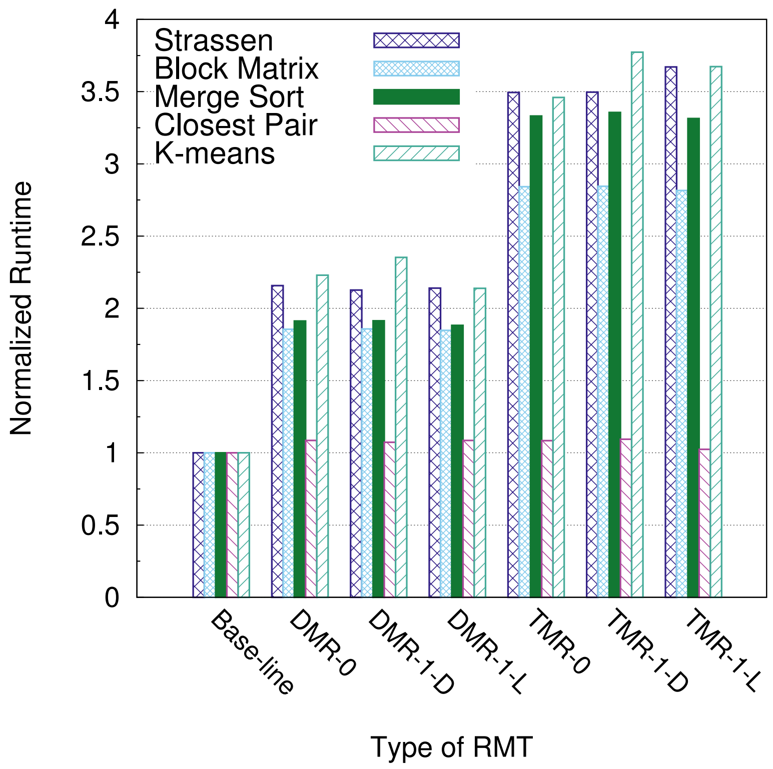

Figure 5.4: Normalized runtime on *green-white* architecture. All six green cores are utilized. Base-line are tests without RMT support. DMR-0 and TMR-0 are tests that enable RMT in a fault-free environment. DMR-1 and TMR-1 are tests that enable RMT while the fault injector injects a single soft error.

As can be seen from Figure 5.4, the overhead imposed by this technique is largely dependent on the ratios between white and green core computation time. While clearly this technique has performance implications as it creates $2\times$ (DMR) and $3\times$ (TMR) the number of green tasks, the inclusion of it in PyDac eases its adoption for programmers. Strassen's algorithm created over $2,400$ green core tasks in the base-line approach. Clearly this number grows to $4,800$ and $7,200$ in the cases of DMR and TMR. One would expect considerably more overhead for scheduling these tasks on only six green cores but PyDac handles all of this easily and with low overhead. As we continue to scale to larger FPGAs, we expect to be able to deploy more green cores and this large number of tasks would be more easily processed.

Notice in Figure 5.4 that the closest-pair algorithm does not incur much overhead from redundancy. This is because, this algorithm runs almost entirely on the white cores (i.e., not well parallelized). As such, there is very little overhead imposed by redundant green tasks.

One thing to consider is that with the ease of implementing resilient computation under PyDac, one could target hardware that is considerably less reliable than conventional architectures. These less-reliable architectures often come with major improvements in power and/or performance. This approach focuses on time to completion rather than instructions per second. Furthermore, PyDac could be a lot more sophisticated with its use of RMT if it had access to probing information about the expected fault rates of the machine. For instance, it might run in base-line mode entirely and only start implementing RMT on some of the cores if they identified hardware problems. Implementing this kind of approach outside of a task-based programming model is costly and complex for a programmer, and PyDac supplies this through a simple switch.

5.2 Evaluating Programming Productivity by Case Study

The programming productivity of the proposed programming model is evaluated in this section through case studies. Specifically, three numerically intensive algorithms coded in PyDac illustrate the programming productivity. A commodity SMP server was used to verify the micro-benchmarks presented in this section. They all stand up to the double-precision floating point tests. (Due to the lack of floating point support on green core in the green-white hardware prototype, symmetrical eigenvalue decomposition and FFT were not tested on the green-white hardware prototype. Strassen's algorithm was tested only with integer inputs on the hardware prototype.) The primary goal of these micro-benchmarks is not to compare absolute performance between a commodity server and the green-white hardware prototype; rather, the goal is to make a subjective argument and rely on the reader's judgment to assert our point. The detailed analysis for the micro-benchmarks is presented at the end of this section.

5.2.1 Strassn's Algorithm

As illustrated in Figure 5.5, Strassen's algorithm [107] partitions a square matrix into four equally sized block matrices. These block matrices are further divided in a recursive manner until they become small enough. The time complexity of the Strassen's algorithm is better than conventional triple-loop matrix multiplication for large matrices; however, it is the large degree of parallelism that makes this operation attractive.

Figure 5.6 illustrates the recursive implementation under PyDac. We used NumPy [100] version 1.6.1—a package for scientific computing with Python—to verify that the implementation produces correct results. In particular, the input matrices contain randomly generated double-precision floating-point numbers with the standard normal distribution. The result of our implementation is equal to the result from the NumPy package up to at least the eleventh decimal.

```
Data: Let $A$ and $B$ be two $2^n \times 2^n$ matrices
Result: The algorithm computes the matrix multiplication of $C = A \times B$
if Matrices A, B are smaller enough then
    Directly compute $C = A \times B$;
    return Matrix $C$;
else
    Partition $A = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix}$ and $B = \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix}$, where each sub-matrix is $2^{n-1} \times 2^{n-1}$ ;
    Reorganize sub-matrices in a way that:
        $Q_0 = (A_{00} + A_{11}) \times (B_{00} + B_{11})$
        $Q_1 = (A_{10} + A_{11}) \times B_{00}$
        $Q_2 = A_{00} \times (B_{01} - B_{11})$
        $Q_3 = A_{11} \times (-B_{00} + B_{10})$
        $Q_4 = (A_{00} + A_{01}) \times B_{11}$
        $Q_5 = (-A_{00} + A_{10}) \times (B_{00} + B_{01})$
        $Q_6 = (A_{01} - A_{11}) \times (B_{10} + B_{11})$
    Call this algorithm with $Q_0, Q_1, Q_2, Q_3, Q_4, Q_5, Q_6$ as output;
    Form matrix $C = \begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix}$ in a way that:
        $C_{00} = Q_0 + Q_3 - Q_4 + Q_6$
        $C_{10} = Q_1 + Q_3$
        $C_{01} = Q_2 + Q_4$
        $C_{11} = Q_0 + Q_2 - Q_1 + Q_5$
    return Matrix $C$;
end
```

Figure 5.5: Strassen's algorithm

5.2.2 Symmetrical Tridiagonal Eigenvalue Decomposition

The symmetric tridiagonal eigenvalue decomposition [108] converts a symmetrical tridiagonal matrix into two half-sized tridiagonal matrices plus a rank-one modification. Each half-sized tridiagonal matrix could be recursively partitioned until the sub-matrix is small enough for a direct solution. The base case returns the eigenvalues and eigenvectors of input sub-matrix. At the *merge* stage, the algorithm combines returned eigenvalues and eigenvectors into a temporary matrix, which is a *diagonal* matrix plus a rank-one update. The temporary matrix is useful in that it simplifies the process of finding the final eigenvalues. The final eigenvectors also take advantage

```
def strassen_base(matrix_a, matrix_b):
    """nxn matrix multiplication
    """
    ret_mat = numpy.dot(matrix_a, matrix_b)
    return ret_mat

def strassen_div(matrix):
    # Divide array into two equal arrays by row
    matrix_1, matrix_2 = numpy.split(matrix, 2, 0)
    # Divide sub arrays into four equal arrays by column
    submat_11, submat_12 = numpy.split(matrix_1, 2, 1)
    submat_21, submat_22 = numpy.split(matrix_2, 2, 1)
    return submat_11, submat_12, submat_21, submat_22

def strassen_concat(m11, m12, m21, m22):
    m1 = numpy.concatenate((m11, m12), 1)
    m2 = numpy.concatenate((m21, m22), 1)
    m = numpy.concatenate((m1, m2), 0)
    return m

def strassen_op(matrix_a, matrix_b, bc_size):
    """Data decomposition in functional style.
    """
    if (matrix_a.shape[0] == bc_size):
        result = ship(strassen_base, [matrix_a, matrix_b])
    else:
        a_submatx_11, a_submatx_12, a_submatx_21, a_submatx_22 =
            strassen_div(matrix_a)
        b_submatx_11, b_submatx_12, b_submatx_21, b_submatx_22 =
            strassen_div(matrix_b)

        subpb = [((a_submatx_11 + a_submatx_22), (b_submatx_11 +
            b_submatx_22), bc_size),\
                ((a_submatx_21 + a_submatx_22), b_submatx_11, bc_size),\
                (a_submatx_11, (b_submatx_12 - b_submatx_22), bc_size),\
                (a_submatx_22, (b_submatx_21 - b_submatx_11), bc_size),\
                ((a_submatx_11 + a_submatx_12), b_submatx_22, bc_size),\
                ((a_submatx_21 - a_submatx_11), (b_submatx_11 +
                    b_submatx_12), bc_size),\
                ((a_submatx_12 - a_submatx_22), (b_submatx_21 +
                    b_submatx_22), bc_size)]

        results = divide(strassen_op, subpb)

        c11 = results[0] + results[3] + results[6] - results[4]
        c12 = results[2] + results[4]
        c21 = results[1] + results[3]
        c22 = results[0] + results[2] + results[5] - results[1]
        result = strassen_concat(c11, c12, c21, c22)
    return result
```

Figure 5.6: Strassen's algorithm implemented in PyDac

of the temporary matrix indirectly. Figure 5.7 illustrates this algorithm in details.

```
Data: Let $T$ be a square ($N \times N$) symmetric tridiagonal matrix.
Result: The algorithm computes the eigenvalue decomposition of $T = Q\Lambda Q^T$,
        where the diagonal $\Lambda$ is the square ($N \times N$) matrix of eigenvalues and
        $Q$ is orthogonal.
if T is smaller enough then
    Directly compute $T = Q\Lambda Q^T$;
    return ($\Lambda$, $Q$);
else
    Partition $T = \begin{bmatrix} T_1 & 0 \\ 0 & T_2 \end{bmatrix} + \rho \boldsymbol{u}\boldsymbol{u}^T$, where $\rho \boldsymbol{u}\boldsymbol{u}^T$ is a rank-one modification.;
    Call this algorithm with $T_1$ as input and $\Lambda_1$, $Q_1$ as output;
    Call this algorithm with $T_2$ as input and $\Lambda_2$, $Q_2$ as output;
    Form $D + \rho \boldsymbol{v}\boldsymbol{v}^T$ from $\Lambda_1$, $\Lambda_2$, $Q_1$, $Q_2$, where $\rho \boldsymbol{v}\boldsymbol{v}^T$ is a rank-one update;
    Find the eigenvalues $\Lambda$ and the eigenvectors $Q'$ of $D + \rho \boldsymbol{v}\boldsymbol{v}^T$;
    Form $Q = \begin{bmatrix} Q_1 & 0 \\ 0 & Q_2 \end{bmatrix} Q'$ which are the eigenvectors of $T$;
    return ($\Lambda$, $Q$);
end
```

Figure 5.7: A divide-and-conquer algorithm for symmetric tridiagonal eigenvalue problem

Figure 5.8 illustrates the recursive implementation under PyDac. NumPy version 1.6.1 was used to verify the implementation producing correct results. In particular, the input symmetrical tridiagonal matrix contains randomly generated double-precision floating-point numbers. The result of our implementation is equal to the one computed by the routine in the NumPy package (i.e., *numpy.linalg.eig()*) up to at least the eleventh decimal.

5.2.3 Recursive FFT

As illustrated in Figure 5.9, the recursive fast Fourier transformation (FFT) algorithm is also known as the Cooley-Tukey algorithm [86]. To demonstrate the programming model, we implement the simplest form—radix-2 decimation-in-time FFT—of the Cooley-Tukey algorithm. With each recursive stage, inputs are partitioned into two equal-sized groups (i.e., odd-indexed inputs and even-indexed inputs). The algo-

```python
def symm_eigen_base(sub_T):
    """Base case for symmetric tridiagonal eigenproblem.
    """
    eigen, Q = numpy.linalg.eig(sub_T)
    return eigen, Q

def symm_eigen_partition(T):
    T_1, T_2 = numpy.split(T, 2, 0)
    T_11, T_12 = numpy.split(T_1, 2, 1)
    T_21, T_22 = numpy.split(T_2, 2, 1)
    rho = T_12.item(T_12.shape[0]-1,0)
    T_11.itemset((T_11.shape[0]-1, T_11.shape[1]-1), T_11.item(T_11.
        shape[0]-1, T_11.shape[1]-1) - rho)
    T_22.itemset((0, 0), T_22.item(0, 0) - rho)
    return T_11, T_22, rho

def form_dia_w_rank_one(lambda_1, lambda_2, rho, v_trans):
    dim = lambda_1.shape[0]
    D = numpy.diag(numpy.concatenate((lambda_1, lambda_2), 1))
    dia_w_rank_one = D + rho*v_trans.T*v_trans
    return dia_w_rank_one

def product(Q1, Q2, Q3):
    dim = Q1.shape[0]
    Q_upper_half = numpy.concatenate((Q1, numpy.zeros((dim, dim))), 1)
    Q_bottom_half = numpy.concatenate((numpy.zeros((dim, dim)), Q2), 1)
    Q = numpy.concatenate((Q_upper_half, Q_bottom_half), 0)
    return Q * Q3

def symm_eigen_op(T, bc_size):
    """Data decomposition in functional style
    """
    if (T.shape[0] == bc_size):
        eigen, Q = ship(symm_eigen_base, [T])
    else:
        T1, T2, rho = symm_eigen_partition(T)
        subpb = [(T1, bc_size), (T2, bc_size)]
        results = divide(symm_eigen_op, subpb)
        lambda_1, Q1 = results[0]
        lambda_2, Q2 = results[1]
        dim = Q1.shape[0]
        v_trans = numpy.concatenate((Q1[dim-1], Q2[0]), 1)
        dia_w_rank_one = form_dia_w_rank_one(lambda_1, lambda_2, rho,
            v_trans)
        eigen, Q3 = numpy.linalg.eig(dia_w_rank_one)
        Q = product(Q1, Q2, Q3)

    return eigen, Q
```

Figure 5.8: Symmetrical tridiagonal eigenvalue decomposition algorithm implemented in PyDac

rithm recursively decomposes the problem until it reaches base cases.

```
Data: Let $x$ be an array $(x_0, x_1, ..., x_{N-1})$ where N is even
Result: The algorithm computes the discrete Fourier transform of
        $X_k = \sum_{n=0}^{N-1} x_n e^{-i2\pi kn/N}$, where $k$ = 0, 1, ..., N-1
if Input array x is smaller enough then
    Directly compute $X_k = \sum_{n=0}^{N-1} x_n e^{-i2\pi kn/N}$;
    return $X_k$;
else
    Partition $x$ into even-indexed $x_{2m}$ and odd-indexed $x_{2m+1}$, where $m$ = 0, 1,
    ..., N/2 -1;
    Call this algorithm with $x_{2m}$ and $x_{2m+1}$ as input and $E$ and $O$ as output;
    for $k = 0$ to $N - 1$ do
        $m = k \bmod N/2$;
        $X_k = E_m + e^{-i2\pi k/N} O_m$;
    end
    return $X_k$;
end
```

Figure 5.9: A Divide-and-Conquer Algorithm for Radix-2 FFT

Figure 5.10 illustrates the recursive implementation under PyDac. The implementation is derived from a code snippet used to generate hardware description language (HDL) version [109]. We use two methods to verify the implementation. Specifically, we use the routine (e.g., *numpy.fft()* provided by NumPy version 1.6.1) and the direct definition of DFT to verify the implementation produce correct results. The root-mean-square (RMS) error does not exceed 10×10^{-12}.

5.2.4 Analysis

Programmer productivity, "ease of use," and other similar traits that make a programming model desirable are notoriously difficult to quantify. Software engineering studies in the 1970s and 1980s tried to find surrogate metrics, such as Lines-of-Code, fog-index, and others, but largely failed. This is because individual programmers' productivity has an enormous variance. The best programmers can be 10× more productive than the worst, and to control for this variance in the population, experi-

```
def rFFT_base(x, N):
    """ Base case of recursive FFT
    """
    y = [1.0 + 1.0j]*N
    y = r_[y]
    for n in range(N):
        wsum = 0 + 0j;
        for k in range(N):
            wsum = wsum + (cos(2*pi*k*n/N) - (1.0j * sin(2*pi*k*n/N)))*x[k]

        y[n] = wsum
    return y

def rFFT_op(x):
    """Data decomposition of recursive FFT
    """

    n = len(x)

    if (n == 2):
        F = ship(rFFT_base, [x, n])
    else:
        w = getTwiddle(n)
        m = n/2;

        X = ones(m, float)*1j
        Y = ones(m, float)*1j

        for k in range(m):
            X[k] = x[2*k]
            Y[k] = x[2*k + 1]

        subpb = [[X], [Y]]
        results = divide(rFFT_op, subpb)

        X = results[0]
        Y = results[1]

        F = ones(n, float)*1j
        for k in range(n):
            i = (k%m)
            F[k] = X[i] + w[k] * Y[i]

    return F
```

Figure 5.10: Recursive FFT implemented in PyDac

ments require very large sample sizes. These type of human subject experiments are logistically challenging and very expensive. Instead of using quantitative experimental data, we will make a subjective argument and rely on the reader's judgment to assert our third point.

The proposed programming model is rooted in Python, a modern high-level programming language. Python was designed to be a Rapid Application Development (RAD) language that incorporates features from the imperative, object-oriented, and functional programming paradigms. From a programmer's perspective, it presents as a scripting language although, technically, it is compiled on-the-fly into bytecodes that are executed by a Virtual Machine. Python's syntax is succinct, and simple statements are very similar to the wildly successful family of C-based programming languages. With careful implementation of global and local namespaces, Python is able to elegantly incorporate object-oriented and functional features with very few additional syntactical flourishes.

Python is reputed to be a "high productivity" language, as evidenced by its widespread acceptance [100, 101] and the numerous contemplative writings [110, 111, 112]. Python language has a significant user base with a 2010 estimation of at least one million Python users in the world [110]. Such a large user base translates into a large collection of freely available Python modules. Not only is the number of free Python modules significant, but also is the number application domains for Python. In fact, the widespread use of Python is shown by the applications domains including graphical user interfaces (GUI), system programming, internet scripting, database programming, component integration, numeric and scientific programming, and more. Python is even applied in embedded system domain [105]. As Python has grown in popularity, it focuses on code quality and readability with no compromise. In fact, most Python programmers today write their code in pure Python and only "a small handful of developers integrate external libraries for the majority to leverage in their

Python code" [110]. Thus, we take the statement "Python is a highly productive language," as a given for the rest of the argument.

Starting with the key assumption that every application consists of mix of *necessarily* serial operations and *potentially* parallel operations, the proposed PyDac programming model is based on two core principles. To fully utilize a chip's potential for parallelism, the programming model encourages potentially parallel operations to be organized in a divide-and-conquer (D&C) style of computation. The second principle is, at first notice, slightly more restrictive: The base case of the D&C computation must be referentially transparent. Referentially transparent, in practical terms, means that given the same input, the base case must always produce the output. In other words, the base case cannot change the global state of the machine on its own. (Its result can suggest a change to the global state of the machine, but it cannot make the change itself.)

The D&C style is recursive and the base case is required to be referentially transparent, which suggests that computational scientists should stop writing imperatively and learn to write programs in functional paradigm languages. However, this is not true. Recursion, and D&C in particular, is not strictly within the domain of functional languages. In fact, it can be used—with great effect—in the paradigm of imperative languages. If D&C seems foreign or mystical, then consider these facts: (a) every CS student learns it in their sophomore year [85], (b) numerical analysis books give D&C algorithms dating back decades [113], and (c) when computation was done with paper and pencil, D&C was used to simplify the computation. D&C is not a difficult or obscure concept! It has been around for much, much longer than electro-mechanical computing and, in some ways, has been marginalized by the limitations of computing devices in the 1950s and 1960s.

The second (and most potentially controversial) restriction of the proposed programming model is that the base case must be referentially transparent. We would argue that most high-end computer (1000s+ processors) programmers use message-passing runtime systems such as message-passing interface (MPI). Even if it is not immediately obvious, these systems have coarse-grain, *referentially transparent* tasks. That is, if a task has a global variable `x`, then it is unique and independent of every other tasks' global variable `x`. To create a global consensus of the value of `x`, the programmer must explicitly resolve all of the individual views by resolving them to a single value in the application. Typically, this is accomplished with either a *broadcast* or an *all_reduce*. Regardless, the onus is on the application programmer. A more recent example is Intel Concurrent Collections (CnC), which directly imposes rules to ease the difficulty of parallel programming. In CnC, not only are dependencies explicitly stated, but computation (called step collection in CnC) may not reference any global values. Such effort is to eliminate race conditions in parallel programming. The proposed programming model imposes the same responsibility on the programmer albeit with a different mechanism (the assembly of the sub-problem solutions). However, it is worth noting that every D&C algorithm we have investigated has naturally exhibited the referentially transparent property; perhaps it is the case that historical mechanical computation have inadvertently restricted us to patterns with unnecessary dependence.

However, the programming model presents a few benefits. First, because the code is compiled into portable byte-code, the programming model offers wide portability. Secondly, the D&C strategy offers an exponential growth in concurrent tasks. Thirdly, the co-designed memory subsystem is re-organized to support the programming model.

PyDac is source code compatible with all known chip architectures in that if it runs Python, our source code will run to completion with the correct answer; the only exception being in non-referentially transparent base cases. The portability will play a significant role since we do not know what architecture the future will bring. For example, an architecture that combines fast and slow processors will be a challenge to some existing programming models that assume symmetric processors. The *de facto* programming model in high-performance computing domain—MPI—currently assumes a chip architecture with symmetric processors. With the emergence of chip multiprocessor, hybrid model called "MPI+X" is expected to better utilize hierarchical feature of hardware (which is still symmetric). For example, "MPI+OpenMP" [59], which combines the library extension approach and the pseudo-comment directive approach, builds distributed memory programming model on top of shared-memory programming model. This hybrid model still requires programmers to invest a great amount of coding effort to utilize cores effectively [60]. In contrast, PyDac runs on both with symmetric processors and with an architecture that combines fast and slow processors. Cross-platform programming—between symmetric processors and with an architecture that combines fast and slow processors without ports—is feasible. Hence, the programming model does not require programmers to invest coding effort to effectively utilize the hardware. With even more revolutionary chip architecture in the future, the portability offered by our programming model will benefit programmers.

The programming model meets the challenge of an exponential growth in cores by effecting exponential growth in concurrent tasks. As is described in section 2.2, the community must contend with many-core architectures that limit the performance of single-threaded applications and instead force programmers to invest in parallelization techniques to increase the performance of their algorithms. PyDac allows for simple parallelization by breaking up problems into a very large number of small problems, each of which results in a thread that can be scheduled on an independent processor core.

In the green-white architecture, the memory subsystem is also re-organized to support the programming model. The programming model frees programmers from the intellectual bottleneck—the *von Neumann bottleneck*, as called by John Backus [83]. The bottleneck arises from the task of exchanging the contents between the processing unit and the storage. Such data traffic becomes a bottleneck when programmers are tied to work-at-a-time thinking instead of being encouraged to think in terms of the larger conceptual units of the task at hand. Object oriented languages, such as C++, are a good example that encourages programmers to think in terms of the larger conceptual units. PyDac also follows this concepts to eliminate the intellectual bottleneck. In PyDac, data exchanged between high level and low level is large conceptual units, such as array or matrix. The memory subsystem in the green-white architecture does the heavy lifting in a way that processing units (i.e., green cores) can transmit an object instead of a single word to the storage.

CHAPTER 6: CONCLUSION

Computer architecture has been evolving for the past few decades in order to take advantage of advancing technology. Based on technology trends, our assumption about computer architecture design over the next decade is three fold: (a) heterogeneous architecture that combines faster, bigger, and more complex cores and slower, smaller, and simpler cores delivers better performance than homogeneous architecture, (b) cores will become more unreliable, and (c) memory subsystems will become more active in managing transactions. Porting conventional software stack to such architecture will be a challenging task, because conventional software design is deeply rooted in a different set of computer architecture assumptions. However, we believe that programmers are willing to adapt to a new programming paradigm once the benefits outweigh the costs.

One such envisioned chip architecture is called green-white architecture. Green-white architecture treats main memory as a collection of named, write-once, variable-sized blocks of data that are transferred in and out of the green cores by DMA engines. Each green core has multiple banks of scratch-pad memory, which is byte-addressable. The architecture also includes one or more fast, complex cores (the white cores). Conceptually, these cores incorporate the latest advances in single-thread performance and incorporate techniques (higher power, protection, hardened) to increase reliability. The on-chip network connects the green cores to a memory subsystem, providing direct access to the blocks of write-once memory. The two memory subsystems share the (off-chip) DRAM memory resources through a multi-ported memory controller.

The green-white architecture is co-designed with a programming framework called

PyDac. The primary goal of the PyDac programming framework is to make it possible to write programs that generate a very large number of parallel tasks without a great deal of programming effort. Specifically, a programmer that wants to use this programming model needs to learn the divide-and-conquer strategy. The runtime system of the PyDac programming framework has three responsibilities: (a) it abstracts the underlying green-white architecture into distributed virtual machines, (b) it implements the core behavior of the PyDac programming model, and (c) it manages resources to meet requirements, such as resilience.

To explore this research area, we have designed and developed many artifacts. First, a hardware prototype of green-white architecture was emulated on an FPGA device. Secondly, a prototype of PyDac programming model was developed. Python was chosen for implementation because it is considered an easy language to learn, it supports both the functional and imperative paradigms, and it has popular modules to support scientific applications. Thirdly, we constructed a runtime system based on distributed Python virtual machines. Python modules were also developed for the programming interface, task scheduling, and fault-recovery. Lastly, the PyDac programming framework was used to developed a set of micro-benchmarks that were then tested on the hardware prototype.

Five micro-benchmarks were run on the hardware prototype and wall clock time was measured for each micro-benchmark. All five micro-benchmarks demonstrated performance improvement over the range of green cores available. In particular, Strassen and block matrix multiplication benefit the most from six green cores and achieved more than $4\times$ speedup. Strassen spawns $2,401$ parallel tasks onto the green core, while block matrix spawns 256 parallel tasks. The percentage of sequential execution in runtime obviously has an impact on speedup. The closest-pair micro-benchmark benefits the least from adding more green cores since its sequential execution takes 97.0% of total runtime. This micro-benchmark appears to be a poor candidate for this architecture or may need more work to exploit parallelism. Results also suggest that it would be advantageous if some of the assembly process could be implemented with parallel tasks and kept on the green cores.

Fault injection mechanism was used to test fault recovery mechanism provided by the runtime system. Our fault model included two scenarios: (a) an unreliable execution affecting the output of instructions, and (b) the unreliable execution affecting the instructions themselves. Different approaches were used to emulate faults for the two scenarios. The goal of the proposed runtime was to add system resilience, even if the underlying hardware is unreliable. To demonstrate this, two key performance questions were investigated. First, the scalability of the micro-benchmarks was examined to see the impact of the runtime system overhead. Secondly, resilience characteristics were explored to see how choices for redundancy impact the time to correct solution of each benchmark. PyDac offers the ability to implement resilient computation of tasks scheduled on the green cores. This is user-defined and entirely transparent, meaning that the application does not need to be modified in any way to compute resiliently. This greatly reduces programming overhead for developing resilient applications and, in particular on lower reliability hardware, can be beneficial. To demonstrate this capability in PyDac, micro-benchmarks were executed in DMR, TMR, and without any redundancy. Micro-benchmarks produced the correct answer in all of these runs, even in the presence of a soft error.

All of these features are uncommon in a conventional monolithic runtime. The experiments and results indicate that the proposed design is not only a viable solution for green-white architecture, but also it provides performance and resilience without compromising the programming productivity. This research is just a beginning for exploring a new era of hardware/software codesign. These ideas will be taken into our next phases of research and development.

CHAPTER 7: FUTURE WORK

In the future, we plan to switch to more mature processors than ARMv2a. Our GCC compiler (version 4.4.6) for ARM generates floating-point instructions incompatible with ARMv2a. With more mature processors, we will be able to port more micro-benchmarks to green-white architecture. If the new processor provides hardware multithreading, we would like to explore how to take advantage of the hardware multithreading in our programming model. Furthermore, we would like to port our design to a multiple-FPGA platform that facilitates as many as 128 green cores. The new platform will also allow us to integrate the active memory subsystem, developed by our colleagues, into the green-white architecture. The integration will also eliminate the software module that emulates the active memory subsystem. Hence, the runtime overhead on the white core will be further reduced. Moreover, a larger FPGA device will grant us more resources, especially on-chip memory, to explore a larger size of scratch-pad memories and a larger number of memory banks in the green-white architecture. With future chips that promise larger on-chip memory, we will be able to increase sub-matrix size and conduct larger-scale tests.

The PyDac programming model is based on divide-and-conquer strategy. Other paradigms, such as streaming or pipelining, are also worth exploring. One approach to do streaming on the green-white architecture is to run different computation kernels on green cores. The programmers will need to define the computation kernels and specify the data flow. New language constructs or libraries may need to be developed for this purpose.

When developing base cases, programmers are not allowed to define a function outside the scope of base case and then call that function in the base case. In the prototype of PyDac programming model, programmers can make a call to only built-in functions. This can be improved by the dynamic translation layer detecting the external function and embedding that function into the naming space of the base case computation.

In the prototype, the criterion for a task to run on the green core is that it fits into the scratch-pad memory. With multiple banks of scratch-pad memories on one green core, combining multiple physical scratch-pad memories into one "virtual" scratch-pad memory is definitely possible. This virtual scratch-pad memory may free programmers from thinking about the physical hardware's size limitations.

One thing to consider is that with the ease of implementing resilient computation under PyDac, one could target hardware that is considerably less reliable than conventional architectures. PyDac could be a lot more sophisticated about its use of RMT if it had access to probing information about the expected fault rates of the machine. For instance, it might run in base-line mode entirely and only start implementing RMT on some of the cores if they identified hardware problems. Also, we are exploring implementing dials and probes on the green cores that allow us to inject hardware faults and report the presence of these faults to PyDac. This feature would allow us to dynamically adapt the resilience of the tasks. Finally, using idle cycles on the white core to replicate work being performed on green cores is currently under consideration. This approach provides more resiliency while also allowing the application to "catch up" when faults are observed in real time constrained applications.

REFERENCES

[1] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller *et al.*, "Exascale computing study: Technology challenges in achieving exascale systems," *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep*, 2008.

[2] G. E. Moore *et al.*, "Cramming more components onto integrated circuits," 1965.

[3] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc, "Design of ion-implanted mosfet's with very small physical dimensions," *Solid-State Circuits, IEEE Journal of*, vol. 9, no. 5, pp. 256–268, 1974.

[4] H. Kaul, M. Anders, S. Hsu, A. Agarwal, R. Krishnamurthy, and S. Borkar, "Near-threshold voltage (ntv) design: opportunities and challenges," in *Proceedings of the 49th Annual Design Automation Conference.* ACM, 2012, pp. 1153–1158.

[5] J. M. Tendler, J. S. Dodson, J. Fields, H. Le, and B. Sinharoy, "Power4 system microarchitecture," *IBM Journal of Research and Development*, vol. 46, no. 1, pp. 5–25, 2002.

[6] S. Gochman, A. Mendelson, A. Naveh, and E. Rotem, "Introduction to intel core duo processor architecture." *Intel Technology Journal*, vol. 10, no. 2, 2006.

[7] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Computer Architecture (ISCA), 2011 38th Annual International Symposium on.* IEEE, 2011, pp. 365–376.

[8] S. Borkar, "The exascale challenge," 2010.

[9] M. D. Hill and M. R. Marty, "Amdahl's law in the multicore era," *Computer*, vol. 41, no. 7, pp. 33–38, Jul. 2008.

[10] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor, "Conservation cores: reducing the energy of mature computations," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 1. ACM, 2010, pp. 205–218.

[11] N. Seifert, P. Slankard, M. Kirsch, B. Narasimham, V. Zia, C. Brookreson, A. Vo, S. Mitra, B. Gill, and J. Maiz, "Radiation-induced soft error rates of advanced cmos bulk devices," in *Reliability Physics Symposium Proceedings, 2006. 44th Annual., IEEE International.* IEEE, 2006, pp. 217–225.

[12] S. Borkar, T. Karnik, S. Narendra, J. Tschanz, A. Keshavarzi, and V. De, "Parameter variations and impact on circuits and microarchitecture," in *Proceedings of the 40th annual Design Automation Conference.* ACM, 2003, pp. 338–342.

[13] B. Schroeder and G. A. Gibson, "Understanding failures in petascale computers," in *Journal of Physics: Conference Series*, vol. 78, no. 1. IOP Publishing, 2007, p. 012022.

[14] K. Ferreira, J. Stearley, J. H. Laros III, R. Oldfield, K. Pedretti, R. Brightwell, R. Riesen, P. G. Bridges, and D. Arnold, "Evaluating the viability of process replication reliability for exascale systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis.* ACM, 2011, p. 44.

[15] J. Chung, I. Lee, M. Sullivan, J. H. Ryoo, D. W. Kim, D. H. Yoon, L. Kaplan, and M. Erez, "Containment domains: A scalable, efficient and flexible resilience scheme for exascale systems," *Scientific Programming*, vol. 21, no. 3, pp. 197–212, 2013.

[16] S. S. Mukherjee, J. Emer, and S. K. Reinhardt, "The soft error problem: An architectural perspective," in *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on.* IEEE, 2005, pp. 243–247.

[17] J. Shalf, S. Dosanjh, and J. Morrison, "Exascale computing technology challenges," in *High Performance Computing for Computational Science–VECPAR 2010.* Springer, 2011, pp. 1–25.

[18] J. Dennis, G. Gao, and X. Meng, "Experiments with the Fresh Breeze tree-based memory model," *Computer Science - Research and Development*, vol. 26, pp. 325–337, 2011, 10.1007/s00450-011-0165-1. [Online]. Available: http://dx.doi.org/10.1007/s00450-011-0165-1

[19] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick, "The potential of the cell processor for scientific computing," in *Proceedings of the 3rd conference on Computing frontiers.* ACM, 2006, pp. 9–20.

[20] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel, "Scratchpad memory: design alternative for cache on-chip memory in embedded systems," in *Proceedings of the tenth international symposium on Hardware/software codesign.* ACM, 2002, pp. 73–78.

[21] B. Egger, J. Lee, and H. Shin, "Dynamic scratchpad memory management for code in portable systems with an mmu," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 2, p. 11, 2008.

[22] H. Sutter, "The free lunch is over: A fundamental turn toward concurrency in software," *Dr. Dobbs Journal*, vol. 30, no. 3, pp. 202–210, 2005.

[23] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software.* Pearson Education, 1994.

[24] "Intel thread building blocks." [Online]. Available: https://www.threadingbuildingblocks.org/

[25] Z. Budimlić, M. Burke, V. Cavé, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. Peixotto, V. Sarkar, F. Schlimbach *et al.*, "Concurrent collections," *Scientific Programming*, vol. 18, no. 3, pp. 203–217, 2010.

[26] H. Kaul, M. Anders, S. Hsu, A. Agarwal, R. Krishnamurthy, and S. Borkar, "Near-threshold voltage (ntv) design: opportunities and challenges," in *Proceedings of the 49th Annual Design Automation Conference*, ser. DAC '12. New York, NY, USA: ACM, 2012, pp. 1153–1158.

[27] T. Mudge, "Power: A first-class architectural design constraint," *Computer*, vol. 34, no. 4, pp. 52–58, 2001.

[28] K. J. Kuhn, M. D. Giles, D. Becher, P. Kolar, A. Kornfeld, R. Kotlyar, S. T. Ma, A. Maheshwari, and S. Mudanai, "Process technology variation," *Electron Devices, IEEE Transactions on*, vol. 58, no. 8, pp. 2197–2208, 2011.

[29] T. C. May and M. H. Woods, "Alpha-particle-induced soft errors in dynamic memories," *Electron Devices, IEEE Transactions on*, vol. 26, no. 1, pp. 2–9, 1979.

[30] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi, "Modeling the effect of technology trends on the soft error rate of combinational logic," in *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on.* IEEE, 2002, pp. 389–398.

[31] M. Hill and C. Kozyrakis, "Advancing computer systems without technology progress," 2012. [Online]. Available: http://www.cs.wisc.edu/~markhill/papers/isat2012_ACSWTP.pdf

[32] B. Sinharoy, R. Kalla, W. Starke, H. Le, R. Cargnoni, J. Van Norstrand, B. Ronchetti, J. Stuecheli, J. Leenstra, G. Guthrie *et al.*, "Ibm power7 multicore server processor," *IBM Journal of Research and Development*, vol. 55, no. 3, pp. 1–1, 2011.

[33] T. Corporation, "Tilepro processor family," 2014. [Online]. Available: http://www.tilera.com/products/processors/TILEPro_Family

[34] I. Corporation, "Intel core i7 processor," 2014. [Online]. Available: http://www.intel.com/content/www/us/en/processors/core/core-i7-processor.html

[35] M. D. Hill and M. R. Marty, "Amdahl's law in the multicore era," *Computer*, vol. 41, no. 7, pp. 33–38, 2008.

[36] F. J. Pollack, "New microarchitecture challenges in the coming generations of cmos process technologies (keynote address)," in *Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture.* IEEE Computer Society, 1999, p. 2.

[37] A. Danowitz, K. Kelley, J. Mao, J. P. Stevenson, and M. Horowitz, "Cpu db: recording microprocessor history," *Communications of the ACM*, vol. 55, no. 4, pp. 55–63, 2012.

[38] S. Borkar, "Thousand core chips: a technology perspective," in *Proceedings of the 44th annual Design Automation Conference.* ACM, 2007, pp. 746–749.

[39] E. S. Chung, P. A. Milder, J. C. Hoe, and K. Mai, "Single-chip heterogeneous computing: Does the future include custom logic, fpgas, and gpgpus?" in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture.* IEEE Computer Society, 2010, pp. 225–236.

[40] T. Corporation, "Ti omap applications processors," 2014. [Online]. Available: http://www.ti.com/lsds/ti/omap-applications-processors/overview.page

[41] P. Greenhalgh, "big.little processing with ARM Cortex-A15 & Cortex-A7," White Paper, ARM, September 2011.

[42] J. Reinders, "An overview of programming for intel® xeon® processors and intel® xeon phi coprocessors," 2012.

[43] A. Heinecke, M. Klemm, and H.-J. Bungartz, "From gpgpu to many-core: Nvidia fermi and intel many integrated core architecture," *Computing in Science & Engineering*, vol. 14, no. 2, pp. 78–83, 2012.

[44] P. Sundararajan, "High performance computing using fpgas," *Xilinx White Paper: FPGAs*, pp. 1–15, 2010.

[45] H. P. Hofstee, "Power efficient processor architecture and the cell processor," in *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on.* IEEE, 2005, pp. 258–262.

[46] A. Ramirez, F. Cabarcas, B. Juurlink, M. Alvarez Mesa, F. Sanchez, A. Azevedo, C. Meenderinck, C. Ciobanu, S. Isaza, and G. Gaydadjiev, "The SARC architecture," *Micro, IEEE*, vol. 30, no. 5, pp. 16–29, 2010.

[47] S. Borkar and A. A. Chien, "The future of microprocessors," *Communications of the ACM*, vol. 54, no. 5, pp. 67–77, 2011.

[48] Y. Rajasekhar, "Changing the memory paradigm: A Novel Memory Architecture and Computational Model for Parallel Reconfigurable Architectures (in progress)," Ph.D. dissertation, University of North Carolina, Charlotte.

[49] T. Mattson, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl *et al.*, "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.* IEEE Computer Society, 2010, pp. 1–11.

[50] J. R. Hauser and J. Wawrzynek, "Garp: A mips processor with a reconfigurable coprocessor," in *Field-Programmable Custom Computing Machines, 1997. Proceedings., The 5th Annual IEEE Symposium on.* IEEE, 1997, pp. 12–21.

[51] H. K.-H. So and R. Brodersen, "A unified hardware/software runtime environment for fpga-based reconfigurable computers using borph," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 2, p. 14, 2008.

[52] E. Anderson, J. Agron, W. Peck, J. Stevens, F. Baijot, E. Komp, R. Sass, and D. Andrews, "Enabling a uniform programming model across the software/hardware boundary," in *Field-Programmable Custom Computing Machines, 2006. FCCM'06. 14th Annual IEEE Symposium on.* IEEE, 2006, pp. 89–98.

[53] E. Lübbers and M. Platzner, "Reconos: An rtos supporting hard-and software threads." in *FPL*, 2007, pp. 441–446.

[54] P. E. McKenney and J. Walpole, "Introducing technology into the linux kernel: a case study," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 5, pp. 4–17, 2008.

[55] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich, "An analysis of linux scalability to many cores," 2010.

[56] D. Wentzlaff and A. Agarwal, "Factored operating systems (fos): the case for a scalable operating system for multicores," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 2, pp. 76–85, 2009.

[57] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, "The multikernel: a new os architecture for scalable multicore systems," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles.* ACM, 2009, pp. 29–44.

[58] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," *SIGPLAN Not.*, vol. 40, no. 10, pp. 519–538, Oct. 2005. [Online]. Available: http://doi.acm.org/10.1145/1103845.1094852

[59] F. Cappello and D. Etiemble, "Mpi versus mpi+ openmp on the ibm sp for the nas benchmarks," in *Supercomputing, ACM/IEEE 2000 Conference.* IEEE, 2000, pp. 12–12.

[60] R. Rabenseifner, G. Hager, and G. Jost, "Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes," in *Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on.* IEEE, 2009, pp. 427–436.

[61] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *Computational Science & Engineering, IEEE*, vol. 5, no. 1, pp. 46–55, 1998.

[62] P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta, "Cellss: a programming model for the cell be architecture," in *SC 2006 Conference, Proceedings of the ACM/IEEE.* IEEE, 2006, pp. 5–5.

[63] E. Ayguadé, R. M. Badia, F. D. Igual, J. Labarta, R. Mayo, and E. S. Quintana-Ortí, "An extension of the starss programming model for platforms with multiple gpus," in *Euro-Par 2009 Parallel Processing.* Springer, 2009, pp. 851–862.

[64] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally *et al.*, "Sequoia: programming the memory hierarchy," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing.* ACM, 2006, p. 83.

[65] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng, "Merge: a programming model for heterogeneous multi-core systems," in *ACM SIGOPS operating systems review*, vol. 42, no. 2. ACM, 2008, pp. 287–296.

[66] D. Tarditi, S. Puri, and J. Oglesby, "Accelerator: using data parallelism to program gpus for general-purpose uses," in *ACM SIGARCH Computer Architecture News*, vol. 34, no. 5. ACM, 2006, pp. 325–335.

[67] H. Kaiser, M. Brodowicz, and T. Sterling, "Parallex an advanced parallel execution model for scaling-impaired applications," in *Parallel Processing Workshops, 2009. ICPPW'09. International Conference on.* IEEE, 2009, pp. 394–401.

[68] E. Mohr, D. A. Kranz, and R. H. Halstead, Jr., "Lazy task creation: A technique for increasing the granularity of parallel programs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 2, no. 3, pp. 264–280, Jul. 1991.

[69] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, *Cilk: An efficient multithreaded runtime system.* ACM, 1995, vol. 30, no. 8.

[70] J. E. Baldeschwieler, R. D. Blumofe, and E. A. Brewer, "A tlas: an infrastructure for global computing," in *Proceedings of the 7th workshop on ACM SIGOPS European workshop: Systems support for worldwide applications.* ACM, 1996, pp. 165–172.

[71] R. D. Blumofe and P. A. Lisiecki, "Adaptive and reliable parallel computing on networks of workstations," in *USENIX 1997 Annual Technical Conference on UNIX and Advanced Computing Systems*, 1997, pp. 133–147.

[72] R. v. Nieuwpoort, J. Maassen, T. Kielmann, and H. E. Bal, "Satin: Simple and efficient java-based grid programming," *Scalable Computing: Practice and Experience*, vol. 6, no. 3, 2001.

[73] K. Yelick, D. Bonachea, W.-Y. Chen, P. Colella, K. Datta, J. Duell, S. L. Graham, P. Hargrove, P. Hilfinger, P. Husbands *et al.*, "Productivity and performance using partitioned global address space languages," in *Proceedings of the 2007 international workshop on Parallel symbolic computation.* ACM, 2007, pp. 24–32.

[74] R. W. Numrich and J. Reid, "Co-array fortran for parallel programming," *SIGPLAN Fortran Forum*, vol. 17, no. 2, pp. 1–31, Aug. 1998. [Online]. Available: http://doi.acm.org/10.1145/289918.289920

[75] B. Chamberlain, D. Callahan, and H. Zima, "Parallel programmability and the chapel language," *International Journal of High Performance Computing Applications*, vol. 21, no. 3, pp. 291–312, 2007.

[76] B. Catanzaro, S. Kamil, Y. Lee, J. Demmel, K. Keutzer, J. Shalf, K. Yelick, and A. Fox, "Sejits: Getting productivity and performance with selective embedded jit specialization."

[77] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih, "Pycuda and pyopencl: A scripting-based approach to gpu run-time code generation," *Parallel Computing*, vol. 38, no. 3, pp. 157–174, 2012.

[78] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[79] F. Bellard, "Qemu, a fast and portable dynamic translator." in *USENIX Annual Technical Conference, FREENIX Track*, 2005, pp. 41–46.

[80] L. Eeckhout, "Computer architecture performance evaluation methods," *Synthesis Lectures on Computer Architecture*, vol. 5, no. 1, pp. 1–145, 2010.

[81] "Amber: Arm-compatible core." [Online]. Available: http://opencores.org/project,amber

[82] "Powerpc 440 embedded core." [Online]. Available: https://www-01.ibm.com/chips/techlib/techlib.nsf/products/PowerPC_440_Embedded_Core

[83] J. Backus, "Can programming be liberated from the von neumann style?: a functional style and its algebra of programs," *Communications of the ACM*, vol. 21, no. 8, pp. 613–641, 1978.

[84] V. Pankratius, F. Schmidt, and G. Garretón, "Combining functional and imperative programming for multicore software: an empirical study evaluating scala and java," in *Proceedings of the 2012 International Conference on Software Engineering*. IEEE Press, 2012, pp. 123–133.

[85] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms*, 2nd ed. McGraw-Hill Higher Education, 2001.

[86] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex fourier series," *Mathematics of computation*, vol. 19, no. 90, pp. 297–301, 1965.

[87] C. A. Hoare, "Quicksort," *The Computer Journal*, vol. 5, no. 1, pp. 10–16, 1962.

[88] V. Strassen, "Gaussian elimination is not optimal," *Numerische Mathematik*, vol. 13, no. 4, pp. 354–356, 1969.

[89] J. J. Dongarra and D. C. Sorensen, "A fully parallel algorithm for the symmetric eigenvalue problem," *SIAM Journal on Scientific and Statistical Computing*, vol. 8, no. 2, pp. s139–s154, 1987.

[90] W. De Leeuw, "Divide and conquer spot noise," in *Supercomputing, ACM/IEEE 1997 Conference*. IEEE, 1997, pp. 19–19.

[91] F. Zhang, X.-Z. Qiao, and Z.-Y. Liu, "A parallel smith-waterman algorithm based on divide and conquer," in *Algorithms and Architectures for Parallel Processing, 2002. Proceedings. Fifth International Conference on*. IEEE, 2002, pp. 162–169.

[92] D. Frosyniotis, A. Stafylopatis, and A. Likas, "A divide-and-conquer method for multi-net classifiers," *Pattern Analysis & Applications*, vol. 6, no. 1, pp. 32–40, 2003.

[93] H.-C. Fu, Y.-P. Lee, C.-C. Chiang, and H.-T. Pao, "Divide-and-conquer learning and modular perceptron networks," *Neural Networks, IEEE Transactions on*, vol. 12, no. 2, pp. 250–263, 2001.

[94] C. Mota, I. Stuke, T. Aach, and E. Barth, "Divide-and-conquer strategies for estimating multiple transparent motions," in *Complex Motion*. Springer, 2007, pp. 66–77.

[95] G. Brinkmann, A. W. M. Dress, S. W. Perrey, and J. Stoye, "Two applications of the divide&conquer principle in the molecular sciences," *Mathematical programming*, vol. 79, no. 1-3, pp. 71–97, 1997.

[96] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," in *Foundations of Computer Science, 1999. 40th Annual Symposium on*. IEEE, 1999, pp. 285–297.

[97] E. D. Demaine, "Cache-oblivious algorithms and data structures," *Lecture Notes from the EEF Summer School on Massive Data Sets*, pp. 1–29, 2002.

[98] F. Petrini, D. Kerbyson, and S. Pakin, "The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of asci q," in *Supercomputing, 2003 ACM/IEEE Conference*. IEEE, 2003, pp. 55–55.

[99] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, A. Fasih, A. Sarma, D. Nanongkai, G. Pandurangan, P. Tetali *et al.*, "Pycuda: Gpu run-time code generation for high-performance computing," *Arxiv preprint*, 2009.

[100] T. Oliphant, *A Guide to NumPy*. Trelgol Publishing USA, 2006, vol. 1.

[101] E. Jones, T. Oliphant, P. Peterson *et al.*, "SciPy: Open source scientific tools for Python," 2001–. [Online]. Available: http://www.scipy.org/

[102] L. Dalcín, R. Paz, and M. Storti, "Mpi for python," *Journal of Parallel and Distributed Computing*, vol. 65, no. 9, pp. 1108–1115, 2005.

[103] "Stackless python." [Online]. Available: http://www.stackless.com/

[104] B. Huang, R. Sass, N. DeBardeleben, and S. Blanchard, "Pydac: A resilient run-time framework for divide-and-conquer applications on a heterogeneous many-core architecture," in *The 6th Workshop on UnConventional High Performance Computing, UCHPC at Euro-Par'13*, 2013.

[105] "Python on a chip." [Online]. Available: http://code.google.com/p/python-on-a-chip/

[106] D. Beazley, "Understanding the python gil," *PRESENTATION AT PYCON*, 2010.

[107] G. H. Golub and C. F. Van Loan, *Matrix Computations*, 3rd ed. JHUP, 1996.

[108] P. Arbenz, D. Kressner, and D.-M. E. Zürich, "Lecture notes on solving large scale eigenvalue problems," 2012.

[109] "Recursive fft in python convertible to verilog/vhdl." [Online]. Available: http://www.dsprelated.com/showcode/16.php

[110] M. Lutz, *Programming python*. O'Reilly Media, Inc., 2010.

[111] M. Lutz, *Learning python*. O'Reilly Media, Inc., 2013.

[112] H. P. Langtangen, *Python scripting for computational science*. Springer, 2006, vol. 3.

[113] W. H. Press, *Numerical recipes 3rd edition: The art of scientific computing.* Cambridge university press, 2007.