BREAKING AWAY FROM THE OS SHADOW: A PROGRAM EXECUTION MODEL AWARE THREAD VIRTUAL MACHINE FOR MULTICORE ARCHITECTURES

by

Juan del Cuvillo

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

Summer 2008

BREAKING AWAY FROM THE OS SHADOW: A PROGRAM EXECUTION MODEL AWARE THREAD VIRTUAL MACHINE FOR MULTICORE ARCHITECTURES

by

Juan del Cuvillo

Approved: _____

Gonzalo R. Arce, Ph.D.
Chairperson of the Department of Electrical and Computer Engineering

Approved: _____

Michael J. Chajes, Ph.D.
Dean of the College of Engineering

Approved: _____

Debra Hess Norris, M.S.
Vice Provost for Graduate and Professional Education

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

Signed: _____

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

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

Signed: _____

Xiaoming Li, Ph.D.
Member of dissertation committee

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

Signed: _____

Stephan Bohacek, Ph.D.
Member of dissertation committee

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

Signed: _____

John Cavazos, Ph.D.
Member of dissertation committee

ACKNOWLEDGEMENTS

First, I would like to thank my advisor, Prof. Guang R. Gao, for his continuous encouragement and support throughout my long journey in graduate school. His contagious enthusiasm and motivation were invaluable, especially during the downturns, and his creativity helped to create the environment that facilitates my research.

Second, I am grateful to the “Cyclops” community, especially to the members of the CAPSL group, for their helpful comments, invaluable feedback, and for helping to move the Cyclops-64 software project forward and, therefore, my research as well. Special thanks to Ziang Hu, Weirong Zhu, Gan Ge, and Fei Chen, with whom I had the privilege to work from the very early stages of the Cyclops-64 project.

Third, I would like to acknowledge all the ETI employees who, in one way or another, are helping to bring the Cyclops-64 system software up.

Finally, I wish to express my deepest gratitude to Elizabeth for her unconditional support, encouragement, and love. Without her, this work simply would not have been possible. To Alexia for bringing more joy that I thought possible, and reminding me when it was time to take a break, and also to my family in Spain.

To Elizabeth

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

The Cyclops-64 (C64) project questions fundamentally the suitability of conventional operating systems to achieve high performance. Domain-specific application experts who have participated in the conception of all aspects of the system software for the C64 supercomputer mandated that Linux was not adequate. Their previous experience with parallel applications that did not scale well for various reasons motivated us to develop a standalone Thread Virtual Machine (TVM) from scratch. Its implementation in the form of the TiNy Threads (TNT) library had the clear goal of allowing applications to achieve full resource utilization.

This dissertation is about the C64 system software in general, and the TNT library in particular. TNT replaces the conventional OS with a non-intrusive runtime system. Even though it is implemented as a user-level library, TNT manages the hardware resources directly. In addition, TNT provides a solid foundation for the development of advanced program execution models. However, for rapid prototyping of applications, TNT also provides a familiar Linux-like programming environment.

As evidence that the TNT model provides a good platform to experiment with innovative execution models, we developed MAGMA. Defined as a memory adaptive program execution model for multicore architectures, MAGMA uses percolation to migrate data that the user (programmer or compiler) identifies, to a level of the memory hierarchy local to the processing element before computation starts. MAGMA takes advantage of the large number of thread units in C64. MAGMA implements a multithreaded percolation engine that runs on a number of cores to maximize bandwidth utilization.

Chapter 1

INTRODUCTION

Throughout history, science and technological progress has been the result of theory and experiment. More recently, with the advent of computer machinery, scientists have been able to analyze and better understand complex physical phenomena and engineering systems through computer models, to such an extent that computer modeling and simulations are nowadays widely recognized as fundamental components in science and technology development.

In parallel to the expansion of computer models and simulation techniques, high-end computing systems, also known as supercomputers, have become increasingly important because they allow scientists and engineers to model such systems in far greater detail and complexity than what main-stream computer systems allow.

Earth and atmospheric sciences, energy and environment, nanoscale science and technology, life sciences, and aerospace vehicle design are some of the application domains that currently demand an increase in both computing power and memory space. This increase ranges from 100 to 1,000 times of today's computing systems resources to tackle certain important scientific and engineering problems [13].

Despite advances in high-end computing technology, the effective use of high-end computing systems is still limited by issues such as poor system performance and reliability, as well as the increasing cost and risk of software development. In fact, there is a widespread agreement among the high-end computing community that those aspects of the hardware and software that impact performance, programmability, ease of use, and

scalability need to be addressed if we are to manage such a set of large computational problems.

From the hardware standpoint two separate matters need to be considered: chip design and system integration. First, as advances in integrated circuits technology allow the feature size to drop, density of transistors on silicon chips are to continue increasing for the next years following Moore's Law[1] [44]. Not surprisingly, billion-transistors chips have been launched already by a few major chip manufacturers [26, 6] and multi-billion transistors chips are expected before 2010 [10, 49, 41, 55, 39, 32, 60]. By that time, the main limitation for the emergence of new micro-architecture functionality will be the computer architect's imagination. However, hardware designers are expected to deal with issues that have started to surface in current technology. For instance, CPU power dissipation imposes already serious constraints on the scaling of clock frequency. In addition, as wires become slower relative to logic gates, the distribution of a single global clock throughout a chip will be a difficult challenge. A paradigm proposed to cope with this constraint is the integration of a large number of simple processors on a single die, in what is known as Chip Multi-Processors (CMP), instead of devoting the entire die to a single and complex processor. In the last years, all major microprocessor manufacturers have been releasing dual- and quad-core versions of their processors, and have also announced their intentions to bring eight-core chips to the market. Therefore, as the semiconductor technology surpasses the integration of a billion transistors on an integrated circuit, we should expect chips based on multicore architectures to become commonplace. Second, there is greater evidence that applications would benefit significantly from an alternative to the commercial off-the-shelf (COTS) based solutions that have dominated the supercomputing arena over the last decade. Indeed, government agencies and hardware vendors are currently working on a major departure from the Beowulf paradigm [2, 34, 46].

[1] Moore's Law describes the trend that the number of transistors integrated on a chip would double about every two years.

From the software point of view, programmability, portability and reliability of both operating and runtime systems are still open issues. These are only aggravated by the requirement for sustainable and scalable real (not peak) performance. Increasing coupling among all the software layers seems an important trend aimed to increase transparency and reduce overheads. If the hardware was efficiently exposed to the programmer, the user could take advantage of the functionality that impacts performance. A question still remains: how to narrow the interface between the raw architecture and the user.

In December 1999, IBM announced the Blue Gene project as an effort to build a new generation of supercomputers. Since then, two different architecture designs have been proposed, which are at different stages of development. Blue Gene/L [3, 24, 25] systems, based on a dual- and quad-core Power4 processors, have been shipped to various research institutions and are among the top ten most powerful supercomputers. Cyclops-64 [19] employs a state-of-the-art multiprocessor-on-a-chip technology to build a completely new chip. Hence, it is in an earlier stage of development with the first system prototype expected by the end of this year.

Cyclops-64 (C64) is a petaflop supercomputer project under development at IBM Research Center. C64 is designed to serve as a dedicated compute engine for running high performance applications such as molecular dynamics, to study protein folding, and image processing, to support real-time medical procedures. Using a cellular organization, a C64 petaflop machine is built out of millions of simple processing cells; a thread unit, the base processing element, is replicated and conformed into several structural organizations. Two thread units, the same number of SRAM memory banks, and a floating point unit constitute a processor. Eighty processors connected to a crossbar network, instruction cache, bidirectional inter-chip routing ports, and an interface to off-chip DDR SDRAM are integrated on a C64 chip. Finally, tens of thousands of C64 nodes, each one consisting of a C64 chip, external DRAM and a small amount of external interface logic, build a C64 supercomputer.

With the advent of a new generation of high performance computing systems, as well as features unique to those found in the Cyclops-64 cellular architecture, it is the objective of this research to answer questions such as:

- What role should the program execution model play in the definition and use of computer systems, in general, and of high performance computing systems, in particular?
- What does the interface between the program execution model and OS look like?
- How can the OS facilitate the adoption of new program execution models?
- Should the OS, as we know it today, be replaced?
- Can a program execution model provide the umbrella under which all the aspects of a computing system (hardware, runtime system, compiler, and user) work together to minimize the effects that limited memory bandwidth and long memory latency have on application performance and, as a result, on the productivity of a computing system?

1.1 The Cyclops-64 Project

The Cyclops cellular architecture, first proposed in late 1990s at IBM's T.J. Watson Research Center, has since evolved substantially in different application contexts and directions. The latest Cyclops-64 (C64) chip architecture employs a multiprocessor-on-a-chip design with a large number of hardware thread units and embedded memory. The C64 has a high computation to memory ratio (number-of-hardware-threads/on-chip-memory); 1 to 2 orders of magnitude higher than a modern microprocessor chip.

The C64 is designed to serve as a dedicated compute engine for running high performance applications such as molecular dynamics, to study protein folding, and image processing, to support real-time medical procedures. The C64 supercomputer is attached

to a host system through a number of Gigabit Ethernet links. The host system provides a familiar computing environment (such as Linux) to application software developers and end users. Each C64 chip has access (through the Ethernet links) to a common file server used for storing input and output data sets used and produced by application programs.

The main objective behind the C64 chip design is to build a petaflop computer by scaling up some millions of simple processing elements and providing massive intra-chip parallelism to tolerate memory and functional unit latencies. On the C64 architecture, the computational cell is a simple thread unit; a 64-bit in-order RISC processor with a small instruction set architecture (60 instruction groups) operating at 500MHz. If a thread stalls on a memory access because of a data dependency between instructions, other threads can proceed independently.

With more than 100 thread units, a C64 chip can be seen as an n-way Symmetric MultiProcessing (SMP) system. Although memory is shared within a chip, communication among nodes is only possible by means of message passing. Hence, a C64 supercomputer can be seen as a cluster of SMPs. Simplicity in hardware design led to a system with no resource virtualization (virtual addresses map directly to physical addresses and execution is non-preemptive) and a non-uniform address space (with several memory levels exposed directly to the user). Caches, which would be hard to keep coherent, have been replaced with on-chip memory mapped into the address space, hence controlled by the programmer.

The Cyclops-64 system software development project began with the objective of designing a full system software infrastructure for the C64 architecture. Such a software infrastructure had to provide a reasonable interface for application developers, yet expose as much functionality as possible to achieve high-levels of performance. Given an architecture like C64, aimed for sustainable performance through simplicity, it was not the intention of this project to build a conventional software development environment. Instead, we built a custom system software from the ground up.

Based on our previous experience in the embedded Cyclops-32 project [52, 18], we believe the first requirement from the system software standpoint is a Thread Virtual Machine that can efficiently manage hardware resources such as a large number of thread units without OS intervention, and will cause no disruption to the user application.

1.2 Contributions

Many people have been involved in various aspects of the Cyclops-64 project, and in the development of the Cyclops-64 system software. The following are the contributions that are solely or primarily the work of the author:

- Revising the role that the OS plays in current high performance computing systems, and proposing to replace the OS with a Program Execution Model aware Thread Virtual Machine (TVM). A TVM not only provides the abstraction layer and the application program interface that programmers expect, but it also supports the direct mapping of program execution models to the architecture without interference from the OS.

- Proposing a system software methodology centered in the premise above, and architecting the design and development of the system software infrastructure for the Cyclops-64 supercomputer according to such methodology.

- Designing and constructing FAST, a functionally accurate simulator for the Cyclops-64 architecture. FAST played a critical role in the project, as it has been supporting all Cyclops-64 system software design, development and testing for the past four years and helped in the verification of the Cyclops-64 logic design as well.

- Designing and implementing TNT, a Thread Virtual Machine for the Cyclops-64 architecture. TNT, provided in the form of a light-weight microkernel and runtime system library, has been in production use for the past two years.

- Studying the OpenMP programming model on Cyclops-64. Given an OpenMP compiler that was ported to the Cyclops-64 architecture, we optimized the OpenMP runtime library to study the feasibility of OpenMP as a possible programming model for Cyclops-64.

- Demonstrating that the Cyclops-64 system software platform, implemented as part of this research, is sound.

- Defining the MAGMA Program Execution Model as an abstract model for running multithreaded applications on multicore based systems. MAGMA applications are able to tolerate the different latencies that are common in multi-level memory hierarchies present in modern multicore architectures via percolation.

- Comparing the MAGMA, EARTH, and Cilk Program Execution Models, and explaining their similarities and differences, as well as their strengths and weaknesses.

- Implementing the MAGMA Program Execution Model for the Cyclops-64 chip architecture, and the corresponding software support in the Cyclops-64 system software toolchain.

- Demonstrating that the TNT model provides a solid foundation for development of advanced Program Execution Models.

- Coding various benchmarks according to the MAGMA Program Execution Model so they may be tested with the Cyclops-64 system software toolchain.

1.3 Synopsis

This dissertation is organized as follows:

In Chapter 2, we review the early history of operating systems as well as the state of the art today. We point out that some of the principles established in the 1960s with

the purpose of using computing systems more efficiently are still in use today. However, like other authors, we question whether the same model is directly applicable to high-end computing systems. We also compare our work with many other threading and multithreading packages available these days, and we highlight why none of them fit our purpose.

Our research proposes a new system software methodology aimed at future high-end computing systems using multicore architectures. Chapter 3 provides a classification of multicore architectures (coarse and fine-grain multicore). This chapter also introduces Cyclops-64 (C64), the many-core architecture used in this research.

Chapter 4 describes the design and implementation of the Cyclops-64 system software. It focuses on three main components: the host software, the C64 toolchain, and the FAST simulator.

Chapter 5 describes the key component of the C64 system software, and the purpose of this research, the TiNy Threads Thread Virtual Machine. First, we present the architecture of the overall TVM, then we describe the implementation of TNT.

In Chapter 6 we present MAGMA, a memory adaptive program execution model for many-core architectures. We use program examples to illustrate the basic features of the program execution model, we define the MAGMA model and we enumerate the operations supported in the MAGMA model, and we finish with a comparison between MAGMA, EARTH and Cilk program execution models.

Chapter 7 summarizes our experimental results. We present our final conclusions and future work in Chapter 8. Appendix A provides the complete source code of the *N-Queens* program that we used in the comparison between MAGMA, EARTH, and Cilk.

Chapter 2

BACKGROUND

In the early days of computing, when mainframes and minicomputers were extremely expensive, the notion of sharing computing resources took root. Multitasking, a collection of methods that facilitate sharing common resources among multiple processes and users, became popular. First, multiprogramming systems were developed, in which a task runs until the program performs an operation that requires waiting for an external event. In order to efficiently use an expensive CPU, in multiprogrammed systems a process that becomes idle waiting for I/O is swapped out, until the I/O operation completes. Multiprogramming (operating) systems required the invention of a number of techniques, including the concepts of virtual memory [27, 38, 5] and time-sharing [12, 11]. Virtual memory not only gives a program the illusion of a large and continuous address space, but it also solves the memory protection problem and permits users to share memory segments containing data or procedures. More importantly, virtual memory provided the foundation for an unparalleled degree of programming generality that is still in use today [20, 21, 15]. Time-sharing provided multiple users simultaneous access to a computing system. In the first time-sharing systems, a user was serviced using some other user's idle time. With the advent of hardware and software support for preemption, the operating system could establish a fixed time slice per process, and distribute the CPU time among all the users of a system in an orderly fashion.

The aforementioned concepts, methods, and techniques described above such as virtual memory, time-sharing, multiprogramming, multitasking, etc. are still in use in

modern operating systems. They are of practical use on mainframes, servers, and workstations wherein the goal is to maximize a computing system throughput, measured as the number of tasks completed by unit of time. However, the operational model of supercomputers is different. A supercomputer focuses on computing power to do one task (for a single user) involving numerically intensive calculations, such as the applications mentioned in Chapter 1.

Over the last decade, computing systems based on commercial off-the-shelf (COTS) microprocessors and the Beowulf paradigm, or clusters, have dominated the supercomputing arena. Among other things, this domination has resulted in the adoption of a conventional operating system such as Linux as the de facto standard kernel for high performance computing.

In the past few years, when the processor count reached the thousand order magnitude, supercomputer manufacturers and users began to notice application performance loss due to interference of the operating system [50, 7]. While the community began to question the appropriateness of Linux (or Linux-like) operating systems for high computing systems, IBM decided to develop a custom kernel for the Blue Gene/L supercomputer [24]. Brightwell et al. also made both technical and social arguments against the adoption of Linux for large scale computing systems, and they proposed a lightweight kernel (LWK) instead [9]. They mentioned issues such as a lack of predictability when the operating system preempts the application, and the adverse impact of virtual memory in the communication library. They also expressed concern with the rapid developments in the kernel, distributions, and development environments, in general. On the other hand, Minnich et al. claimed that LWKs are optimized for one type of application activity, and they remove many needed capabilities such as file systems, sockets, and security. They proposed a rightweight kernel (RWK) based on an off-the-shelf kernel (i.e. Linux) [43]. Following suit, IBM has recently launched a study to evaluate the effect of replacing the custom kernel with Linux on the compute nodes of Blue Gene/L [53]. IBM claims that

certain applications require capabilities such as those mentioned by Minnich. Beckman et al. propose a third point of view [4]. They suggest that the focus should be on the problems that will prevent high performance computing systems from reaching the Petaflop barrier. Among others, they mention issues such as synchronization and collective operations, parallel I/O, and fault tolerance.

Regardless of whether the kernel is heavyweight (HWK), lightweight (LWK) or rightweight (RWK), none of the previous work takes into consideration the importance of the Program Execution Model, or the additional challenges to come with the arrival of many-core architectures. On the other hand, the goal of our research is precisely to identify the critical aspects of a Program Execution Model, in particular those likely to become relevant when working with many cores. We also plan to demonstrate that the Program Execution Model can be integrated into the low-level system software, primarily for the reason that we believe said model should be an integral part of a computing system, including high-end computing systems.

There is also a myriad of work related to the design and implementation of thread libraries found in multithreaded runtime systems, or provided as stand-alone thread packages such as: Coda [51], Pthreads [45], Quick-Threads [37], TAM [14], uThreads [54], Converse [35], Lazy Threads [29], Nano-Threads [47], OpenThreads [31], Active Threads [61], Cilk [28], NPTL [22], Cappricio [59].

These thread packages have been developed as part of the runtime system for multithreaded parallel programming languages. Their goals have been to provide portability, interoperability and open implementation with regard to design decisions (e.g. scheduling and preemption). To achieve portability across parallel machines and environments, a number of them assume a common software substrate that the OS or a machine-dependent layer will provide. Unlike them, TNT is a standalone user library that provides high efficiency at the expense of portability, running directly on top of the C64 architecture without kernel support.

Some multithreaded programming models rely on sophisticated compiler analysis to achieve efficiency [14, 29]. On the other hand, TNT obtains efficiency by integrating hardware and virtual thread management.

The BlueGene/L Compute Node Kernel (CNK) also replaces the conventional OS [24]. The kernel provides 2 modes of operation (coprocessor and virtual node modes) aimed at maximizing the overlapping between computation and communication to expose the parallelism that MPI and UPC, the programming models available for this system, may expose. However, these programming languages allow to express fine-grain parallelism. On the other hand, TNT supports multithreaded execution as the natural way to make efficient use of the 160 processing elements in a C64 chip.

The implementation of the EARTH-MANNA multithreaded system runs directly on top of the hardware without the assistance of the OS [33]. TNT is a flexible special-purpose multithreaded library, whereas the EARTH-MANNA system was the specific implementation of the EARTH program execution model. Additionally, the implementation of some key aspects of the EARTH model are not directly applicable to C64. For instance, in EARTH-MANNA threads allocate their frame in the heap. In C64, such a feature would result in poor performance because of the limited DRAM bandwidth.

Cyclops-64 is not the only system with shipping I/O. Blue Gene/L compute nodes also ship I/O requests to the I/O nodes. However, in Blue Gene/L, I/O nodes are dedicated for I/O and run a Linux kernel [24]y. On Cyclops-64, I/O nodes run the TNT kernel and forward I/O requests to the front-end cluster, where the I/O operation is actually performed.

To the best of the author's knowledge, there is not previous work similar to RMEM. Maybe because the configuration of the C64 supercomputer, consisting on a front-end cluster attached to a C64 back-end engine, is uncommon. However, we believe, RMEM provides the foundation of a programming environment for heterogeneous architectures.

Chapter 3

HARDWARE ARCHITECTURE

In the last decade, performance gain experienced by the end-user of commercial computing systems was riding on advances in integrated circuits technology, in particular on increases in clock speeds. Since the clock frequency is now constrained by issues such as CPU power consumption and dissipation, computer architects started to look at better ways to use the ever increasing transistor density. As the number of transistors on a chip continues to double every two years, performance improvements are expected to come from the development of multicore processors, among other innovations.

3.1 Coarse Multicore Architectures

Nowadays there are two trends in specifying and designing multicore architectures. On one end, manufacturers of common-off-the-shelf processors take advantage of the advances in the semiconductor manufacturing technology to reduce the size of a commercial microprocessor and replicate several of these processors into the same die size. This type of multicore architecture is also known as coarse multicore. By integrating multiple cores into the same die, manufacturers provide customers with better performance per watt solutions. However, these manufacturers are developing few architecture innovations. In particular, as the package pin count remains constant, so does the memory bandwidth. As a consequence, applications are more likely to notice the effect of bandwidth and latency limitations on performance. In other words, in coarse multicore architectures, applications will often have to face the Memory Wall problem [62, 42].

Table 3.1: Examples of Many-Core Architectures

Manufacturer	Processor Family	No. of cores
IBM	Cyclops-64	160 thread units
IBM	Cell	1 PowerPC and 8 SPEs
Intel	Tera-scale	80 cores
Intel	Larrabee	16 – 24 cores
Nvidia	Tesla GPU	128 streaming processors
ClearSpeed	CSX600	96 processing elements
Cisco	CSR Metro	192 processing elements
Tilera	TILE64	64 processor cores

3.2 Many-Core Architectures

In addition to coarse multicore architectures, more recently there has also been an industry trend towards the design and fabrication of fine-grain multicore or many-core architectures. Compared to coarse multicore, many-core architectures integrate a much larger number of small cores on a chip. In addition, fine-grain multicore chips come with architectural innovations such as special purpose processing elements and novel intra-chip interconnection devices. A large number of cores together with intra-chip communication networks may provide the intra-chip parallelism and bandwidth required to tolerate the impact of limited off-chip memory bandwidth.

Table 3.1 shows some examples of many-core architectures that are already available in the market or will be available soon. These processors have three common features: (1) the processing elements are much simpler than a commercial-off-the-shelf microprocessor; (2) the processing element design is usually geared towards an application domain; (3) while on-chip bandwidth is enormous, off-chip bandwidth remains limited.

3.3 Cyclops-64 Architecture

This research uses Cyclops-64 (C64) to illustrate the issues that many-core architectures are either facing or will face in the near future.

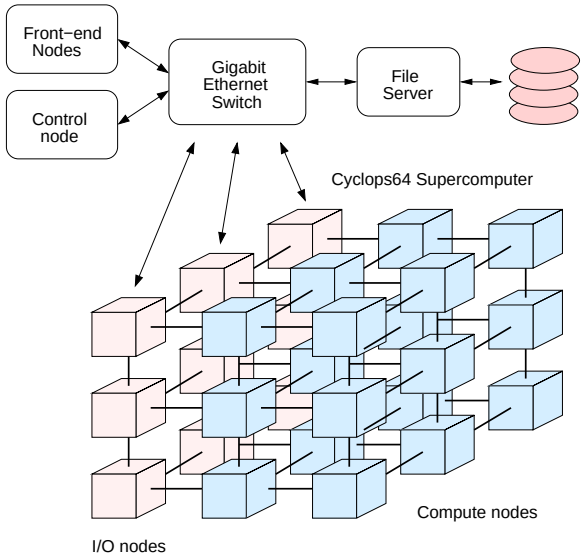


Figure 3.1: Cyclops-64 Computing Environment

Cyclops-64 (C64) is a flexible special purpose supercomputer comprised of a host connected to a C64 computing engine by a Gigabit Ethernet network, as shown in Figure 3.1. The host system (shown as consisting of a number of control nodes and front-end nodes) supports application development and program execution as well as system administration, monitoring, and boot. The file system, which may also contain multiple (external) file server nodes, provides file support for the C64 supercomputer. The C64 back-end consists of 13,824 C64 blades arranged in a $24 \times 24 \times 24$ logical configuration, see Figure 3.2. The peak performance of the C64 computing engine will exceed one PetaFLOPS [17].

C64 nodes are arranged in a 3D-mesh network. A fraction of these nodes, labeled as I/O nodes, use the Gigabit Ethernet port (present in all C64 chips) to connect the C64 supercomputer to the host and external file systems. Each I/O node will service a number of C64 nodes (called compute nodes) and relay requests and data between the compute nodes and the host and file server systems. The I/O nodes and compute nodes communicate via packets over the 3D-mesh network only. This 3D-mesh provides the high bandwidth necessary for inter-node communication in running application programs.

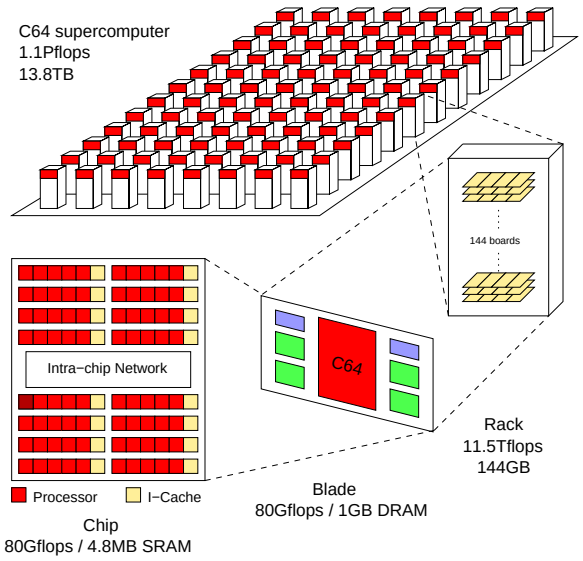


Figure 3.2: Cyclops-64 Supercomputer

There is a separate control network that connects the C64 system to the host system. This control network carries commands from the control nodes to each C64 node. A C64 node attaches to this control network via a special communication port. The host system uses this control network to initialize the C64 system, monitor its status while programs are in execution, and reconfigure and restart C64 after hardware failures. Details of the initialization and configuration procedures are not the focus of this research and will be discussed elsewhere.

Each C64 blade consists of a C64 chip, external DRAM, and a small amount of glue logic, as shown in Figure 3.3. A C64 chip employs a multiprocessor-on-a-chip architecture containing 80 processors. Each processor contains two thread units, a floating-point unit, and two SRAM memory banks of 30KB each. A thread unit is a simple 64-bit in-order RISC processor core with a small instruction set architecture operating at a moderate speed. In a C64 chip, there are 16 32KB instruction caches, each shared among five processors. Each group of 4 instruction caches share a crossbar port. In a C64 chip architecture, there is no data cache. Instead a portion of each SRAM bank can be configured as

scratch-pad memory. Such a memory provides a fast temporary storage to exploit locality under software control. Processors are connected to a crossbar network that enables intra-chip communication, i.e. access to other processor's memory and off-chip DRAM, as well as enabling inter-chip communication via two communication devices, called the A-switch and B-switch. The A-switch and B-switch connect each C64 chip to its nearest neighbors in the 3D-mesh. The intra-chip network also facilitates access to special hardware devices such as the Gigabit Ethernet port and the control network interface attached to each C64 node.

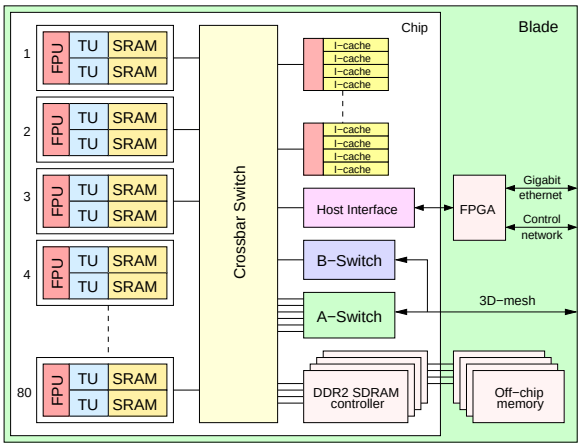


Figure 3.3: Cyclops-64 Blade

Besides its cellular organization, four distinct hardware features distinguish Cyclops-64 from other general purpose processors:

- The computation to on-chip memory (160/4.6MB) and computation to off-chip memory bandwidth (160/16GBps) ratios are much larger than in a commercial microprocessor. That is why we consider memory bandwidth a scarce resource and thread units rather inexpensive.

- Execution is non-preemptive. While running on user state, a context switch into supervisor state might happen whenever an exception occurs. However, this is intended as a protection mechanism only. In other words, the OS will never interrupt

the user program execution unless the user explicitly demands so (by executing a supervisor call) or an exception occurs.

- There is no hardware virtual memory manager, which means the memory hierarchy of the C64 chip is exposed to the programmer. Processors can directly address any memory location of the non uniform shared address space formed by the on-chip and off-chip memory banks within on a chip.

- In the C64 chip architecture there is no data cache. Instead, a portion of each SRAM bank can be configured as scratch-pad memory. Such a memory provides a fast temporary storage to exploit locality under software control.

Although the C64 has a special purpose ISA, for this research we only rely on features we believe will be mainstream in future high dense multicore architectures. In particular, we take advantage of the following two key features of the C64 architecture: (1) an instruction set architecture design that incorporates efficient support for thread level execution and a set of hardware supported in-memory atomic operations; (2) the ability to configure a portion of every SRAM bank as scratchpad memory, providing a fast temporary storage to exploit locality under software control.

Chapter 4

CYCLOPS-64 SYSTEM SOFTWARE

The Cyclops-64 (C64) project questions fundamentally the suitability of conventional operating systems to achieve high performance. Domain-specific application experts who have participated in the conception of all aspects of the system software for the C64 supercomputer mandated that Linux was not adequate. Their previous experience with parallel applications that did not scale well for various reasons motivated us to develop a standalone Thread Virtual Machine (TVM) from scratch. Its implementation in the form of the TiNy Threads (TNT) library had the clear goal of allowing applications to achieve full resource utilization [23].

The C64 programming environment and the first implementation of the TNT library have been in production use for the past two years. More than 15,000 programs of various sophistication levels have been ported to, and implemented on, the C64 architecture, and successfully tested in both simulation and emulation platforms. The experience and feedback from the C64 community can be summarized as follows:

- For flexible special-purpose architectures such as C64, and for high-end computing systems in general, a radical departure from the conventional OS is now being seriously considered. High performance applications run exclusively on several CPUs for extended period of times and require as little disruption as possible from the OS, let alone from other users. Single-user operating systems implemented as light-weight kernels that achieve full resource utilization by a single running process will be commonplace in the near future [24, 17].

- A wide range of issues from virtual memory management to protection between concurrent processes need to be revisited. For an application to receive full resource utilization, the system software must be non-intrusive to applications. There should be no observable degradation in performance for applications that do not request services from the runtime system.

- As long as a familiar programming environment is provided to the user, applications do not require software virtualization that adds layers of software that are not strictly required. For instance, the complexity of context switching on a conventional OS that is needed for multitasking is unnecessary when a single process application runs exclusively on a processor.

- Modern parallel programming languages exploit parallelism by means of multithreading. In some cases, a fairly large number of threads are created by the application regardless of the number of processors or processing elements available.

4.1 System Software Architecture

In many supercomputing projects, when an OS such as Linux is ported to a new architecture, a great deal of resources are spent trimming the OS's functionality. The objective is to ensure that services that are not strictly required interfere as little as possible with the applications. This reduced functionality results in a lower computational noise, which measures the degree to which an application is disturbed by the asynchronous execution of daemons and other OS processes.

The C64 architecture does not support preemption or virtual memory management. These features allow the design of a non-intrusive system software. But they are also essential aspects of a conventional OS. If porting and trimming an OS to a common-off-the-shelf microprocessor architecture requires considerable effort, porting and trimming an OS to the C64 architecture without the hardware features expected by the OS would be formidable, to say the least.

The C64 system software was designed taking into account the hardware architecture of the C64 supercomputer and the lack of a conventional operating system for the C64 architecture. In addition, to achieve full resource utilization, we decided that only services that are performance critical (from the application standpoint) should run on the C64 back-end. These services are provided by the TNT Thread Virtual Machine described in Chapter 5. As a result of this design decision, file I/O operations are shipped to the front-end, for instance. The standard C library provides the programmer with a standard I/O interface. However, when a system call is executed, TNT encapsulates the call arguments into a request and sends it to the front-end. Once the file I/O operation is performed, the host sends the results back to TNT, which are then forwarded to the application.

The remainder of this chapter highlights the features of the C64 system software. Note that all the components described in the next sections, including the C64 toolchain, run on the front-end cluster. TNT, the C64 Thread Virtual Machine, together with the standard C library and the communication libraries, are the only components that run on the C64 back-end. This ensures that applications do not experience observable degradation in performance if they do not request any service from TNT and a familiar programming environment.

4.2 Host Software

This section outlines the system software that is specific to the host, i.e., that runs on the front-end cluster. We refer to it as the host control software and its three main components are: job scheduler and launcher, resource manager and host to C64 communication.

4.2.1 Job scheduler

Like in other large computing systems [8, 48, 36, 56], the goal of the job scheduler is to maximize the utilization of the computing system by minimizing waiting and

idle times. On C64, job scheduling supports both interactive and batch modes. A C64 system may be partitioned into development and production sections. For a fast turn-around, the development partition may be used interactively while the production partition is restricted to batch submission. In interactive mode, users are granted access to a small number of C64 nodes for the purpose of debugging and/or tuning their applications. Batch jobs that users submit are put into a job queue by the queue manager process. Certain parameters are associated with each job, including priority and resource requirements such as number of C64 nodes. Every time nodes in the production partition are released, the job scheduler wakes up and decides which job runs next. The decision is made based on the list of parameters submitted with the job as well as runtime factors such as time waiting on the queue. In addition, the scheduler invokes a placement algorithm that determines the set of C64 nodes assigned to run a job. Placement accounts for faulty nodes and guarantees the number of nodes that the user requested.

4.2.2 Resource manager

A resource manager is deployed to manage the system resources, including C64 and front-end nodes. Its objective is to minimize system downtime due to hardware failures and hence, to improve system utilization. On a system as complex as the C64 computing environment, the sources of failures are numerous. For instance, a thread unit, floating point unit, or memory bank of a chip may be bad. An entire chip may be inaccessible due to a malfunction of the A-switch or some link of the 3D-mesh may not work as expected. The C64 system software and the resource manager, in particular, detect and try to work around all these and many other issues. For instance, at boot time each C64 node is thoroughly tested to determine its aptitude to run programs. C64 architecture provides a hardware mapping table (accessible to the resource manager only) where bad components are marked and effectively removed from the set of active elements. The resulting chip with a reduced number of resources is still eligible for computation. Similarly, faulty nodes and links may be assigned to partitions allocated to run jobs. However, these may

be avoided by means of a routing algorithm. Given the nodes and links status information generated by diagnostics programs run under the resource manager control, the role of the routing algorithm is to find at least one path between any two C64 nodes in a partition and among C64 and front-end nodes. The ability to remain operational despite hardware failures is unique to the C64 architecture and provides a cost-effective solution with unparallel efficiency among common off-the-shelf microprocessor-based supercomputing systems.

Additionally, the resource manager maintains a central database, which provides a reliable and comprehensive view of the system. Such information simplifies the design of the system software. For instance, the job scheduler requires the knowledge of bad chips to ensure that the user requirement for a minimum number of working nodes is met. In the event of a hardware failure, for instance a C64 chip stops responding during the execution of a program, this view of the system allows recovery in minimum time. As soon as a node within the partition where the job was running is identified as faulty, the remaining C64 nodes are moved again to the pool of available resources. Notice that while the status of a partition is verified, other jobs may be assigned to other partitions independently.

4.2.3 Host to Cyclops-64 communication

The C64 supercomputer is attached to the host system through a number of Gigabit Ethernet links. These links, in addition to the 3D-mesh, support all the communication between C64 and front-end nodes. Therefore, system software developers are required to handle the specifics of both Ethernet and A-switch protocols to carry out any communication successfully. To avoid this trouble a uniform communication protocol layer, called the Cyclops Datagram Protocol (CDP), is added. CDP provides a global address space across the front-end host and the C64 back-end. Based on CDP, application level protocols are implemented, including file I/O, debugging, performance monitoring and host to C64 remote memory communication. For instance, when a C64 node attempts to open a

file, a request is shipped to the front-end in the form of a CDP packet. At the host, a daemon performs the operation on behalf of the back-end and sends the result (file handler) back to the C64 node where the I/O operation originated. The C64 computing engine always starts file I/O operations. However, there are services that the front-end initiates instead. For instance, when a job is scheduled to start execution on a set of C64 nodes, the job scheduler contacts the process control thread running on each C64 node and transfers among other information the program's image, the user environment, command line parameters, etc. All this data communication relies on the CDP protocol as well.

When file I/O processing is expected to be intensive, it would not be judicious to allow the C64 side to drive the computation. That would result in numerous I/O requests being shipped to the front-end that could easily make the Ethernet links the bottleneck of the entire system. To cope with this situation, a novel computing paradigm is supported, in which an application consists of two processes: one running on the front-end, another on the C64 back-end. The former is responsible for I/O and takes care of preprocessing and off-loading computation to the latter, which accomplishes the computational intensive part. Once computation is done, if any post-processing is required the front-end will handle it. We enable this scenario with a remote memory operations library (RMEM) that facilitates inter-process communication (between host and C64 engine). According to our current model, the application part running on the front-end cluster sends data to (push) and gets results from (pull) the C64-side. All the communication and synchronization primitives provided by the RMEM library are implemented on top of CDP.

4.3 Cyclops-64 Toolchain

Figure 4.1 illustrates the software toolchain currently available for application development on the C64 platform. The C compiler has been ported from the GCC-4.1 suite. The assembler, linker and other binary utilities are based on binutils-2.18. To fully exploit C64 multi-layered memory hierarchy, the toolchain is designed to support segmented memory spaces that are not contiguous. In other words, multiple sections of

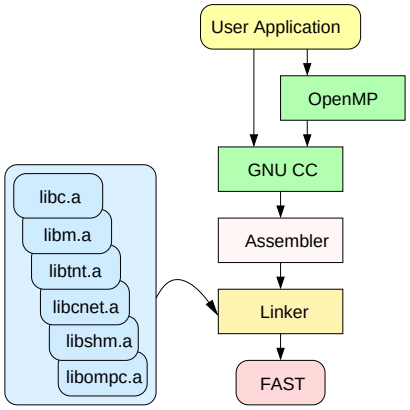


Figure 4.1: Cyclops-64 Software Toolchain

code, initialized, and uninitialized data may be allocated on each memory region, just like in some toolchains for embedded processors. To direct the allocation of sections, pragmas are provided to specify the memory areas where the user would like to place certain variables or procedures. For instance, frequently used data structures can be put in the scratchpad memories, closer to the processor/thread units. In general, applications should be designed keeping in mind the on-chip and off-chip memories latency and bandwidth, so that they make the best use of the memory. The current toolchain with pragma support for segmented memory spaces is the first step towards this goal.

The standard C and math libraries are derived from those in newlib-1.16.0. Functions (libc/libm) are thread safe, i.e. multiple threads can call any of the functions at the same time. Nonetheless, mutual exclusion is guaranteed by efficient spin locks. In addition, memory functions have been optimized, taking into account the memory hierarchy and C64 ISA support for multiple load and store operations that make more efficient use of the memory bandwidth.

The TNT microkernel/runtime system library, discussed in detail in Chapter 5, provides the software and application developer with the functionality to write multithreaded programs: thread management, support for mutual exclusion, synchronization

among threads, etc. In order to achieve high performance and scalability, the implementation of such functionality tries to match the architecture underneath the microkernel/RTS as closely as possible, as explained in the next section.

The CNET communication protocol is also part of the microkernel. This software component controls the A-switch, and supports SHMEM, a one-sided communication library, on top of it. SHMEM provides a shared global address space, data movement operations between locations in that address space, and synchronization primitives that greatly simplify programming on a multi-chip system such as C64.

To carry out our research until a hardware platform is available, we developed FAST, an functionally accurate simulator of a multi-chip multithreaded C64 system. The following section explains the FAST simulator.

4.4 FAST: Cyclops-64 Architectural Simulator

FAST is an execution-driven, binary-compatible simulator of a multi-chip multithreaded C64 system. It accurately reproduces the functional behavior and count of hardware components such thread units, on-chip and off-chip memory banks, and the 3D-mesh network, as shown in Table 4.1. The actual number of simulated chips is limited for practical reasons, because the memory corresponding to all the chips needs to be allocated in the host machine.

Although FAST is not cycle accurate, we have shown that it is useful for performance estimation [16]. In addition, FAST played a critical role in the system software development process as it supported all C64 system software design, development and testing for the past four years and helped in the verification of the C64 logic design.

We developed FAST according to a modular approach, such that additional features could be easily incorporated into the existing design. To help the architecture team with the verification of the C64 chip design, the simulator executes instructions (4.4.1), models the architecture exceptions (4.4.2), reproduces the C64 memory map (4.4.3) and produces histograms of the instruction mix as well as detailed traces of all instructions

Table 4.1: Simulation Parameters

Component	# of units	Params./unit
Threads	160	single in-order issue, 500MHz
FPUs	80	floating point/MAC, divide/square root
I-cache	16	32KB
SRAM (on-chip)	160	30KB
DRAM (off-chip)	4	256MB
Crossbar	1	96 ports, 4GB/s port
A-switch	1	6 ports, 4GB/s port

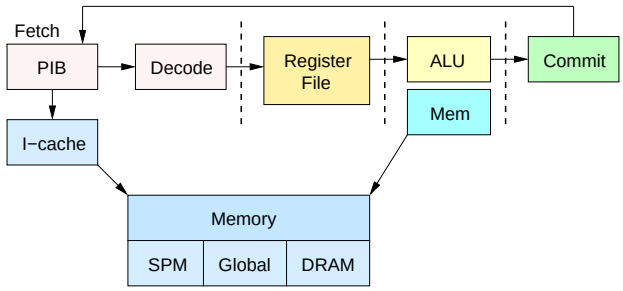


Figure 4.2: Four-Stage Instruction Pipeline

executed (4.4.4). For the purposes of early system and application software design and evaluation, FAST also accounts for memory and interconnect contention (4.4.5), and supports intra-chip communication through the A-switch device (4.4.6). Finally, an overview of the simulator internals is provided (4.4.7).

4.4.1 Instruction execution

FAST simulates the four-stage pipeline employed in the C64 architecture, as shown in Figure 4.2.

At the first stage of the pipeline, an instruction (see Table 4.2) is fetched from the program instruction buffer (PIB) and decoded. FAST may account for the access to the PIB and, should a miss occur, the subsequent delay while the instruction is read from the

Table 4.2: Cyclops-64 Instruction Set Summary

Core Integer and Branch	**Floating Point**
Load, Store	Add, Subtract
Load, Store Multiple	Multiply, Divide
Add, Subtract [Immediate]	Multiply and Add
Multiply, Divide	Conversions
Compare [Immediate]	Square Root
Trap on Condition [Immediate]	
Logic [Immediate]	
Shift [Immediate]	
Shift left 16 then OR immediate	
Insert, Extract	
Move if Condition	
Branch on Condition	
Branch and Link	
Exotic	**Control**
Bit Gather (permute bits)	I-Cache Invalidate
Count Leading Zeros	Move From/To SPR
Count Population	Return from Interrupt
Parity	Sleep
Load then Op	Stop
Move Indirect (register-register)	Supervisor Call
Multiply and Accumulate	

instruction cache or memory. Whenever the branch prediction is incorrect, execution in a thread unit stalls for three cycles while the pipeline is flushed. However, FAST does not reflect the operation of the branch predictor and regards all conditional branches as correctly predicted.

In the second pipeline stage, the instruction input operands are read from the register file. For all the C64 instructions, except the floating multiply and add (FMA), one or two register operands are read in one cycle. FMA instructions have three input operands; hence, an extra cycle is required to read the third operand since the register file has two read ports.

In the third stage, the instruction is executed. RISC-like instructions such as integer, floating-point, branch and memory operations are modeled, based on execution times expressed by x/d pairs, where x is the execution time in the ALU, and d represents the delay before the result of the instruction becomes available. Instruction timing reported in Table 4.3 is based on information provided by the C64 chip design team. For instance, signed integer division is said to take one cycle in the ALU, but a subsequent instruction will not be able to use the result until 69 cycles later. During this delay, execution of independent instructions can proceed normally. However, if the result of an instruction is to be used by another instruction before it is available, the pipeline will stall. It is the compiler and programmer's responsibility to cover these delays as much as possible with the appropriate instruction scheduling.

The result is finally committed in the fourth stage if no exception is generated. Otherwise, a context switch causes execution to continue from the address specified by the interrupt vector. When the results are to be written, conflicts may occur, since the register file has two write ports. However, these events are not expected to happen frequently and FAST does not account for them.

In terms of instruction execution, FAST allows thread units to fetch, decode and execute instructions independently, following the sequence of events dictated by each thread's instruction stream. However, care needs to be taken for some special instructions. The sleep instruction, the wakeup signal, the inter-thread interrupt, etc., all imply a synchronization between threads. For instance, a thread unit, while asleep, does not execute any instructions. During this time, the simulator will not update its clock counter. When a wakeup signal is received, the clock counter is set to that of the remote thread that executed a store in the wakeup memory area (plus some delay). To handle these synchronizations, threads will commit instructions once the simulated chip clock reaches the time point at which the instruction is executed by the thread. In other words, although instructions are executed asynchronously, they are committed in a synchronized fashion.

Table 4.3: Cyclops-64 Instruction Timing

Instruction type	x	d
Bit gather	1	1
Branches	2	0
Count population	2	0
Integer multiplication	1	6
Integer division signed	1	69
Integer division unsigned	1	68
Integer remainder signed	1	70
Integer remainder unsigned	1	69
Move indirect register	3	0
Floating add, subtract and conv.	1	5
Floating multiplication	1	6
Floating multiply and add	1	11
Floating divide double	1	63
Floating divide single	1	34
Floating square root double	1	62
Floating square root single	1	33
Floating mult. and accumulate	1	6
Memory operation (local SRAM)	1	2
Memory operation (global SRAM)	1	31
Memory operation (off-chip DRAM)	1	57
All other operations	1	0

4.4.2 Exception handling

Exceptions are thread-specific events. Some are caused by instructions and trigger what we call synchronous interrupts that cannot be disabled. For instance, an attempt to execute an instruction with an invalid opcode generates an illegal interrupt. Others, known as asynchronous, are caused by events such as a timer alarm and can be disabled. While disabled, only the first exception of each type generated by a sequence of events is held pending; subsequent ones are lost. Throughout the instruction's execution, multiple exceptions of both classes may occur. FAST checks for exceptions at the end of the execution stage. Before the results are written, if one or more enabled exceptions exist,

FAST generates an interrupt according to the priority order specified by the architecture.

4.4.3 Segmented memory space

The C64 chip hardware supports a shared address space model: all on-chip SRAM and off-chip DRAM banks are addressable from all thread units/processors within a chip. That is, all threads see a single shared address space.

Architecturally, each thread unit has an associated 30KB SRAM bank. Each memory bank can be partitioned (configured) into two sections: one called the “global” (or “interleaved”) section, the other the “local” (or “scratchpad”) section. All such global sections together form the (on-chip) global memory in an interleaved fashion that is free of holes and uniformly addressable from all thread units. Although scratchpad memory, global memory and off-chip DRAM memory are addressable from any thread within the chip, the access is not uniform. Besides having different latencies, these three memories have a separate address space, resulting in a three-level hierarchy. Furthermore, there is no virtual memory manager in the C64 architecture, hence, this memory hierarchy is directly exposed to the programmer.

The FAST simulator accurately reproduces the C64 memory map by implementing the above-mentioned, non-uniform shared address space. It also includes the address upper limit special purpose registers (AULx) that define the highest existing location in scratchpad memory, global memory and DRAM memory, respectively. FAST also implements three protection boundary special purpose registers (PBx). These registers define regions in scratchpad, interleaved, and DRAM memory that can only be written in supervisor state, which effectively provide a basic mechanism to protect the kernel against il-behaved programs. In FAST, all memory-specific parameters, such as the number of banks, size of each bank, latency, and bandwidth, are easily configurable.

4.4.4 Execution trace and instruction statistics

Given the appropriate command line option, the toolset generates the execution trace of a program. There are two mechanisms to select the instructions that are to be stored in the trace. The user can either specify the time interval (in clock cycles) for which the program execution is to be traced, or enclose the instructions to be output to the trace within TraceOn/TraceOff macros. These macros access unarchitected special purpose registers (SPRs) that control the simulator's functionality, but are not present in the C64 chip design. The output, consisting of a text file per active thread on the C64 system, contains detailed information such as clock cycle, instruction executed, source and target register values, address of the memory location touched by the instruction, if applicable, and specific information regarding events that could have delayed the execution of the instruction (contention in the crossbar network, operand not available yet, etc).

FAST may also collect instruction statistics over an execution interval and produce histograms of the instruction mix. Similar to the procedure available for tracing, the user can specify an interval in clock cycles or use StatsOn/StatsOff macros to start/stop collecting statistics, respectively. A combined report for each node, as well as individual reports for all active threads, are generated.

4.4.5 Memory and interconnect contention

One of the latest additions to the FAST simulator is a module that accounts for the contention in the crossbar network and in the memory system.

Figure 4.3 illustrates the data path between processors and memory banks on a C64 chip. Every memory instruction executed on a processor results in a network packet delivered by the crossbar network to the appropriate memory bank (global SRAM or off-chip DRAM). For load operations, the memory replies with another packet containing the data retrieved from memory.

In order to obtain reasonable accuracy without increasing too much the simulation time, FAST models the following sources of contention.

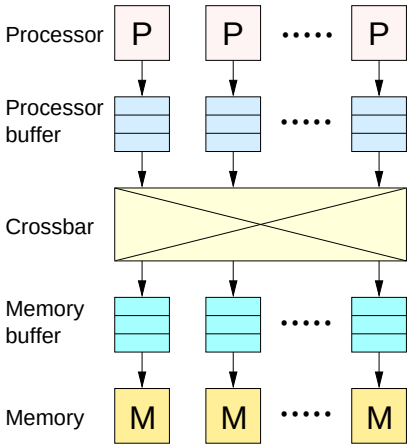


Figure 4.3: Interconnection to the On-Chip Crossbar

- Packets issued by threads on the same processor are queued on a 7-slot FIFO (processor buffer) until they are retrieved by the crossbar. If a thread issues a memory operation when the FIFO is full, the pipeline will stall until space is available.

- The crossbar retrieves packets from the input ports and delivers packets to the output ports, one per cycle. If at the same cycle, two packets are to be delivered to the same output port, the crossbar blocks one of them arbitrarily.

- Between the crossbar and each memory bank, there is another 7-slot FIFO (memory buffer) where packets are held until processed by the memory. Whenever this buffer becomes full, the crossbar stops delivering packets to this destination. At the same time, it stops retrieving packets from any input that tries to send packets to the blocked output port.

- Memory latencies are also taken into account. SRAM memory banks can perform a load or store operation every cycle, i.e., 4GB/s per bank. Whereas DRAM memory can sustain a much lower bandwidth. DRAM memory consists of four banks, and each bank is subdivided into four subbanks. Subbanks can service requests simultaneously, one every 57 cycles, on average. While a memory subbank is in service,

any incoming request is held pending in the memory buffer. Therefore, the DRAM bandwidth is 2GB/s for single loads and stores. For multiple transfers, using load multiple (LDM) and store multiple (STM) instructions, the DRAM bandwidth is 16GB/s instead.

4.4.6 A-switch device

In FAST, the functioning of the A-switch communication device is simulated at a functional level only. When a chip has an A-switch message to send, the simulator copies the whole message directly to the destination node. In other words, the simulator does not model the details of all the hardware mechanisms involved in transferring packets, double word by double word, through the 3D-mesh network.

In addition to not accounting for the interaction among multiple C64 chips, FAST does not account for the interaction between the A-switch and the crossbar network. Sending or receiving messages via the A-switch does not cause any disturbance in the crossbar network. Therefore, performance estimations obtained with FAST for multi-chip simulations should be regarded as less accurate than single-chip simulations.

4.4.7 Simulator internals

The simulated C64 system starts running when one of the three main simulator functions is called. To maximize performance, each function specifically handles a C64 system consisting of a single processing core, a C64 chip fully populated, or a system built out of several nodes. Therefore, the decision is simply based on the system configuration.

In multi-node simulations, the main function starts with a loop that iterates over all the active threads on all the nodes. Each thread unit attempts to execute an instruction. For a new instruction, the simulator calls functions responsible for the instruction fetch, instruction decode, read the input operands from the register file, and instruction execution. If the thread unit is asleep, stalled waiting for an operand or due to a resource hazard, or waiting to commit the previous instruction, it does nothing but return.

Back in the main function, the chip clock is moved forward, just enough to allow one thread unit, at least, to commit the current instruction. Once the clock is updated, the crossbar and memory banks proceed to flush packets and memory operations that are to be performed by this time.

Then a second loop iterates over all the threads, regardless of their status. First, thread units check whether an exception occurred, and if it did, the corresponding interrupt is serviced with the appropriate context switch. If no interrupt was triggered, they try to commit the last instruction. At this stage, threads compare the chip clock with their own internal clock. When the execution on the chip reaches the time step at which a thread can commit an instruction, the results are written. Otherwise, the thread waits.

Finally, after the status of the A-switch is updated, execution returns to the beginning of the main loop. The process is repeated until thread units on every node execute the STOP instruction in supervisor state.

To simplify the communication among components of the simulator, the representation of the simulated C64 system is kept in a single multi-level data structure. At the chip level, it contains information regarding thread units, floating point units, on-chip SRAM and off-chip DRAM memories, I-caches, the crossbar model, and the A-switch. At the thread level, it accounts for general, special purpose, and accumulator registers, in addition to timing information as to when the value stored in a general purpose register will be available, the last decoded instruction, program counter, exception flags, thread status, and a third-level data structure with statistics counters.

Chapter 5

TNT: CYCLOPS-64 THREAD VIRTUAL MACHINE

In this chapter, we present the design and implementation of TiNy Threads (TNT), the Thread Virtual Machine for the Cyclops-64 architecture. We highlight the features of TNT as follows:

- TNT replaces the conventional OS with a custom-made kernel: Instead of trimming a conventional OS such as Linux, the C64 kernel and the TNT library have been implemented from the ground up. Only the functionality that is crucial to achieve and sustain high levels of application performance has been included.

- TNT is a non-intrusive runtime system: TNT is implemented as a user-level library that manages the hardware resources directly. TNT supports a non-preemptive thread execution model needed for applications to achieve full resource utilization.

- TNT provides an efficient Linux-like programming environment: TNT relocates services to the user-layer to simplify the runtime software environment and to make it more efficient. TNT also supports a familiar fork/join programming API for quick prototyping of parallel applications.

- TNT supports the development of program execution models: TNT does not impose any limitation on the number of threads available for parallel programming models and applications. TNT seamlessly orchestrates dozens of hardware thread units and thousands of virtual threads with high efficiency.

Given the C64 special hardware features described in Chapter 3, it is not our intention to develop a conventional OS for this platform. Instead, we focus our efforts on the design and implementation of a Thread Virtual Machine that provides a familiar but efficient application program interface. In the following sections we discuss the design and implementation of TNT. In Section 5.1, we present a high level overview of the three key components of the C64 TVM; thread, memory, and synchronization models, and we discuss implementation details in Section 5.2.

5.1 TNT Design

The Cyclops-64 Thread Virtual Machine (TVM) can be seen as an multi-chip multiprocessor "extension" of the C64 ISA. It has been designed to replace the OS with a narrow interface layer. Such a layer of system software directly manages the hardware resources and provides an interface that shields the application programmer from the complexity of the architecture whenever possible. However, unlike a conventional OS, a TVM exposes those resources that are critical to achieve performance.

The C64 TVM not only provides an abstraction layer and the application program interface expected by programmers, it also provides the common baseline for future research on program execution models. In Chapter 6, we illustrate how a memory-adaptive program execution model can be mapped to the TNT TVM to efficiently run without interference from the OS.

The C64 TVM includes three components: a thread model, a memory model and a synchronization model, as well as their corresponding APIs. The thread model presents thread management issues. The memory model includes the specification of the consistency model for the C64 system. The synchronization model describes the functionality to implement mutual exclusion regions, and perform direct thread-to-thread and barrier type of synchronization.

5.1.1 Thread Model

A program section, namely a function, can be declared as a thread. A thread can be activated for execution by binding to a hardware thread unit within a certain chip, a thread activation pointer. This activation pointer is defined as the tuple: <program pointer, state pointer>. The program pointer is the address specified by the program counter associated with the corresponding hardware thread unit, and the state pointer is the thread specific information stored in the C64 memory map (e.g. stack pointer, etc.)

A thread activation pointer can also be "global" if the thread handler is extended with a node (or chip) identifier; a system-wide identifier of the chip where the corresponding thread unit resides. The binding of a thread activation to a thread unit can be dynamic as long as the system software properly maintains the binding information.

Thread model API

In the first release of TNT, we provided an interface inspired by the popular Pthread model, to ease application and system software developers' first hands-on experience. Initially, the user is responsible for creating, terminating and synchronizing threads by inserting appropriate function calls to the TNT runtime library. Subsequent releases, in addition to the Pthread-like model, supported a Single Program Multiple Data (SPMD) style of execution. In this mode, TNT spawns all the threads available on a chip when a program starts running. In this mode, the user has the option to reserve a number of thread units for other purposes. The SPMD mode of execution demands less effort on behalf of the programmer, since TNT automatically manages all the threads. We now describe some functions that are part of the TNT API.

- **tnt_create(tnt_desc_t *th, const void *(*fn)(void *), const void *arg)**

  Runs the user provided function (`fn`) in the next available thread unit. If the thread cannot be spawned the function returns an error condition, otherwise it returns 0 and `th` points to a thread unique identifier (descriptor). One parameter (`arg`) can be passed to the thread function.

- **tnt_exit(const void *rc)**

  The caller thread terminates its execution returning and the exit code specified by `rc` is made available to any successful join with the terminating thread.

- **tnt_join(const tnt_desc_t th, void **th_ret)**

  The caller waits for the target thread to terminate. If it returns successfully, the value passed to `tnt_exit` by the terminating thread will be placed in the location referenced by the parameter `th_ret`.

- **tnt_self(void)**

  Obtains the descriptor of the current thread.

5.1.2 Memory Model

TNT employs a memory consistency model close to the underlying C64 architecture support.

The most widely accepted memory model for the multiprocessor machine is Lamport's sequential consistency (SC) model. Lamport described it in the following well-known statement:

> [A system is sequentially consistent if] the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program order [40].

The above quote becomes the commonly used definition of sequential consistency in most textbooks and research papers.

Under the current C64 single-chip architecture design, the following two conditions are valid:

1. Each processor issues memory requests in the order that its program specifies.

2. Two operations designated to the same memory module M will be delivered to M's input FIFO queue in the same order as they entered into the network.

Notice the latter refers to the time a memory request enters into the network, not when it is issued by a processor, and it is true due to the equal-latency property of Cyclops' intra-chip network.

It has been shown that the above two conditions are sufficient to ensure that the C64 architecture behaves as sequentially consistent [63]. Because the C64 architecture is sequentially consistent, there is no need to issue fence-like instructions after each memory operation to ensure SC.[1] However, the hardware cannot guarantee a "Lamport order" of the accesses to the scratchpad memory space; hence no sequential consistency can be assumed.

Each thread has a private memory region (in scratchpad memory), which can be used by the thread as its local storage for shared variables that reside in the shared memory space. The allocation, management and synchronization needed to keep the consistency between shared and private memory is solely the user's responsibility.

5.1.3 Synchronization Model

TNT synchronization model includes support for several types of synchronizations. The first type of synchronization is used to ensure mutual exclusion of memory accesses to shared memory locations/space. This can be expressed using TNT lock and unlock operations, which are directly implemented using C64 hardware atomic test-and-set operations. Users can declare spin lock variables using the TNT library and operate upon them with the functions provided for that purpose. In addition to spin locks, TNT supports the mutex construct. It is up to the programmer to decide which construct is more appropriate given the application at hand.

[1] In fact C64 has no sync instruction.

```
 1  void producer(tnt_desc_t consumer_th)
 2  {
 3    while(1) {
 4      produce_data();
 5      tnt_signal(consumer_th);
 6    }
 7  }
 8
 9  void consumer(tnt_desc_t producer_th)
10  {
11    while(1) {
12      tnt_wait(producer_th);
13      consume_data();
14    }
15  }
```

Figure 5.1: Producer-Consumer Sample Program

A second type of synchronization in TNT is introduced to express precedence relations between operations from two different threads. In TNT there is a signal-wait type of synchronization that will be placed between a pair of specific program points within the two threads.

The sample program in Figure 5.1, based on a producer-consumer model, shows the basic use of the signal/wait primitives. The `producer` thread produces data that the `consumer` thread consumes. The latter starts by calling `tnt_wait` and blocks until a signal from the thread, whose thread handler matches that given as argument to the function, is received. The former produces a datum and sends a signal to the thread, whose thread handler is specified by `tnt_signal`'s only parameter. Once the signal is received, the `consumer` thread is awakened and consumes the datum.

A third type of synchronization is a collective synchronization in which a group of threads will participate. For example, a barrier synchronization primitive can be invoked by a group of threads. Threads block until all participants in the operation (participants are defined by a single object passed as parameter to the barrier function) have reached this routine.

The function in Figure 5.2 is part of a TNT program that uses a barrier primitive. Multiple threads execute the `worker` routine, which starts with each thread generating

```
1  void worker(int set_id, tnt_barrier_t barrier)
2  {
3    produce_data_set(set_id);
4    tnt_barrier(barrier);
5    if (set_id == 0)
6      reduce_data_set();
7  }
```

Figure 5.2: Barrier Sample Program

some data according to a thread-specific parameter. Once all the data has been generated, an unspecified operation is applied to it (in our example it is some type of reduction). Before the operation can be applied, we must ensure that all threads have produced the corresponding data. For that purpose, we call the `tnt_barrier` function, so all threads block until all the threads participating in the barrier reach the same point before continuing.

5.2 TNT Implementation

The TNT library defines two layers of thread management: the hardware thread layer and the software or virtual thread layer, as shown in Figure 5.3. The hardware layer provides direct access to the hardware resources. It manages the hardware thread units (HT) and implements a non-preemptive, thus non-intrusive runtime system. The virtual layer handles thousands of software or virtual threads (VT) on behalf of the application. It provides a familiar fork/join programming API that is simple to use, yet sufficiently general to write multithreaded applications. The integration of these two simple, but well structured layers of thread management,[2] makes it possible to replace a conventional OS with TNT on C64. We now describe these two layers in more detail.

5.2.1 Hardware Threads (HT)

The hardware thread units can be idle or active. While idle, a thread unit remains asleep so that it consumes little power and does not waste memory bandwidth. Once

[2] Less than 3,000 lines of code including header files and comments.

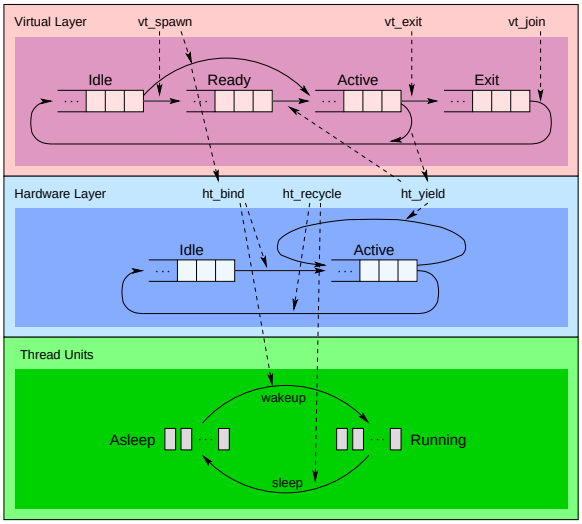


Figure 5.3: TNT Flow Chart

active, a thread unit starts running the user program. The hardware thread layer supports a non-preemptive thread execution model. Once a thread unit starts running, it will continue until the user explicitly relinquishes control over the thread unit or the program performs an illegal operation (core dumps).

Upon initialization, each physical thread unit is given control over its own scratch-pad memory. The 32 bytes at the beginning of the scratchpad memory are reserved for the hardware thread descriptor. The scratchpad memory area above the reserved area is allocated for the thread stack. The compiler and runtime system share a general purpose register, which points to the end of the stack and beginning of the reserved area. This register is used for two purposes: (1) check for stack overflows, (2) provide a fast self-identification mechanism for hardware threads. The TNT library allocates one stack per hardware thread unit at boot time. Therefore, when a software thread is about to start or terminate execution the stack does not need to be relocated, which allows faster thread management.

5.2.2 Virtual Threads (VT)

The virtual layer receives requests to create and/or terminate software or virtual threads (VT) as required by the application. To manage the software threads, the virtual library defines a thread descriptor that is usually allocated in the on-chip memory. If needed, virtual descriptors could be allocated in off-chip memory as well. The virtual layer leverages on the fast self-identification method provided by the hardware layer. When a virtual thread is bound to a thread unit, pointers in the hardware and virtual thread descriptors are setup to point to each other. In this way, self-identification of a virtual thread involves identification of the thread unit and dereferencing the pointer to the virtual descriptor.

Once ready for execution, virtual threads are bound to hardware thread units as they become available. When this happens, TNT implicitly assigns the stack associated with the thread unit to the virtual thread. This method of stack assignment, together with the fast self-identification mechanism, are the two characteristics for which *Descriptor on Stack* (DOS) enabled libraries are known to be fast [58]. Upon thread termination, TNT reuses the stack, which was assigned to the hardware thread unit at boot time, to run the next virtual thread. In that sense TNT is memory efficient like *L*azy Stack Allocation (LSA) + *D*irect Stack Reuse (DSR) enabled libraries [58].

5.2.3 Thread Scheduling

From a scheduling standpoint, one of the features of TNT worth noting is that hardware and virtual threads are scheduled at the same time. Since thread execution is non-preemptive, a virtual thread can only run when a hardware thread unit is available, and a thread unit remains active as long as there are virtual threads to be executed.

The thread scheduling algorithm that the TNT library implements is as follows. To launch the user application the runtime system creates a virtual thread descriptor for the main function of the program and schedules the master thread to start its execution. After the main function returns, the master thread is rescheduled and starts retrieving work

with the other hardware thread units. While the program is running, the virtual layer may receive a request to spawn a thread. After creating a descriptor to handle the new software thread, the virtual layer calls the hardware layer, which determines whether a hardware thread unit is available. If a thread unit is idle, virtual and hardware threads are bound and execution starts immediately. Because execution is non-preemptive, any virtual thread runs to completion on the thread unit initially assigned. The hardware thread is awakened and given the address of the virtual thread's entry point. However, if there are not thread units idle, i.e. all the thread units are active already, the thread descriptor is pushed to a queue of virtual threads ready to run. Once a software thread finishes execution, the virtual descriptor is recycled and the thread unit is returned to the runtime system. The hardware unit then checks whether threads are waiting to run. If there are no threads ready at that moment, the hardware unit goes to sleep, until another request is received. If virtual threads are pending for hardware resources, the hardware unit is reassigned to a new virtual thread and execution starts without suspending the hardware thread. Once all the virtual threads have been executed and the hardware thread units do not have more work to do, the TNT library returns control to the C64 kernel.

TNT does not implement the scheduling algorithm in a centralized fashion. In other words, the TNT library does not reserve a thread unit to carry out the task of scheduling hardware and virtual threads. To minimize overhead and achieve scalability, the TNT runtime system allows any thread unit that returns to the runtime system to run the thread scheduler code. As a consequence, requests for thread creation may arrive at the same time a hardware thread unit is being recycled and trying to determine whether software threads are ready. In some rare conditions, it is possible that the hardware thread gets suspended and goes to sleep assuming there are no additional virtual threads, at the same time a virtual descriptor is pushed into the virtual ready queue assuming no hardware resources are available. However, as soon as the application attempts to spawn another

virtual thread, the thread unit that mistakenly remained idle will be awakened. In a system with only two thread units, the above situation implies a temporary loss of 50% of the hardware resources. On a multicore architecture such as C64 with 160 hardware thread units, the loss is negligible so did not implement a work around to avoid introducing additional overhead in the thread scheduler.

Chapter 6

MAGMA: A MEMORY-ADAPTIVE MULTITHREADED ARCHITECTURE MODEL

This chapter defines the MAGMA (Memory Adaptive Multithreaded Architecture) Program Execution Model (PXM) as the interface between users (e.g. programmers and compilers) of high-level languages, and the implementation of a computing system. An abstract model describing the operations involved in the execution of a multithreaded program on a Cyclops-like cellular architecture illustrates this definition.

MAGMA proposes a memory-centric computing model (as opposed to a processor-centric model), in which the memory latency and bandwidth determine the computing pace. The relationship between processors and memory is organized for parallel applications to effectively manage and tolerate the latency and bandwidth constraints of the multi-level memory hierarchy present in modern cellular computing systems.

Section 6.1 provides an overview of the MAGMA Program Execution Model and illustrates its basic features through program examples. Section 6.2 precisely defines the MAGMA Program Execution Model, including the thread model and the MAGMA operations. Section 6.3 describes MAGMA main features by comparing the MAGMA, EARTH and Cilk models.

6.1 Introduction to MAGMA

This section introduces the basic features of the MAGMA Program Execution Model and illustrates their use through program examples. It begins with simple but illuminating examples (the ubiquitous $Hello\ \ World$ program and simple

```
 1  #include <stdio.h>
 2  #include <magma.h>
 3
 4  void print_hello(void)
 5  {
 6    printf("From PE %d: Hello World!\n", MY_PE);
 7  }
 8
 9  void main(int argc, char *argv[])
10  {
11    int i;
12    for (i=0; i<NUM_PES; i++) {
13      SPAWN(print_hello);
14    }
15  }
```

Figure 6.1: MAGMA *Hello World* Program

$Producer$-$Consumer$ programs) and ends with a small but real application, *daxpy*, a level 1 BLAS routine that computes a linear combination of two vectors. We intentionally leave the discussion of the advanced features for Section 6.3.

6.1.1 Thread Execution

Like sequential programs, MAGMA programs have a `main` function. This function accepts the same arguments as the sequential `main`. When a program starts running, the first thread that the system spawns executes the `main` function. However, a MAGMA program does not necessarily stop running after `main` returns. On the contrary, a MAGMA program runs as long as threads spawned by the application remain active, unless the program purposely exits or aborts the execution.

Figure 6.1 shows the code for a simple $Hello\ World$ program. In lines 12–13, the `main` function tries to spawn a thread on each processing element of the machine. The `SPAWN` command starts execution of a thread function on any available processing element. The `SPAWN` arguments consist of the name of the function we intend to run and the function's arguments.

In MAGMA, threads cannot return a value. For this reason, in lines 4 and 9 the return type is simply `void`. When the program creates a thread, MAGMA allocates a frame

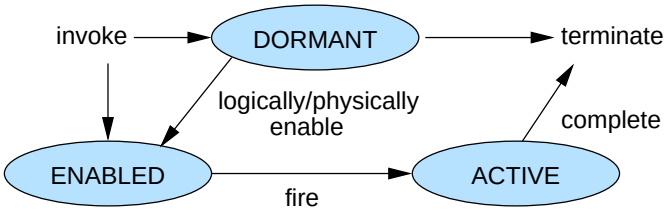


Figure 6.2: MAGMA Thread States

from the heap for a thread handle. However, the handle is only needed until the thread starts running. Right before that happens, the information stored in the thread handle, such as the thread arguments, is copied to the runtime stack and the handle is released. In MAGMA, thread execution is non-preemptive. Once a thread starts execution, it runs to completion. Unlike a sequential function, a thread can be spawned without providing all the arguments at the thread invocation site. However, the thread will not start execution until all the arguments are available, as if they had been supplied when the thread was first invoked. Continuing with the $Hello\ World$ example, `print_hello` does not have any arguments. This means that this thread function can start running immediately, as long as there is a processing element available.

The `SPAWN` command creates a thread. Arguments will be produced either before or after the call to `SPAWN`, in which case they can be provided either when the thread is spawned or afterward using thread synchronization operations. We say a thread is *logically* enabled after values for all the thread parameters have been supplied. Until then, we say a thread is *dormant*. If a thread is *logically* enabled, it can be scheduled for execution. As we explain in Section 6.1.3, MAGMA allows the user to specify a stronger condition before a thread can be fired. That is when a thread is not only *logically* enabled but it is also *physically* enabled.

Figure 6.2 shows the complete state diagram for a MAGMA thread. When a thread is invoked and all the arguments are initialized, the thread begins in the *enabled* state. However, if an argument is left uninitialized, the thread begins in the *dormant* state. Once the values for all the arguments are produced, a *dormant* thread becomes

enabled and is scheduled for execution. An *enabled* thread is moved to the *active* state when a processing element, on which the thread runs, is available. When an *active* thread completes execution, the runtime system moves it to the *terminate* state.

To help a program determine at runtime the number of processing elements available, as well as the processing element where a thread is running, MAGMA defines the following two integer constants:

- `NUM_PES` – The number of processing elements that are executing the program.
- `MY_PE` – The identifier of the processing element ($0 \leq$ `MY_PE` $\leq$ `NUM_PES-1`).

These constants are initialized when the program is loaded and cannot be modified throughout the program execution. Given a machine with a number of processing elements, the $Hello\ World$ program will output as many "From PE #: Hello World!" strings as there are processing elements. However, it is quite possible that we observe duplicates of PE values; in other words, a processing element may print more than one message. The `SPAWN` command creates a thread on behalf of the $Hello\ World$ program that starts execution in the first processing element that becomes available. It is possible that a thread completes execution of `print_hello` by the time the `main` function spawns another thread. In this case, MAGMA may spawn the second thread on the same processing element; that is why we see duplicate PEs. In addition, the $Hello\ World$ program runs until all the threads spawned by the program complete. That means that by the time `main` returns in line 15, `NUM_PES` threads would have been executed and output a message to the terminal.

6.1.2 Thread Synchronization

In this context, synchronization refers to the mechanism that allows the programmer to impose a certain order in which threads execute. For instance, if we have two threads, `THREAD_1` and `THREAD_2`, and there is a data dependence to indicate that

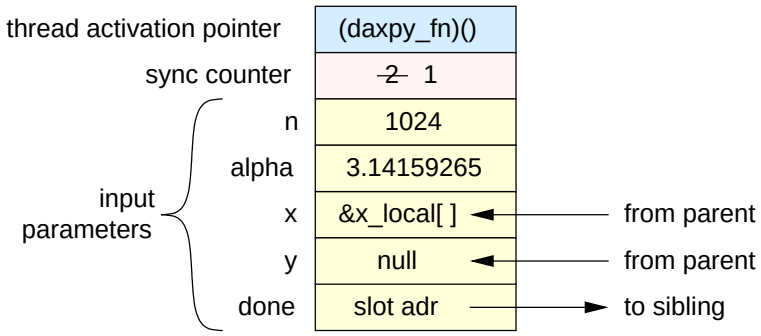


Figure 6.3: MAGMA Thread Handle

`THREAD 1` must produce a datum to be consumed by `THREAD_2`, then we must use synchronization to guarantee that `THREAD_1` executes and produces the needed datum before `THREAD_2` can start.

In MAGMA, synchronization slots handle thread synchronization, and are the key element of the thread handle. In addition to the thread activation pointer, the thread handle has as many slots as the function has arguments, as well as a synchronization counter, see Figure 6.3. Arguments for which `SPAWN` provides a value are copied into the corresponding slot of the thread handle. The remaining slots are left uninitialized, waiting for the value to be supplied at runtime. These are the synchronization slots. In addition, the synchronization counter is initialized with the number of synchronization slots. Only when the synchronization counter reaches zero is the thread eligible to start execution. The thread handle shown in Figure 6.3 represents a snapshot of the thread `daxpy_fn`'s handle, as shown in Figure 6.6. The synchronization counter is initialized to 2, since the addresses of vectors x and y are to be provided at runtime. After the address of vector x is provided, the synchronization counter is decreased to 1. This is the state depicted in Figure 6.3.

As noted above, threads run to completion. However, before a thread is scheduled for execution, values for all the thread parameters have to be supplied. When a thread is spawned, some arguments may not be available. An argument that is not provided at

```
 1  #include <magma.h>
 2
 3  void producer(SYNC_SLOT int sync_result)
 4  {
 5    int result = produce();
 6    DATA_SYNC(result, sync_result);
 7  }
 8
 9  void consumer(int result)
10  {
11    consume(result);
12  }
13
14  void main(int argc, char *argv[])
15  {
16    SPAWN(consumer, SYNC(res));
17    SPAWN(producer, SLOT_ADR(res));
18  }
19
```

Figure 6.4: Thread Synchronization

thread creation time represents a dependence that will be satisfied at runtime. The programmer declares a *missing* argument by simply using the SYNC keyword instead of the argument in the SPAWN call. The SYNC macro accepts an optional parameter. The parameter, enclosed in parenthesis, is a keyword used to uniquely identify each SYNC macro. After a *missing* argument is defined, it can referenced with the macro SLOT_ADR. The compiler pairs SLOT_ADR with SYNC macros to determine the position of the *missing* argument within the thread argument list. Based on the argument's position, the compiler computes the offset of the *missing* argument slot within the thread handle, and therefore the address of the synchronization slot needed by subsequent thread synchronization operations. When a thread is spawned, the global counter is initialized with the number of *missing* arguments. At runtime, as synchronization among threads occurs, the counter is decreased by one unit every time the value of an argument is produced. Once the counter reaches zero, meaning that all the dependencies have been satisfied, the thread is eligible for execution.

The simple program shown in Figure 6.4 demonstrates how to use the SYNC operator to define a dependency and synchronization between two threads.

Line 16: The program spawns the consumer thread. However, no argument is passed

to the SPAWN command. Instead SYNC(res) indicates the argument is missing, and it is expected to be provided at runtime. Note that res is not a variable, but an identifier that the compiler needs to match SLOT_ADR and SYNC commands, as explained below.

Line 17: The program spawns the producer thread. Note there is not an argument either but the keyword SLOT_ADR(res). SLOT_ADR is a built-in MAGMA compiler command. The compiler pairs SLOT_ADR and SYNC macros. When the identifiers match, the compiler replaces SLOT_ADR with the actual address of the synchronization slot corresponding to the missing argument. In our example, the address of result's synchronization slot of the consumer.

Line 3: The keyword SYNC_SLOT precedes the input argument. The purpose of this keyword is to indicate that the argument is a pointer to a synchronization slot, expected to be filled in with an integer number.

Line 6: Synchronization occurs between producer and consumer threads. The function DATA_SYNC sends the scalar value result to the consumer synchronization slot. Once the value is stored in the synchronization slot, the global counter is decreased. Since only one argument was missing, the counter becomes zero, and the runtime system schedules the consumer for execution.

Suppose that in order to solve a problem, it has to be decomposed into a relatively large but fixed number of subproblems, let us say N. Let us further suppose that the results from all these subproblems must be combined and processed all together to determine the final answer. Initially, the declaration of the consumer thread function would have to have N parameters, one for each value to be produced. Declaring functions with a large number of input arguments is not only cumbersome, but also prone to errors. To improve programmability, MAGMA supports indexed synchronization slots that are defined using the notation "<* ... *>". The notation attached to a parameter in a function declaration tells the compiler that the function has multiple parameters.[1] The same notation attached to a function argument tells the compiler that multiple arguments

[1] The difference between multiple arguments and variable argument lists is that with multiple arguments, the number of arguments cannot change from one thread invocation to another. The MAGMA compiler may translate multiple arguments to variable argument lists. However, that is an implementation detail.

```
 1  #include <magma.h>
 2  #define N (NUM_PES)
 3
 4  void producer(SYNC_SLOT int sync_result)
 5  {
 6    int result = produce();
 7    DATA_SYNC(result, sync_result);
 8  }
 9
10  void consumer(int result <*N*>)
11  {
12    int i;
13    for (i=0; i<N; i++) {
14      consume(result[i]);
15    }
16  }
17
18  void main(int argc, char *argv[])
19  {
20    int i;
21    SPAWN(consumer, SYNC(res<*N*>));
22    for (i=0; i<N; i++) {
23      SPAWN(producer, SLOT_ADR(res<* i *>));
24    }
25  }
```

Figure 6.5: Thread Synchronization with Indexed Slots

are missing. Finally, the user can also refer to each of the individual synchronization slots with the built-in command SLOT_ADR. In this case "<* ... *>" encloses a variable or constant that specifies the actual synchronization slot.

Figure 6.5 shows another version of the $Producer$-$Consumer$ program. In this example there are multiple producers and a single consumer synchronized through indexed synchronization slots. The following explains the use of indexed synchronization slots in this example:

Line 10: Defines indexed synchronization slots. It declares the consumer thread with N parameters.

Line 21: The program spawns the consumer thread. All the arguments expected by the consumer are declared as missing.

Line 22: The main function spawns N producers.

Line 23: Each producer receives a pointer to a synchronization slot. The compiler computes the offset to each particular synchronization slot using the index i.

Line 7: The synchronization operation does not change to handle indexed synchronization slots. After all, the producer gets a pointer to only one synchronization slot.

6.1.3 Data Percolation

The previous examples demonstrated how threads use synchronization to pass simple data. MAGMA allows parallel applications to exchange blocks of data as well. MAGMA provides an explicit `BLK_SYNC` operation to move contiguous blocks of arbitrary size. Similarly, `GATHER_BLOCK_SYNC` moves blocks of data of arbitrary size but the data need not be located in a contiguous block. This operation is described in detail in Section 6.2.2. Since this section only covers the basic features, we will focus on the `BLK_SYNC` operation. Even though it is a reduced version of `GATHER_BLOCK_SYNC`, `BLK_SYNC` is more commonly used.

Threads cannot run until they are *logically* enabled. In other words, threads cannot start execution until all the data dependencies have been met. Because threads run to completion, scheduling a thread while arguments are missing will just cause the thread to stall at some point in the middle of the computation.

The `DATA_SYNC` operation copies a scalar value to the target synchronization slot. When the destination thread receives the synchronization signal, the scalar value is already *local* and the thread can be executed without experiencing any long delay. However, when a thread accesses a block of data, i.e., the thread parameter refers to an array, it may not be wise to fire the thread that is only *logically* enabled. If the block of data is in off-chip memory, the thread will experience continuous delays due to the higher latency of this memory. In a distributed memory system, a thread may not be able to directly access the data located in a different node. Therefore, the programmer needs to be cautious with *logically* enabled threads. Instead of simply providing the pointer to remote memory where the data block is, which *logically* enables the thread for execution, it would be better to first migrate the data closer to the processing element and then

provide the pointer to local memory. In MAGMA, after this process takes place, we say that the thread is *physically* enabled.

`BLK_SYNC` is the operation that handles data block dependencies, and ensures that a thread is not fired until it becomes *physically* enabled. `BLK_SYNC` takes three arguments: the source memory area, the number of bytes to transfer, and a pointer to the target synchronization slot. Note that it does not have a target memory area. To guarantee that the block of data is local, `BLK_SYNC` first allocates memory in the level of the memory hierarchy considered to be *local* to the processing element. After memory is allocated, the data block is transferred, for instance from off-chip to on-chip memory. Then the target synchronization slot is synced, and so the function argument is filled with a pointer to the temporary buffer. At that moment, the thread is not only *logically* enabled, but it is also *physically* enabled, and now is ready for execution.

Figure 6.6 shows an implementation of the *daxpy* routine that computes a linear combination of two vectors, i.e., a constant alpha times a vector plus another vector ($y_i = \alpha \times x_i + y_i$). It is a simplified version in the sense that it does not have the integer arguments inc_x and inc_y for the increment between elements of vectors x and y, respectively. The code is a straightforward implementation using the iterative programming paradigm to demonstrate how `BLK_SYNC` works.

In this example, we assume that the user can take advantage of the regularity of the loop structure and the data access pattern and uniformly distribute the computation among processing elements. In order to simplify the example and without loss of generality, we also assume that the length of the vectors is a multiple of the number of processing elements. The main aspects of the *daxpy* implementation according to the MAGMA model can be summarized as follows:

Line 24: Given our assumption that the vector length is a multiple of the number of processing elements, we evenly divide the computation into chunks.

Lines 26–31: The main loop of the *daxpy* routine spawns twice as many threads as there are processing elements. The `daxpy_fn` threads each do part of the computation,

```
 1  #include <magma.h>
 2
 3  daxpy_fn(int n, double alpha, double x[], double y[],
 4           SYNC_SLOT double * done)
 5  {
 6    int i;
 7
 8    for (i=0; i<n; i++) {
 9      y[i] = alpha * x[i] + y[i];
10    }
11    RELEASE(x);
12    DATA_SYNC(y, done);
13  }
14
15  scatter_fn(int n, double y_rem[], double y_loc[])
16  {
17    memcpy(y_rem, y_loc, n*sizeof(double));
18    RELEASE(y_loc);
19  }
20
21  daxpy(int n, double alpha, double x[], double y[])
22  {
23    int i;
24    int chunk_len = n/NUM_PES;
25
26    for(i=0; i<NUM_PES; i++) {
27      SPAWN(scatter_fn, chunk_len, &y[i*chunk_len], SYNC(y_local));
28      SPAWN(daxpy_fn, chunk_len, alpha, SYNC(x), SYNC(y), SLOT_ADR(y_local));
29      BLK_SYNC(&x[i*chunk_len], chunk_len*sizeof(double), SLOT_ADR(x));
30      BLK_SYNC(&y[i*chunk_len], chunk_len*sizeof(double), SLOT_ADR(y));
31    }
32  }
```

Figure 6.6: MAGMA *daxpy* Program

whereas the `scatter_fn` threads will copy the y vector back to its original location in memory. After a `daxpy_fn` thread computes its assigned chunk of vector y, a `scatter_fn` thread can start transferring the resulting y chunk back. The dependency between threads `daxpy_fn` and `scatter_fn` is communicated explicitly to the compiler with the macros `SYNC(y_local)` and `SLOT_ADR(y_local)`. Note that `y_local` is neither a variable nor a constant. It is only an identifier that the compiler uses to match the `SLOT_ADR` in line 28 with the `SYNC` in line 27.

Line 28: The `daxpy_fn` function expects vectors x and y of length n. However, these arguments are defined as missing with the `SYNC` macro.

Lines 29–30: The `daxpy` routine knows the address of vectors x and y when it spawns the `daxpy_fn` threads. However, these vectors are probably not in local memory. In order to *physically* enable the `daxpy_fn` threads, the `daxpy` routine invokes two `BLK_SYNC` operations to transfer chunks of vectors x and y from their current location (probably in remote memory) to a level of the memory hierarchy local to the processing element. Triggered by a `BLK_SYNC` operation, the system allocates

some temporary storage, and once the block of data has been transferred, it will synchronize the corresponding slot of a `daxpy_fn` thread.

Lines 8–9: Once chunks of x and y are local to `daxpy_fn`, i.e., the thread is *physically* enabled, it performs the normal *daxpy* computation.

Line 11: Releases the temporary storage that holds the local copy of vector x. The chunk was allocated by the system as the result of a `BLK_SYNC` operation.

Line 12: Sends a synchronization signal to thread `scatter_fn`. This thread is waiting since it was spawned in line 27, for a pointer to the local buffer where the resulting y vector is stored. Note that `scatter_fn` is fired after it has been *logically* enabled since the purpose of this thread is precisely to scatter data back to remote memory.

Line 17: The `scatter_fn` thread copies the results (chunk of vector y) from local memory to its original location.

Line 18: Releases the local copy of vector y.

6.2 MAGMA Program Execution Model

MAGMA refers to a multithreading model suitable for large scale multicore systems. MAGMA is based on a memory-adaptive model that incorporates the behavior of the multi-level memory hierarchy found in modern multicore architectures. The runtime system is responsible for data allocation and migration, ensuring that data is *locally* available (to the processing element) before the computation starts.

The MAGMA Program Execution Model has the following important attributes:

- Programs are divided into small sequences of instructions, which we call threads.
- Threaded procedures or threads are defined like sequential functions, with an argument list just like that of a function.
- Upon thread creation, a small thread handle is allocated from the heap. But once a thread starts execution, the local context is allocated from the runtime stack and the thread handle is released.
- Threads are non-preemptive, i.e., once fired they run to completion.

- There is a single-level hierarchy of threads.
- Data dependencies explicitly identified in the program determines the execution order among threads.
- In order to satisfy the data dependencies identified in the application, the program instructs the runtime system to percolate data from remote to local memory.

6.2.1 MAGMA Thread Model

The MAGMA thread model is based on event-driven (or signal-driven) non-preemptive threaded procedures with dataflow-like synchronization as opposed to control flow-driven asynchronous calls. A thread is said to be *logically* enabled when the precedence conditions (data and control) are met. However, this condition is not sufficient. In MAGMA, a thread is eligible to begin execution once it becomes *physically* enabled. This only happens when all the data referenced by the thread is *physically* located in *local* memory, where *local* memory is determined by the user (programmer and compiler).

To make MAGMA suitable for large scale multicore systems, the model considers a single level of threads, spawned like function calls. In sequential programs, a function is called with a set of arguments specified at the function call site. Similarly, in MAGMA, a thread can only start running when all the arguments have been produced. A thread may have function-like arguments that are also specified when the thread is created. But a thread may also have arguments that are not specified at that point. These arguments are expressed as explicit dependencies in the program. The dependencies are to be met at runtime for a thread to start execution.

More importantly, date dependencies usually refer to data residing in remote memory, and needing to be brought to local memory, before computation can start. In MAGMA, the user (e.g. programmer and compiler) guides the runtime system to allocate and gather data before an operation can be performed, and to scatter data back once the work is done. Additionally, the compiler generates code assuming the data will be placed

in a *local* region of memory. Once the data is transferred to *local* memory, and if all the dependencies are satisfied, the runtime system activates the thread for execution. But before a thread starts running, the runtime system fixes the thread's argument so that it points to the temporary storage.

In MAGMA, threads are uniquely addressable units referenced by a thread handle. The thread handle is allocated from the heap. But once a thread starts running, it allocates its own local context from the runtime stack and releases the thread handle. The runtime stack is initialized when the thread is fired and released once the thread completes execution. Therefore, a thread has no previous state when it starts running. In addition, threads do not share data via a frame in (remote) shared memory. Instead, all the data a thread needs is to be passed as arguments. This solution inherently exploits on-chip memory bandwidth (an abundant resource especially in many-core architectures) rather than relying on off-chip memory (a scarce resource in multicore architectures).

As shown in the $Hello\ World$ program in Figure 6.1, multiple instances of the same thread can be created by passing the same function name to multiple `SPAWN` commands. However, each thread is only fired once and it terminates after execution completes. On the other hand, because threads are uniquely addressable units that do not share data via a frame, the model guarantees that multiple instances of a thread can be executed concurrently with no side-effects or data race conditions.

In addition, MAGMA programs, which strictly obey the thread firing rules, are deadlock-free. The dataflow computational model is known to be deadlock-free as long as the program is "well-structured". The MAGMA Program Execution Model is based on the same operational semantics of dataflow firing rules. Additionally, MAGMA threads, which can be regarded as macro dataflow nodes, cannot deadlock. This is because by definition, all the data a thread requires is available for the thread to start execution. Therefore, it can be proved that "well-structured" MAGMA programs are also deadlock-free.

6.2.2 MAGMA Operations

This section describes the operations supported by the MAGMA Program Execution Model. These can be implemented in hardware, software, or a combination of both, depending on the multicore architecture at hand.

- *SPAWN*: Initializes the descriptor of a new MAGMA thread and provides the thread arguments. It also specifies the arguments that are missing (data dependencies), which are to be resolved at run time.

  SPAWN($function\_name$, ...)

  When a program calls *SPAWN*, the MAGMA runtime system allocates a handle for a new MAGMA thread. The virtual thread is bound to the given function. The runtime system fills in the available arguments and leaves those that are missing empty. To specify a missing argument, the user precedes a unique identifier (an undeclared variable) with the *SYNC* keyword. The runtime system also initializes a counter with the total number of missing arguments.

  If the thread descriptor does not have missing arguments, the thread is immediately activated for execution. Otherwise, it remains *dormant* until values for the missing arguments are produced. Missing arguments are filled in with values produced by subsequent calls to the MAGMA runtime system. Precisely, such missing arguments represent the long latency operations that the runtime system will try to hide using percolation.

  The programmer may use the keyword *SLOT_ADR* to refer to the missing argument of another *SPAWN* function. Both the *SYNC* and *SLOT_ADR* keywords require a unique identifier. The compiler uses the identifier to pair *SYNC* and *SLOT_ADR* macros, and to generate the correct synchronization slot address.

- *DATA_SYNC*: Sends a value to fill in a missing argument.

  DATA_SYNC($datum$, *SLOT_ADR*)

If a data dependency refers to a single datum (i.e. a single value as opposed to a complex data structure such as an array), *DATA_SYNC* provides the value for the missing argument to a *dormant* thread. Once the value is copied to the MAGMA thread descriptor, the runtime system updates the dependencies counter. When the counter reaches zero, meaning that all data dependencies have been satisfied and values for all arguments have been produced, the runtime system schedules the thread for execution.

Based on the list of missing arguments in a previous *SPAWN* operation, the MAGMA compiler automatically fills in the *SLOT_ADR* argument of *DATA_SYNC*.

- *BLK_SYNC*: The runtime system begins to percolate a contiguous block of data of arbitrary size that a complex data structure requires.

  BLK_SYNC(void * src, size_t $length$, *SLOT_ADR*)

  The MAGMA runtime system automatically allocates a buffer from local memory to store the data to be percolated. Then the runtime system starts to percolate *length* bytes of data from *src* to the internal buffer. Once the data is transferred to the temporary buffer, the missing argument is filled in with the address of the temporary buffer. Then the runtime system decrements the dependencies counter, and if it becomes zero, the runtime system activates the thread for execution.

  The MAGMA compiler automatically generates the *SLOT_ADR* argument for *BLK_SYNC* calls based on the list of missing arguments passed to a *SPAWN* function.

- *GATHER_BLK_SYNC*: The runtime system percolates a non-contiguous block of data of arbitrary size.

  GATHER_BLK_SYNC($function\_name$, $function\_arg$,
  void * src, size_t $length$, *SLOT_ADR*)

The MAGMA runtime system automatically allocates a buffer from local memory to store the data to be percolated. Then the runtime system calls the user-provided function to percolate *length* bytes of data from *src* to the internal buffer. The runtime system passes this function the argument also provided by the user. Once the data is transferred to the temporary buffer, the runtime system fills in the missing argument with the address of the temporary buffer. Then the runtime system decreases the dependencies counter, and if it becomes zero, the runtime system activates the thread for execution.

The MAGMA compiler automatically generates the *SLOT_ADR* argument for *GATHER_BLK_SYNC* calls based on the list of missing arguments passed to the *SPAWN* command.

- *ALLOC_SYNC*: The runtime system allocates a buffer from local memory that is to be used by a MAGMA thread.

  ALLOC_SYNC(size_t $length$, *SLOT_ADR*)

  The MAGMA runtime system allocates a buffer from local memory to store some intermediate data needed by a MAGMA thread. Once the buffer is allocated, the runtime system fills in the value of the missing argument and checks whether all the data dependencies are met. When they are, the runtime system schedules the thread for execution.

  The compiler fills in the *SLOT_ADR* argument for *ALLOC_SYNC* calls based on the list of missing arguments of the *SPAWN* function.

- *RELEASE*: The runtime system releases internal storage allocated by previous function calls.

  RELEASE(void * ptr)

  Local memory previously allocated with a call to *ALLOC_SYNC*, *BLK_SYNC*, or *GATHER_BLK_SYNC* is returned to the runtime system. The runtime system may

keep the information used for bookkeeping of this memory buffer and may decide to reuse the data in the future. However, the programmer should not rely on this feature to continue accessing the data after the buffer has been released.

6.3 Main Features of MAGMA

In MAGMA, threads are like macro dataflow nodes. They contain a sequence of instructions that are executed in a von Neumann fashion, while thread activation follows the dataflow style. Thread activation (or synchronization) is decoupled from the execution stage. The former takes place in memory (mainly local memory) by writing tokens into the synchronization slots of a thread handle. Execution is performed by the hardware processing elements once a thread is fired. Firing and execution stages are connected with a queue. This queue, which the the runtime system scheduler manages as a FIFO, holds the threads that once activated are waiting for a processing element on which to run.

Threads are declared like sequential functions with a function argument list. However, a thread can be spawned without providing values for all its arguments. When arguments are *missing*, the thread remains inactive until all the data dependencies have been resolved, i.e., values for all the arguments have been produced. If the argument is a scalar variable, the value is directly copied into the argument slot. When the argument references a block of data, the system first transfers the data to local memory, then it writes the address of the internal storage in the argument slot.

As soon as the arguments are available, the thread is ready to be executed. When a thread is fired, the runtime system reads the arguments from the thread handle and sets both the thread registers and runtime stack according to the system ABI. Then the thread handle is released and recycled. A thread runs to completion without incurring any long latency operations. That is because all the data the thread accesses has been previously percolated and is *locally* available. Note that unlike pre-fetching, percolation allows to gather and scatter data blocks across the different levels of the memory hierarchy, even before the procedure or function that requires the data starts execution.

The remainder of this section illustrates the main features of MAGMA through program examples. We use three familiar programs (*Fibonacci*, *N-Queens* and *daxpy*) to illustrate the comparison between MAGMA, EARTH, and Cilk models. All the programs according to the EARTH model presented in this section are implemented using the Threaded-C language release 2.0 [57], and the Cilk examples are coded based on the Cilk 5.4.6 release [30].

In particular, it is worth noting that MAGMA gives the user the flexibility to solve a problem using either recursion or iteration. Thread synchronization (or token matching) is straightforward once the address of the target argument slot is known. Although the address is based on the thread handle, and that is only known at runtime, the compiler calculates the argument slot's offset based on the location of the missing argument within the function argument list statically. As a result, threads and synchronization slots are uniquely identified at runtime. This feature supports both programming paradigms with ease. Additionally, MAGMA percolates data from remote to local memory, allowing threads to run to completion without experiencing long delays. In addition to hiding latency, percolation improves data locality, and in some cases, facilitates data reuse, as we demonstrate in Section 6.3.2.

We use *Fibonacci* as an example of a recursive program that involves little computation to illustrate operations such as thread creation and synchronization. The solution to the enumeration of the *N-Queens* is also recursive. However, it involves movement of data blocks and therefore, demonstrates how MAGMA percolates data. The third example presents the *daxpy* routine. Interestingly, the Cilk implementation uses recursion, whereas MAGMA and EARTH programs use iteration.

6.3.1 *Fibonacci* Example

Figure 6.7 shows an example of a recursive sequential C code to calculate *Fibonacci* numbers. The code is a naive implementation of the mathematical equation for

```
 1  #include <stdio.h>
 2  #include <stdlib.h>
 3
 4  int fib(int n)
 5  {
 6    if (n<2) return (n);
 7    else {
 8      int x, y;
 9      x = fib (n-1);
10      y = fib (n-2);
11      return (x+y);
12    }
13  }
14
15  int main(int argc, char *argv[])
16  {
17    int n, result;
18    n = atoi(argv[1]);
19    result = fib (n);
20    printf ("Result: %d\n", result);
21    return 0;
22  }
```

Figure 6.7: Sequential *Fibonacci* Program

Fibonacci numbers. However, it serves the purpose of showing the differences and similarities among the three program execution models. Figures 6.9, 6.10, and 6.11 show three programs that compute *Fibonacci* numbers according to MAGMA, EARTH and Cilk models, respectively.

Since we take the mathematical equation for *Fibonacci* numbers, and use a recursive approach to write the parallel programs, in the three parallel versions there is a fib function that is called recursively. If $fib(n)$ determines it is not a leaf, it calls $fib(n-1)$ and $fib(n-2)$, and adds the results.

Sequential C and Cilk programs are very similar. In fact, the only differences between them, besides the inclusion of the library header file `cilk.h`, are a few keywords: `cilk`, `spawn`, and `sync`. When a Cilk program runs on one processor, it has the same semantics as the C program that results from deleting the Cilk keywords. This is called C elision of the Cilk program.

Of the three *Fibonacci* parallel programs, the Cilk version is the easiest to understand. fib is a compact function that resembles the sequential version with almost

transparent synchronization among threads. On the other hand, MAGMA has two distinct functions: fib and sum. $fib(n)$ spawns the thread sum and then directs the recursion until a leaf is reached. sum collects the results from $fib(n-1)$ and $fib(n-2)$, adds them together and passes the result to the sum thread spawned by $fib(n)$'s parent. Figure 6.8 shows the call graph for $fib(4)$. Note that all instances of `fib` spawn an instance of `sum`, except $fib(1)$ and $fib(0)$. In this case, the `fib` thread sends the result directly to the `sum` thread spawned by $fib(2)$. The EARTH model does not have two explicit functions. However, the threaded procedure contains two fibers. These fibers are the counterparts to the fib and sum threads in the MAGMA program. Additionally, the Threaded-C precompiler translates fibers into functions, although that is an implementation detail.

In Cilk and MAGMA, communication and synchronization are indistinguishable. EARTH is more flexible because data communication and synchronization signals are handled separately. In EARTH, the data is first written to memory. Then, a fiber is synced using its synchronization slot.

Both Cilk and EARTH programs declare two local variables to receive the results from $fib(n-1)$ and $fib(n-2)$. On EARTH, these variables are allocated from the frame that all fibers within a threaded procedure share. The requirement for a this frame prevents EARTH from exploiting the memory closest to the processing elements, because this memory usually is not uniformly accessible. On the other hand, the Cilk compiler actually breaks the function into two threads, each with its own stack similar to MAGMA threads. MAGMA does not need such variables, since the results are passed directly to the sum thread. The compiler computes the synchronization slot address directly from the information that the macros $SYNC$ and $SLOT\_ADR$ provide, such as the unique identifier.

The following explains the details of the MAGMA program shown in Figure 6.9 to compute *Fibonacci* numbers and the commands introduced in this example:

Line 5: The arguments to `fib` include a *SYNC_SLOT* to be synced after the result is computed. The synchronization signal is implicit with the function result.

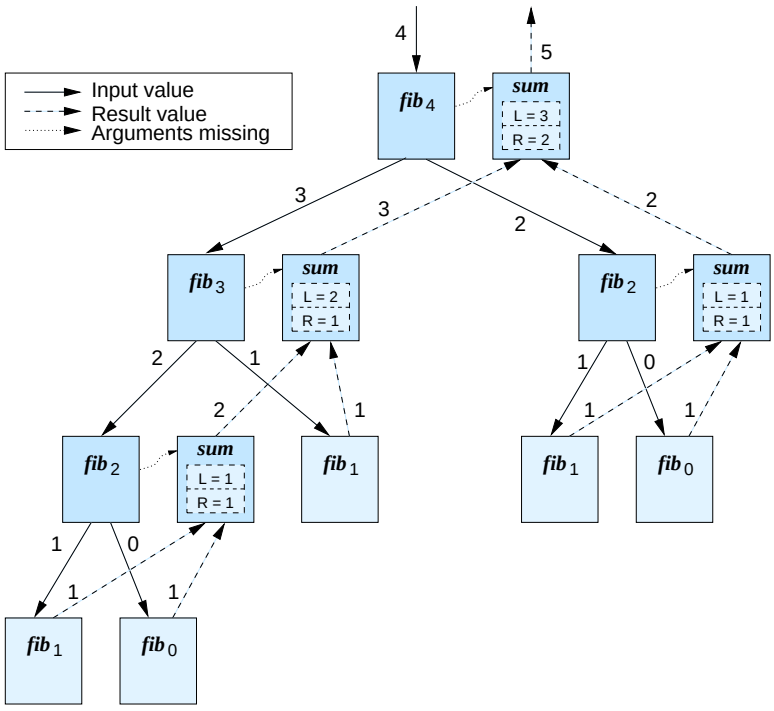


Figure 6.8: MAGMA *Fibonacci* Call Graph

Lines 7–8: This function call is a leaf (no more recursion). It passes the value n directly to the *SYNC_SLOT* $result$. The threaded procedure `sum` that the parent thread spawned receives this value as either `x` or `y`.

Lines 10–13: This function call requires recursion. It spawns the thread that will receive the results, and recursively calls the `fib` function twice. One function's result is directed to `left` (`sum` function argument `x`), and the other is directed to `right` (`sum` function argument `y`). The compiler pairs each `SLOT_ADR` with a `SYNC` macro based on matching identifiers, and generates the address of each synchronization slot so that the instances of `fib` have places to send their results. Note that each call receives a slot address to either `left` or `right`, and that the `sum` thread requires two values (Line 17) before it can start.

Lines 17–19: Thread `sum` runs after two values have been received in `x` and `y` because both children (Lines 12–13) have sent back their result. The thread adds `x` and `y` together and use a `DATA_SYNC` (on a remote slot) to send the sum to the caller's `sum` thread.

```
 1  #include <stdio.h>
 2  #include <stdlib.h>
 3  #include <magma.h>
 4
 5  fib(SYNC_SLOT int result, int n)
 6  {
 7    if (n<2) {
 8      DATA_SYNC(n, result);
 9    }
10    else {
11      SPAWN(sum, result, SYNC(left), SYNC(right));
12      SPAWN(fib, SLOT_ADR(left), n-1);
13      SPAWN(fib, SLOT_ADR(right), n-2);
14    }
15  }
16
17  sum(SYNC_SLOT int result, int x, int y)
18  {
19    DATA_SYNC(x+y, result);
20  }
21
22  int main (int argc, char *argv[])
23  {
24    int n;
25    n = atoi(argv[1]);
26    SPAWN(done, SYNC(res), n);
27    SPAWN(fib, SLOT_ADR(res), n);
28  }
29
30  done(int result, int n)
31  {
32    printf ("fib(%d) = %d\n", n, result);
33  }
```

Figure 6.9: MAGMA *Fibonacci* Program

The aspects worth noting of the EARTH program shown in Figure 6.10 to compute *Fibonacci* numbers, are the following:

Line 4: The parameters to the `fib` threaded function include a global handle (`result`) which will receive the result of the function (the n^{th} *Fibonacci* number); they also include a slot to be signaled when that result has been computed. In other words, `result` is used for data communication whereas `done` carries the synchronization signal.

Line 6: Local variables `r1` and `r2` are introduced to receive the results of the recursive calls.

Lines 8–10: This function call is a leaf (no more recursion). It writes the values 0 and 1 into `r1` and `r2` and directly spawns fiber `READY`.

Lines 11–13: This function call requires recursion. It calls the fib function twice, with one result directed to `r1` and the other directed to `r2`. The macro `TO_GLOBAL`

```
 1  #include <stdio.h>
 2  #include <stdlib.h>
 3
 4  THREADED fib( int n, int *GLOBAL result, SPTR done )
 5  {
 6    int r1, r2;
 7
 8    if (n <= 1) {
 9      r1 = 0; r2 = 1;
10      SPAWN( READY );
11    } else {
12      TOKEN( fib, n-1, TO_GLOBAL(&r1), TO_SPTR(READY) );
13      TOKEN( fib, n-2, TO_GLOBAL(&r2), TO_SPTR(READY) );
14    }
15
16    FIBER READY <* 2 *> {
17      PUT_SYNC( r1 + r2, result, done );
18      TERMINATE;
19    }
20  }
21
22  THREADED MAIN( int argc, char* argv[] )
23  {
24    int n, res;
25
26    n = atoi(argv[1]);
27    TOKEN( fib, n, TO_GLOBAL(&res), TO_SPTR(FIB_DONE) );
28
29    FIBER FIB_DONE <* 1 *> {
30      printf( "fib(%d) = %d\n", n, res );
31      TERMINATE;
32    }
33  }
```

Figure 6.10: EARTH *Fibonacci* Program

converts the local addresses `&r1` and `&r2` into global handles so that the two instances of `fib` have places to send their results. Note that both calls receive the same synchronization slot (`TO_SPTR(READY)`), and that this fiber requires 2 synchronization signals (Line 16) before it can start.

Lines 16–19: Fiber `READY` runs after two signals have been received in synchronization slot `READY`, indicating that both results are ready in `r1` and `r2` because both children (Lines 12–13) have sent back their result. Fiber `READY` may also be spawned directly (Line 10) if this invocation of the function `fib` is a leaf of the recursion tree. Fiber `READY` adds the sub-results together and uses a `PUT_SYNC` (on a remote slot) to send the sum to the caller.

The main aspects of the Cilk program shown in Figure 6.11 are as follows:

```
 1  #include <stdio.h>
 2  #include <stdlib.h>
 3  #include <cilk.h>
 4
 5  cilk int fib(int n)
 6  {
 7    if (n<2) return n;
 8    else {
 9      int x, y;
10      x = spawn fib (n-1);
11      y = spawn fib (n-2);
12      sync;
13      return (x+y);
14    }
15  }
16
17  cilk int main(int argc, char *argv[])
18  {
19    int n, result;
20    n = atoi(argv[1]);
21    result = spawn fib(n);
22    sync;
23    printf ("Result: %d\n", result);
24    return 0;
25  }
```

Figure 6.11: Cilk *Fibonacci* Program

Lines 10–11: In Cilk, when the keyword `spawn` precedes a function, the procedure is executed similarly to a C function call. However, execution of the parent can continue in parallel with the child, producing parallelism.

Line 12–13: The parent needs to execute `sync` in order to safely use the values that the child returns. This synchronization is like a local barrier. It only waits for the children spawned by the procedure that executes `sync`.

6.3.2 *N-Queens* Example

Figure 6.13 shows the core routine of a sequential program that solves the *N-Queens* problem. The full source code is provided in Appendix A. The *N-Queens* code counts the number of ways to place N queens on a $N \times N$ chess board so that none of them can hit any other in one move under normal chess rules. We chose a recursive implementation, similar to the previous *Fibonacci* example, but that requires block transfer operations.

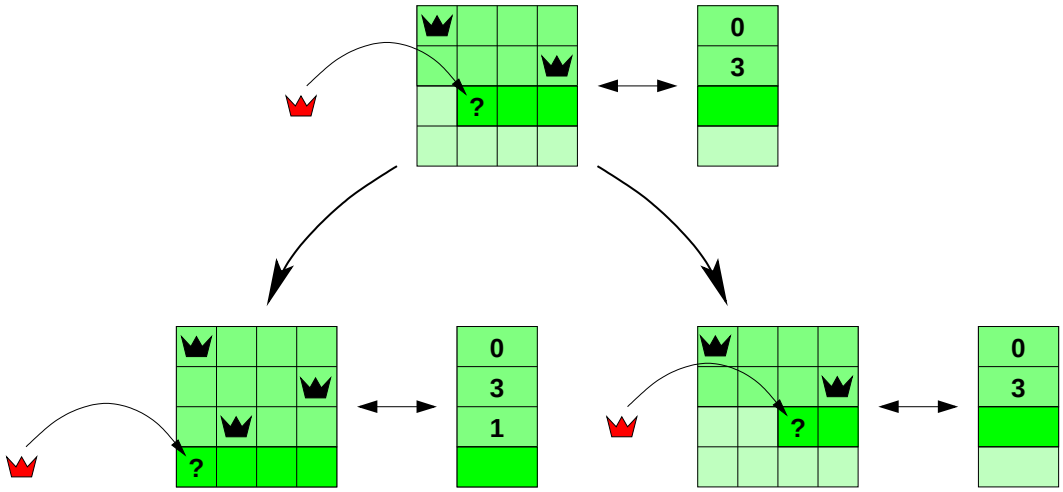


Figure 6.12: *N-Queens* Recursion

To solve the *N-Queens* problem a search function is called that tries placing queens in different columns of a given row of the board. When the function finds a valid position, it splits the search into two subproblems. One subproblem adds the new queen to the chessboard and starts searching the next row. The other keeps trying positions to the right of the current position. For the original search, the function returns the sum of the solutions to both subproblems, see Figure 6.12.

The sequential C implementation is straightforward. For a row received as one of the input arguments in line 1, the function tries to find a valid position for a queen in the column range between `start_col` and `n`, line 6. If the search is successful the problem is divided into two subproblems as explained above by making recursive function calls, lines 9–10. Once the two searches complete, the function adds the solutions returned by each of the subproblems, line 11.

Figures 6.15, 6.17, and 6.16 show the parallel version of the *N-Queens* solution according to MAGMA, EARTH, and Cilk models, respectively. Due to the nature of the problem, the *N-Queens* code is amenable to parallelization. In the three models, each of the recursive function calls in the sequential program is executed on an independent thread (or fiber depending on the model) so they can run concurrently.

```
 1  int sequeens(int n, int row, int start_col, int board[])
 2  {
 3    int col, sols_this_col, sols_other_cols;
 4
 5    if (row >= n) return 1;
 6    for (col = start_col; col < n; col++) {
 7      if (safe(board, row, col)) {
 8        board[row] = col;
 9        sols_this_col = sequeens(n, row+1, 0, board);
10        sols_other_cols = sequeens(n, row, col+1, board);
11        return (sols_this_col + sols_other_cols);
12      }
13    }
14    return 0;
15  }
```

Figure 6.13: Sequential *N-Queens* Program

The Cilk program is again simpler than the programs written according to the EARTH and MAGMA models. However, this simplicity comes at a cost. At runtime, each instance of the `nqueens` function stalls while it makes a copy of the current state of the board. While data is transferred, no other thread can make use of the processing element. On the contrary, MAGMA and EARTH models support split-phase memory copy transactions. While the runtime system transfers the data, another thread or fiber could run on the processing element, achieving better resource utilization.

Figure 6.14 shows the call graph for the MAGMA *N-Queens* program. Each instance of `nqueens` that is not a leaf spawns two new instances of the same function. However, the new threads remain *dormant* until the status of the board is percolated. Note that after the status of the board has been percolated from remote memory once, subsequent block transfers are from local memory. This example demonstrates how percolation can improve memory bandwidth via data reuse.

The main features of the *N-Queens* routine (see Figure 6.15) implemented according to the MAGMA model are as follows:

Line 8: After the search function finds a valid position for a queen, we spawn the thread `sum` that will receive the results from each of the subproblems. If the main loop in line 5 does not find a valid position for a queen, the search function returns 0 directly to its parent in line 27.

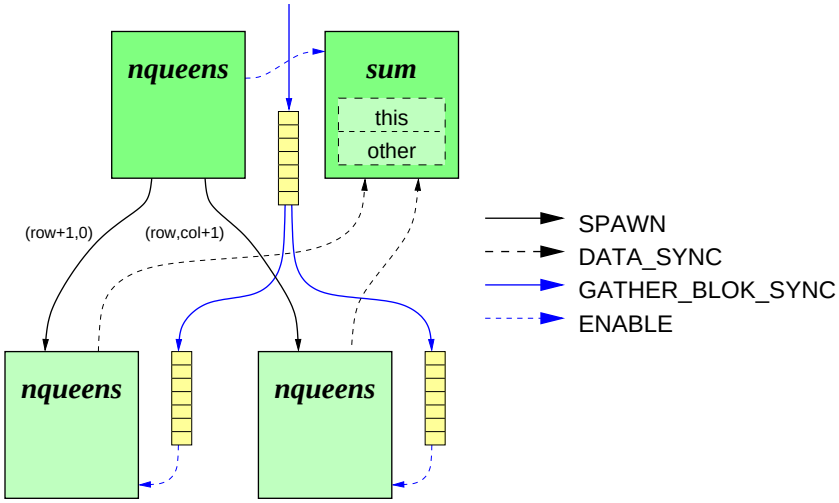


Figure 6.14: MAGMA *N-Queens* Call Graph

Lines 9–10: When the search reaches the last row, it is complete, and the current solution is reported as one solution to the `sum` thread.

Lines 12–14: Otherwise, we spawn a new thread to continue the search of the subproblem. We add the new queen to the board and start searching from the first column of the next row.

Lines 16–17: When the search reaches the rightmost column of the board, it means that a solution has not been found. At this point, the function tells the `sum` thread spawned in line 8 that there is not a solution to the second subproblem.

Lines 19–21: Otherwise, we spawn another thread to continue trying positions to the right of the current column.

Lines 26–27: When the search throughout a row is unsuccessful, this function tells the `sum` thread spawned by its parent that the current subproblem did not find any solution.

Lines 33: After the threads spawned in lines 13 and 20 complete execution or when the current function reaches an edge of the board, lines 10 and 17 the `sum` thread adds the number of solutions to both subproblems and communicates the result to the `sum` thread spawned by its parent.

Lines 14 and 21: Before the thread spawned to compute one of the subproblems is fired, the status of the current board must be transferred. In this case the user defines a function (lines 36–39) and a single argument, that the runtime system invokes to copy data from remote to local memory. We use `GATHER_BLK_SYNC` instead of

```
 1  nqueens(int n, int row, int start_col, int board[], SYNC_SLOT result)
 2  {
 3    int col;
 4
 5    for (col=start_col; col<n; col++) {
 6      if (safe(board, row, col)) {
 7        board[row] = col;
 8        SPAWN(sum, result, SYNC(this_col), SYNC(other_cols));
 9        if (row+1 == n) {
10          DATA_SYNC(1, SLOT_ADR(this_col));
11        }
12        else {
13          SPAWN(nqueens, n, row+1, 0, SYNC(board), SLOT_ADR(this_col));
14          GATHER_BLK_SYNC(gather, row, board, n*sizeof(int), SLOT_ADR(board));
15        }
16        if (col+1 == n) {
17          DATA_SYNC(0, SLOT_ADR(other_cols));
18        }
19        else {
20          SPAWN(nqueens, n, row, col+1, SYNC(board), SLOT_ADR(other_cols));
21          GATHER_BLK_SYNC(gather, row, board, n*sizeof(int), SLOT_ADR(board));
22        }
23        break;
24      }
25    }
26    if (col == n) {
27      DATA_SYNC(0, result);
28    }
29  }
30
31  sum(SYNC_SLOT int result, int sols_this_col, int sols_other_cols)
32  {
33    DATA_SYNC(sols_this_col+sols_other_cols, result);
34  }
35
36  gather(void *dst, void *src, int n, void *arg)
37  {
38    memcpy(dst, src, (int)arg);
39  }
```

Figure 6.15: MAGMA *N-Queens* Program

BLK_SYNC because the amount of data to move and the size of the buffer that needs to be allocated are different. We allocate $n \times sizeof(int)$ bytes but we only transfer $row \times sizeof(int)$ bytes.

The implementation according to the EARTH and MAGMA models are similar because the semantics of the operations are the same. In this case, the differences are that in the EARTH program (see figure 6.17), a fiber collects the results from the subproblems instead of a threaded procedure. Additionally, the first fiber of the threaded procedure copies the board from the parent's thread in line 10. Once the transfer completes, the

```
 1  cilk int nqueens(int n, int row, int start_col, int *previous)
 2  {
 3    int own_board[MAX_BOARD_SIZE], col, sols_this_col, sols_other_cols;
 4
 5    sols_this_col = sols_other_cols = 0;
 6    memcpy(own_board, previous, row*sizeof(int));
 7
 8    if (row >= n) return 1;
 9    for (col=start_col; col<n; col++) {
10      if (safe(own_board, row, col)) {
11        own_board[row] = col;
12        sols_this_col = spawn nqueens(n, row+1, 0, own_board);
13        sols_other_cols = spawn nqueens(n, row, col+1, own_board);
14        sync;
15        return (sols_this_col + sols_other_cols);
16      }
17    }
18    return 0;
19  }
```

Figure 6.16: Cilk *N-Queens* Program

DATA_RECEIVED fiber does the actual search in lines 12–37. Because in EARTH, the user is responsible for all the memory management, the current implementation declares an array in the frame of the threaded procedure. However, the program could dynamically allocate the own_board array elsewhere.

Like the implementation of the recursive *Fibonacci* program, *N-Queens* sequential C and Cilk programs are very similar. However, besides the inclusion of the library header file and the Cilk keywords, the Cilk program makes a call to memcpy in line 6, to copy the status of the board. Note that the Cilk program can only run on SMP systems, precisely because the $memcpy$ call. In this case, the simplicity of the Cilk model and the C elision property comes at the expense of portability. The operational semantics of the Cilk program can be summarized as follows:

Lines 12–13: Generates parallelism when the nqueens functions are executed in a separate thread and run concurrently with the parent. That is because the function is preceded by the keyword spawn.

Line 14–15: The parent needs to execute sync in order to safely use the values that the children returned.

```
 1  THREADED nqueens(int n, int row, int start_col,
 2                   int *GLOBAL previous, int *GLOBAL result, SPTR done)
 3  {
 4    int own_board[MAX_BOARD_SIZE],
 5        col,
 6        sols_this_col, sssols_other_cols;
 7
 8    sols_this_col = 0;
 9    sols_other_cols = 0;
10    BLKMOV_SYNC(previous, TO_GLOBAL(own_board), row*sizeof(int), DATA_RECEIVED);
11
12    FIBER DATA_RECEIVED <* 1 *> {
13      for(col=start_col; col<n; col++) {
14        if (safe(own_board, row, col)) {
15          own_board[row] = col;
16          if (row+1 == n) {
17            sols_this_col = 1;
18            SYNC(DONE);
19          }
20          else {
21            TOKEN(nqueens, n, row+1, 0, TO_GLOBAL(own_board),
22                  TO_GLOBAL(&sols_this_col), TO_SPTR(DONE));
23          }
24          if (col+1 == n) {
25            SYNC(DONE);
26          }
27          else {
28            TOKEN(nqueens, n, row, col+1, TO_GLOBAL(own_board),
29                  TO_GLOBAL(&sols_other_cols), TO_SPTR(DONE));
30          }
31          break;
32        }
33      }
34      if (col == n) {
35        SPAWN(DONE);
36      }
37    }
38
39    FIBER DONE <* 2 *> {
40      PUT_SYNC(sols_this_col + sols_other_cols, result, done);
41      TERMINATE;
42    }
43  }
```

Figure 6.17: EARTH *N-Queens* Program

```
1  void daxpy(int n, double alpha, double x[], double y[])
2  {
3    int i;
4
5    for (i=0; i<n; i++) {
6      y[i] = alpha * x[i] + y[i];
7    }
8  }
```

Figure 6.18: Sequential *daxpy* Program

6.3.3 *daxpy* Example

Figure 6.18 shows an example of a sequential *daxpy* routine. The C code computes a constant alpha times a vector plus another vector ($y_i = \alpha \times x_i + y_i$). It is a simplified version in the sense that it assumes that the increments between vectors x and y are equal to one. In other words, the routine does not have the integer arguments inc_x and inc_y for the increment between vectors x and y, respectively. Figures 6.19, 6.20, and 6.21 show the parallel version of the *daxpy* routine according to MAGMA, EARTH, and Cilk models, respectively.

The sequential C implementation is straightforward using a *for* loop. In lines 5–6, the routine iterates over the vectors x and y of length n, adding a scalar multiple of a double precision vector element to another double precision vector element. The result overwrites the initial values of vector y.

For the MAGMA and EARTH implementations, we assume that the user can take advantage of the regularity of the loop structure and the data access pattern in order to uniformly distribute the computation among threads (or fibers depending on the model). In order to simplify the example and without loss of generality, we also assume that the length of the vectors is a multiple of the number of threads (or fibers). Quite differently, the Cilk program recursively divides the computation of an n-size *daxpy* problem into 2^k subproblems. k is determined by the initial problem size, n, and a predefined threshold, `BLK_SIZE`. In the three models, the programmer is responsible for determining the optimal block size.

The Cilk program requires fewer number of lines of codes than do the other two. Even though the Cilk program without the Cilk keywords is still a valid C program, this time it is not like the sequential C version. The sequential *daxpy* routine is a single loop whereas the Cilk program uses recursion. At each step, the Cilk program divides vectors x and y into two halves. To keep track of the vectors' boundaries, the program requires an additional function argument (`offset`). Then the program spawns two threads that perform the *daxpy* operation on each half of the vectors x and y.

The implementations according to the MAGMA and EARTH models are somewhat more complicated, mainly because the vector chunks are transferred from a "remote" to a "near" location. The Cilk program is simpler but at runtime, there may be significant execution delays if the accesses are to remote memory. The EARTH model ensures that the "near" location is on the same node, although it could be in off-chip memory. MAGMA goes one step further and guarantees that the "near" location is indeed local memory. In addition, the MAGMA program is simpler than the EARTH code because some of the memory management is handled directly by the runtime system. For instance, the MAGMA program does not do any memory allocation. The runtime system internally allocates temporary storage for vectors x and y blocks, which are released by the program once the partial results are obtained. The EARTH program on the contrary, handles all memory manually. Note that the EARTH program shown in Figure 6.20 corresponds to an unoptimized implementation with a naive buffer management.[2] On the other hand, the Cilk program does not involve any data transfer, thus its simplicity. However, on a cache-less system, such as Cyclops-64, the program would have to explicitly handle all buffer allocation and data transfers to exploit locality. It is very likely that the resulting Cilk program will be as complex as the EARTH program. It is also worth noting that the

[2] In order to simplify synchronization among fibers, the EARTH *daxpy* routine has not been optimized with double buffers or alike. As a consequence, the execution appears to be pipelined but it actually runs sequentially.

```
 1  #include <magma.h>
 2
 3  daxpy_fn(int n, double alpha, double x[], double y[],
 4           SYNC_SLOT double * done)
 5  {
 6    int i;
 7
 8    for (i=0; i<n; i++) {
 9      y[i] = alpha * x[i] + y[i];
10    }
11    RELEASE(x);
12    DATA_SYNC(y, done);
13  }
14
15  scatter_fn(int n, double y_rem[], double y_loc[])
16  {
17    memcpy(y_rem, y_loc, n*sizeof(double));
18    RELEASE(y_loc);
19  }
20
21  daxpy(int n, double alpha, double x[], double y[])
22  {
23    int i;
24    int chunk_len = n/NUM_PES;
25
26    for(i=0; i<NUM_PES; i++) {
27      SPAWN(scatter_fn, chunk_len, &y[i*chunk_len], SYNC(y_local));
28      SPAWN(daxpy_fn, chunk_len, alpha, SYNC(x), SYNC(y), SLOT_ADR(y_local));
29      BLK_SYNC(&x[i*chunk_len], chunk_len*sizeof(double), SLOT_ADR(x));
30      BLK_SYNC(&y[i*chunk_len], chunk_len*sizeof(double), SLOT_ADR(y));
31    }
32  }
```

Figure 6.19: MAGMA *daxpy* Program

elision property, which makes Cilk highly efficient for divide-and-conquer-programs, limits the types of program paradigms that are supported and the types of machines where it can run efficiently to shared-memory machines. From the discussion above, the MAGMA model offers a reasonable trade-off between programmability and portable performance for cache-less distributed memory systems.

The main features of the *daxpy* routine (see Figure 6.19) implemented according to the MAGMA model are as follows:

Line 24: Given our assumption that the vector length is a multiple of the number of processing elements, we evenly divide the computation among processing elements.

Lines 26–31: The main *daxpy* routine loop spawns twice as many threads as there

are processing elements. The `daxpy_fn` threads each do part of the computation, whereas the `scatter_fn` threads will copy the y vector back to its original location in memory. After a `daxpy_fn` thread computes its corresponding chunk of vector y, a `scatter_fn` thread can start transferring the resulting y chunk back. The programmer explicitly tells the compiler the dependency between threads `daxpy_fn` and `scatter_fn` with the macros `SYNC(y_local)` and `SLOT_ADR(y_local)`. Note that `y_local` is neither a variable nor a constant. It is only an identifier that the compiler uses to match `SLOT_ADR` and `SYNC` macros in lines 27 and 28, respectively.

Line 28: The `daxpy_fn` function expects vectors x and y of length n. However, these arguments are defined as missing with the keyword `SYNC`.

Lines 29–30: The `daxpy` routine knows the address of vectors x and y when it spawns the `daxpy_fn` thread. However, these vectors likely are not in local memory. In order to *physically* enable the `daxpy_fn` thread, the `daxpy` routine invokes two `BLK_SYNC` operations to transfer vectors chunks x and y for its current location (probably in remote memory) to a level of the memory hierarchy local to the processing element. Triggered by a `BLK_SYNC` operation, the system allocates some temporary storage, and once the block of data has been transferred, it will synchronize the `daxpy_fn` thread filling the appropriate argument slot with the address of the temporary storage to which the chunk has been copied.

Lines 8–9: Once the `daxpy_fn` thread is *physically* enabled, i.e., vectors x and y chunks are in local memory, it performs the normal *daxpy* computation.

Line 11: After `daxpy_fn` completes the computation, and the partial result is in y, the thread releases the temporary storage that holds the local copy of vector x. The chunk was allocated by the system as the result of a `BLK_SYNC` operation.

Line 12: Sends a synchronization signal to thread `scatter_fn`. This thread is *dormant* since it was spawned in line 27, waiting for a pointer to the local buffer where the partial result (y) is stored.

Line 17: The `scatter_fn` thread copies the results (chunk of vector y) from local memory to its original location.

Line 18: Releases the local copy of vector y.

```
 1  THREADED daxpy(int n, double alpha, double x[], double y[], SPTR * done)
 2  {
 3    int i;
 4    int chunk_len = n/NUM_NODES;
 5    double *lx, *ly;
 6
 7    lx = malloc(chunk_len*sizeof(double));
 8    ly = malloc(chunk_len*sizeof(double));
 9
10    for(i=0; i<NUM_NODES; i++) {
11      FIBER COPY <* 0, 1 *>:
12      BLKMOV_SYNC(TO_GLOBAL(x+i*chunk_vec), lx, chunk_len, TO_SPTR(DAXPY));
13      BLKMOV_SYNC(TO_GLOBAL(y+i*chunk_vec), ly, chunk_len, TO_SPTR(DAXPY));
14    }
15
16    FIBER DAXPY <* 2 *> {
17      for (i=0; i<n; i++) {
18        ly[i] = alpha * lx[i] + ly[i];
19      }
20      BLKMOV_SYNC(ly, TO_GLOBAL(y+i*chunk_vec), chunk_len, TO_SPR(COPY));
21    }
22
23    free(lx);
24    free(ly);
25    SYNC(done);
26    TERMINATE;
27  }
```

Figure 6.20: EARTH *daxpy* Program

The main features of the *daxpy* routine (see Figure 6.20) implemented according to the EARTH model are as follows:

Lines 7–8: The initial fiber, which runs automatically when the threaded procedure *daxpy* is `INVOKE`d by the main program, starts with the allocation of a temporary buffer for chunks of vectors x and y.

Lines 10–11 : Iterations of the threaded procedure's main loop terminate when the control flow reaches the `FIBER` keyword. After the first iteration is executed, however the `COPY` fiber is executed because the fiber's initial synchronization counter is zero. Subsequent firing of instances of this fiber is controlled by the synchronization signal sent by the `DAXPY` fiber in line 20.

Lines 12–13: Fiber `COPY` transfers two chunks of vectors x and y from remote memory into this node's memory. Once the data is copied into lx and ly the runtime system will sync the *DAXPY* fiber twice, one for each chunk.

Line 16: Once `DAXPY`'s synchronization slot counter reaches zero, the fiber is enabled for execution. When the fiber starts running, it first computes the linear combination $ly_i = \alpha \times lx_i + ly_i$. Then it transfers the partial results in ly to the location from

```
 1  #include <cilk.h>
 2
 3  cilk void daxpy(int n, int offset, double alpha,
 4                  double x[], double y[])
 5  {
 6    int i;
 7    int n2;
 8
 9    if (n<BLK_SIZE) {
10      for (i=0; i<n; i++)
11        y[i+offset] = alpha * x[i+offset] + y[i+offset];
12    }
13    else {
14      n2 = n >> 1;
15      spawn daxpy(n2, offset, alpha, x, y);
16      spawn daxpy(n-n2, offset+n2, alpha, x, y);
17    }
18  }
```

Figure 6.21: Cilk *daxpy* Program

which the chunk was originally read, line 20. After the data has been scattered, it is safe to reuse buffers lx and ly for another iteration. Once COPY receives this synchronization signal, it will transfer two new blocks of data from remote memory and will continue through another iteration of the main loop. Again, once control flow reaches the keyword FIBER the ongoing iteration terminates.

Lines 23–26: After the main loop completes all the iterations, execution continues in line 23. The threaded procedure releases the memory allocated for lx and ly, synchronizes the parent's threaded procedure, and terminates execution.

As mentioned above, this implementation of the *daxpy* routine according to the EARTH model could be further optimized. The current implementation allocates a single buffer and therefore cannot apply techniques such as double-buffering in order to hide latency. As a result, execution is serialized, COPY, DAXPY, and the initial fiber execute one iteration at a time. With double or triple buffering, interleaving of the stages would be possible, thus hiding the communication latency between COPY and DAXPY. However, the programmer would be responsible for all memory management. On MAGMA, memory management is almost transparent.

The main characteristics of the *daxpy* routine implemented in Figure 6.21 according to the Cilk model are as follows:

Lines 13–17: As long as the problem size is bigger than the predefined threshold `BLK_SIZE`, the Cilk routine continues applying recursion and decomposes an n-size problem into two $n/2$-size subproblems.

Line 15: The program spawns a thread to compute the result for the $n/2$-size subproblem corresponding to vectors x and y first halves.

Line 16: The program spawns a second thread to compute the result for the $n/2$-size subproblem corresponding to vectors x and y second halves.

Lines 9–12: Once the $daxpy(n)$ function determines that it is a leaf, it performs the *daxpy* operation on the two input vectors using a loop. Because the *daxpy* function operates directly on the x and y vectors, it has to keep track of each chunk's position within the vector. The loop iteration uses this position (`offset`) to access the correct section of the vectors.

Chapter 7

EXPERIMENTAL RESULTS

In this chapter, we present the experimental results for TNT TVM. The main results of our experimental study can be summarized as follows:

- **High efficiency**: Our microbenchmarks demonstrate that TNT primitives for thread creation and recycling complete in a few hundred cycles with low overhead.
- **Scalability**: An increased workload (number of threads spawned by a microbenchmark) and/or additional hardware resources (thread units) are easily handled by the runtime system with negligible impact on the operations of the runtime system.
- **Usability**: Our experience with MAGMA demonstrates that the TNT user-level library without kernel intervention (hence, no disruption), can effectively support a multithreaded program execution model.

Experimental Platform

We conducted our experiments with the Cyclops-64 software toolset 2.4 release, which includes the C64 GCC-4.1 compiler, the FAST simulator and the C64 kernel. For the purpose of these experiments, we replaced the default C64 kernel with the TNT library and a small amount of code that bootstraps the new library.

7.1 TNT Results

To measure the overhead imposed by thread management operations, we wrote a simple microbenchmark. The main function consists of a single loop to create a number

```
 1  #include <tnt.h>
 2
 3  #define NUM_THREADS 10000
 4
 5  void empty_fn(void)
 6  {
 7    return;
 8  }
 9
10  int main()
11  {
12    tnt_desc_t th_desc;
13    int64_t    error = 0;
14
15    for (i=0; i<NUM_THREADS && !error; ++i) {
16      if (tnt_create(&th_desc, &empty_fn, NULL) != 0) {
17        error++;
18      }
19    }
20  }
```

Figure 7.1: TNT *Empty* Microbenchmark

of threads. After all the threads have been spawned, the program returns. The threads spawned by the microbenchmark execute an empty function, see Figure 7.1. They return almost instantaneously and they are immediately recycled by the runtime system. We define "user time" as the elapsed time between the invocation of the main function and its termination, which mainly accounts for the loop that creates all the threads. We also measure the "system time", defined as the wall time it takes to complete the execution of the program. The system time accounts for the execution of the main function in addition to all the work generated from it. Because the thread functions are empty, the difference between the system and user times represents the overhead of the runtime system, i.e. the time spent in the creation, termination, and recycling of threads.

For the purpose of this experiment, we modified the TNT library so that it only allocates 1,000 thread descriptors, and these are allocated from the on-chip memory (initially TNT can allocate descriptors from off-chip memory too). Accordingly, the program detects whether a thread could not be spawned, i.e. TNT failed to create another virtual thread because all the thread descriptors were being used, and terminates.

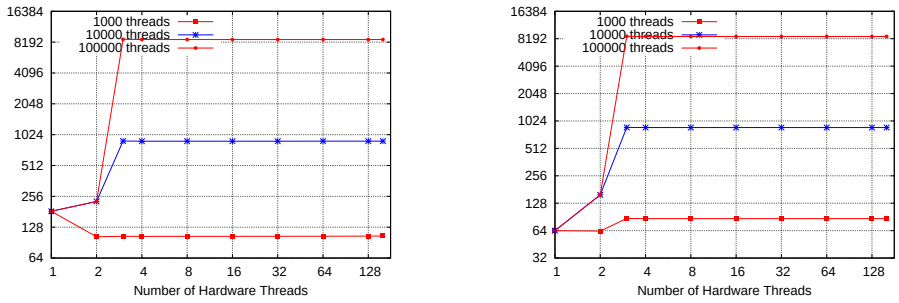
Table 7.1 and Figure 7.2 summarize the system and user times measured by the

Table 7.1: *Empty* Microbenchmark Execution Times [in Ten Thousand Clock Cycles]

VTs	1,000		10,000		100,000	
HTs	System	User	System	User	System	User
1	183	63	183	63	183	63
2	103	63	228	158	228	158
3	104	87	887	870	8,717	8,700
4	104	86	884	867	8,687	8,670
8	104	86	884	867	8,687	8,670
16	104	86	884	867	8,687	8,670
32	104	86	884	867	8,687	8,670
64	104	86	884	867	8,687	8,670
128	104	86	885	867	8,688	8,670
160	104	86	885	867	8,688	8,670

microbenchmark that creates 10^3, 10^4, and 10^5 threads, on a platform with an increasing number of hardware thread units. Because of the TNT library modifications described above, when there is a single hardware thread unit, the program only spawns 1,000 threads, and when there are two hardware thread units, the program spawns 2,498 threads, instead of 10^4 and 10^5.

As expected, the benchmark's execution time increases linearly with the number of threads spawned by the program. A point worth noting is that with three or more thread units, the system and user times remain constant. That means that a system with at least three thread units is able to recycle threads as fast as they are created. How fast can TNT spawn and recycle threads? From the user time in the last column, we determine that TNT can spawn a thread in about 867 clock cycles. We define the recycle time as the elapsed time from the moment a virtual thread returns until the thread unit starts execution of the next virtual thread. That time is precisely the difference between the system and user times when the microbenchmark runs on a single thread unit. From the values of the first row in Table 7.1, we determine that the recycle time is less than 1,200 cycles. In light of these numbers, the execution of the microbenchmark can be summarized as follows. The



(a) System time (b) User time

Figure 7.2: *Empty* Microbenchmark Execution Time [in Ten Thousand Clock Cycles]

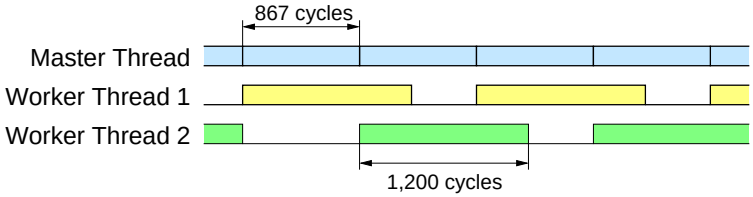


Figure 7.3: Thread Execution Interleaving

master thread executes a loop iteration every 867 cycles. In a system with three thread units, each of the two remaining thread units will be able to execute every other virtual thread so by the time the master thread returns, the work will have been almost completed, see Figure 7.3.

When the hardware resources scale from 1 to 160 thread units, we notice that the thread creation time increases from 630 to 867 cycles. This happens because when there are not enough hardware thread units to pick up the work created by the master thread, creating a new thread involves the initialization of a thread descriptor and pushing the descriptor to the virtual ready queue. However, if there are thread units, one will be bound to the new thread and start its execution. This last step requires additional processing (queue operations), thus the increased execution time.

In the *Empty* microbenchmark, the main function creates a number of threads sequentially. Because threads execute an empty function, they return almost instantaneously

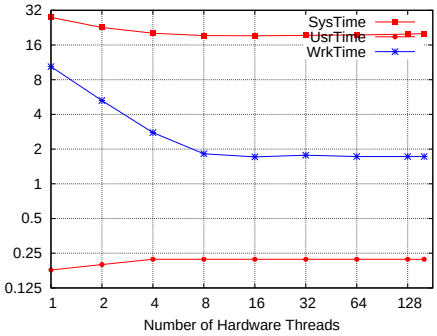
```
 1  #include <tnt.h>
 2
 3  #define TREE_DEPTH 10
 4
 5  void tree_fn(int64_t *ptr_depth)
 6  {
 7    tnt_desc_t th_desc;
 8    int64_t    depth = (int64_t)ptr_depth;
 9    int64_t    error = 0;
10
11    if (depth > 1) {
12      if (tnt_create(&th_desc, &tree_fn, (void *)depth-1) ||
13          tnt_create(&th_desc, &tree_fn, (void *)depth-1)) {
14        error++;
15      }
16    }
17  }
18
19
20  int main(int argc, char *argv[])
21  {
22    int64_t depth = TREE_DEPTH;
23
24    tree_fn((void *)depth);
25  }
```

Figure 7.4: TNT Binary Tree Program

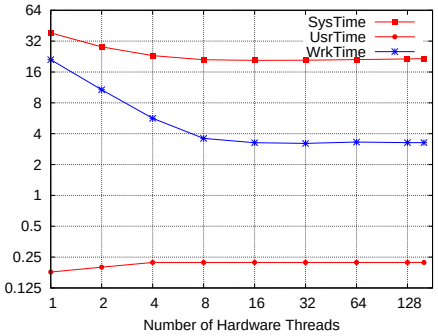
and they are immediately recycled by the runtime system. To measure the overhead imposed by thread management operations with a large number of active thread units, we wrote a second microbenchmark. This program spawns threads in a binary tree fashion. The main function executes a procedure that spawns two threads and terminates. Each thread then executes the same procedure so four additional threads are created. The process continues until a specified depth is reached, at which point, the procedure returns immediately, see Figure 7.4.

We define “work time” or “work” as the elapsed time between the termination of the main function and the completion of all the work generated by the program. Because the main function only spawns two threads, it returns almost immediately. Therefore, the “user time” is meaningless. Similarly, the “system time” accounts for the initialization of the system, which for small problem sizes, dominates the execution of the microbenchmark. Thus, the “system time” is not meaningful either. For the tree microbenchmark the “work time” is a better indicator of the overhead incurred by the runtime system.

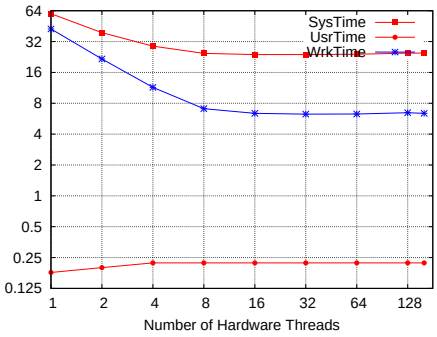
Figure 7.5 shows the execution of the binary tree microbenchmark for depths between six and ten. As expected, the "work time" increases with the problem size (depth), and for a given depth the "work time" decreases as we increase the number of hardware thread units. However, with more than eight thread units, the "work time" does not improve any further, which seems to indicate the system is not scalable. On the contrary, given the speed at which the program spawns threads, and the speed at which the runtime system can recycle them, the "work time" does not improve with more than eight threads because there is a bottleneck in the TNT library. In particular, there was a single queue to manage all the virtual threads. This result means that TNT should have 16 queues to recycle threads in parallel. In conventional SMP machines, the runtime system usually maintains one queue per processor to avoid this type of bottleneck. TNT does not require as many, which demonstrates its efficiency. In TNT, any thread that returns to the runtime system, executes the thread scheduler procedure described in Section 5.2.3. Even though the scheduler can be invoked by many threads in parallel, in the end all the threads access the same queue of virtual threads, hence the initial lack of scalability. It is worth noticing that even though the "work time" does not improve (beyond 8 thread units) it does not worsen either. This result means that the system is able to maintain a constant throughput, regardless of the load. Such a result confirms the efficiency of the runtime system.



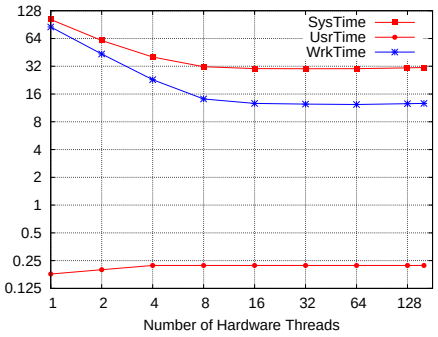
(a) Depth = 6



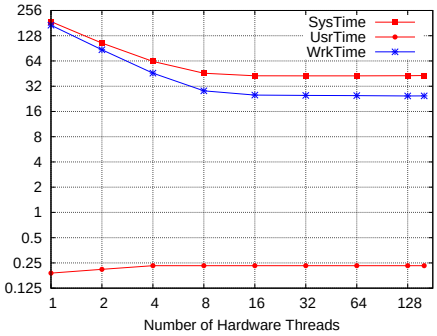
(b) Depth = 7



(c) Depth = 8



(d) Depth = 9



(c) Depth = 10

Figure 7.5: Tree Microbenchmark Execution Time [in Ten Thousand Clock Cycles]

Chapter 8

CONCLUSIONS

This dissertation has presented TiNy Threads (TNT), a Program Execution Model Aware Thread Virtual Machine (TVM) for the Cyclops-64 cellular supercomputer.

We believe that the Program Execution Model should be an integral part of a computing system, especially on high-end computing systems, to avoid unnecessary interference from the OS and subsequent performance degradation. For this reason, we propose a system software methodology that ensures that critical capabilities such as fine-grain multithreading and synchronization are exposed to the Program Execution Model via a narrow interface. Given the features of the Cyclops-64 many-core architecture, we implemented TNT as a user library that replaces the OS completely, yet provides direct access to critical hardware resources.

We define the operational semantics of the MAGMA Program Execution Model, and we complete an early implementation of MAGMA using TNT, demonstrating that the Cyclops-64 TVM provides a sound model for the research and development of advanced Program Execution Models. MAGMA thread model is based on event-driven non-preemptive threaded procedures with dataflow-like synchronization. MAGMA exploits locality using percolation to migrate data among different levels of the memory hierarchy before the computation starts.

8.1 Future Work

This dissertation has defined the TNT Thread Virtual Machine to support the development of Program Execution Models on many-core architectures. One obvious continuation of the current work is to extend the TNT Thread Virtual Machine to a multi-chip environment. There are some natural limitations restricting the maximum number of chips that FAST can simulate. These limitations, rather than restrictions on the model itself, prevented us from developing a multi-chip framework. Once C64 systems become available in the upcoming Fall, these limitations will cease to be an issue.

Once MAGMA adopts the TNT multi-chip environment, we intend to extend the percolation model so that the programmer has finer controls over local memory. The destination address of a percolation operation involving two nodes could be either off-chip or on-chip memory. In both cases the memory will be local to the processing element residing on the target node.

In the Cyclops-64 architecture, ten thread units share a 32KB I-cache. To improve I-cache utilization, TNT's thread scheduler could be optimized to run threads with the same thread activation pointer on thread units that share an I-cache.

Another area to continue working on is the MAGMA precompiler. Unlike EARTH and Cilk, currently there is not a MAGMA precompiler. A precompiler would accept MAGMA programs as described in Chapter 6 and would generate standard C code with function calls to the MAGMA runtime system. Such a precompiler could be integrated into the C64 toolchain to facilitate the development of MAGMA applications.

Appendix

N-QUEENS SOURCE CODE

A.1 Sequential

Figure A.1 shows an example of a sequential *N-Queens* program.

A.2 MAGMA

Figures A.2 and A.3 show the full MAGMA *N-Queens* program.

A.3 EARTH

Figures A.4 and A.5 show the source code of the *N-Queens* code in EARTH.

A.4 Cilk

Figures A.6 and A.7 show the Cilk program for the enumeration of the *N-Queens*.

```
 1  #include <stdlib.h>
 2  #include <stdio.h>
 3  #define MAX_QUEENS 24
 4
 5  int safe (int board[], int row, int col)
 6  {
 7    int rowchk, colchk;
 8
 9    for (rowchk = 0; rowchk < row; rowchk++) {
10      colchk = board[rowchk];
11      if ((col == colchk) || (row - rowchk == col - colchk) ||
12          (row - rowchk == colchk - col))
13        return 0;
14    }
15    return 1;
16  }
17
18  int sequeens(int n, int row, int start_col, int board[])
19  {
20    int col, sols_this_col, sols_other_cols;
21
22    if (row >= n) return 1;
23    for (col = start_col; col < n; col++) {
24      if (safe(board, row, col)) {
25        board[row] = col;
26        sols_this_col = sequeens(n, row+1, 0, board);
27        sols_other_cols = sequeens(n, row, col+1, board);
28        return (sols_this_col + sols_other_cols);
29      }
30    }
31    return 0;
32  }
33
34  main (int argc, char *argv[])
35  {
36    int n, result, board[MAX_QUEENS];
37
38    n = atoi(argv[1]);
39    printf("queens (%d) running on %d processors\n", n, 1);
40    result = sequeens(n, 0, 0, board);
41    printf("Number of solutions: %d\n", result);
42  }
```

Figure A.1: Sequential *N-Queens* Program

```
 1  nqueens(int n, int row, int start_col, int board[], SYNC_SLOT result)
 2  {
 3    int col;
 4
 5    for (col=start_col; col<n; col++) {
 6      if (safe(board, row, col)) {
 7        board[row] = col;
 8        SPAWN(sum, result, SYNC(this_col), SYNC(other_cols));
 9        if (row+1 == n) {
10          DATA_SYNC(1, SLOT_ADR(this_col));
11        }
12        else {
13          SPAWN(nqueens, n, row+1, 0, SYNC(board), SLOT_ADR(this_col));
14          GATHER_BLK_SYNC(gather, row, board, n*sizeof(int), SLOT_ADR(board));
15        }
16        if (col+1 == n) {
17          DATA_SYNC(0, SLOT_ADR(other_cols));
18        }
19        else {
20          SPAWN(nqueens, n, row, col+1, SYNC(board), SLOT_ADR(other_cols));
21          GATHER_BLK_SYNC(gather, row, board, n*sizeof(int), SLOT_ADR(board));
22        }
23        break;
24      }
25    }
26    if (col == n) {
27      DATA_SYNC(0, result);
28    }
29  }
30
31  sum(SYNC_SLOT int result, int sols_this_col, int sols_other_cols)
32  {
33    DATA_SYNC(sols_this_col+sols_other_cols, result);
34  }
35
36  gather(void *dst, void *src, int n, void *arg)
37  {
38    memcpy(dst, src, (int)arg);
39  }
```

Figure A.2: MAGMA *N-Queens* Program

```
 1  #include <stdio.h>
 2  #include <stdlib.h>
 3  #include <magma.h>
 4
 5  #define MAX_BOARD_SIZE 24
 6
 7  int safe(int board[], int row, int col)
 8  {
 9    int rowchk, colchk;
10
11    for (rowchk=0; rowchk<row; rowchk++) {
12      colchk = board[rowchk];
13      if ((col == colchk) ||
14          (row - rowchk == col - colchk) ||
15          (row - rowchk == colchk - col))
16        return 0;
17    }
18    return 1;
19  }
20
21  void print_usage_and_die(void)
22  {
23    fprintf(stderr,"usage (with 1 <= size <= %d):\n"
24            "  queens size\n", MAX_BOARD_SIZE);
25    exit(1);
26  }
27
28  int main (int argc, char *argv[])
29  {
30    int n;
31    int place[MAX_BOARD_SIZE];
32
33    if (argc != 2)
34      print_usage_and_die();
35
36    n = atoi(argv[1]);
37    if (n<1 || n>MAX_BOARD_SIZE)
38      print_usage_and_die();
39
40    SPAWN(done, SYNC(res), n);
41    SPAWN(nqueens, n, 0, 0, SYNC(board), SLOT_ADR(res));
42    BLK_SYNC(board, n*sizeof(int), 0, SLOT_ADR(board));
43  }
44
45  done(int result, int n)
46  {
47    printf ("queens(%d) = %d\n", n, result);
48  }
```

Figure A.3: MAGMA *N-Queens* `main` Function

```
 1  THREADED nqueens(int n, int row, int start_col,
 2                   int *GLOBAL previous, int *GLOBAL result, SPTR done)
 3  {
 4    int own_board[MAX_BOARD_SIZE],
 5        col,
 6        sols_this_col, sssols_other_cols;
 7
 8    sols_this_col = 0;
 9    sols_other_cols = 0;
10    BLKMOV_SYNC(previous, TO_GLOBAL(own_board), row*sizeof(int), DATA_RECEIVED);
11
12    FIBER DATA_RECEIVED <* 1 *> {
13      for(col=start_col; col<n; col++) {
14        if (safe(own_board, row, col)) {
15          own_board[row] = col;
16          if (row+1 == n) {
17            sols_this_col = 1;
18            SYNC(DONE);
19          }
20          else {
21            TOKEN(nqueens, n, row+1, 0, TO_GLOBAL(own_board),
22                  TO_GLOBAL(&sols_this_col), TO_SPTR(DONE));
23          }
24          if (col+1 == n) {
25            SYNC(DONE);
26          }
27          else {
28            TOKEN(nqueens, n, row, col+1, TO_GLOBAL(own_board),
29                  TO_GLOBAL(&sols_other_cols), TO_SPTR(DONE));
30          }
31          break;
32        }
33      }
34      if (col == n) {
35        SPAWN(DONE);
36      }
37    }
38
39    FIBER DONE <* 2 *> {
40      PUT_SYNC(sols_this_col + sols_other_cols, result, done);
41      TERMINATE;
42    }
43  }
```

Figure A.4: EARTH *N-Queens* Program

```
 1  #include <stdio.h>
 2  #include <stdlib.h>
 3
 4  #define MAX_BOARD_SIZE 24
 5
 6  int safe( int board[], int row, int col )
 7  {
 8    int rowchk, colchk;
 9
10    for( rowchk = 0; rowchk < row; rowchk++ ) {
11      colchk = board[rowchk];
12      if ( (col == colchk) ||
13           (row - rowchk == col - colchk) ||
14           (row - rowchk == colchk - col) )
15        return( 0 );
16    }
17    return( 1 );
18  }
19
20  void print_usage_and_die()
21  {
22    fprintf( stderr, "usage (with 1 <= size <= %d):\n"
23             "    queens size\n", MAX_BOARD_SIZE );
24    exit(1);
25  }
26
27  THREADED MAIN( int argc, char *argv[] )
28  {
29    int n, result, place[MAX_BOARD_SIZE];
30
31    if (argc != 2)
32      print_usage_and_die();
33
34    n = atoi(argv[1]);
35    if ( n < 1 || n > MAX_BOARD_SIZE )
36      print_usage_and_die();
37
38    INVOKE( 0, nqueens, n, 0, 0, TO_GLOBAL(place),
39            TO_GLOBAL(&result), TO_SPTR(DONE) );
40
41    FIBER DONE <* 1 *> {
42      printf( "Number of solutions for %d queens = %d\n",
43              n, result );
44      TERMINATE;
45    }
46  }
```

Figure A.5: EARTH *N-Queens* `main` Function

```
 1  cilk int nqueens(int n, int row, int start_col, int *previous)
 2  {
 3    int own_board[MAX_BOARD_SIZE], col, sols_this_col, sols_other_cols;
 4
 5    sols_this_col = sols_other_cols = 0;
 6    memcpy(own_board, previous, row*sizeof(int));
 7
 8    if (row >= n) return 1;
 9    for (col=start_col; col<n; col++) {
10      if (safe(own_board, row, col)) {
11        own_board[row] = col;
12        sols_this_col = spawn nqueens(n, row+1, 0, own_board);
13        sols_other_cols = spawn nqueens(n, row, col+1, own_board);
14        sync;
15        return (sols_this_col + sols_other_cols);
16      }
17    }
18    return 0;
19  }
```

Figure A.6: Cilk *N-Queens* Program

```
 1  #include <stdio.h>
 2  #include <stdlib.h>
 3  #include <cilk.h>
 4
 5  #define MAX_BOARD_SIZE 24
 6
 7  int safe( int board[], int row, int col )
 8  {
 9    int rowchk, colchk;
10
11    for( rowchk = 0; rowchk < row; rowchk++ ) {
12      colchk = board[rowchk];
13      if ( (col == colchk) ||
14           (row - rowchk == col - colchk) ||
15           (row - rowchk == colchk - col) )
16        return( 0 );
17    }
18    return( 1 );
19  }
20
21  void print_usage_and_die()
22  {
23    fprintf( stderr, "usage (with 1 <= size <= %d):"
24             " queens size\n", MAX_BOARD_SIZE );
25    exit(1);
26  }
27
28  cilk int cilk_main(int argc, char *argv[])
29  {
30    int n, result, place[MAX_BOARD_SIZE];
31
32    if (argc != 2)
33      print_usage_and_die();
34
35    n = atoi(argv[1]);
36    if ( n < 1 || n > MAX_BOARD_SIZE )
37      print_usage_and_die();
38
39    result = spawn nqueens(n, 0, 0, place);
40    sync;
41    printf("Number of solutions for %d queens = %d\n",
42           n, result );
43    return 0;
44  }
```

Figure A.7: Cilk *N-Queens* `main` Function

BIBLIOGRAPHY

[1] CAPSL technical memo, Department of Electrical and Computer Engineering, University of Delaware, Newark, Delaware. In ftp://ftp.capsl.udel.edu/pub/doc/memos.

[2] Defense Advanced Research Projects Agency. DARPA selects three high productivity computing systems projects, July 9, 2003. Press Release.

[3] F. Allen, G. Almási, W. Andreoni, and D. Beece et. al. Blue Gene: A vision for protein science using a petaflop supercomputer. *IBM Systems Journal*, 40(2), November 2001.

[4] Pete Beckman, Kamil Iskra, Kazutomo Yoshii, and Susan Coghlan. Operating system issues for petascale systems. *ACM SIGOPS Operating Systems Review*, 40(2):29–33, April 2006.

[5] A. Bensoussan, C. T. Clingen, and R. C. Daley. The MULTICS virtual memory: Concepts and design. *Communications of the ACM*, 15(5):308–318, May 1972.

[6] Bogdan Botezatu. Nvidia confirms its 1 billion-transistor GT-200 GPU. Softpedia, April 15, 2008. URL http://news.softpedia.com/news/Nvidia-Confirms-its-1-Billion-Transistor-GT-200-GPU-83462.shtml.

[7] R. Brightwell, R. Riesen, K. Underwood, T.B. Hudson, P. Bridges, and A.B. Maccabe. A performance comparison of Linux and a lightweight kernel. In *Proceedings of the IEEE International Conference on Cluster Computing*, pages 251–258, Hong Kong, China, December 1–4, 2003. Argonne National Laboratory.

[8] Ron Brightwell and Lee Ann Fisk. Scalable parallel application launch on Cplant. In *Proceedings of SC2001: High Performance Networking and Computing*, page 263, Denver, Colorado, November 10–16, 2001.

[9] Ron Brightwell, Arthur B. MacCabe, and Rolf Riesen. On the appropriateness of commodity operating systems for large-scale, balanced computing systems. In Werner, editor, *Proceedings of the 17th International Parallel and Distributed Processing Symposium*, page 68, Nice, France, April 22–26, 2003. IEEE Computer Society.

[10] Doug Burger and James R. Goodman. Billion-transistor architectures. *Computer*, 30(9):46–49, September 1997. Guest Editors' Introduction.

[11] Computation Center. The darmouth time-sharing system, October 19, 1964.

[12] The MIT Computation Center. The compatible time-sharing system. a programmer's guide, 1963.

[13] Computing ResearchAssociation. *Proceedings of the Workshop on The Roadmap for The Revitalization of High-End Computing*, June 16–18, 2003. HEC user survey.

[14] David E. Culler, Seth C. Goldstein, Klaus E. Schauser, and Thorsten von Eicken. TAM – a compiler controlled threaded abstract machine. *Journal of Parallel and Distributed Computing*, 18:347–370, July 1993.

[15] Robert C. Daley and Jack B. Dennis. Virtual memory, processes and sharing in MULTICS. *Communications of the ACM*, 11(5):306–312, May 1968.

[16] Juan del Cuvillo, Weirong Zhu, Ziang Hu, and Guang R. Gao. FAST: A functionally accurate simulation toolset for the Cyclops64 cellular architecture. In *Proceedings of the Workshop on Modeling, Benchmarking and Simulation*, pages 11–20, Madison, Wisconsin, June 4, 2005. Held in conjunction with the 32nd Annual International Symposium on Computer Architecture.

[17] Juan del Cuvillo, Weirong Zhu, Ziang Hu, and Guang R. Gao. Toward a software infrastructure for the Cyclops-64 cellular architecture. In R.G. Deupree and J.A. Adams, editors, *Proceedings of the 20th International Symposium on High-Performance Computing in an Advanced Collaborative Environment*, pages 55–61, St. John's, Newfoundland and Labrador, Canada, May 14–17, 2006.

[18] Juan B. del Cuvillo, Robert Klosiewicz, and Yingping Zhang. A software development kit for DIMES. CAPSL Technical Note 10, Department of Electrical and Computer Engineering, University of Delaware, Newark, Delaware, September 2003.

[19] Monty Denneau. Blue Gene. In *Proceedings of SC2000: High Performance Networking and Computing*, Dallas, Texas, November 4–10, 2000. ACM SIGARCH and IEEE Computer Society. URL http://www.supercomp.org/sc2000/proceedings/.

[20] Jack B. Dennis. Segmentation and the design of multiprogrammed computer systems. *Journal of the ACM*, 12(4):589–602, October 1965.

[21] Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–155, October 1966.

[22] Ulrich Drepper and Ingo Molnar. The native POSIX thread library for linux. Technical report, Read Hat, Inc., January 30, 2003.

[23] Joel Emer, Mark D. Hill, Yale N. Patt, Joshua J. Yi, Derek Chiou, and Resit Sendag. Single-threaded vs. multithreaded: Where should we focus? *IEEE Micro*, 27(6):14–24, November/December 2007.

[24] Adiga et al. An overview of the BlueGene/L supercomputer. In *Proceedings of SC2002: High Performance Networking and Computing*, Baltimore, Maryland, November 16–22, 2002. IEEE Computer Societyand ACM SIGARCH.

[25] George Almási et al. Blue Gene/L, a system-on-a-chip. In *Proceedings of the IEEE International Conference on Cluster Computing*, pages 349–350, Chicago, Illinois, September 2002. Argonne National Laboratory.

[26] Jonathan Fildes. Chips pass two billion milestone. BBC News, February 4, 2008. URL http://news.bbc.co.uk/2/hi/technology/7223145.stm.

[27] John Fotheringh. Dynamic storage allocation in the Atlas computer, including an automatic use of a backing store. *Communications of the ACM*, 4(10):435–436, October 1961.

[28] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 212–223, Montréal, Québec, June 17–19, 1998. *SIGPLAN Notices,* 33(6), June 1998.

[29] Seth Copen Goldstein, Klaus Erik Scheuser, and David E. Culler. Lazy threads: Implementing a fast parallel call. *Journal of Parallel and Distributed Computing*, 37(1):5–20, August 1996.

[30] Supercomputing Technologies Group. Cilk 5.4.6 reference manual, 2008. URL = http://supertech.lcs.mit.edu/cilk.

[31] Mattew Haines and Koen Langedoen. Platform-independent runtime optimizations using OpenThreads. In *Proceedings of the 11th International Parallel Processing Symposium*, pages 460–466, Geneva, Switzerland, April 1–5, 1997. IEEE Computer Society and ACM SIGARCH.

[32] Lance Hammond, Basem A. Nayfeh, and Kunle Olukotun. A single-chip multiprocessor. *Computer*, 30(9):79–85, September 1997.

[33] Herbert H. J. Hum, Olivier Maquelin, Kevin B. Theobald, Xinmin Tian, Guang R. Gao, and Laurie J. Hendren. A study of the EARTH-MANNA multithreaded system. *International Journal of Parallel Programming*, 24(4):319–347, August 1996.

[34] Cray Inc. Cascade team selected for phase II of DARPA HPCS program, July 9, 2003. URL http://www.cray.com/products/programs/cascade.html.

[35] L.V. Kalè, J. Yelon, and T. Knauff. Threads for interoperable parallel programming. In David C. Sehr, Uptal Banerjee, David Gelernter, Alexandru Nicolau, and David A. Padua, editors, *Proceedings of the 9th International Workshop on Languages and Compilers for Parallel Computing*, number 1239 in Lecture Notes in Computer Science, pages 534–552, Ithaca, New York, August 8–10, 1996. Springer-Verlag.

[36] Suzanne M. Kelly and Ron Brightwell. Software architecture of the light weight kernel, Catamount. Technical report, Sandia National Laboratories, Albuquerque, New Mexico, 2005.

[37] David Keppel. Tools and techniques for building fast portable threads packages. Technical Report UW-CSE-93-05-06, Department of Computer Science and Engineering,University of Washington, May 1993.

[38] T. Kilburn, D.B.C. Edwards, M.J. Lanigan, and F.H. Summer. One-level storage system. In *IRE Transactions*, volume 2, pages 223–235, April 1962.

[39] Christoforos E. Kozyrakis, Stylianos Perissakis, David Patterson, Thomas Anderson, Krste Asanovic, Neal Cardwell, Richard Fromm, Jason Golbus, Benjamin Gribstad, Kimberly Keeton, Randi Thomas, Noah Treuhaft, and Katherine Yelick. Scalable processors in the billion-transistor era: IRAM. *Computer*, 30(9):75–78, September 1997.

[40] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, September 1979.

[41] Mikko H. Lipasti and John Paul Shen. Superspeculative microarchitecture for beyond AD 2000. *Computer*, 30(9):59–66, September 1997.

[42] Sally A. McKee. Reflexions on the memory wall. In *Proceedings of the 1st Conference on Computing Frontiers*, pages 162–166, Ischia, Italy, April 14–16, 2004.

[43] Ronald G. Minnich, Matthew J. Sottile, Sung-Eun Choi, Erik Hendriks, and Jim McKie. Right-weight kernels: an off-the-shelf alternative to custom light-weight kernels. *ACM SIGOPS Operating Systems Review*, 40(2):22–28, April 2006.

[44] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(18), April 19, 1965.

[45] Frank Mueller. Pthreads library interface. Technical report, Department of Computer Science, Florida State University, July 1993.

[46] IBM Research News. IBM wins DARPA funding, July 9, 2003. URL http://domino.research.ibm.com/comm/pr.nsf/pages/news.20030710_darpa.html.

[47] Dimitrios S. Nikolopoulos, Eleftherios D. Polychronopoulos, and Theodore S. Papatheodorou. Efficient runtime thread management for the Nano-Threads programming model. In *Proceedings of the 2nd IPPS/SPDP Workshop on Runtime Systems for Parallel Programming*, pages 183–194, Orlando, Florida, March 30, 1998. IEEE Computer Society and ACM SIGARCH.

[48] A. Oliner, Ramendra K. Sahoo, José E. Moreira, Manish Gupta, and Anand Sivasubramaniam. Fault-aware job scheduling for BlueGene/L systems. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, page 64, April 2004.

[49] Yale N. Patt, Sanjay J. Patel, Marius Evers, Daniel H. Friendly, and Jared Stark. One billion transistors, one uniprocessor, one chip. *Computer*, 30(9):51–57, September 1997.

[50] Fabrizio Petrini, Darren J. Kerbyson, and Scott Pakin. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q. In *Proceedings of SC2003: High Performance Networking and Computing*, page 55, Phoenix, Arizona, November 15–21, 2003. IEEE Computer Societyand ACM SIGARCH.

[51] Jonathan Rosenberg. LWP user manual. Technical Report CMU-ITC-85-037, Information Technology Center, Carnegie-Mellon University, June 1985.

[52] Hirofumi Sakane, Levent Yakay, Vishal Karna, Clement Leung, and Guang R.Gao. DIMES: An iterative emulation platform for multiprocessor-system-on-chip designs. In *Proceedings of the IEEE International Conference on Field-Programmble Technology*, pages 244–251, Tokio, Japan, December 15–17, 2003. University of Tokio.

[53] Edi Shmueli, George Almási, Jose Brunheroto, José Castaños, Gabor Dozsa, Sameer Kumar, and Derek Lieber. Evaluation the effect of replacing the CNK with linux on the compute-nodes of Blue Gene/L. In *Conference Proceedings, 2008 International Conference on Supercomputing*, Island of Kos, Greece, June 7–12, 2008. ACM SIGARCH.

[54] Wei Shu. Runtime support for user-level ultra lightweight threads on massively parallel distributed memory machines. In *Proceedings of Frontiers '95: The Fifth*

Symposium on the Frontiers of Massively Parallel Computation, page 448, Annapolis, Maryland, February 6–9, 1995. IEEE Computer Society.

[55] James E. Smith and Sriram Vajapeyam. Trace processors: Moving to fourth-generation microarchitectures. *Computer*, 30(9):68–74, September 1997.

[56] Jon Stearley. Towards a specification for measuring Red Storm reliability, availability, and serviceability (RAS). Technical report, Sandia National Laboratories, Albuquerque, New Mexico, May 2005.

[57] Guy Tremblay, Kevin B. Theobald, Christopher J. Morrone, Mark D. Butala, José Nelson Amaral, and Guang R. Gao. Threaded-C language reference manual (release 2.0). CAPSL Technical Memo 39, Department of Electrical and Computer Engineering, University of Delaware, Newark, Delaware, September 2000. In ftp://ftp.capsl.udel.edu/pub/doc/memos.

[58] Ioannis E. Venetis and Theodore S. Papatheodorou. Tying memory management to parallel programming models. In Wolfgang Lehner Wolfgang E. Nagel, Wolfgang V. Walter, editor, *Proceedings of the 12th International Euro-Par Conference*, number 4128 in Lecture Notes in Computer Science, pages 666–675, Dresden, Germany, August 28–September 1, 2006. Springer-Verlag.

[59] Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer. Capriccio: Scalable threads for internet services. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pages 268–281, Bolton Landing, New York, October 19–22, 2003. ACM SIGOPS.

[60] Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, Jonathan Babb, Saman Amarasinghe, and Anant Agarwal. Baring it all to software: RAW machines. *Computer*, pages 86–93, September 1997.

[61] Boris Weissman. Active threads: An extensible and portable light-weight thread system. Technical Report ICSI TR-97-036, International Computer Science Institute, University of California at Berkeley, October 1997.

[62] Win A. Wulf and Sally A. McKee. Hitting the memory wall: Implications of the obvious. *ACM SIGARCH Computer Architecture News*, 23(1):20–24, March 1995.

[63] Yuan Zhang, Weirong Zhu, Fei Chen, Ziang Hu, and Guang R. Gao. Lamport order revisit: A study on how to efficiently achieve sequential consistency on a modern multiprocessor-on-a-chip architecture. CAPSL Technical Memo 53, Department of Electrical and Computer Engineering, University of Delaware, Newark, Delaware, March 2004. In ftp://ftp.capsl.udel.edu/pub/doc/memos.