Hardware Microkernels - A Novel Method for Constructing Operating Systems for Heterogeneous Multi-Core Platforms
Hardware Microkernels - A Novel Method for Constructing Operating Systems for Heterogeneous Multi-Core Platforms

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Engineering

By

Jason M. Agron
University of Kansas
Bachelor of Science in Computer Engineering, 2004
University of Kansas
Master of Science in Computer Engineering, 2006

August 2010
University of Arkansas
Abstract

Today’s standard operating systems (OSes) do not have the proper structure, nor support for executing on heterogeneous multi- and many-core platforms. These deficiencies manifest in symptomatic issues that range from relatively low-level application binary interface (ABI) compatibility problems, to higher-level concerns such as application programmer interface (API) compatibility, synchronization, and caching issues. These issues in turn reflect back up to the programmer through a lack of a standard set of OS services. Non-uniformities in the hardware architecture are also reflected back into the programming model. In total these issues combined break down the most fundamental benefit of modern operating systems; abstracting the differences between hardware platforms under a portable and uniform virtual machine model. This breakdown is disconcerting as historical precedence established during the earlier parallel era in the 1980s showed that without a uniform virtual machine model a strong customer base cannot be established. The extreme difficulty associated with rewriting an application for each new architecture put the use of parallel machines outside the grasp of all but the most proficient computer scientists. Thinking machines went as far as placing a computer scientist on site to work with domain scientists and other users to help port code onto their machine. For the current wave of architecture changes sweeping the general purpose computing domain to not simply repeat history a careful rethinking is required of the very structure of our traditional software protocols stacks, and the hardware primitives on which they are implemented. This rethinking must address both older left over issues from the prior parallel processing era now mixed with new challenges that result from heterogeneous sets of scalable numbers of cores that will expand at a rate following Moore’s law. Many of the issues associated with supporting parallel processing on heterogeneous cores can be traced back to both microarchitectures and software protocol stacks developed for single CPU systems. Simply adding additional software abstraction layers on top of lower level software and hardware infrastructure that provide inefficient
and incomplete mechanisms for supporting our concurrency models is not a long term solution. In thesis, we explore how operating system methods must change if we are to provide programmers with their familiar uniform virtual machine model across next generation heterogeneous manycores. We observe that meeting this requirement may very well require a shift in the current hardware/software (HW/SW) boundary that has been in existence for over 45 years. Shifting the HW/SW interface may be key to building a common set of OS services that are equally accessible to all components in a heterogeneous system, regardless of differences in instruction-set architecture (ISA), micro-architecture, etc. This may be critical to enable standard parallel programming models and abstractions to be used in heterogeneous systems, and promote portability. Additionally, a HW/SW co-designed OS kernel could provide low-latency system services, providing an efficient base for constructing ultra-scalable many-core platforms.
This dissertation is approved for recommendation to the Graduate Council

Dissertation Director:

________________________________________

Dr. David Andrews, Ph.D.

Dissertation Committee:

________________________________________

Dr. Jia Di, Ph.D.

________________________________________

Dr. John Gauch, Ph.D.

________________________________________

Dr. Scott Smith, Ph.D.
Dissertation Duplication Release

I hereby authorize the University of Arkansas Libraries to duplicate this dissertation when needed for research and/or scholarship.

Agreed

______________________________
Jason M. Agron
Acknowledgements

I would like to thank my parents, sister, Julie, my grandparents, and all of my relatives that have helped to keep me driven as well as support me over the years.

I would also like to thank my Cousin Frank and my Uncle David and Aunt Marilyn for encouraging me to continue in my pursuit of education.

Additionally, my critical thinking skills, breadth of knowledge, and quality of research has been greatly improved through the friendship and help of Wesley Peck, Garrin Kimmell, Andy Schmidt, Klaus Danne, Enno Lübbers, Heiner Giefers, and Paul Kaufmann.

I would also like to thank my advisor, Dr. David Andrews, for providing me with so many great opportunities, experiences, and continued mentorship over the years.

Finally, I would like to thank my dissertation committee, including my advisor-from afar, Dr. Marco Platzner, for participating in my academic research career as both a supervisor, mentor, and friend.
Contents

1 Introduction 4
  1.1 Multi-Core: Fighting Against Power, Heat, and Bandwidth Constraints . . 5
  1.2 Heterogeneity: Both a Need and a Bane . . . . . . . . . . . . . . . . . . 6
  1.3 Scalability: The New Metric for Multi-Core . . . . . . . . . . . . . . . 10
  1.4 Dissertation Contributions and Organization . . . . . . . . . . . . . . . 12

2 Background 14
  2.1 Shared Memory: Synchronization and Caching . . . . . . . . . . . . . . . 15
    2.1.1 Canonical Example . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
    2.1.2 Mutual Exclusion - Lock Policy Vs. Mechanism . . . . . . . . . . . . 18
    2.1.3 Cache Coherency . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
    2.1.4 New Synchronization Methods . . . . . . . . . . . . . . . . . . . . . . 29
  2.2 Parallel Programming Models . . . . . . . . . . . . . . . . . . . . . . . . . 37
    2.2.1 Shared Memory Models . . . . . . . . . . . . . . . . . . . . . . . . . . . 39
    2.2.2 Message Passing Models . . . . . . . . . . . . . . . . . . . . . . . . . . 42
  2.3 Heterogeneous Multi-Core Systems . . . . . . . . . . . . . . . . . . . . . 44
    2.3.1 Arachne . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45
    2.3.2 IBM Cell . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47
    2.3.3 Intel EXOCHI . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52
    2.3.4 NVIDIA CUDA . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
    2.3.5 Inria StarPU . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59
    2.3.6 Microkernels . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61

3 Design Overview 69

4 Hthreads Microkernel Framework 71
  4.1 Microkernel Cores . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 72
    4.1.1 Thread Manager . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 76
    4.1.2 Scheduler . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 77
    4.1.3 Synchronization Manager . . . . . . . . . . . . . . . . . . . . . . . . . . 82
    4.1.4 Condition Variable Manager . . . . . . . . . . . . . . . . . . . . . . . . 86
    4.1.5 CPU-Bypass Interrupt Scheduler (CBIS) . . . . . . . . . . . . . . . . 88
  4.2 Tying It All Together . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 90

5 Heterogeneous Coordination Layer 91
  5.1 Thread Interfaces . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 92
  5.2 Hardware Abstraction Layer . . . . . . . . . . . . . . . . . . . . . . . . . . . 96
  5.3 HAL + Cores = Heterogeneous OS . . . . . . . . . . . . . . . . . . . . . . . 100
6 Design Flow
6.1 Related Work ........................................... 103
6.2 Multiple, Architecture-Specific Executables .............. 105
6.3 Embedding Executables .................................. 105
6.4 Heterogeneous Compilation Utilities ...................... 108

7 Thesis Evaluation ........................................... 111
7.1 Evaluation Criteria ........................................ 111
  7.1.1 Defining Efficiency .................................. 111
  7.1.2 Defining Uniformity ................................. 112
7.2 Evaluation Platform Description .......................... 112
  7.2.1 Common Hardware Platform ......................... 112
  7.2.2 Remote Procedure Call Configuration .............. 114
7.3 Efficiency Experiments ................................... 116
  7.3.1 System Call Latency .................................. 119
  7.3.2 Application Scalability ............................... 121
7.4 Uniformity Experiments .................................... 124
  7.4.1 Parallel π Approximation ............................ 124
  7.4.2 Mailbox-based Communication ....................... 128
  7.4.3 Parallel IDEA Encryption ............................ 132
7.5 Tying It All Together: Real Examples ..................... 134
  7.5.1 Thread Pools and Work Queues ...................... 134
  7.5.2 Image Processing with Heterogeneous Device Drivers .. 136
  7.5.3 Dynamic Thread Dispatch ............................ 139
7.6 Summary .................................................. 149

8 Conclusion .................................................. 151
8.1 Research Contributions .................................... 151
8.2 Future Work .............................................. 153
  8.2.1 Heterogeneity Detection and Migration .............. 154
  8.2.2 Heterogeneous Compiler Technology ................ 155
  8.2.3 Platform Flexibility and Degree of Heterogeneity .... 156

References .................................................... 157
List of Figures

2.1 High-Level Code - Incrementing a Shared Variable ........................ 16
2.2 Low-Level Code - Incrementing a Shared Variable ........................ 16
2.3 Using Locks When Incrementing a Shared Variable ........................ 18
2.4 Pseudo-code Mutex Operations .............................................. 19
2.5 Mutex Operations Using Test-And-Set ....................................... 20
2.6 Mutex Operations Using Compare-And-Swap ................................. 21
2.7 Mutex Operations Using Load-Linked/Store-Conditional .................. 22
2.8 Incoherent Multi-Processor Cache ............................................. 23
2.9 MESI Protocol - State Transitions .......................................... 25
2.10 MOESI Protocol - State Transitions ......................................... 27
2.11 Multi-Word Atomic Operation Using LL/SC ................................. 32
2.12 ASF Implementation of Double-Word Compare-and-Swap (DCAS) [12] . 35
2.13 ASF Implementation of Load-Linked/Store-Conditional (LL/SC) [12] . 35
2.15 ASF Composition of Atomic Operations [12] ................................. 37
2.16 IBM Cell B.E. Architecture .................................................... 48
2.17 IBM Cell SPUFS Programming Environment .................................. 49
2.18 EXOCHI Programming Environment [121] .................................. 54
2.19 Example Codelet for Cell [21] .................................................. 60
2.20 OS Structures: Monolithic Vs. Microkernel ................................. 61

4.1 Original hthreads Architecture ............................................... 71
4.2 Encoded Requests: Placing Parameters in Address and Data Lines ..... 74
4.3 IPC Requests: Asynchronous CPU-to-CPU Communication ............... 75
4.4 Block Diagram of Thread Manager and Scheduler Components .......... 78
4.5 Block Diagram of Synchronization Manager

4.6 Block Diagram of Condition Variable Manager

4.7 Block Diagram of CBIS

5.1 Pseudocode for HAL Bootloop Process

5.2 Protocol Stack for a Heterogeneous System Call: Translating User-Level APIs to HW/SW Interactions

5.3 Uniform HAL Targets Common Set of OS Services

6.1 ELF File: 2 Non-Contiguous Sections

6.2 Embedding Before Flattening - Incorrect Instruction Placement

6.3 Embedding After Flattening - Correct Instruction Placement

6.4 Symbol Table Extraction Process and Generation of Thread Handles

6.5 Heterogeneous Compilation and Embedding Flow

7.1 Heterogeneous hthreads System

7.2 Processing Steps for RPC Mechanism

7.3 Synchronization Latency Histograms

7.4 Thread Create Latency Histograms

7.5 Scalability Test: Direct Vs. RPC

7.6 Pseudocode - Scalability Test

7.7 \(\pi\) Approximation Thread

7.8 Heterogeneous Fork/Join Pattern

7.9 Pi Approximation Scalability: Direct Vs. RPC

7.10 Heterogeneous Circular Pipeline

7.11 Pipeline Thread

7.12 Pipeline Scalability Equation

7.13 Mailbox Scalability: Direct Vs. RPC

7.14 Heterogeneous IDEA Encryption Performance Results
## List of Tables

2.1 Correct Interleaving - Incrementing a Shared Variable .......................... 16  
2.2 Incorrect Interleaving - Incrementing a Shared Variable ......................... 17  
2.3 MESI Legal State Matrix [123] ......................................................... 26  
2.4 MOESI Legal State Matrix [124] ....................................................... 28  
2.5 Complexity of Synch. Operation Implementations [110] .......................... 34  
4.1 Comparing Operations: IPC Amongst Processors Vs. Processor-to-Core IPC 76  
4.2 Thread Management Operations ......................................................... 79  
4.3 Scheduler Operations ................................................................. 81  
4.4 Synchronization Manager Operations ............................................... 84  
4.5 Condition Variable Manager Operations ......................................... 86  
4.6 CPU-Bypass Interrupt Scheduler (CBIS) Operations ............................ 88  
5.1 Virtual Hardware Thread Interface (V-HWTI) Memory-Map .................. 92  
7.1 Mean System Call Latencies (Clock Cycles) .................................... 121  
7.2 Dot-Product Results with Dynamic Dispatch .................................... 145
Terms and Definitions

**ABI** Application Binary Interface. Refers to the calling convention and data layout of a particular processor-compiler pair.

**API** Application Programmer Interface. The defined interface of a piece of software, often times a library or operating system.

**CISC** Complex Instruction Set Computer.

**CLB** Configurable Logic Block. A programmable block within the architecture of Field Programmable Gate Arrays (FPGAs).

**CPU** Central Processing Unit. A programmable hardware component, often referred to as processor or core.

**DMA** Direct Memory Access. Often referring to hardware devices that can perform memory-to-memory operations without processor assistance.

**DSP** Digital Signal Processor. A processor specialized for signal processing, often featuring vector and multiply-accumulate operations.

**FIFO** First-In, First-Out. A hardware component or data structure that exhibits First-In, First-Out behavior (e.g. a queue).

**FPGA** Field Programmable Gate Array. A hardware chip whose functionality can be changed post-fabrication.

**GPU** Graphics Processing Unit. A hardware component specialized for graphics processing.

**HAL** Hardware Abstraction Layer. A layer of software used to hide hardware-specific implementation details.
HDL  Hardware Description Language (e.g. VHDL or Verilog).

HdS  Hardware-Dependent Software. Software that is tightly integrated with a specific hardware component or platform.

HLL  High-Level Language.

HW   Abbreviation for Hardware.

ISA  Instruction Set Architecture. Also known as the instruction set of a particular processor.

IPC  Inter-Process Communication, when referring to software; or Inter-Processor Communication, when referring to hardware systems.

I/O  Input/Output.

Kernel The core component(s) of an operating system.

LIFO  Last-In, First-Out. A hardware component or data structure that exhibits Last-In, First-Out Behavior (e.g. a stack).

LUT  Lookup Table. A digital building block used to implement N-bit binary functions via lookup operations.

MIMD  Multiple-Instruction, Multiple-Data. A category of parallel programming/computational model in Flynn’s Taxonomy that is represented by systems capable of executing different instruction streams that are able to operate on different streams of data simultaneously. Multiprocessor (multi-core) systems fit into the MIMD category.

MISD  Multiple-Instruction, Single-Data. A category of parallel programming/computational model in Flynn’s Taxonomy that is represented by a machine that executes multiple instructions on a single piece of data. It can be argued that pipelines and systolic arrays fit into this category.
**MMIO** Memory-Mapped I/O. The process of performing I/O through memory-mapped interactions such as reads and writes (loads and stores).

**MMU** Memory-Management Unit.

**MPSoC** Multi-processor System-on-Chip. An SoC made up of multiple processors.

**OS** Operating System.

**RPC** Remote-Procedure Call.

**RTL** Register Transfer Level. An abstraction level of the hardware design process.

**SoC** System-on-Chip. A system in which all components are co-located on a single chip.

**SIMD** Single-Instruction, Multiple-Data. A category of parallel programming/computational model in Flynn’s Taxonomy that is represented by a vector processor that executes a single instruction on multiple data at once.

**SISD** Single-Instruction, Single-Data. A category of parallel programming/computational model in Flynn’s Taxonomy that is represented by a typical scalar processor.

**SW** Abbreviation for Software.

**RISC** Reduced Instruction Set Computer.
Chapter 1

Introduction

In its basic form, the operating system is merely a layer of software that abstracts a computing platform into a single virtual machine \([77, 113]\). System calls represent the virtual machines standard set of operations implemented as libraries for each specific machine. Programmers utilize the operating system’s set of virtual operations, with the knowledge that these same operations will be available on all machines. One of the most successful examples of portability provided by the operating system is the Basic Input/Output System or \textit{BIOS} created for IBM Personal Computers (PCs). The BIOS set of system services was adopted by all PC manufactures to form the de-facto standard for desktop computer systems. This same approach to standardization has been adopted in the operating system and middleware communities as evidenced by Windows/Linux, POSIX, MPI, OpenMP, to further enable application-level portability.

Code portability provided by operating systems has played a large role in enabling software programmers to ride the wave of technology changes and continual advancements predicted by Moore’s Law for over half a century. On average, the same piece of code could enjoy a doubling in performance with no change, every 18 to 24 months as a by-product of Moore’s Law \([90, 89]\)! But what was enabling such increases? Was it intense focus of electrical engineers on producing smaller and faster circuits? Was it the work of computer architects as they perfected CPU pipelines, caches, and out-of-order execution engines \([111]\)? Or, was it the work of those involved in improving memory fabrication technology? The answer is all of the above, but all happening underneath the virtual machine provided by the operating system.
1.1 Multi-Core: Fighting Against Power, Heat, and Bandwidth Constraints

In the late 1990s and early 2000s, the performance increases enabled by technology scaling began to meet new constraints: namely power, heat, and memory bandwidth [30, 111]. The rampant frequency scaling of the 1980s and 1990s resulted in faster CPUs but began requiring active cooling systems and huge power supplies to support the 100+ watt peak power draw of the power-hungry CPUs [98]. Additionally, memory bandwidth was languishing, as processor clock frequencies grew to a magnitude higher than bus and memory clocks. Architects were at a stopping point; super-scalar techniques were already exploiting the available parallelism from a single-threaded application, Caches were extremely efficient at exploiting available locality, branch prediction techniques were keeping pipelines filled, and clock frequencies were as high as possible without beginning to melt chips. These factors forced computer architects to turn 180 degrees away from large power hungry scalar processors towards providing continued performance increases through chips with multiple processors. Commonly called the many-core era, we are once again walking down the path of parallel architectures [30, 63]. Now, thread-level parallelism has become a dominating factor in computing performance.

This architectural jump may at first seem straightforward. However the reality is anything but, considering the current void in the availability of the software infrastructure needed to make the jump transparent to end-users and programmers [63]. Operating systems, compilers, and high-level languages must now adapt to new types of parallel machine architectures [52, 65]. High-level languages must allow programmers to specify task level concurrency, including the ability to efficiently synchronize and communicate between tasks running on scalable numbers of potentially heterogeneous processors. Compilers must adapt in order to translate higher-level languages to these new architectures as well. Most fundamental of all, operating systems must adapt to provide their fundamental virtualization capabilities in systems with multiple processing cores, new interconnect systems, and scalable memory hierarchies. It is not at all well understood how these changes should
bubble up the protocol stack. Many still question why at all the OS needs to change for the manycore era, as it has managed to resist change during the last half-century. The answer comes by considering two new issues brought with the manycore era that the operating system must now face: heterogeneity and scalability.

1.2 Heterogeneity: Both a Need and a Bane

One of the most important outcomes of the prior era of parallel computing is the recognition that performance increases are not at all guaranteed by simply adding more processors. Amdahl’s law helps us understand that speedup is restricted to only the section of code that can be parallelized. Just as important as Amdahl’s law is the understanding of how and where parallelism occurs from coarse grained process and thread levels, through finer grained instruction and data levels. These lessons are helping architects to create manycore chips with multiple numbers and types of processors to efficiently exploit parallelism where it exists. The multi-core era has already produced system architectures that contain multiple types of processing cores creating asymmetry at the hardware level. The inclusion of heterogeneous processing elements recognizes the needs of modern application classes. Some applications, such as graphical user interfaces, consist of branchy code that executes very well on traditional out-of-order execution processors. However, data-intensive applications, such as media-processing kernels are able to execute on SIMD/VLIW machines with much greater efficiency. It is very difficult for computer architects to produce a single processor that excels at executing programs from all application classes efficiently. Instead, architects have turned to systems that contain multiple types of compute elements built to serve the different needs of applications [47, 121, 80].

Unfortunately, our accepted operating systems were never created to abstract this type of heterogeneity within a unified virtual machine model. The internals of most operating systems are built with mechanisms that assume homogeneity at the hardware level [27, 80, 63]. The introduction of multiple, heterogeneous processing cores creates many opportunities,
but also presents many challenges that make integration and abstraction of new systems difficult for both system and application developers [80]. The presence of processors with different ISAs introduces code compatibility problems that disallow code sharing, and may even introduce data layout problems due to differences in endianess and the use of different application binary interfaces (ABIs) [117]. Differences in ISA prevent processors from directly invoking code of a different instruction set, while differences in ABI prevent a processor from directly reading and writing shared data structures. These types of incompatibilities can affect both OS-level and application-level code, and must be properly handled in order to ease heterogeneous system programmability. While heterogeneous multi-core architectures have been available in the past, never before have they become the standard offering in both the embedded, desktop, server, and high-performance computing domains [111]. Overall, Heterogeneous computing environments can be characterized by several challenges that are simply not present in homogeneous systems, such as:

- ISA Incompatibility
  
  - Inability to natively share code amongst processors with different instruction sets. This affects both application-level and system-level (OS) code.
  
  - Differences in ISA, including atomic operations, can prevent processors from synchronizing with one another.

- ABI Incompatibility
  
  - Code generated for different processors may interpret data, functions, and objects in different ways [117].

- Microarchitectural Incompatibilities
  
  - Differences in cache organization and memory hierarchy can create hardware incompatibilities.
Incompatible coherence protocols, some processors may not have coherent cache interfaces, etc.

These challenges may be tackled with both software- and hardware-based solutions, but the trade-offs for solving one often contradict another. For instance, interpreter-based software virtual machines can alleviate incompatibilities at both ISA- and ABI-levels by creating a portable code definition that can be interpreted on a variety of processors. Unfortunately, interpretation can introduce severe execution overheads due to non-native code execution [48], and just-in-time compilation techniques involve complex software stacks that may not be well-suited for lower-power embedded systems such as cell phones and other mobile platforms. Hardware-assisted solutions can support integration of heterogeneous processors while still supporting native code execution by providing ISA-agnostic services in the form of hardware accelerators for common functions. In the past, hardware-assisted operating system functions were often built into the internal microarchitecture of CPUs at the ISA-level [79, 76, 37]. This approach does provide efficiency benefits in terms of operating system overhead, however it can have negative effects when attempting to build OS support for heterogeneous processors with different ISAs. The work in this dissertation aims to show how OS functionality can be built in the form of hardware components that feature ISA-neutral interfaces. The goal is to provide services that remain accessible by any processor core capable of device interaction via memory-mapped I/O. This has the effect of decoupling OS services from CPU internal microarchitectural details, thus allowing a variety of CPU architectures to access a uniform set of run-time services [104]. Regardless of the style of hardware implementation, migration of important OS-level services into hardware must be well-thought out, as these hardware components will become permanent and unchangeable if integrated into silicon. The hardware components should be quite small, so as to not take away too much chip real estate from traditional processor and interconnect logic. Additionally, the components should be flexible, possibly even programmable, so they can be used and reconfigured in a variety of different ways. Finally, any hardware
cores should provide additional performance benefits when compared to traditional software implementations to justify their existence in silicon. Overall, a fine balance must be struck to build heterogeneous operating systems that incorporate the flexibility of software along with the speed and interface-neutrality of device hardware.

While the spectrum of solutions is wide, the amount of time for developing solutions is small. Processor vendors such as IBM and Intel are already producing heterogeneous multi-core hardware platforms, without the requisite software infrastructure to allow software programmers to target the new processing facilities. While some amount of software infrastructure does already exist, it is based on homogeneous systems with little to no physical concurrency [93, 80]. The current generation of manycore systems already violate many of these efforts simplifying assumptions as they already contain large amounts of physical concurrency, and contain heterogeneous processing elements with different ISAs, memory interfaces, and underlying programming models [80].

To add further complexity for operating systems designers, heterogeneity is not the only problem, as parallelism and concurrency also introduce challenges involving synchronization, performance, and scalability [63]. As an example, the Sony PlayStation 3, a gaming system that contains IBM’s Cell processor, has been plagued with bad press due to the difficulty programmers face when targeting the system. One of the main reasons for this difficulty is that the heterogeneous incompatibilities of the hardware platform have been exposed directly to the programmer [36, 29]. The lack of a well developed operating system infrastructure for the Cell, and other heterogeneous platforms, forces programmers to learn, understand, and use non-uniform, platform-specific features when programming. Not only is this extremely difficult, but it is also extremely non-portable [30]. Difficulty in programming creates challenges for writing new software for heterogeneous systems, and lack of portability makes it challenging to execute older, well-established code on new and future platforms [63]. The overarching reason for the challenges that currently face programmers, is the lack of a well-developed operating system for heterogeneous platforms.
1.3 Scalability: The New Metric for Multi-Core

In the past, processor utilization, ILP, and throughput metrics were key when evaluating new processors [111]. In the new multi-core era, full system utilization can only be achieved through proper application parallelization as suggested by Amdahl’s Law [13, 61]. While copious amounts of efforts have focused on parallel programming models and parallelization strategies, very little attention has been paid to the efficiency of the models on current- and next-generation multi-core hardware. As an example, auto-parallelization compiler technologies have existed for decades. However the focus has been mostly on auto-vectorization strategies to exploit instruction- and data-level parallelism (ILP/DLP) targeting a single-processor [30, 95]. Performance in modern multi-core systems is affected by ILP and DLP, but depends mostly on the exploitation of thread-level parallelism (TLP).

Parallel programming models almost always build upon task (process/thread) management facilities of the underlying operating system. This is true of the most popular of today’s parallel programming models: including PThreads, OpenMP, and MPI. These models will continue to be used in the future, but where will the underlying support for these models come from? First, operating systems themselves must be parallelized and ported to multi-core platforms. The scalability of applications is directly affected by the scalability of the underlying OS support mechanisms. The computing community must produce scalable operating systems frameworks in order to reap the full benefits of heterogeneous multi-core hardware. However, it is important that the community simultaneously considers heterogeneity, programmability, and portability alongside scalability when developing such frameworks. Ignoring heterogeneity can lead to non-uniform programming models that are both difficult to use and non-portable. Ignoring programmability can alienate programmers that have adhered to standard programming models [111]. Finally, ignoring scalability can lead to applications that are easy to write, but gain no performance increases as the number of cores scale with Moore’s law.
This is concerning as current operating systems frameworks for heterogeneous multi-core system are being constructed using the concept of a remote-procedure call (RPC) to lessen and abate heterogeneous incompatibilities [80]. While this approach is flexible, and can be used to adapt a traditional OS framework to operate in a heterogeneous environment, it does have its drawbacks [80]. For one, it is built upon the notion of non-uniform access to OS services, as only a host CPU can directly access OS services, while non-host (slave) CPUs must request services via RPC – essentially forcing host CPUs to act as servers for other heterogeneous processing cores [69, 70, 121, 57, 93, 107, 80].

The introduction of the RPC mechanism causes slave processors to incur all of the traditional overhead associated with OS requests as well as the overhead of the RPC mechanism itself. RPC mechanisms are currently constructed on top of mechanisms such as cache coherence protocols as well as inter-processor interrupts [101, 93, 121, 57], which are far from lightweight in terms of overhead and predictability [120]. Additionally, RPC mechanisms create increased contention for host CPU resources; as slave CPUs, as well as host-based tasks, are now competing for host CPU resources. This can create a bottleneck for servicing RPC requests, while also significantly upsetting the timing and behavior of host-level tasks. Furthermore, this bottleneck will only be exacerbated by scaling the number of processors as we move farther into the multi-core era. While the focus of the forthcoming parallel programming crisis has had its primary focus set on that of parallelizing applications, it is important to remember that an OS or run-time system is also an application that must be parallelized. Application-level programs cannot and will not scale without an underlying operating system that is both efficient and scalable.

To complicate things further, non-uniform access to OS services can and has been reflected into the programming model of heterogeneous multi-core systems. This has the effect of producing programming model abstractions that cannot be used uniformly within a heterogeneous multi-core system; forcing programmers to manually place hard partitions into their code destined for execution on CPUs of differing capability. This not only makes
programming heterogeneous systems exceedingly difficult, but it creates severe portability challenges for both newly developed applications and legacy code. A uniform programming model greatly facilitates the porting of code that was previously developed on homogeneous systems, while simultaneously producing a familiar programming environment for current- and next-generation multi-core architectures. Clearly, a framework that provides uniform services to programmers will aid in porting our entire software infrastructure to the new generation of heterogeneous multi-core devices.

1.4 Dissertation Contributions and Organization

The thesis of this work is that a hardware-based microkernel operating system can provide both efficient and uniform services in heterogeneous multi-core environments, in ways that are not readily achievable in software-only operating systems.

To support this thesis statement, the dissertation provides the following set of contributions:

- Heterogeneous extensions to the hthreads operating system kernel and cores.

- Development of an ISA-neutral interface to enable operating system interactions with heterogeneous processors.

- Multithreaded programming model extensions to enable POSIX-style threaded programs to execute in heterogeneous environments.

- A heterogeneous compilation and linking flow that enables programmers to build executable binary files for heterogeneous multi-core systems.

- Experimental heterogeneous multi-core platform built on Xilinx Platform FPGA technology.

- A set of example programs and benchmarks to examine the differences between RPC-based and direct-access heterogeneous operating system frameworks.
Overall, this dissertation aims to quantify the performance differences between traditional software-based heterogeneous operating systems structured around RPC-based invocation methods, and a hardware-based microkernel operating system that provides direct access to operating system services through ISA-neutral interfaces. Chapter 2 presents background information concerning heterogeneous multi-core hardware platforms and their associated software infrastructures. Additionally, it presents an overview of computer architecture, including multi-core caching and synchronization approaches that are relevant to challenges faced in developing operating systems frameworks for heterogeneous multi-core systems. Chapter 3 serves as an introductory guide to the following chapters. Chapter 4 introduces the original design concepts of the hthreads system as well as the details of the hthreads OS cores. Chapter 5 describes the hardware abstraction layer (HAL) that has been developed in order to extend the hthreads operating system for use in heterogeneous multi-core systems. Chapter 6 describes the augmented design flow that is used to target the heterogeneous platform. This chapter also describes the binary utilities that have been developed for assisting in the heterogeneous compilation and linking process. The overview and analysis of experimental results are detailed in Chapter 7, alongside performance and scalability tests used to compare the performance and scalability of the hardware-based microkernel to RPC-based operating system frameworks. Chapter 8 contains conclusions that can be drawn from this dissertation as well as future extensions to the work.
Chapter 2

Background

Multiprocessor systems are computing systems composed of multiple, independent processing units or CPUs. Many features of basic computer architecture require modification or augmentation when used in platforms with multiple processors. Cache-based memory hierarchies aim to hide memory latency in computing systems. While this solves performance problems by exploiting locality, it introduces memory coherence and consistency problems in systems with multiple processors. Additionally, processors that may work together by sharing data must be able to synchronize to keep from interfering with the work of other competing processors. Synchronization issues of this type become key factors in the architecture of multiprocessor systems. Finally, programmers require an avenue to utilize each processor in a multiprocessor system. This necessitates the use of parallel programming models: an augmentation to traditional serial programming models that allows a programmer to specify sections of code capable of executing in parallel. However, by defining parallel sections of code, a programmer is often required to add synchronization statements to a program in order to guarantee that parallel execution of code does not result in race conditions that can corrupt both individual data elements or even entire data structures.

Design issues and caveats become even more complex in heterogeneous multi-core systems. Heterogeneity introduces non-uniformities at the hardware level in the form of differences in ISA, memory interfaces, cache architecture, and atomic operations. These issues confound traditional multiprocessor architecture solutions as they prevent different CPUs from native communication and synchronization. Additionally, processors with different ISAs are unable to natively share a common code base, thus creating challenges in the construction of common operating system services. The following sections discuss these issues, and how they relate to computer architecture and operating systems for multi-core
platforms.

2.1 Shared Memory: Synchronization and Caching

Most computer systems are memory-based, in that almost all program state is resident within the memory hierarchy of the computer. In a multi-processor system, the memory system may be used by many different processors; storing the individual programs and data of each. If processors are to communicate, they can do so by sharing data within the memory system. The shared data may be on disk, in memory, or even in the cache of a processor within the system. Data can be shared amongst cooperating processors, or amongst cooperating tasks, or threads, on a single processor. The use of shared-memory communication allows for data to be shared seamlessly amongst different threads within a multi-processor system without the need to use explicit communication commands required by message-passing based sharing systems. Additionally, sharing in-memory data can be quite efficient as it does not necessarily require the data to be copied in order for it to be shared. Without copies, all threads are actually sharing a common set of data; therefore ownership of data must be arbitrated in order to prevent corruption via concurrent accesses.

Arbitration of ownership over shared resources is most often accomplished using mutual exclusion primitives and the notion of a critical section. This idea establishes that a piece of data can only be in use, or owned, by a single thread at a time within specific regions, or critical sections, of code. The notion of atomicity comes surfaces here, as only one thread can enter a critical region at a time, thus making a critical section appear to rest of the system as a single atomic block of code, or a transaction. Once a thread enters a critical region, it usually performs a bulk set of operations, and then leaves the critical section to allow other threads to use, share, or see the protected data. Without mutual exclusion primitives, parallel programs can encounter data corruption and race conditions due to atomicity violations.
Imagine a two thread system, where both threads share a single variable, $x$. In this example, both threads are infinitely incrementing the shared variable, $x$, by one as shown in Figure 2.1. The increment statement can be written in a single line of code in a high-level language (HLL) such as C or Java, however at the machine-level, this statement requires several sequential operations as shown in Figure 2.2. Thus, the entire increment statement is not considered to be *atomic*. This is problematic in concurrent systems as intermediate results can be seen by other threads or processes. The intermediate results can become visible for a myriad of reasons; including different code interleavings, inter-processor timing, as well as interrupts and exceptions.
Table 2.2: Incorrect Interleaving - Incrementing a Shared Variable

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Thread</th>
<th>Instruction</th>
<th>R1 (0)</th>
<th>R1 (1)</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>load R1, x</td>
<td>0</td>
<td>n/a</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>addi R1, R1, 1</td>
<td>1</td>
<td>n/a</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>load R1, x</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>store R1, x</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>addi R1, R1, 1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>store R1, x</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Invalid Result</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Two possible interleavings of the threads are shown in Tables 2.1 and 2.2. In this example, if \( x \) is initialized to zero, then after two iterations, \( x \) should have the value of 2, as \((0++)++)\) equals 2. The interleaving shown in Table 2.1 yields the correct result, while the sequence of events in Table 2.2 does not as the interleaving broke the atomicity of the increment operation by allowing one thread to read the older value of \( x \) before the previously executing thread had updated it. In order to prevent such invalid interleavings, mutual exclusion mechanisms must be used.

In this example, the critical section is where the shared variable \( x \) is read, modified, and written back to memory. This entire read-modify-write process must be protected to ensure that it appears to be atomic to all threads who have access to \( x \). This can be accomplished using mutual exclusion primitives such as lock/unlock as shown in Figure 2.3. The mutex, short for mutual exclusion, \( m \) is locked before entering the critical section code, and unlocked when leaving the critical section. The use of a lock prevents multiple threads from entering the critical section at a time, therefore making the code in the critical section execute atomically. If a thread tries to acquire a lock that is already owned by another thread, then it must either continue to retry the lock the operation (spinning, or busy-waiting) or it can be put to sleep until the lock is released (blocking). Either type of lock mechanism can be used, however each has their own sets of pros and cons in different concurrency scenarios.


Figure 2.3: Using Locks When Incrementing a Shared Variable

2.1.2 Mutual Exclusion - Lock Policy Vs. Mechanism

Mutual exclusion is a policy specification stating that a resource (often global) can only be accessed by a single entity at a time. In the domain of parallel programming, this often translates to meaning a resource (device, variable, data, etc.) can only be accessed by one thread at a time. The actual mechanism for upholding mutual exclusion properties is referred to as a mutex, or lock. In general, locks have a simple interface that consists of two major operations: acquisition (locking) and release (unlocking). Pseudo-code for standard lock and unlock primitives is shown in Figure 2.4. One must note that the action of testing for, and acquiring the lock must be atomic in practice, otherwise multiple threads would be able to test that the lock is free, leading each thread to believe that it was the new owner of the lock.

In order to prevent this race condition, the code must make use of a mechanism to enforce atomicity when acquiring a lock. In general, this means there must be hardware support for enforcing atomicity of the test-and-set operation; usually involving the use of special atomic instructions provided by the ISA of the processor being used. Examples of such atomic instructions are test-and-set, compare-and-swap, fetch-and-add, and load-linked/store-conditional.
#define FREE (0)
#define TAKEN (1)

typedef struct
{
    bool locked;
    int owner;
} mutex_t;

lock(int id, mutex_t *m) unlock (mutex_t * m)
{
    // Wait for mutex to be unlocked
    while (m->locked == FREE);
    m->locked = FREE;

    // Lock mutex
    m->locked = TAKEN;

    // Mutex is now ours, set owner
    m->owner = id;
}

Figure 2.4: Pseudo-code Mutex Operations

Test-And-Set

The atomic test-and-set (TAS) instruction tests the current value of a memory location and conditionally changes its value based on the result of the comparison. The atomicity of this instruction allows for only a single thread to acquire a mutex lock, and therefore enforces mutual exclusion of a critical section of code. The use of a test-and-set instruction for implementing mutex operations can be seen in Figure 2.5.

Compare-And-Swap

The atomic compare-and-swap (CAS) instruction tests the current value of a memory location and conditionally swaps out its value for another based on the result of the comparison. The CAS instruction can be used to emulate test-and-set functionality as shown in Figure 2.6.

Load-Linked/Store-Conditional

The load-linked (LL) and store-conditional (SC) instructions can be used to implement a 2-phase atomic operation. In fact, the use of two instructions to implement a single atomic operation can be more efficient than a single atomic instruction as it does not require the


```c
#define FREE (0)
#define TAKEN (1)

typedef struct
{
    bool locked;
    int owner;
} mutex_t;

<atomic> int test_and_set(int * mem_loc)
{
    if (*mem_loc == FREE)
    {
        *mem_loc = TAKEN;
        return FREE;
    }
    else
    {
        return TAKEN;
    }
}

lock(int id, mutex_t *m)
{
    // Wait for mutex to be unlocked
    while (test_and_set(&m->locked));
}

unlock(mutex_t * m)
{
    // Release mutex
    m->locked = FREE;
}

// Mutex is now ours, set owner
m->owner = id;
```

Figure 2.5: Mutex Operations Using Test-And-Set

use of bus-locking to enforce atomicity. Atomicity is enforced in LL/SC by making use of a reservation, or link register, that is used to make the SC portion of the atomic transaction conditional. In general, the LL instruction performs a traditional load from memory, while also marking the address being read in a reservation register. The link register is volatile, in that any SC instruction executed in a system will invalidate the link register on any other processor.

A thread can continue within its critical section even with an invalid link register, but when it comes time to commit its results via the SC instruction, the store will fail. Upon failure, a thread must repeat the entire atomic transaction over again, starting at the initial LL instruction. The use of LL/SC instructions in mutual exclusion is optimistic, as threads can continue into a critical section even though they may not be able to commit their results. Atomicity is enforced by allowing only a single thread to go through the LL/SC process at a time. The first thread to get to the SC instruction is the first to complete its critical section,
#define FREE 0
#define TAKEN 1

typedef struct {
    bool locked;
    int owner;
} mutex_t;

<atomic> int compare_and_swap(int * mem_loc, int comp, int new) {
    if (*mem_loc == comp) {
        *mem_loc = new;
        return FREE;
    } else {
        return TAKEN;
    }
}

int test_and_set(int * mem_loc) {
    compare_and_swap(mem_loc,FREE,TAKEN);
}

lock(int id, mutex_t *m) {
    // Wait for mutex to be unlocked
    while (test_and_set(&m->locked));
    // Release mutex
    m->locked = FREE;
    // Mutex is now ours, set owner
    m->owner = id;
}

Figure 2.6: Mutex Operations Using Compare-And-Swap
#define FREE (0)
#define TAKEN (1)

typedef struct {
  bool locked;
  int owner;
} mutex_t;

int test_and_set(int * mem_loc) {
  bool reserve;
  retry:
  LL(mem_loc,& reserve);
  if (mem_loc == FREE) {
    SC(mem_loc,TAKEN,& reserve);
    if (!reserve)
    {
      goto retry;
    }
    return FREE;
  } else {
    return TAKEN;
  }
}

lock(int id, mutex_t *m) { 
  // Wait for mutex to be unlocked
  while (test_and_set(&m->locked));
  // Release mutex
  m->locked = FREE;
  // Mutex is now ours, set owner
  m->owner = id;
}

unlock(mutex_t * m) {
  // Release mutex
}

Figure 2.7: Mutex Operations Using Load-Linked/Store-Conditional

thus invalidating all other threads’ attempts, and forcing them to retry. The use of LL/SC in mutex operations can be seen in Figure 2.7.

2.1.3 Cache Coherency

The use of caches in computer systems leads to situations in which multiple copies of data can be in existence. In a uni-processor system, this does not cause any problems as the global view of memory, from the perspective of the processor, is coherent. However, in multi-processor systems, data coherency does not come for free, and in fact, must be purposefully maintained [86]. Data can become incoherent when multiple processors have copies of the same piece of data in their caches. As soon as one processor makes a change
to the piece of data, the two copies will have different values, and are thus made incoherent as shown in Figure 2.8. Cache coherency mechanisms work by making sure that common copies of data are always updated as soon as a single processor changes a portion of cached data [86]. The caches work together by communicating which pieces of data are stored, and negotiating which piece of data should be considered the current coherent value.

![Incoherent Multi-Processor Cache](image)

Figure 2.8: Incoherent Multi-Processor Cache

Cache coherency is vital to the correct execution of parallel programs. Without coherency, the values of program data, including both program variables and synchronization objects, can diverge and lead to incorrect results. Cache coherency can be maintained by software or hardware, but is most often done at the hardware level in order to achieve acceptable levels of performance. The two most popular cache coherency mechanisms are snooping caches and directory schemes.

**Snooping-Based Cache Coherency**

Snooping-based cache coherency systems monitor, or snoop, a global broadcast medium, often a shared bus [86]. Each cache access is broadcast to all other cache controllers in
order to notify each cache as to what pieces of information are currently being processed. Information concerning what data is held in each cache is distributed, thus necessitating the use of a broadcast mechanism so that each cache can have a global view of memory. The need for atomic broadcasts makes the snooping approach ideal for bus-based systems, as it is naturally a broadcast medium, however this approach does not scale well in distributed systems due to the lack of an efficient broadcast mechanism.

The scalability of snoop-based cache coherency protocols is also limited in terms of the number of processors attached to a shared bus. All bus transactions must appear to be atomic, so the rate of address generation and response is directly related to the aggregate snoop rate of the system. As an example from Shen and Lipasti [112], assume a system in which each processor generates $r^A$ address requests per second. In a system with $N$ processors, each processor is responsible for completing $N \times r^A$ snoop operations per second. Each local snoop operation requires, at a minimum, a local cache look-up in order to determine what, if any, actions are to be carried out. These operations do not occur in zero time, and will take several cycles, at a minimum, in order to process. So as the number of processors in a system grows, the bandwidth required to satisfy a snooping-based coherency protocol increases quite dramatically. This effect makes it difficult to scale snoop-based systems past double-digit numbers of processors [2, 86].

Snooping requires that each cache controller be able to continuously monitor both the address and data lines of a bus. Additionally, tracking cache coherency information requires a few extra bits per cache entry in order to establish how the data is being shared in a system. The amount of data varies by protocol, but in general, the amount of data required is quite small and on the order of a few bits. The two most popular snooping-based cache coherency protocols in production machines are MESI and MOESI protocols [56]. The acronyms for these protocols refer to the possible states that a piece of cached data can be in. The letters of these acronyms have the following meanings:

- MESI = Modified, Exclusive, Shared, Invalid
- MOESI = Modified, Owned, Exclusive, Shared, Invalid

**MESI Protocol**  
The MESI protocol is a 4-state protocol which can be coded using 2 additional bits per cache line. The invalid, or $I$, state refers to a line in the cache that does not currently hold valid data. When an entry is removed from a cache, or a cache is flushed, invalid lines do not have to be processed as they do not currently hold valid cached data. The exclusive, or $E$, state refers to a piece of data this is present in only a single cache in the system, and that the value of the data is *clean*, or un-changed from that in main memory. The shared, or $S$, state refers to a cache line that may be stored in multiple caches within a system, however all copies of the cached data remain *clean*. The modified, or $M$, state refers to a cache line that is present in only a single cache in the system, and that the value of the line is *dirty*, or differing from that in main memory. The possible state transitions in the MESI protocol are shown in Figure 2.9. Given a pair of caches, the legal states achievable in the MESI protocol are shown in Table 2.3 [123].

![Figure 2.9: MESI Protocol - State Transitions](image)

At startup, all cache lines are marked as invalid. Upon an initial data read, a cache line
will be marked as exclusive. If this same processor writes to this cache line, the data will be considered dirty (when compared to main memory) and the cache line will move to the modified state. If another processor attempts to read this same piece of data, the snooping process on the cache that maintains the modified cache line must catch the read. The dirty data is first written back to main memory, and the modified cache line is moved to the shared state. Meanwhile, the reading cache misses, and retrieves the updated value in main memory and updates its cache line to shared as well.

**MOESI Protocol**  The MOESI protocol is a 5-state protocol which can be coded using 3 additional bits per cache line. This protocol is actually an optimized version of the MESI protocol that allows for cache-to-cache transfers of updated data instead of having to first commit data back to main memory. The extra owned, or \( O \), state refers to a cache line that contains the most recent copy of a piece of data; additionally, other processors may have this same copy of data in their caches, but the value in main memory is allowed to be different [11]. Only a single processor can hold a cache line in the owned state at once, although in general, this state is similar to the shared state, except that the value in main memory can be incoherent. The possible state transitions in the MOESI protocol are shown in Figure 2.10. Given a pair of caches, the legal states achievable in the MOESI protocol are shown in Table 2.4 [124].

If a cache line is held in the modified state and another cache (probe) attempts to read this data, the snooping protocol will issue a read hit to the probe, supply the probe with the correct value of the cache line, all while simultaneously moving its own cache line from the modified to the owned state. In the MESI protocol, a read probe would result in the issuance
of a read miss along with a writeback of the cache line to main memory, paired with moving
the cache line from modified to shared state. The additional state in the MOESI protocol
circumvents the need to update main memory, by using an efficient cache-to-cache transfer
mechanism. This is an important optimization, as it reduces the amount of traffic to off-chip
memory.

![MOESI Protocol - State Transitions](image)

Figure 2.10: MOESI Protocol - State Transitions

**Directory-Based Cache Coherency**

The use of a bus-based broadcast medium allows for easy construction of distributed, snoop-
based, coherency protocols [86]. However, bus-based interconnects have scalability limits,
and are often replaced with more network-centric interconnects when the number of proc-
essors increases or when additional bandwidth is desired. The lack of an efficient broadcast
medium in networks can make it difficult to implement snoop-based coherency, so instead, a centralized approach is used. A centralized repository, also known as a directory, can be used to hold all relevant cache info for properly implementing cache coherency protocols in a distributed system.

Directory-based cache coherence protocols keep all coherence state for each processor in a single location, rather than distributing the information amongst all caches in a system. All cache-related coherence traffic is first forwarded to a directory, and the directory handles all message forwarding to other processors only when necessary. This results in large bandwidth savings, as cache coherence traffic is no longer routed to every processor in a system. Additionally, directory-based coherence bandwidth scales with the amount of available memory bandwidth, as opposed to snoop-based approaches whose bandwidth is directly tied to the number of processors in a system [112]. This allows directory-based systems to scale past double-digit numbers of processors, a feat that is extremely difficult to accomplish in snoop-based systems [86].

Directory-based coherence protocols are very similar to those used in snoop-based systems, however, explicit point-to-point communication is used to route coherence messages from the directory to each processor, and vice-versa. Directories must contain information about cache lines on a per-processor basis. This can be implemented using fixed-size vectors, or by using linked-lists of cache-line information.
2.1.4 New Synchronization Methods

Lock-based synchronization methods have dominated in the parallel processing domain, as they are quite simple in terms of hardware resource usage, and higher-level abstractions can be readily built on top of them (monitors, barriers, etc.). Unfortunately, from a programmer’s perspective, locks are difficult to use. In practice, using locks requires strict discipline and it still remains error-prone. Errors may include unprotected sections, race conditions, deadlock, livelock, priority inversion, and self-deadlock. Additionally, the granularity of locks can adversely affect program performance. Tuning lock performance is usually done in iteratively starting with relatively coarse-grained locks, and iteratively replacing highly-contended locks, with finer-grained alternatives.

Many have argued that parallel programming should be done with message-passing instead of using shared memory and locks. In message-passing systems, ownership of data is explicit, making it impossible to have race conditions on data. While message-passing has gained significant popularity in cluster and high-performance computing communities, it has not yet been used extensively in smaller scale systems; especially desktop and embedded systems where efficient access to shared memory is often a guarantee. Additionally, lock-free synchronization methods have existed since the advent of locks, however these methods are often data-structure dependent, and may not be readily available for a given application. Alternatives to locks that make use of speculation and timeouts, such as transactional memories [31], and transient blocking synchronization primitives [110] have been proposed recently to make parallel programming easier, safer, and more productive. While the semantics of these operations may have benefits that are not easily achievable through the use of traditional locks, the interface used by the programmer will most likely appears as if locks are being used [31].
Transactional Memory

The major goal of a transactional memory model is to allow programmers to easily declare and use arbitrary-sized atomic memory transactions [60]. Transactions are considered to be lock-free, in that system-wide progress is guaranteed. The LL/SC atomic operation can be considered a form of transaction, in which the amount of data involved in the transaction is just a single word. The operation is lock-free in that all threads are able to optimistically enter a critical section, thereby allowing system-wide progress. Only a single thread is able to commit their results at a time, making the entire LL/SC operation atomic. In general, transactional memory is an extension of LL/SC semantics; with the added ability of defining arbitrary-sized transactions. Transactional memory implementations require the use of several different types of instructions that include:

- **Load Transactional** - Read the value of memory into a local register.

- **Store Transactional** - Store the value of a local register to memory, however this new value is not visible to others until an entire transaction is successfully committed.

- **Commit** - Conditional completion of a transaction. Can succeed or fail depending on the action of other processors in the system.

- **Abort** - Abandon a transaction, discarding any changes the transaction may have made.

- **Validate** - Check the abort status of a transaction. Informs the caller that the current transaction may succeed, or that is will definitely abort.

The transactional memory operations are meant to be used to construct both traditional as well as custom mechanisms not readily achievable with traditional synchronization mechanisms. For instance, a test-and-set, compare-and-swap, or LL/SC operation can only atomically protect a single memory location at a time. On the other hand, a transaction can atomically protect multiple memory locations at once. A single *lock variable* must be
used in order to protect multiple locations using a traditional synchronization mechanism as shown in Figure 2.11. This forces threads to acquire a lock before entering a critical section, thus leading to situations in which deadlock may prevent system-wide progress.

While transactions are indeed more general than traditional synchronization operations, they require additional state information to be collected and used to decide when transactions must commit, and when they need to abort. This is often implemented using a transaction cache that can be software or hardware controlled. Software-based transactional memory (STM) systems are both portable and flexible, however they are often less efficient than systems that make use of hardware-based synchronization operations. Hardware-based transactional memory systems promise to increase the efficiency of TM systems, at the cost of increased area resource usage and increased interconnect complexity used implement transaction caches. So far, very few hardware-based transactional memory systems exist outside of research platforms. Sun has recently added hardware transactional memory (HTM) support in its Rock [35] processor line that is was to be released in late 2007, however it has been delayed numerous times, and is still not yet available. Initial experimental results have been made public by Sun to demonstrate the HTM instruction-set as well as initial performance figures [41].

While TM semantics may be richer than traditional single-word atomic operations, they also have many drawbacks. While lock-free synchronization guarantees system-wide progress, it does not make any ordering guarantees. This means that a lower-priority thread may consistently beat a higher-priority thread to a critical section if the lower-priority thread happens to do less work, thereby forcing the higher-priority thread to consistently retry. In a lock-based system, blocking semantics help to define an ordering based on access ordering as well as priority, leading to a more fair system in the eyes of a programmer; especially in real-time environments. Another drawback of the lock-free semantics of TM systems is related to speculative execution of critical sections. Threads destined to abort can continue executing; thereby generating unnecessary interconnect traffic, and burning power and com-
#define FREE (0)
#define TAKEN (1)

multi_word_atomic(lock_t lock)
{
    bool reserve;
    retry:
    LL(lock,&reserve);
    if (lock == FREE)
    {
        SC(lock,TAKEN,&reserve);
        if(!reserve)
        {
            goto retry; // Spin, retry lock
        }
        // **** Critical Section Begin ****
        <modify multiple words of data>
        // **** Critical Section End ****
        lock = FREE;
    }
    else
    {
        goto retry; // Spin, retry lock or block and context switch
    }
}

Figure 2.11: Multi-Word Atomic Operation Using LL/SC

pute cycles. In lock-based systems, speculative execution of critical sections is not allowed, and with the help of an operating system, spin locks can be replaced with blocking followed by context switching to non-speculative code.

**Transient Blocking Synchronization**

Sun’s proposal of Transient blocking synchronization (TBS) is an attempt at establishing a set of hardware synchronization primitives that can be used in machines intended to scale to large numbers of interconnected processors. The goal of TBS is to try to minimize synchronization overhead in terms of both time and space by making modifications to the semantics of existing synchronization protocols [110]. The *transient* nature of their protocol allows ownership to be revoked at any time. Most synchronization mechanisms work by establishing either permanent ownership (locks) or by allowing speculative execution paired with aborts to establish a serializable ordering. Additionally, most mechanisms specialize in single-word synchronization, while the designers of TBS aim to support both single-word and multi-word operations.
The terminology used in TBS refers to *leases* and the ability to both acquire and relinquish leases. TBS is a 2-phase protocol similar to that of LL/SC, however it differs in that leases can be revoked at any point in time, and the load-lease operation is allowed to fail. Additionally, all leases are accompanied by a timeout period specified by the programmer. These modifications aim to prevent un-bounded numbers of leases on a single memory location, which can aid in building efficient hardware mechanisms for supporting TBS. For instance, a comparison of the complexity of TBS to other synchronization mechanisms is shown in Table 2.5 [110]. Compare-and-swap (CAS) suffers from the ABA problem, and the solution requires additional tag bits for every possible memory location in a system [59]. LL/SC operations eliminate the ABA problem by decoupling the value of a memory location from the act of updating the value of a memory location. Unfortunately, LL/SC allows unbounded numbers of threads to be holding a reservation on a memory location; thus requiring a mechanism to notify large numbers of threads atomically. A mechanism for this type of broadcast is built in to cache coherent machines, however in distributed machines, specialized hardware and protocols must be used in order to support possibly unbounded sizes of reservation lists. TBS solves this problem by ensuring that leases are guaranteed to have a limited life span. Timeouts guarantee that an individual lease has a finite life span, and the ability for load-and-lease operations to fail allows for bounded (system dependent) lease reservation lists.

Another benefit of TBS is its ability to support k-Reads-Single-Write operations atomically with lock-free semantics. While this is true of transactional memory systems as well, TBS can be efficiently supported on distributed memory machines, while transactional memory systems currently depend on snoop-based cache coherency protocols [110]. TBS is also able support multi-word atomic operations by adding a single transaction bit to their memory subsystem. Without this extra bit, locking would be required to perform multiple memory writes atomically.
Table 2.5: Complexity of Synch. Operation Implementations [110]

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Controller Space Complexity</th>
<th>Communication Complexity</th>
<th>Fault Tolerance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compare&amp;Swap</td>
<td>Tag (Each Location)</td>
<td>$O(N)$</td>
<td>lock-free</td>
</tr>
<tr>
<td>LL/SC</td>
<td>Unbounded</td>
<td>$O(1)$</td>
<td>lock-free</td>
</tr>
<tr>
<td>Full/Empty</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>none</td>
</tr>
<tr>
<td>TBS</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>lock-free</td>
</tr>
</tbody>
</table>

AMD’s Advanced Synchronization Facility

AMD has recently announced a set of experimental instruction-set extensions for supporting multi-word atomic operations [12]. The extensions allow for speculative execution of critical regions similar to that of transactions in that they are able to fully commit their work, or they abort, and must retry. The Advanced Synchronization Facility (ASF) extensions are transaction-based and represent a realistic look at how major vendors intend to implement transactional memory systems. The ASF specification defines a set of assembly instructions used to build multi-word atomic memory transactions that include:

- Speculate - Declaration of the beginning of a speculative region.
- Lock Mov (Load) - Declarator that specifies a location for which atomic access is desired.
- Lock Mov (Store) - Memory store instruction that must be used along with a previous declarator. Updates are not visible to other CPUs until a successful commit operation.
- Commit - Denote the end of a speculative region. Used when attempting to finalize a speculative region.
- Abort - Abandons a speculative region.

The ASF specification allows for a programmer to declare nested speculative regions up to a limit of 256, and with a maximum speculative region size of 4 objects, in which each object is less than 64 bytes [12]. Execution of a speculative region can stop or fail due to
; DCAS Operation:
; IF ((mem1 = RAX) && (mem2 = RBX)) {
;  mem1 = RDI
;  mem2 = RSI
;  RCX = 0
;}
; ELSE {
;  RCX = 1
;}
; (RAX, RBX, R8, R9 modified)

DCAS:
MOV R8, RAX
MOV R9, RBX
MOV RCX, 1
SPECULATE ; speculative region begins
JNZ fail ; Bail out if rolled back
LOCK MOV RAX, [mem1] ; Specification begins
LOCK MOV RBX, [mem2]
CMP R8, RAX ; DCAS semantics
JNZ out
CMP R9, RBX
JNZ out
LOCK MOV [mem1], RDI ; Update protected memory
LOCK MOV [mem2], RSI
XOR RCX, RCX

Figure 2.12: ASF Implementation of Double-Word Compare-and-Swap (DCAS) [12]

SPECULATE ; LL/SC section begins
JNZ ll_sc_failed
LOCK MOV RAX, [mem]

; compute new value for mem in RAX
...
LOCK MOV [mem], RAX
COMMIT ; End of speculative region
...

; Error handling
ll_sc_failed:
...

Figure 2.13: ASF Implementation of Load-Linked/Store-Conditional (LL/SC) [12]
; ENQUEUE Operation:
; (INPUT: element ptr in RAX)
; (INPUT: list ptr in RBX)
; RAX->next = 0
; IF (RBX->tail != 0)
;
; tmp_ptr_next = & RBX->tail->next
; } ELSE {
; tmp_ptr_next = & RBX->head
;
; *tmp_ptr_next = RAX
; RBX->tail = RAX
;
; ENQUEUE:

MOV [RAX + next], 0

retry:
SPECULATE
JNZ retry
LOCK PREFETCH [RBX + head]
LOCK MOV RCX, [RBX + tail]
TEST RCX, RCX
JZ empty_list
LOCK PREFETCHW [RCX + next]
LEA RCX, [RCX + next]
JMP ok

empty_list:
LEA RCX, [RBX + head]
ok:
LOCK MOV [RCX], RAX
LOCK MOV [RBX + tail], RAX
COMMIT
RET

; DEQUEUE Operation:
; (INPUT: list ptr in RBX)
; (RETURN: element ptr in RAX)
; RAX = RBX->head
; IF (RBX->head != 0)
;
; RBX->head = RAX->next
; IF (RBX->head = 0)
;
; RBX->tail = 0
;
;
; DEQUEUE:
retry:
SPECULATE
JNZ retry
LOCK MOV RAX, [RBX + head]
LOCK PREFETCH [RBX + tail]
TEST RAX, RAX
JZ end
LOCK MOV RDX, [RAX + next]
LOCK MOV [RBX + head], RDX
TEST RDX, RDX
JNZ end
LOCK MOV [RBX + tail], RDX
end:
COMMIT
RET

Figure 2.14: ASF Implementation of a FIFO [12]
an explicit abort command, or implicit aborts caused by the interrupts, exceptions, or the action of other processors. Abortion of a speculative region is followed by the roll-back of protected memory locations, however unprotected memory locations must be correctly handled manually by the programmer. Additionally, register state is not rolled-back upon abortion aside from the program counter and stack pointer. Speculative regions have implicit roll-back semantics in that abortion causes an immediate retry by returning control back to the instruction following the Speculate instruction. At this point a set of registers contain status information as to why execution of the speculative region failed, allowing the program to route control to a specialized error handler via a conditional jump. Two usage examples of ASF, double-word compare-and-swap (DCAS) and LL/SC, can be seen in Figures 2.12 and 2.13 respectively. An example of advanced, multi-word synchronization operations used to implement a FIFO (queue) can be seen in Figure 2.14, and atomic composition of FIFO operations can be seen in Figure 2.15. The ability to easily compose atomic operations into larger atomic operations without the use of explicit lock variables is one of the main advantages of transactional memory over the use of locking.

2.2 Parallel Programming Models

Parallel programming models allow a programmer to define sections of code that are able to execute in parallel. A programmer is free to create multiple tasks, essentially different flows, or threads, of execution. Tasks are able to execute completely independent from one
another, but they must also be able to synchronize and communicate in some manner. The types of task-level interaction found in a parallel programming model vary widely depending on the constraints of the model. In some models, tasks can implicitly communicate and synchronize through shared memory, while in other models, data may be passed from task to task explicitly. In general, all parallel programming models promise the ability to speedup serial computations by splitting up a set of work amongst tasks that can run in parallel. On a uni-processor machine, speedup may not be easily achieved as all parallel tasks are time-multiplexed on a single processor. However, on a multi-processor machine or distributed system, tasks can truly execute in parallel leading to significant speedups.

Parallel programming models are often classified in terms of Flynn’s Taxonomy as well as the granularity of the parallelism found in the model. Flynn’s Taxonomy defines 4 different classes of parallel programming models [50, 51]:

- **SISD** - Single Instruction, Single Data. General sequential programming model.

- **SIMD** - Single Instruction, Multiple Data. Vector, or data-parallel, programming model.

- **MISD** - Multiple Instruction, Single Data. Very uncommon, different tasks working on the same set of data.

- **MIMD** - Multiple Instruction, Multiple Data. Task-based, multiple tasks working on multiple sets of data. Can be used to emulate both SIMD and MISD.

These models can also be used to describe types of parallel computing platforms, as certain hardware architectures fit into these exact categories. SISD models make use of only a single flow of instructions on a single set of data. This represents the classical sequential programming model. Most processors fit into this category, aside from vector and superscalar machines that may make use of multiple instructions and data at once. SIMD models make use of a single stream of instructions processing multiple, parallel streams of data. SIMD models often execute in completely synchronized, lock-step manner. All parallel
processing units in a SIMD model execute the same instruction simultaneously on different data items. This has lead to use of the term *vector* machine to refer to SIMD computers. The MISD model is the inverse of the SIMD model, in that a single stream of data is processed by parallel tasks, each capable of performing different operations. MISD computers and models have not been widely deployed as the more popular MIMD model can easily emulate a MISD model. MIMD models make use of multiple independent streams of instructions and data. MIMD models are considered to be task based, in which each task is capable of performing unique sets of operations on unique sets of data. The MIMD, task-based model is considered to be the most flexible in that individual tasks can execute asynchronously, as opposed to the fully synchronous SIMD model.

The size, or granularity, of parallelism in each model also varies. SIMD models often exploit fine-grained parallelism, where the unit of parallelism can be measured in number of data items processed, or the number of instructions performed in parallel. MIMD models are more coarse-grained, where the unit of parallelism is often the number of parallel tasks. It is important to note that models can be composed in order to exploit multiple levels of parallelism simultaneously. For instance, SIMD and MIMD can be combined in order to form a system in which independently executing tasks are also able to exploit intra-task data- and instruction-level parallelism. This type of model composition is used to produce computing systems that are able to exploit multiple levels of parallelism.

### 2.2.1 Shared Memory Models

Shared memory allows for parallel sections of code to share data structures directly. If memories are private, or distributed, data must be copied from one memory to another in order for it to be shared. In a shared memory system, copies are not required; however proper synchronization of data access is needed to prevent data races. Synchronization mechanisms including mutexes and condition variables can be used to serialize access to shared data. The two most popular forms of shared-memory parallel programming models
are threads and OpenMP.

**Processes**

Most operating systems contain the concept of parallel tasks, of which the most common form is a *process*. A process is an independent flow of execution paired with its own private address space. Processes cannot interfere with the data of another process, unless some type of explicit inter-process communication (IPC) is used. IPC is often file- or network-centric; using communication mechanisms that involve file descriptors, pipes, and sockets. Processes are considered to be one of the more *heavyweight* forms of parallel programming due to the amount of context (data) associated with a typical OS process. This context includes a full Process Control Block (PCB) that consists of an entire program context, list of all owned OS objects (file descriptors, etc.), and all address space information for the process. Over the years, many OS-level optimizations, such as copy-on-write semantics, have improved the performance of multi-process parallel programming. However, processes are still considered to be one of the most heavyweight approaches to parallel programming.

**Threads**

Threads are individual concurrent execution paths, each with their own context. Within a process, all threads share the same shared memory space; allowing them to implicitly share in-memory data structures. The use of a shared address has led to threads being equated with the notion of *lightweight* processes [42]. Threads must be explicitly created by a programmer, and they are able to synchronize and communicate using shared memory, mutexes, condition variables, and barriers. Early OS development was characterized by a wide variety of threaded programming models, each with a different set of APIs and underlying semantics. The POSIX threads (PThreads) standard was developed to create a uniform, portable standard for threading [32]. It is widely accepted and used in almost all common operating systems; including Unix, Linux, Solaris, Windows, and a myriad of
other embedded operating systems. The lack of address space protection makes threads suitable for use in MMU-less systems; a common feature of many embedded computing systems.

Multithreaded programming models are characterized by the ability to dynamically create parallel threads of execution, all capable of independently communicating and synchronizing through shared-memory constructs. Upon creation, a thread is an immediate peer of other running threads, and is also considered to be the child of the thread that created it. The creating thread, or parent thread explicitly creates a child thread with a desired functionality, often by specifying a function that it should execute upon startup. Multi-process programming is similar to this, however the creation of a child process leads to an exact copy of the parent process. A child process must explicitly invoke the code that it will execute. This is often accomplished with a traditional function call, or through an exec system call. The ability to dynamically create parallel, and relatively lightweight execution flows lends itself well to the description of both regular and irregular forms of parallel processing [92].

**OpenMP**

OpenMP is a set of language extensions to allow a programmer to describe parallel sections of code without requiring major modifications to the existing code base. OpenMP pragmas are used by a compiler to generate a parallelized version of a program [38]. The use of pragmas allows OpenMP programs to remain portable by allowing compilers that do not support OpenMP to simply ignore the parallelization hints. OpenMP was based upon a set of SMP compiler directives proposed in the late 1980s by Cray/SGI, however it was not standardized until 1997 [87]. The original standard allowed for a programmer to easily declare parallel loop bodies and sections, as well as basic synchronization; specializing in single-program-multiple data (SPMD) or a SIMD style of programming. Now, revision 3.0 of the OpenMP standard has evolved to allow for explicit descriptions of irregular task-level parallelism. The OpenMP run-time system handles all task creation, synchronization, and
scheduling duties. OpenMP is built on a shared memory programming model, that allows tasks (threads) to share data directly, and also provides private (thread-specific) data to prevent false-sharing. OpenMP compilers rely on the underlying OS framework and thread libraries including POSIX threads [63].

2.2.2 Message Passing Models

Distributed memory systems are often referred to as systems with non-uniform memory access (NUMA). Access times may be non-uniform due to board- or chip-level architecture issues, or memories may be on different physical platforms, interconnected via a network. Shared-memory abstractions were not originally intended for NUMA systems as the programming models do not account for non-uniform memory latency. Most shared-memory models cannot distinguish between local and remote memory accesses, making it difficult to reason about the performance of a shared-memory program on a NUMA architecture. Other programming models provide better representations of NUMA systems by distinguishing between local and remote accesses. Message passing systems often rely solely on local data, while all remote data must be communicated using a special set of APIs. This encourages programmers to think about locality as remote accesses require the use of special APIs. Distributed memory parallel programming is largely used in networked environments and clusters, however most multi- and many-core systems are shared-memory architectures that may have slight NUMA characteristics [63].

Message Passing Interface

The Message Passing Interface, or MPI, is a standard distributed memory programming model for message-based systems. MPI is the most widely used message-passing standard and is extremely popular for programming clusters of computers [63]. MPI established a standard set of message-passing APIs, replacing numerous other vendor-specific APIs with a single, portable standard. Message-passing based programming models encourage data
locality at the expense of requiring communication to be done explicitly via copying data. This forces the idea of consistency and coherency into the programming model; making the programmer responsible for communicating updated data objects to other processes at the correct times.

The lack of shared data in MPI programs resembles that of process-based parallel programming. In fact, both MPI and process-based communication primitives are built to extend over network boundaries, making them well-suited for distributed environments. MPI programs can be executed on shared memory machines, just as shared-memory programs can be run on distributed memory machines, but this is not MPI’s ideal use case [63]. Additionally, the full MPI standard is quite large and complex, so lightweight versions of MPI have been created for use in embedded domains that are subject to memory, power, and interconnect constraints [63].

**Erlang**

Erlang is a functional programming language that originated from Ericsson in the mid 1980s [18]. The language has built-in concurrency primitives for lightweight processes and message passing. The Erlang programming model stresses the use of data localization and cooperation between processes [119]. Data is never implicitly shared between processes, as it can only be shared explicitly via message passing. Erlang can be compiled or interpreted, and it was designed to be used in distributed systems. Differences between local (on-node) and remote (inter-node) communications are hidden by the Erlang run-time system; allowing process distribution to remain transparent to the user. The lightweight nature of Erlang processes paired with an extremely efficient message-passing protocol allow for extreme scalability. Design of Erlang also focused on reliability, leading to built-in support for hot code updates and notification mechanisms for process failure. In general, Erlang represents a comprehensive language and run-time system that was originally built for distributed systems, but may also find a niche in multi- and many-core environments.
Unified Parallel C

Unified Parallel C (UPC) is an extension of the C programming language that attempts to unify shared- and distributed-memory programming models for large-scale parallel machines. The programming model provides a single shared address space to allow for ease of programmability, while integrating some of the performance benefits of message-passing on distributed memory systems. The UPC model is thread-based, and makes use of a custom run-time layer (GASNet) that allows for shared-memory programs to operate in distributed-memory networked environments [33]. UPC, like OpenMP, is compiler controlled, and not a library as PThreads is implemented. UPC compilers perform a source-to-source translation that results in a C program that makes use of either threads or processes along with the custom GASNet run-time layer [44]. The resulting programs are hybrid parallel programs in that they use a combination of shared-memory and distributed-memory constructs. While UPC provides a uniform programming model for a variety of different types of computer architectures, performance of an application may vary widely due to underlying hardware features, operating system implementations, networking framework, as well as source-to-source transformations.

2.3 Heterogeneous Multi-Core Systems

Heterogeneous computing systems are composed of different types of computational units. All heterogeneous systems are inherently parallel, as they must contain at least two types of computational units in order to have heterogeneous characteristics. Computational units are generally programmable in nature, and include general-purpose processors, DSP units, GPUs, and microcontrollers. However, non-programmable fixed-function units may also exist, including hardware accelerators for encryption/decryption, packet processing, network offload, and fixed graphics pipelines. The presence of multiple numbers and types of computational units introduces challenges not found in typical homogeneous systems. These challenges are a superset of those involved in traditional parallel systems, but the
presence of multiple instruction set architectures (ISAs) forces developers to make use of multiple compilers and linkers to generate heterogeneous executables. This significantly complicates the development process, and leads to significant obstacles when developing programming model support for heterogeneous systems; including generation of libraries and run-time systems for heterogeneous platforms. The next sections are dedicated to the description of many of the key challenges involved with heterogeneous system development, programming, and integration, as well as the current status of heterogeneous architectures.

2.3.1 Arachne

Arachne is a user-level threading package aimed at providing heterogeneous thread migration amongst networked machines [42]. The Arachne system is based on that of Ythreads and Ariadne, two systems that support for thread migration in distributed systems. The Ariadne system only contained support for homogeneous thread migration, whereas the Ythreads system allowed for heterogeneous thread migration, but limited the use of thread primitives to top-level thread functions [105]. The Arachne system supports heterogeneous thread migration of user-level threads while placing no restrictions on the use of thread primitives within threads bodies.

The Arachne system is composed of two major components: (1) _app_ the Arachne preprocessor, and (2) the Arachne run-time system. The Arachne preprocessor, _app_ is a source-to-source compiler that transforms a program written in Arachne-specific dialect of C++ into a native C++ program. The Arachne dialect of C++ extends native C++ with 3 new keywords: thread, strand, and call. These keywords provide important hints to the preprocessor that allow for threads to be distinguished by name rather than by function pointer; thereby making threads first class objects in the Arachne dialect of C++. This is important as function pointers are absolute addresses within an architecture-dependent binary, and are not useful in heterogeneous settings. The preprocessor generates native C++ code, sans the
Arachne-specific keywords, while also generating a set of helper files used in packing/unpacking thread context during migration as well as an array of function pointers for each target architecture that are indexable by thread identifiers [105, 42].

The generated code contains jump-table ascriptions and context push/pop operations to allow for thread suspension, migration, and resumption. All context push/pop operations translate architecture-dependent activation records and data types on a thread’s stack into architecture-independent forms suitable for transmission over a communication network. Upon migration, the thread’s context is un-packed into the receiving processor’s format, and the thread is made ready to run. The preprocessing framework of Arachne introduces the notion of first-class threads and thread-primitives while circumventing the need for supporting a full-blown compiler. This greatly eases integration of future architectures, and relieves users from having to maintain and support special compilers for Arachne systems [42].

The Arachne run-time system is a user-level threading system built on top of PVM to achieve distribution across networks of machines. Each processor in an Arachne system executes within a OS-level process that hosts that processor’s Arachne threads. Arachne threads do not support preemption as they are user-level threads that are not visible to the underlying OS. Each process uses PVM communication facilities to communicate with one another during migration operations. For multi-core systems, representing virtual CPUs as processes may be too heavyweight of a solution, and could eliminate the possibility of exploiting hardware multithreading capabilities including HyperThreading, or other forms of simultaneous multithreading (SMT). The Arachne run-time system supports the following set of thread primitives:

- **a_create** - Creates a new thread.
- **a_destroy** - Destroys a thread identifier.
- **a_migrate** - Migrates the calling thread to another process.
- **a_resume** - Resumes (joins) on a given thread.
• `a_set_priority` - Alters the priority of a given thread.

• `a_suicide` - Destroys the calling thread.

• `a_suspend` - Suspends (blocks) the calling thread.

• `a_yield` - Conditionally suspends the calling thread.

• `a_yield_to` - Conditionally suspends the calling thread for a specified thread if it is ready to run.

Missing from this list of primitive are basic synchronization and signaling functions. These types of operations are necessary for protecting shared data objects from concurrent access from multiple threads, but not necessary for demonstrating thread migration capabilities. The lack of synchronization operations is based on the fact that these operations may be OS- and hardware-specific, thus making them difficult to migrate between heterogeneous machines [42]. The distributed nature of Arachne does not seem to support shared data objects, especially in migratory threads, as only the threads local context (activation records, stack) is migrated. Global variables are not common amongst all nodes in an Arachne system, thereby making threads essentially independent, isolated computations. Future development aimed to support global object migration, however results have never been published [42].

2.3.2 IBM Cell

IBM’s Cell B.E. (Broadband Engine) is a heterogeneous platform that features a general-purpose Power-architecture processor along with a set of synergistic vector processing units; all interconnected with a high-speed network-on-chip as shown in Figure 2.16. The Cell’s SoC architecture allows a programmer to spread computation across different types of processors. Programmers utilize a dedicated set of application-programmer interfaces (APIs) to create heterogeneous threads that run on the SPEs (spe Xxx functions). A programmer can make a single executable for the Cell platform that contains code that executes
on both the PPE units and SPE units. This is accomplished via embedding SPE executables (ELFs) into a PPE executable [47]. Work is currently being done to automate the process of embedding SPE executables through higher-level pragmas in the CellSs source-to-source compilation framework [28, 100].

Heterogeneous threads on the Cell are able to share data via DMA transfers over the high-speed EIB interconnect. SPE threads only have access to a 256KB local store, therefore coherent DMA transfers must be used to move data back-and-forth to global memory [99]. Additional software libraries can be used to emulate shared-memory without requiring explicit use of DMA commands. Synchronization between threads can be done via special SPE-specific constructs found within the Cell SDK that includes signals, mutexes, and mailboxes. Additionally, the RPC mechanism allows SPE-based programs to invoke higher-level OS commands on the PPE processor in a cooperative fashion. This RPC mechanism is built upon an inter-processor signaling capability native to the Cell architecture [69, 70].

LIBSPE

IBM provides a special set of SPE APIs packaged in libspe [69] to create and manage SPE-based threads. Running SPE-based threads is synchronous, so if a program wants to launch
N SPE-threads, it must do so in the context of N+1 PPE-based threads, where N PPE threads are used to run the SPE threads, and 1 PPE thread is used as the main thread. Therefore, to spawn a truly *asynchronous* SPE-thread, one must first spawn an asynchronous OS-level thread (i.e. POSIX thread), and then this thread will spawn and synchronously run the SPE thread. This programming model decouples host-level threading from SPE multithreading, but it forces programmers to know and understand two programming models, as opposed to a single uniform programming model.

Figure 2.17: IBM Cell SPUFS Programming Environment

The most popular operating system framework for the Cell platform is Linux paired with the SPUFS virtual filesystem. The SPUFS filesystem is a repository for all SPU-related data required to manage and schedule threads destined to execute on the SPEs. All management and scheduling routines are distinct in that they are not related to the core Linux scheduler as shown in Figure 2.17. SPUFS is essentially a mini operating system that executes solely on the PPE. SPUFS makes use of the status and control registers within each SPE’s MFC in order to monitor the current status of underlying SPE architecture. SPE threads are added
to the SPUFS ready-to-run queue as they are created. When SPUFS detects an idle SPE, it calculates an updated scheduling decision, and dispatches a new thread on the idle SPE. The SPUFS uses several scheduling algorithms including the traditional round-robin as well as gang-scheduling algorithms for simultaneously dispatching sets of SPE threads at once. SPUFS is able to preemptively schedule SPE threads, however it is not a recommended practice as context switching on SPEs involves considerable amounts of overhead due to the local store architecture. A running thread has *full* control of the 256KB local store as well as the large SPE vector register file. Preempting an SPE thread involves temporarily storing all of the vector register file contents in memory, as well as all data portions of the local store. SPUFS and Linux on Cell developers suggest that one should only create as many SPE threads as there are physical SPE units in order to prevent preemption invocation in the SPUFS scheduler.

**Virtualizing The Cell: CellVM and HeraJVM**

Several research projects focusing on providing higher-level parallel programming models for Cell have begun; both of which aim to build a heterogeneous JVM infrastructure [94, 88]. The virtual machine approach used by both of these projects allow for multiple JVMs; one PPE-based and another SPE-based to interpret a JVM program stored in bytecode form. While this approach eliminates the problem of code compatibility, it introduces interpretation overhead and a complex just-in-time compilation system. Additionally, both approaches depend on underlying hardware and OS support for synchronization and communication amongst the heterogeneous processors. In CellVM, The lack of OS services available to SPE processors forces each SPE-based JVM to have a corresponding signal handler thread that executes on the PPE [94]. For any call that cannot be carried out on the SPE, i.e. synchronization and native methods, the SPE must signal the PPE thread to perform the work on its behalf, while the SPE stalls until the PPE signals to it that the operation is complete [94, 88]. This dependence on the PPE for all synchronization calls
presents a huge bottleneck for multithreaded applications, leading to significant slowdowns due to the cost of switching control from SPE to PPE and back. Both the CellVM and Hera-JVM projects identify the current lack of uniform OS support in heterogeneous systems as a serious drawback; and a future research goal.

**SHIM**

SHIM is a language rooted in model-based design of parallel programs [45, 118]. The design of the language aims to eliminate many of the problems inherent to already existing parallel programming models; namely non-determinism, data races, consistency issues, and deadlock [78]. The aim of this work is to use advanced compiler technology to transform SHIM-based programs into native code capable of exploiting the resources of modern heterogeneous MPSoCs, including IBM’s Cell [118].

SHIM is built on top of the notion of concurrently executing tasks that communicate via asynchronous message-passing over point-to-point channels; a model whose basis can be found in both Communicating Sequential Processes (CSP) Kahn Process Networks (KPN) [74, 62]. All channels are FIFO-based, thereby making sure that the transmission and receipt of data through a channel is seen in a deterministic order. This model enables deterministic execution traces and eliminates the chance for data races.

While SHIM aims to make the process of creating parallel programs easier for the end user by eliminating the use of non-deterministic threads; the SHIM system itself depends on the existence of thread-based primitives in order to execute [46, 118]. While the inherent problems with multithreaded programming may be hidden from the user in this model, the underlying limitations of homogeneous run-time systems have a direct effect on the architecture of SHIM. For instance, the cross-processor synchronization operations available for use on the Cell platform are not able to yield to the operating system’s scheduler [118, 68]. The lack of appropriate heterogeneous compatible OS facilities required a workaround to be constructed. The workaround resulted in the use of an RPC-based mechanism to route
all synchronization requests from each SPE to the PPE. This requires the PPE to take on the burden of control for all heterogeneous processing elements. In order to handle asynchronous invocation of control routines from each SPE, all SPE-related requests must have an associated proxy thread running on the PPE. The proxy thread, implemented as a traditional POSIX thread running on the host processor, is responsible for transforming OS-level events between the two heterogeneous domains.

2.3.3 Intel EXOCHI

Intel’s EXOCHI [121] is another effort to enable programmers to use heterogeneous components using OpenMP along with a special set of pragmas and the CHI (C for Heterogeneous Integration) run-time environment and compiler. Any processor, or sequencer, can become a part of the EXOCHI system, by implementing a simple interface, or exoskeleton, defined by the EXOCHI system [121, 57]. The exoskeleton interface provides a set of mechanisms that enable slave processors to request services from a master processor. This allows for the construction of heterogeneous operating system services through the use of cooperative RPC mechanisms. The MISP extensions are meant to be used in systems with two classifications of processors [57]:

- OS-Managed Sequencers (OMS).
- Application-Managed Sequencers (AMS).

Within MISP, a sequencer is defined as a hardware thread context that is fully capable of executing and independent instruction stream. An OS-managed sequencer (OMS) is a traditional CPU that is fully managed by an operating system, and is able to execute both user-level, and system-level code in different privilege levels. In MISP systems, the OMS is a traditional IA-32 host CPU. Conversely, an application-managed sequencer (AMS) is a user-level resource that is not necessarily able to execute system-level code. AMS units are managed by applications, as the OS is not cognizant of them, except when saving
and restoring AMS state during context switches. The only extension required to enable traditional operating systems frameworks to manage AMS units is a memory-space used for saving/restoring cumulative AMS state [57].

MISP Extensions

Intel has defined an ISA extension termed MISP, or Multiple Instruction Stream Processor. This extension defines an inter-processor communication (IPC) mechanism called signal. The exoskeleton interface is implemented as a thin software wrapper that utilizes the MISP signal extension for bootstrapping, and running shreds, or user-level threads on multiple types of processing units. The exoskeleton also defines a method for routing exceptions and page faults, from the exo-sequencers to the master CPU for processing. This mechanism, called Collaborative Exception Handling (CEH), will most likely also be used to allow exo-sequencers to invoke system services [121, 57]. The CEH mechanism is similar to an on-chip RPC mechanism; depending on the MISP inter-processor signaling function to perform work on the behalf of sequencer on a host processor [57].

Intel has chosen a fully cache coherent virtual memory system for use in its MISP prototype. The virtual memory system requires that all CPUs agree on virtual-to-physical page mappings, therefore all page table entries in the system must remain coherent, even amongst CPUs with different TLB architectures. To implement coherent virtual memory, Intel has chosen to use collaborative exception handling (CEH) to forward all page faults to the host processor. This provides a centralized coherency mechanism for handling page fault exceptions, but requires a translation mechanism to transform standard IA-32 TLB entries to those entries suited for sequencers. This translation mechanism is called Address Translation Remapping, or ATR [57, 121]. This RPC-style mechanism can be used to provide other forms of proxy execution as it allows one to define a set of architectural triggering conditions that cause a fault on an exosequencer. The fault exception or system call request can then be relayed to a host processor to be handled, and then the results can be passed
back to the faulting sequencer. The host processor must save its current state, assume the state of the faulting sequencer, and re-execute the faulting instructions, thus causing a local exception and invoking OS services [57]. The fault is handled by the OS, the results are returned to the host, and the host is responsible for marshaling the results back to the sequencer so that it can resume execution. The host processor must then restore its state so that it can continue executing until the next sequencer fault activation. This is quite similar to the migrate-on-fault approach used by Intel in ISA-asymmetric systems [80], however CEH depends on the MISP ISA extensions, while migrate-on-fault depends on x86-specific opcode filtering hardware [80].

**EXO + CHI**

![EXOCHI Programming Environment](image)

Figure 2.18: EXOCHI Programming Environment [121]

The EXOCHI system extends the MISP system into a cohesive parallel programming model that makes use of OpenMP. The primary goal of this system is to enable programmers to efficiently develop code for heterogeneous platforms while retaining the look-and-feel of traditional programming models [121]. The framework reuses the MISP extensions in the form of the Exoskeleton Sequencer (EXO), but adds a new compilation framework and run-time system [121]. The new run-time system, called CHI, stands for C for Heterogeneous
Integration [121]. Similar to AMS units in MISP, non-standard CPU cores are exposed to programmers as an exo-sequencer resource. The CHI run-time system and compilation flow enables a mixed language development environment; allowing for C/C++ to be mixed with accelerator-specific assembly and domain-specific languages [121]. The run-time system extends the OpenMP [38, 121] parallel programming model with pragmas that allow a programmer to specify sections of code targeted for heterogeneous acceleration units. All host CPUs (OMS units) are managed by the operating system, while exo-sequencers are managed through a combination of user-level control and the CHI run-time library as shown in Figure 2.18 [121].

Heterogeneous threads in EXOCHI operate in a shared virtual-memory environment, therefore only OS-support for address translation is needed for communication, which is handled by CEH and ATR [121]. This means that all memory accesses are cache coherent, unlike the Cell, in which only DMA operations are coherent. While full cache coherency is useful for programmer productivity, the overhead of cache coherency protocols can cause reductions in overall system performance [116]. Additionally, it is not yet clear if fully cache-coherent multi-processor systems will remain viable in the future as the number of processors continues to scale [132, 71, 26]. Furthermore, the cost of maintaining TLB coherency within EXOCHI includes the additional overhead of CEH/ATR, and may also experience scalability problems in the future [104]. However it is important to note that while the design of EXOCHI depends on a shared address space, the design does not depend on the the existence of coherence support in hardware. Software-based cache coherency, or explicit data-copying models can be used within EXOCHI, but at the cost of programmer efficiency. Evaluations of the EXOCHI system under various data copying models show that the run-time system is applicable in a variety of scenarios including: full hardware coherency, software-managed coherency, and explicit data copying models [121]. Both software-managed coherency and explicit data-copying techniques suffer from overhead incurred during cache flushing and memory copy routines. However, the use of heteroge-
neous multithreading, even in non-hardware coherent systems, is able to achieve significant speedups compared to traditional host-only execution [121].

**Pangaea**

Intel’s Pangaea project is an extension of the EXOCHI work, including an entirely FPGA-based prototype system as well as a small instruction set extension to support heterogeneous inter-processor communication. The traditional method of notifying a processor of external events via interrupts can involve large amounts of latency and jitter. Currently, many systems use inter-processor interrupts (IPIs) as the base communication mechanism in multicore systems. The Pangaea project has recognized the inefficiency of IPI mechanisms, and has proposed an ISA extension to support lighter-weight, user-level interrupts (ULIs) [125]. While the ULI extensions are very similar to traditional interrupt handling facilities, they differ in that they define a standard architecture and methodology for registering handlers and responding to interrupt events, or scenarios. This brings interrupt processing out of the operating system and into the programming model; allowing ULIs to remain fully under an application programmer’s control. Another key difference between ULIs and traditional interrupts, is that ULIs have been defined to be handled completely in user-mode, requiring only a minimal amount of processor state to be saved in order to invoke the ULI handler routine. The Pangaea ULI extension includes the following instructions:

- **EMONITOR**(scenario, handler) - registers a ULI handler for a given event (scenario).
- **ERETURN()** - final instruction of a ULI handler, pops handler state off of stack, and returns PC to interrupted instruction.
- **SIGNAL**(id, instr, stack) - inter-processor signal mechanism, often used to spawn threads on execution units.

The architectural modifications to enable ULI processing also includes a set of monitor registers, called channels, that store scenario-to-handler mappings. ULI interrupts are
lower priority that all other IA32 interrupt mechanisms, therefore they only fire when other
interrupts are not pending. The CPU pipeline is flushed upon invocation of a ULI scenario,
and all ULI events are disabled. Next, the contents of the channel register associated with
the given scenario is examined. This value is used to jump to the ULI handler. The ULI
handler executes to completion and invokes an ERETURN instruction to return to normal
program execution and to re-enable ULI events. To an application program, the invocation
of the ULI handler resembles that of an asynchronous function call [57].

The main use-case for ULIs is to allow programmers to write small event handlers and
managers to launch and schedule threads on heterogeneous execution units [125]. Effi-
ciency is achieved by eliminating OS intervention during interrupts as well as allowing a
user to define custom handlers, schedulers, and thread spawning routines. While ULIs are
much more efficient than traditional OS-managed inter-processor interrupts, they still re-
quire interrupting a host (application) processor to service the needs and requests of other
heterogeneous processing elements.

2.3.4 NVIDIA CUDA

NVIDIA’s CUDA, or Compute-Unified Device Architecture, is both a platform and compiler-
suite for allowing graphics processing units (GPUs) to be used for general-purpose compu-
tations [97]. This type of general purpose use of graphics cards has led to the coinage of
a contradictory new term: GPGPU, or general-purpose graphics processing units. Graph-
ics cards have always been quite dense in computational power, but their traditional com-
pute pipelines were dedicated towards graphics-centric computations. Current generations
of graphics processing units are more flexible in that individual processing elements re-
semble traditional SIMD/vector processors, programmable through a general-purpose ISA.
Programmers decided that they wanted to exploit the massive computational power found
within GPUs, and developed compilation frameworks to allow for non-graphics related pro-
grams to execute on the GPU hardware.
CUDA devices are application accelerators that must be used in a hosted environment. The CUDA card acts as a pure slave, only able to perform computations on its local on-chip memories. CUDA cards are dependent on a host processor to marshal data into and out of the GPU’s memory banks. While most CUDA cards contain upwards of 1 gigabyte of on-chip memory, they can only be used in a master/slave format as there is no support for CUDA-initiated task synchronization, signaling, OS calls, or system-wide DMAs.

The underlying CUDA compute architecture is composed of a complex, software-managed memory hierarchy paired with a large amount of SIMD execution units. The execution units lack hardware-support for branch prediction, and make heavy use of simultaneous multi-threading (SMT) mechanisms to hide memory latency. Architecture-specific features of the SMT mechanisms are reflected directly in the CUDA programming model through the notion of thread warps. A thread warp is a group of 32 threads that share a single execution unit within a CUDA card. While threads within a warp are allowed to follow different execution paths, only a single thread is active at a time. Divergent computations force all other threads within the warp to wait until the computation re-converges. On the other hand, all threads within a warp can execute simultaneously when they are all following the same execution path; so in practice, efficient CUDA programming requires proper thread-to-warp mapping.

Efficient CUDA programming requires extremely fine-grained tuning of application code to run specifically within the constraints of the underlying architecture. This includes a wide variety of factors: memory bank size, data layout and alignment, number of threads per warp, work per warp, vector sizes, etc. CUDA’s SIMD architecture is well-suited for regular, data-parallel applications that focus on the decomposition of parallel loop bodies [53]. However, the master/slave model paired with poor support for irregular, or dynamic branching parallelism makes it extremely difficult to use CUDA for other parallelization strategies.
2.3.5 Inria StarPU

INRIA’s StarPU run-time system uses the idea of a codelet [21] as the basis for a heterogeneous parallel programming model. StarPU targets systems with heterogeneous sets of accelerators; including systems featuring FPGAs, programmable GPUs (GP-GPUs), and synergistic processing elements similar to that of IBM’s Cell. A codelet is a fully encapsulated task that can be run on a variety of different hardware computing units based on dynamic factors.

The StarPU run-time system is responsible for scheduling codelets, as well as dynamically managing the movement of input and output data for executing codelets. The ability of the run-time system to aid in data movement is important as many heterogeneous accelerator technologies do not provide fully shared, coherent memories. Forcing programmers to explicitly manage non-uniform memory hierarchies can result in serious deficiencies in terms of programmability and code portability. StarPU intends to relieve programmers from this arduous task by allowing a workload to be described in terms of groups of codelets; or portable descriptions of tasks that are automatically managed by the StarPU run-time system.

StarPU defines codelets as non-preemptible, offloadable tasks [21]. The codelets operate within a unified execution model, in which all data management is also handled by the StarPU run-time system [22]. All codelet descriptions must be accompanied by a complete description of their input and output data structures [21]. This description is used by StarPU during codelet initialization and finalization. Each codelet has a callback mechanism that can be invoked on a host processor upon termination of a codelet. The callback mechanism can be used to enforce dependencies between codelets by delaying codelet execution until all dependencies have been satisfied. The actual location of where a codelet runs in StarPU is dynamic, therefore requiring that a codelet must actually contain multiple implementations of the same function. For example, a Cell-based codelet may have two executables: (1) for execution on the PPU, (2) another for execution on SPU as shown in
// Allocate space for codelet
codelet *c1 = malloc(sizeof(codelet));

// Setup codelet parameters and functions
c1->where = ANY;
c1->ppu_func = matrix_mult_ppu;
c1->spu_func = matrix_mult_spu;
c1->cb = matrix_mult_callback_func;
c1->argcb = &jobcounter;

// Setup codelet data description
c1->nbuffers = 3; [...] 
c1->buffers[2].state = subdata_ref(&C, 2, i, j);
c1->buffers[2].mode = W;

// Schedule codelet for execution
push_task(c1);

Figure 2.19: Example Codelet for Cell [21]

Figure 2.19 [21].

All accelerator-host interactions in StarPU are handled by the drivers executing on each accelerator. StarPU accelerator drivers must support fetching codelets from the StarPU codelet queues, support for data management, and callback notification. For GPU-based systems this is quite simple, as the accelerator is purely host-controlled. In fact, the StarPU run-time for GPU-based systems is synchronous, and blocks during codelet execution [22]. However, driver architecture can be much more complex for systems with multiple accelerators that are not pure-slaves, such as the SPUs in Cell-based systems. In the case of the Cell, codelet callbacks are scheduled on the host via inter-processor interrupts [21].

The StarPU run-time system unifies a set of heterogeneous processing facilities using a new codelet abstraction. Codelets encapsulate enough information so that they are able to execute on any of heterogeneous processing resources available in a given system. This is enabled by programmer supplied descriptions of the input/output needs of each codelet along with its individual implementations for each accelerator type. The run-time system is work-queue based; where codelets are dynamically scheduled on accelerators as they become free. While there may be multiple queues, with multiple concurrently executing codelets, all inter-codelet synchronization is implemented on the host using RPC-style callback mechanisms.
2.3.6 Microkernels

There are many opinions concerning how one should structure and organize the internals of operating systems [40, 34], but in general the opinions fall into two major categories: monolithic kernels and microkernels. In general, a monolithic kernel is an operating system that is built as a single unit that may not contain hard interfaces between particular OS modules or functions [55, 83]. On the other hand, a microkernel is an operating system that is built using the principles of modularity and encapsulation; building each OS module or function as a separate server that may communicate with other servers through a communication mechanism provided by a small lightweight kernel [55, 91].

![Figure 2.20: OS Structures: Monolithic Vs. Microkernel](image)

Unlike monolithic kernels, which execute completely in privileged mode, Microkernels aim to minimize the amount of code that must be run in privileged (supervisor) mode by enabling user-level definitions of traditional OS services [91] as shown in Figure 2.20. Microkernels are extensible in that higher-level functionality can be built and layered at the user-level through server communication and coordination; thus allowing for both horizontal and vertical integration of OS modules [91]. Microkernels seek to abide by Liedtke’s minimality principle; only tolerating a concept inside of the microkernel if placing it outside the kernel would adversely effect system implementation and functionality [82].
practice, many microkernels are able to limit privileged code functionality to [83]:

- Address Space Management.
- Inter-Process Communication (IPC).
- Basic Scheduling Services.

The IPC interface provided by the microkernel enforces modularity principles, as it is the only mechanism available to communicate data and control between OS modules. However, it is important to note that IPC performance is a key factor in microkernel OS performance [83, 81, 91]. The following sections discuss relevant microkernel research projects and how each attempts to solve challenges within the realm of operating systems performance, scalability, and applicability to heterogeneous platforms.

Mach

Mach is a first-generation microkernel that finds its lineage in the Accent operating system at Carnegie Mellon University [113]. The structure of the Accent operating system was based upon the principles of modularity, protection, and transparent process communication [103]. All communication in Accent is accomplished via messages, whether it is process-process communication or process-kernel communication. Accent also pioneered the use of memory-management units (MMUs) to build copy-on-write mechanisms to build efficient message transfer mechanisms [103]. Accent’s introduction of copy-on-write semantics along with message-based communication has greatly influenced operating system and microkernel designs even to this day.

Mach sought to build upon the structure of Accent, but to create a microkernel-based system that was binary compatible with UNIX. The goal was to create a lightweight kernel that could not only support normal UNIX system facilities, but also one that could serve as an extensible foundation on which traditional UNIX facilities can be constructed [1].
Additionally, the designers wanted to create a kernel that contained several important new features [1]:

- Multiprocessor support for both tightly- and loosely-coupled parallel machines.
- Improved virtual memory support, including support for shared- and distributed-memory machines.
- Capability-based communication mechanisms.

Mach was initially developed within the BSD-UNIX kernel, by replacing standard UNIX components with Mach components. However, all replaced components still remained inside of the kernel. In its 3rd release, Mach 3 stripped the kernel of all UNIX code by migrating all BSD kernel code into user-level components [113]. This resulted in an extremely small microkernel compared to the previous versions of Mach, as the microkernel now only implemented the basic features of Mach. This type of microkernel structure enables replacement of OS components, as well as the ability to host multiple operating systems interfaces simultaneously [113]. This concept is analogous to para-virtualization, where higher-level virtual machines, or operating systems, target a software-defined interface provided by a microkernel or hypervisor. Mach’s integration of the virtual-memory system with its messaging facilities allows for flexible implementations of memory management tasks. This flexibility allows for the system to be highly-tuned for different types of memory hierarchies, including both shared- and distributed-memory machines [113].

Overall, the Mach operating system pioneered the use of a small microkernel to provide primitive system calls on which higher-level functions can be constructed [113]. This structure reduces the size of the typical operating system kernel by pushing the majority of system functionality into user-level libraries and servers. The resulting system is modular and extensible, and can support a wide variety of operating system interfaces [113].
L4

The L4 microkernel family has evolved from the L3 microkernel originally developed by Jochen Liedtke [81]. Liedtke recognized that microkernels represented a viable operating system framework as long as the performance of such kernels could compete with popular monolithic kernels of the time. His work resulted in the construction of efficient IPC mechanisms that enabled the construction of high-performance microkernel operating systems [81, 84, 83, 82]. Liedtke focused on streamlining IPC mechanisms; striving for simple routines that had small cache footprints.

The original version of the L4 microkernel was coded entirely in x86 assembly language in order to produce a highly-optimized and fast microkernel [84]. However, coding an entire operating system in low-level assembly leads to obvious portability and maintenance issues, so future work on L4 emphasized the use of higher-level languages (HLLs) including C and C++ [84]. While the performance of initial versions of L4 coded in HLLs suffered, further optimization and careful redesign of the internals of L4 have produced implementations that are comparable in performance to the initial assembly-only implementations [84, 39]. The use of high-level languages in L4 development has dramatically improved the portability of the kernel; and it is has now been ported to a variety of server, desktop, and embedded platforms. More recently, L4 microkernel development has branched in several different, but related, directions:

- **Real-Time and Embedded Systems [85]**
  - Exploiting the relative simplicity of microkernels to gain increased system predictability.
  - Creating ultra-lightweight microkernels for embedded systems.

- **Secure Run-Time Systems [75]**
  - Exploiting the modularity of microkernels to enforce security policies.
Leveraging the explicit communication flows in microkernels to create formal security proofs.

**Singularity**

Microsoft’s Singularity project’s main goal is to create a research operating system framework fit for prototyping and evaluating new operating system and language features [65]. The current prototype OS is unique in that it has been built using a modern type-safe high-level language, namely C# [64]. Another unique property of Singularity is that it operates with the notion of *Software Isolated Processes* (SIPs); a mechanism which provides encapsulation and data protection of user-level processes without a dependence on hardware features associated with memory-management units [64]. The isolation properties provided by SIPs ensure that individual processes can neither share code nor data, and all message-based communication between SIPs is moderated by the kernel in a type-safe manner. SIPs are able to operate with individual run-time systems, managed by the Singularity microkernel. The microkernel provides the essential communication, synchronization, scheduling, and memory management services to SIPs via a system call interface.

An offshoot of the Singularity project, named Helios, has been created in order to target heterogeneous multi-core systems with non-uniform memory architectures (NUMA) [93]. The Helios operating system introduces the notion of *satellite kernels*, or cooperative kernels that are able create a single set of uniform OS abstractions across sets of heterogeneous processing units [93]. This concept is reminiscent to that of satellite processors [23], which provides a lightweight software layer (satellite kernel) that enables slave processors to request UNIX-based OS services from a single master processor via interrupts. Helios differs slightly from this approach as it attempts to reduce remote service requests by servicing them locally if possible – however in the case that a request cannot be handled locally, the request is forwarded to a coordinator kernel (master) via RPC that is implemented using inter-processor interrupts. Additionally, Helios attempts to ease the heterogeneous com-
pilation process by using a common language intermediate form that can be just-in-time compiled to target the various ISAs of a heterogeneous platform. The intermediate form used within Helios is the common intermediate language (CIL) – the standard intermediate form of Microsoft’s .NET platform [93].

A unique feature of Helios is the use of affinity to indicate message-passing communication patterns. Affinity in Helios is represented by a single integer, where the sign and magnitude of the affinity value indicate the desired degree of communication performance between processes [93]. A positive affinity value denotes a tight coupling between SIPs, indicating to the system that a fast zero-copy (local) message passing channel should be used. Conversely, a negative affinity value denotes that a SIP will be executing in isolation. Additionally, the default affinity value of zero indicates no preference to the OS, allowing the OS to make it’s own decision as to how to place the SIP. Affinity is specified through process manifests which are automatically generated by the the Sing#-based programming model that is used in both Helios and Singularity. While manifests are automatically generated, programmers may still manually tune affinity values to their liking in order to alter the communication characteristics of processes executing within Helios. The separation of application from manifest enables application communication patterns to be tuned without having to change application code. Affinity values are meant to be used to influence placement decisions at run-time. Positive affinities can indicate that frequently communicating SIPs should be co-located on a single CPU. Negative affinities can be used to indicate distribution of SIPs amongst independent CPUs. Additionally, affinity hierarchies can be used to indicate levels of preference in various heterogeneous systems.

**Barrelfish**

The Barrelfish project has began investigating and prototyping new multikernel OS structures for heterogeneous systems [108, 26]. The project recognizes that heterogeneous manycore hardware may not be able to depend on the existence of fully uniform, cache
coherent interconnects in the future due to scalability reasons [26]. Hardware platforms already consist of message-based point-to-point interconnects, which has resulted in the Barreelfish effort focusing on a shared-nothing approach with 3 basic tenets [26, 27]:

1. All inter-core communication must be explicit.

2. Operating system structure must be hardware-neutral.

3. State should not be considered as shared, however it can be replicated.

The approach aims to build a message-passing based multikernel that distributes functionality amongst cores. Prototype systems have been built on several different homogeneous platforms using a user-level RPC mechanism (URPC) that currently utilizes a region of shared memory for routing point-to-point messages. The multikernel and microkernel approaches are similar in that they both use distributed message-passing schemes, however the Barreelfish designers distinguish the two styles along the following lines:

“We see a multikernel as distinct from a microkernel, which also uses message-based communication between processes to achieve protection and isolation but remains a shared-memory, multithreaded system in the kernel. For instance, Barreelfish has some structural similarity to a microkernel, in that it consists of a distributed system of communicating user-space processes which provide services to applications. However, unlike multiprocessor microkernels, each core in the machine is managed completely independently – the CPU driver and monitor share no data structures with other cores except for message channels [26].”

While the multikernel approach attempts to share less data structures than a typical microkernel approach, the URPC mechanisms built thus far are tightly coupled with shared memory and snoopy cache coherency schemes [26]. This tight coupling is very similar to the way efficient IPC mechanisms are built in microkernel-based operating systems. This is
concerning as cache coherency schemes are tightly coupled with processor architecture and may not be globally uniform amongst computational components in a given heterogeneous platform.
Chapter 3

Design Overview

This chapter lays out the road map for the work contained in this dissertation. The research accomplishments in this dissertation build upon the original hthreads system developed for hybrid CPU/FPGA components. The original hthreads design was developed to allow designers to create custom accelerator cores within the FPGA fabric, and integrate these cores under the familiar POSIX threads (PThreads) programming model \[32\]. The original hthreads system is described in more detail in Chapter 4.

The original hthreads followed the distributed microkernel approach which partitioned operating system services into independent and parallel functional units. The hthreads OS then took the additional step of transitioning key functional units into hardware cores. Custom hardware accelerators accessed these cores using a hardware abstraction layer known as the hardware thread interface. The hardware abstraction layer allowed threads created as custom accelerators to plug into, or access, the set of distributed run-time service functions \[5, 4, 17\]. Adopting hthreads for heterogeneous MPSoCs required a new abstraction layer for general purpose processors. This layer, called the heterogeneous coordination layer is described in more detail in Chapter 5.

The modified set of hthreads cores with the heterogeneous coordination layer allowed programs running on processors with different ISAs to call a uniform set of operating system APIs. However, to create a virtual machine model that abstracts the platform-specific details from the user also requires changes to the compilation of the source code for each thread. Each application-level thread must be compiled into the assembly language of the processor on which it will execute. Furthermore, different threads within a common source code file may need to be compiled to different ISAs in order to accommodate systems with heterogeneous types of processing cores. A new augmented design flow was created to
serve this purpose [6, 4]. Chapter 6 describes the creation of an augmented design flow that enables a programmer to target a POSIX-compatible source application for execution across a heterogeneous multi-core platform.

Finally, Chapter 7 describes the prototype heterogeneous platform, and the set of tests, benchmarks, and case studies that were used to evaluate the effectiveness and usability of the new hardware-based microkernel framework [8, 9, 7, 5, 4, 6].
Chapter 4

Hthreads Microkernel Framework

The structure of traditional software microkernels was discussed in Section 2.3.6. In this chapter we introduce hthreads, which further decomposes the distributed microkernel structure across the hardware/software boundary. The system, called hthreads, was first developed to allow functions to be created as custom hardware circuits under control of the operating system. The basic structure of hthreads is well-suited for heterogeneous MPSoCs as the core operating system functionality is already hardware agnostic. The core of the hthreads OS is implemented as a set of independent hardware accelerators, each in charge of a different piece of OS functionality [10, 17]. The operating system was originally designed to enable the multithreaded programming model to be applied to both software and hardware components. Currently, the hthreads scheduler is capable of scheduling both homogeneous software threads and hardware threads, thus producing a platform suitable for handling limited heterogeneity. While custom hardware threads are promising for many applications, they are still quite difficult to develop for both hardware and software designers. Additionally, even after initial development, custom hardware accelerators remain difficult to maintain, extend, and reuse.

The architecture of hthreads is itself a multi-core platform, as it is composed of several dedicated OS processing cores. These cores communicate and interact with one another to...
create an OS infrastructure for other computational components. The general structure of the hthreads operating system is shown in Figure 4.1. The cores are currently implemented as custom finite-state machines (FSMs) but an hthreads-like system could also be built using general-purpose, programmable processors. Dedicated components currently include:

- Thread Management
- Thread Scheduler
- Synchronization Manager
- Condition Variable Manager
- CPU-Bypass Interrupt Scheduler (CBIS)

Together, these components form a basic set of distributed OS primitives that do not depend on global data structures, nor processor-specific atomic instructions. Additionally, each core presents an ISA-neutral interface that is equally accessible to all types of processors in a heterogeneous system. This is advantageous, as the core OS services of hthreads can function properly in a system with a variety of types of processors, regardless of differences in ISA. The cores communicate with one another using a simple message-passing protocol. Inter-core communication does not require processor intervention, enabling OS processing for interrupt scheduling, and the unblocking of threads to occur in parallel with application execution. This eliminates the need for periodic processor interrupts, thereby reducing overall OS overhead and freeing up additional processor cycles for use by application threads.

4.1 Microkernel Cores

Microkernels break apart the monolithic structure of an operating system into a set of distributed functions, or servers, that can be executed independently. Each function can be written in software and run on a general purpose processor. This eliminates the central
choke point of a monolithic kernel, but does impose new requirements for interprocess communications between the functions. Methods for implementing interprocess communications between the CPUs hosting the different functions must be built upon the physical capabilities of the CPUs. These methods usually revolve around passing data and request information through shared memory and the use of interrupts for invoking processing on a remote CPU. The overhead of these methods can be reduced by moving the functions into a custom hardware solution that is accessed through simple load and store operations.

Thus, each of the hthreads microkernel cores are equipped with a memory-mapped interface that is the target for all service requests and responses. Each service request is packaged in an encoded memory-mapped command in the form of a load or store. Most requests are implemented using loads, as this allows for both the request and return value to be packaged in a single atomic operation. The encoded commands work by embedding service opcodes and arguments within the address and data fields of a load or store operation as shown in Figure 4.2. When an operation is invoked, the corresponding hthreads microkernel core will detect the transaction, and extract any needed data from the encoded fields. The core will then process the request, modifying any internal core state necessary. The core returns results to the requester by acknowledging the request with the return value(s) of the call. Load operations allow the implementation of fully atomic single-read transactions complete with synchronous return results. Store operations allow for more data to be passed from the requester to the core by using both address and data lines for parameter passing, but at the expense of not having a synchronous return value. The cores themselves use these same memory-mapped I/O mechanisms to communicate with one another at runtime. The ability to bundle an entire request in a single operation is a unique feature of hardware-based microkernel cores.

In a traditional software-based microkernel, communication between OS servers is built on top of the notion of IPC. In multi-core systems, IPC mechanisms must be able to asynchronously request services from a different processor. Asynchronous processor notifica-
tions are only possible through the use of interrupts, or more specifically routable inter-
processor interrupts (IPIs). Figure 4.3 outlines the operations required to perform request
services from another processor in a multi-core system. The act of parameter passing and
request invocation are now separate steps that require data to be marshaled to and from
shared memory. Additionally, an asynchronous IPI must be used to notify the destination
processor of the request. This style of invocation mechanism is not only heavyweight (mul-
tiple interconnect transactions + interrupts), but it is also far from a single atomic operation.
The relative heft of the constituent factors of overall execution time for encoded requests is
compared to that of traditional multi-core IPC in Table 4.1.
IPC Operations

- Processor places request parameters in shared memory.
- Asynchronous request to other processor via IPI
- Processor loads request params, performs operation, and places result in shared memory
- Requesting processor detects result via polling or via completion interrupt (ack).

Figure 4.3: IPC Requests: Asynchronous CPU-to-CPU Communication
Table 4.1: Comparing Operations: IPC Amongst Processors Vs. Processor-to-Core IPC

<table>
<thead>
<tr>
<th></th>
<th>CPU-to-CPU</th>
<th></th>
<th>CPU-to-Core</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU&lt;sub&gt;1&lt;/sub&gt; Stores Parameter</td>
<td>10’s</td>
<td>Encode Parameters</td>
<td>10’s</td>
</tr>
<tr>
<td>Inter-CPU Interrupt</td>
<td>1000’s</td>
<td>Load Result</td>
<td>10’s</td>
</tr>
<tr>
<td>CPU&lt;sub&gt;2&lt;/sub&gt; Loads Parameters</td>
<td>10’s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU&lt;sub&gt;2&lt;/sub&gt; Processes</td>
<td>Variable</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU&lt;sub&gt;2&lt;/sub&gt; Stores Result</td>
<td>10’s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU&lt;sub&gt;1&lt;/sub&gt; Poll/Interrupt</td>
<td>10-1000’s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU&lt;sub&gt;1&lt;/sub&gt; Loads Result</td>
<td>10’s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>1000’s</td>
<td>Total</td>
<td>10’s</td>
</tr>
</tbody>
</table>

These simple memory-mapped interfaces provide a uniform interface to OS services that can be invoked by any component capable of communicating over the system interconnect. This enables any type of CPU or hardware accelerator, regardless of differences in ISA and microarchitecture, to directly interact with a common set of run-time services. Invoking microkernel services only involves a few low overhead boolean operations along with a single memory-mapped I/O command. The result is a set of services that are equally accessible to all system components; thereby eliminating the need for a dedicated host processor, and an RPC-centric operating system framework. Additionally, mapping an entire OS request into a single load/store operation reduces the overhead of an OS request to a single I/O transaction – which represents the absolute minimum possible in traditional Von Neumann style CPU architectures.

4.1.1 Thread Manager

The thread management (TM) core serves to create, allocate, and recycle thread identifiers, and acts as a repository for heterogeneous thread information. The TM also serves as the interface to the scheduler as many thread management routines, such as the process of creating a thread, directly result in scheduler invocations. The core provides built-in support for POSIX-compatible create, join, detach, and exit commands as shown in Table 4.2. Internally, the core has two major data structures:
- Thread Status Array: A repository for each thread’s management parameters, and run-time status.

- Thread Identifier Stack: A LIFO data structure used for the allocation and recycling of thread identifiers.

The thread status array is stored in Block RAM and contains status information that can be queried by other OS services at run-time. Information contained in each entry includes joinability, thread identifiers, parent-child relationship information, and exit status. This information is used by the thread manager to perform run-time checks during management requests. Many management operations result in the addition or removal of threads from the ready-to-run queue, so a dedicated interface has been built between the hthreads thread manager and scheduler components as shown in Figure 4.4. As an example, when a new thread is created, it must be added to the ready-to-run queue to make it schedulable. Additionally, the exiting of a thread that has previously been joined on will cause the parent (joining) thread to be added to the queue to make it runnable again. Overall, the thread manager is a vital system component as it allows OS-level scheduling requests to be fielded, recorded, error-checked, and routed without requiring CPU intervention.

4.1.2 Scheduler

An important step in eliminating host-centric RPC mechanisms is to bring all processor resources within the control envelope of an OS scheduler. The original scheduler was modified to maintain CPU and thread type information so that it could properly map threads to heterogeneous hardware resources. All other hthreads cores are type-agnostic and need not know specific processor numbers or types. The scheduler is the only component that must know processor resources as it performs thread dispatch operations across the system’s given computational resources. The scheduler contains an internal partitioned ready-to-run queue structure that provides constant execution time enqueue and dequeue operations, built-in preemption checking, and fast multi-core scheduling decisions for all heterogeneous
Figure 4.4: Block Diagram of Thread Manager and Scheduler Components
<table>
<thead>
<tr>
<th>Operation</th>
<th>Type ($T = 256$)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create_Thread</td>
<td>Read-only,</td>
<td>Creates a new management thread structure for a detached thread and returns the TID of the new thread.</td>
</tr>
<tr>
<td>(Detached)</td>
<td>depth = 1</td>
<td></td>
</tr>
<tr>
<td>Create_Thread</td>
<td>Read-only,</td>
<td>Creates a new management thread structure for a joinable thread and returns the TID of the new thread.</td>
</tr>
<tr>
<td>(Joinable)</td>
<td>depth = 1</td>
<td></td>
</tr>
<tr>
<td>Add_Thread</td>
<td>Read-only,</td>
<td>Adds a thread to the scheduling queue.</td>
</tr>
<tr>
<td></td>
<td>depth = $T$</td>
<td></td>
</tr>
<tr>
<td>Current_Thread</td>
<td>Read-only,</td>
<td>Returns the TID of the thread currently running on the CPU.</td>
</tr>
<tr>
<td></td>
<td>depth = 1</td>
<td></td>
</tr>
<tr>
<td>Next_Thread</td>
<td>Read-only,</td>
<td>Operation that consumes a scheduling decision. If a valid scheduling decision exists, then that TID will be placed in the current_thread_ID register.</td>
</tr>
<tr>
<td></td>
<td>depth = 1</td>
<td></td>
</tr>
<tr>
<td>Clear_Thread</td>
<td>Read-only,</td>
<td>De-allocates a thread management structure for a given thread so that it’s TID can be recycled.</td>
</tr>
<tr>
<td></td>
<td>depth = $T$</td>
<td></td>
</tr>
<tr>
<td>Is_Queue</td>
<td>Read-only,</td>
<td>Scheduler query for a given thread’s queued status.</td>
</tr>
<tr>
<td></td>
<td>depth = $T$</td>
<td></td>
</tr>
<tr>
<td>Yield_Thread</td>
<td>Read-only,</td>
<td>Voluntary yield operation that places the currently running thread back in the ready-to-run queue, makes a new scheduling decision (if possible) so that a different TID can be placed in the current_thread_ID register.</td>
</tr>
<tr>
<td></td>
<td>depth = 1</td>
<td></td>
</tr>
<tr>
<td>Exit_Thread</td>
<td>Read-only,</td>
<td>Notifies the manager that a thread has terminated (exited). If the thread is joinable, then the parent thread that has joined on this thread is added to the scheduling queue.</td>
</tr>
<tr>
<td></td>
<td>depth = $T$</td>
<td></td>
</tr>
<tr>
<td>Join_Thread</td>
<td>Read-only,</td>
<td>Allows a parent thread to block on a child thread until it has exited.</td>
</tr>
<tr>
<td></td>
<td>depth = $T$</td>
<td></td>
</tr>
<tr>
<td>Detach_Thread</td>
<td>Read-only,</td>
<td>Operation that changes a thread’s status from joinable to detached.</td>
</tr>
<tr>
<td></td>
<td>depth = $T$</td>
<td></td>
</tr>
</tbody>
</table>
resources. The scheduler provides an interface to add and remove threads from the ready-to-run queue, calculate scheduling decisions, enable and disable preemption, and the ability to dynamically adjust thread scheduling parameters as shown in Table 4.3. The scheduler contains two major internal components that aid providing constant time ready-to-run queue operations:

- **Ready-To-Run Queue**: Partitioned doubly-linked list for each priority level.

- **Scheduling Decision Function**: A high-speed priority encoder for calculating highest active priority levels.

The ready-to-run queue structure, illustrated in Figure 4.4, is composed of three memory blocks: (1) the priority Block RAM (BRAM), (2) the thread entry BRAM, and (3) the scheduling parameter BRAM. The priority BRAM contains the linked-list information for each priority level such as head and tail pointers. These pointers are thread identifiers used to index into both the thread entry and scheduling parameter BRAMs. The thread entry and scheduling parameter BRAMs contain thread-specific data such as priority, scheduling parameters, and doubly linked-list pointers for navigating within a given priority level’s queue. This data layout allows for constant time access to each priority-level’s head and tail entries, allowing for the construction of efficient enqueue and dequeue operations.

The components used to implement the scheduling decision function consists of a 128-bit register as well as a high-speed priority encoder. Each bit in the 128-bit register represents the presence of an active thread in a given priority level. The priority encoder takes the 128-bit register as input, and generates a 7-bit output that represents the best active priority level in the system. The priority encoder is able to calculate the highest priority level in a deterministic 4 clock cycles, allowing for extremely fast, and jitter free scheduling decisions.

Once the highest priority level is calculated, the scheduler can then read the head pointer out of the priority BRAM and use this value as an index into the thread entry BRAM. The
<table>
<thead>
<tr>
<th>Operation</th>
<th>Type ( (T = 256) )</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enqueue</td>
<td>Dedicated TM Interface</td>
<td>Adds a thread to the R2RQ, updates the next scheduling decision, and performs a preemption check if necessary</td>
</tr>
<tr>
<td>Dequeue</td>
<td>Dedicated TM Interface</td>
<td>Removes a thread from the R2RQ and updates the next scheduling decision</td>
</tr>
<tr>
<td>Is_Queueed</td>
<td>Dedicated TM Interface</td>
<td>Returns a status bit indicating whether or not a thread is currently in the R2RQ</td>
</tr>
<tr>
<td>Is_Empty</td>
<td>Dedicated TM Interface</td>
<td>Returns a status bit indicating whether or not the R2RQ in empty</td>
</tr>
<tr>
<td>Toggle_Preemption</td>
<td>Write-only, depth = 1</td>
<td>Enable/disable the scheduler’s preemption interrupt. The bit will be set to the value of the MSB of the data being written. A '1' represents an enabled interrupt, while a '0' disables the interrupt.</td>
</tr>
<tr>
<td>Set_SchedParam</td>
<td>Write-only, depth = T</td>
<td>Updates a thread’s scheduling parameter. A value between 0 and 127 signifies a priority-level for a SW thread, and a value greater than 127 represents the base address of a HW thread.</td>
</tr>
<tr>
<td>Get_SchedParam</td>
<td>Read-only, depth = T</td>
<td>Returns the 32-bit scheduling parameter for a thread.</td>
</tr>
<tr>
<td>Check_SchedParam</td>
<td>Read-only, depth = T</td>
<td>Performs a safety check on the thread’s scheduling parameter. If a thread is not queued, then the call is guaranteed to pass. However, if the thread is queued, then a check is performed to make sure that thread’s scheduling parameter is in the valid range for SW threads (0 to 127).</td>
</tr>
<tr>
<td>Set_Idle_Thread</td>
<td>Read-only, depth = 1</td>
<td>Sets up the thread ID of the idle thread, and sets the internal idle_thread_valid flag.</td>
</tr>
<tr>
<td>Get_Idle_Thread</td>
<td>Read-only, depth = 1</td>
<td>Returns the TID of the idle thread as well as the internal idle_thread_valid flag.</td>
</tr>
</tbody>
</table>
The identifier of this thread is placed in the scheduler’s decision register, while simultaneously comparing the thread’s priority levels to the priority levels of all currently running threads. The scheduler’s preemption check is able to compare the current state of all running threads to the state of the ready-to-run queue and the next scheduling decision without interrupting a CPU. This completely eliminates the need for speculative preemption interrupts. In fact, this framework will only generate a preemption interrupt when preemption is imminent; i.e. the thread that is scheduled to run next has a higher priority than one of the currently running threads. The preemption check always makes sure to preempt the lowest priority thread of all running threads, so as to prevent threads from thrashing between processors.

4.1.3 Synchronization Manager

The synchronization manager core resolves inter-thread synchronization and signaling operations for threads running on different heterogeneous processors. The core provides POSIX-compliant mutex and condition variable operations as system primitives that are accessible via simple memory-mapped I/O commands. This allows any combination of cores to efficiently synchronize with one another in a heterogeneous system. The synchronization manager contains built-in blocking queues that provide fair and fast lock hand-off without requiring CPU intervention for the processing of waking up threads. All internal state is held in embedded block RAMs, allowing the core to scale with embedded RAM technology increases. The core provides native support for POSIX synchronization including fast, recursive, and error-checking mutex objects as shown in Table 4.4. The synchronization manager communicates indirectly with the scheduler via the thread manager’s add_thread operations. Internally, all synchronization objects are backed by linked-list data structures, as shown in Figure 4.5, that are kept in FIFO order to ensure fair lock hand-off.

Each of the synchronization manager’s operations are based on standard single-beat encoded read commands; which allows for the inclusion of synchronous return values in an atomic fashion. As an example, mutex lock and unlock commands encode the opcode (lock
Figure 4.5: Block Diagram of Synchronization Manager
Table 4.4: Synchronization Manager Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Type ((T = 256, M = 64))</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock</td>
<td>Read-only, depth = MxT</td>
<td>Attempts to lock a mutex for a given thread. Queues thread if mutex has been previously locked.</td>
</tr>
<tr>
<td>Try-lock</td>
<td>Read-only, depth = MxT</td>
<td>Attempts to try-lock a mutex for a given thread.</td>
</tr>
<tr>
<td>Unlock</td>
<td>Read-only, depth = M</td>
<td>Unlocks a mutex. Wakes up next mutex owner if necessary.</td>
</tr>
<tr>
<td>Owner</td>
<td>Read-only, depth = M</td>
<td>Returns the current owner’s TID of a given mutex.</td>
</tr>
<tr>
<td>Kind</td>
<td>Read/Write, depth = M</td>
<td>Returns or updates the kind of a given mutex. Where kind is one of the following: FAST, RECURSIVE, or ERROR.</td>
</tr>
<tr>
<td>Count</td>
<td>Read-only, depth = M</td>
<td>Returns the current lock count of a given mutex. Only valid for RECURSIVE mutexes.</td>
</tr>
</tbody>
</table>

or unlock), the calling thread’s identifier, and the mutex identifier within a single read transaction. When invoked, the synchronization manager’s address decode logic unpacks these values, performs the necessary processing, and returns any results back through the core’s bus interface in a single transaction without the need to use ISA-specific or platform-specific atomic operations. Thus providing an interface for atomic synchronization operations well-suited for use in heterogeneous environments. The use of synchronous single-beat transactions may be concerning at first, however the core’s operations are very efficient, leading to a worst-case latency of 5 clock cycles from operation invocation to acknowledgment. If contention does become a concern, the synchronization manager can be distributed so that multiple managers are instantiated and active in parallel.

Successful lock operations result in the marking of a mutex as owned paired with an update of the mutex owner field. Unsuccessful lock operations can occur when a mutex is already owned by another thread, and result in the addition of the calling thread to the blocked queue for the given mutex. Successful unlock operations perform several internal and inter-core operations including:
• Updating the mutex status to be unlocked if no blocked threads exist.

• Updating the owner of the mutex to the next blocked thread if available.

• Making the next owner of the mutex (if it exists) runnable.

In general, the synchronization core can be thought of as a specialized lock-cache, containing both the lock objects themselves, their blocked queues, as well as the synchronization logic. This results in a highly efficient multiprocessor lock and queuing system that does not depend on processor-specific atomic instructions, or cache coherency protocols; making them well-suited for heterogeneous multi-processor systems. Additionally, interconnect contention during synchronization is essentially halved as only a single load operation is used to synchronize rather than two separate operations as in LL/SC, test-and-set, and fetch-and-add.

Breaking the dependence of synchronization mechanisms on both processor-specific coherency protocols and atomic operations allows any processor to utilize the synchronization primitives provided by hthreads; regardless of heterogeneity levels. This is very important as many families of processors do not have compatible atomic operations [109]. Additionally, processors such as the MicroBlaze do not have any low-level atomic synchronization operations. Compatibility of cache coherency protocols is also problematic in heterogeneous systems, and requires non-trivial amounts of hardware support to allow processors with different types of native coherency protocols to interact with one another [115].

The synchronization logic natively supports recursive, fast, and error-checking mutexes with built-in support for hardware queuing. The logic is efficient in terms of both speed and area; performing all operations in under 5 clock cycles, and requiring less than 2% of the resources of a mid-sized FPGA device. The hardware-based queues enable rapid lock hand-off not easily achieved in software-based queuing systems [16]. Currently all mutexes are associated with a single, centralized IP core, however hthreads could be extended to support the use of multiple parallel synchronization cores. This allows for different sets of locks to be distributed; easing contention for un-related locking operations.
Table 4.5: Condition Variable Manager Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Type ((T = 256, C = 64))</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wait</td>
<td>Read-only, depth = CxT</td>
<td>Waits on a condition variable. Blocking the calling thread.</td>
</tr>
<tr>
<td>Signal</td>
<td>Read-only, depth = C</td>
<td>Signals a single waiting thread. Wakes up the next blocked thread on the condition variable queue if one exists.</td>
</tr>
<tr>
<td>Broadcast</td>
<td>Read-only, depth = C</td>
<td>Signals all waiting threads. Wakes up all threads blocked on the condition variable queue.</td>
</tr>
</tbody>
</table>

### 4.1.4 Condition Variable Manager

The condition variable manager is responsible for managing the internal state of POSIX-compliant condition variables. The core provides native condition variable wait, signal, and broadcast mechanisms that communicate directly with the hthreads thread management and scheduling subsystems. This allows for fast end-to-end signal/wait communication amongst threads, even in mixed-ISA heterogeneous systems. The core contains BRAM-based wait queues for each condition variable, and is structured in such a way that allows for the distribution of condition variable managers in order to partition heavily used condition variables from one another. The condition variable manager communicates indirectly with the scheduler via the thread manager’s `add_thread` operations.

Internally, the condition variable core manages a singly-linked wait list for each of the available condition variable objects as shown in Figure 4.6. Each entry on the list represents a thread waiting on a given condition variable object. Entries are added to the wait-list via `signal` (enqueue) operations. Conversely, entries are removed from the list in FIFO order using either `signal` (single dequeue) or `broadcast` (dequeue all) operations. The condition variable manager’s key operations are summarized in Table 4.5. As threads are removed from the wait-list, they become schedulable again by sending `add_thread` requests to the thread manager. The thread manager then makes each thread runnable by adding them to the scheduler’s ready-to-run queue through the dedicated interface connecting the thread
Figure 4.6: Block Diagram of Condition Variable Manager
Table 4.6: CPU-Bypass Interrupt Scheduler (CBIS) Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Type ( T = 256, I = 2^N )</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Associate</td>
<td>Read-only, depth = ( I \times T )</td>
<td>Associates a thread with a given interrupt line. Blocks the thread until an interrupt event occurs on the line.</td>
</tr>
<tr>
<td>Clear Interrupt</td>
<td>Read-only, depth = ( I )</td>
<td>Clear the pending interrupt flag for a specific interrupt.</td>
</tr>
<tr>
<td>Read Interrupt</td>
<td>Read-only, depth = ( I )</td>
<td>Read the pending status of a specific interrupt.</td>
</tr>
</tbody>
</table>

manager to the scheduler.

4.1.5 CPU-Bypass Interrupt Scheduler (CBIS)

The CPU-Bypass Interrupt Scheduler (CBIS) core is responsible for the conversion of asynchronous interrupt events into thread scheduling invocations. This has two major positive benefits: (1) it brings the process of defining and working with interrupt handlers into the higher-level programming model, and (2) it brings interrupts into the domain of the operating system, allowing them to be scheduled and prioritized alongside other application threads.

The effect of asynchronous interrupts with priority levels of \( \infty \) can wreak havoc on the determinicity of application programs. Currently, the only way to prevent interrupt events from preempting user-level tasks is to disable interrupts all together. This is quite dangerous to do, as this can prevent any future interrupts from being serviced for long periods of time. If interrupts are left enabled, repetitive interrupt events can cause a program to be preempted over and over again. While the program will still be able to make progress, the interrupt events destroy the timeliness characteristics of the task. This is especially concerning for real-time and embedded systems, where determinicity and timeliness are first-class concerns.

All CBIS operations are shown in Table 4.6. The CBIS contains interrupt capture logic as well as a Block RAM based association table to store thread-interrupt relationships as shown in Figure 4.7. Upon an interrupt event, the CBIS will scan through the association
Figure 4.7: Block Diagram of CBIS
BRAM, and if a match is found, it will wake the thread by sending an `add_thread` message to the thread manager via its master communication port.

The CBIS acts as a translator; transforming asynchronous interrupt events into asynchronous scheduling events that can be fielded by the hthreads scheduler core. The ability to schedule interrupt events alongside application threads in parallel with application execution eliminates the need to speculatively interrupt CPUs in order to perform top-half processing. Additionally, it brings bottom-half interrupt handlers into the control of application-level programmers, as interrupt handlers can now be defined as traditional threads that merely associate with interrupt events using the new CBIS APIs, such as `hthread_intrassoc`.

### 4.2 Tying It All Together

Each microkernel core has been built to support a specific class of OS-level tasks. The cores work and communicate in unison in order to provide a cohesive set of OS services. However, it is unreasonable to expect programmers to understand the interfaces presented by each microkernel core. Each core has been encapsulated using a driver-level interface that provides easy-to-use software functions that can be used at both the kernel and application levels. The driver-level functions alleviate the need to know the specifics of each core’s memory-mapped register offsets, address/data encoding, and low-level functional behaviors. Instead, programmers can use the driver-level functions to construct higher-level services – using the cores as composable system-level primitives.
Chapter 5

Heterogeneous Coordination Layer

In order to coordinate service requests originating from heterogeneous processors with different ISAs, one must create an abstraction layer common to each processor. This abstraction layer must be uniform in order to enable transparent multithreaded programming from the perspective of the user. Additionally, it must encapsulate and hide the low-level details of the underlying hardware platform so as to not burden the programmer with the intricacies of heterogeneous multi-core systems. The coordination layer will field requests from all processors in a system, directing their requests to a common run-time system provided by the hthreads microkernel cores. The existence of uniform APIs allows programmers to write standard multithreaded programs for heterogeneous multi-core systems while retaining portability.

Two new system components have been created in order to maintain cohesive OS support for heterogeneous processors within hthreads: (1) the hardware abstraction layer (HAL), a thin software wrapper used to abstract the low-level details of the hthreads system, and (2) the virtual hardware thread interface (V-HWTI), a hardware/software co-designed interface to support interaction and communication between the OS and heterogeneous processors. The software wrapper is quite simple, transforming each high-level API call into a lower-level memory-mapped I/O command to interact with the hthreads system. Together, the HAL and V-HWTI fully encapsulate the details of the hthreads system; allowing the uniform use of system calls within a heterogeneous system. The V-HWTI is an hthreads-specific interface as it directly communicates with the OS IP cores. On the other hand, the HAL takes platform-independent APIs, and translates them into V-HWTI commands, just as a traditional OS translates system calls into trap instructions to invoke the kernel. Together, the HAL and the V-HWTI components enable the extension of hthreads oper-
Table 5.1: Virtual Hardware Thread Interface (V-HWTI) Memory-Map

<table>
<thead>
<tr>
<th>Byte Offset</th>
<th>Field</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>TID</td>
<td>Thread Identifier</td>
</tr>
<tr>
<td>0x04</td>
<td>Utilized</td>
<td>CPU Utilized Flag</td>
</tr>
<tr>
<td>0x08</td>
<td>STA</td>
<td>Thread Status</td>
</tr>
<tr>
<td>0x0C</td>
<td>CMD</td>
<td>Thread Command</td>
</tr>
<tr>
<td>0x10</td>
<td>ARG</td>
<td>Thread Argument</td>
</tr>
<tr>
<td>0x14</td>
<td>RES</td>
<td>Thread Result</td>
</tr>
<tr>
<td>0x18</td>
<td>FCN</td>
<td>Thread Start Function</td>
</tr>
<tr>
<td>0x1C</td>
<td>Unused</td>
<td>Reserved Field</td>
</tr>
<tr>
<td>0x20</td>
<td>Stack Base</td>
<td>Base of Kernel Stacks</td>
</tr>
<tr>
<td>0x24</td>
<td>TCB Base</td>
<td>Base of TCB Structures</td>
</tr>
<tr>
<td>0x28</td>
<td>Context Base</td>
<td>Base of Thread Contexts</td>
</tr>
<tr>
<td>0x2C</td>
<td>Bootstrap Ptr</td>
<td>Base of Bootstrap Function</td>
</tr>
</tbody>
</table>

Operating system services across heterogeneous processor boundaries. They replace custom fixed-logic hardware interface components with flexible and extensible software interfaces capable of hiding the irregularities of systems composed of heterogeneous processors with different ISAs.

5.1 Thread Interfaces

Within hthreads, a hardware thread is defined as an independent, uninterruptible computation that has pre-allocated, reserved resources on which to execute [10]. Each hardware thread is provided a hardware thread interface (HWTI), a memory-mapped IP core which serves as the thread’s system-level interface to the hthreads OS cores [14, 15]. The HWTI provides a set of hardware-level APIs to hardware threads in the form of a simplified command interface. The system-facing side of the HWTI provides a slave interface that allows the hthreads OS cores to communicate with each hardware thread. The original definition of a hardware thread is easily extended to serve as a more generalized heterogeneous thread using small software wrappers around the threads interface. Just as hardware threads have dedicated FPGA resources, heterogeneous threads are bound to a single heterogeneous core for the duration of the thread’s lifetime, similar to a codelet [20].
The virtual HWTI (V-HWTI), a replacement for the state machines and hard-coded interface of the HWTI, was defined in order to support generalized heterogeneous processors within the hthreads OS framework. The memory-map of the V-HWTI is shown in Table 5.1. The V-HWTI serves as the interface between a processor and the hthreads OS. All OS-level responses to system call requests are stored in a small section of RAM used by V-HWTI; essentially making the V-HWTI a system-level, software-managed scratchpad memory. The FPGA architecture allows for the scratchpad memories to be implemented using a bus-attached Block RAM, composed of embedded SRAM cells that have 2 clock cycle read access, and 1 clock cycle write access. The implementation is area efficient, and provides the OS with an expandable interface that is able to easily scale with technology advances. As a comparison, an 8 kilobyte V-HWTI constructed of embedded Block RAMs only requires 163 FPGA slices and 2 Block RAMs, whereas the HWTI requires 1,500+ slices [14]. Additionally, a single V-HWTI memory space is able to support many heterogeneous threads, while a traditional HWTI is only able to support a single hardware thread.

The V-HWTI defines the interface between processors and the hthreads operating system. It is analogous to the way a thread control block (TCB) or process control block (PCB) in a traditional operating system includes all of the state necessary to fully describe a thread or process. Instead, the V-HWTI defines all of the necessary state to start or resume the execution of a thread across heterogeneous boundaries. POSIX-compliant threads are defined as an independent stream of computation that take a single argument, a pointer to a typeless memory location, or void* pointer. Therefore, the V-HWTI defines both an argument field and a function pointer field that will contain architecture-specific values for the thread being invoked. The target processor can then unpack these entries from the V-HWTI, using them to bootstrap the execution of the thread using the target processor’s application binary interface (ABI). Additionally, the V-HWTI fields can be extended for publishing dynamic information that can be used by the system at run-time for scheduling, profiling, and di-
agnostics. This can enable the production of advanced run-time service discovery features including dynamic processor changes, and application-specific scheduling and management functions.

The process of creating a heterogeneous thread populates the V-HWTI with all of the information required for it to begin execution on a non-native CPU. The hthreads OS kernel supplies the V-HWTI with the identifier of the newly created thread, it’s thread argument, as well as the thread’s entry point or function pointer. Additionally, the OS may supply the V-HWTI with information concerning the location of important kernel-level data structures including the addresses of the global thread-context block (TCB) array, the architecture-specific CPU context array, the global array of thread stacks, and the location of any bootstrap functions required to aid in the creation of threads for other architectures. A detailed description of each V-HWTI field can be found below:

- **TID - Thread Identifier**
  
  - Contains the thread identifier of the thread utilizing this V-HWTI.
  
  - Identifier value is setup by hthreads kernel at thread creation time.

- **Utilized - Utilization Flag**
  
  - Contains a binary (Free/Utilized) flag that indicates to the system whether or not the V-HWTI is currently in use.
  
  - Can be extended to hold non-binary values in the future in order to indicate weighting.
  
  - Flag is set upon thread creation (by kernel) and cleared upon thread exit (HAL bootloop).
  
  - Protocol:

    - Free = 0x0000.
    - Used = 0x0001.
• **Status** - Thread Status
  
  – Current status of running thread.
  
  – Legacy register used by original HWTI definition. Not used in heterogeneous system.

• **Command** - Thread Command
  
  – External command request field.
  
  – Use by HAL for bootstrapping and unblocking threads.
  
  – Protocol:
    * NoOp = 0x0000 (No command).
    * Go = 0x0001 (Go, unblock).
    * Reset = 0x0002 (Reset HAL).

• **Argument** - Thread Argument
  
  – Thread input argument.
  
  – Contains a *(void*) pointer to a thread argument.
  
  – Setup by hthreads kernel at thread creation time.

• **Result** - Thread Result
  
  – Return value for a terminated thread.
  
  – Automatically setup by HAL upon thread exit.

• **Function** - Thread Function Pointer
  
  – Thread start function pointer.
  
  – Utilized by the HAL as the entry point for a newly created thread.
  
  – Setup by hthreads kernel at thread creation time.
- **Stack Base** - Kernel Stack Base
  - Pointer to global kernel stack structure.
  - Utilized by HAL when accessing thread execution stack entries held within the kernel.
  - Setup by hthreads kernel at thread creation time.

- **TCB Base** - Kernel TCB Base
  - Pointer to global kernel TCB structure.
  - Utilized by HAL when accessing architecture-independent TCB entries held within the kernel.
  - Setup by hthreads kernel at thread creation time.

- **Context Base** - Kernel Context Base
  - Pointer to global kernel thread context structure.
  - Utilized by HAL when accessing architecture-dependent thread context entries held within the kernel.
  - Setup by hthreads kernel at thread creation time.

- **Bootstrap Pointer** - Bootstrap Function Pointer
  - An architecture-specific function pointer used for bootstrapping threads.
  - A function pointer utilized by the HAL when wrapping fresh thread contexts during the process of thread creation.
  - Setup by hthreads kernel at thread creation time.

### 5.2 Hardware Abstraction Layer

The hardware abstraction layer (HAL) consists of two major components: V-HWTI interactions, and system call handling. The V-HWTI interactions component contains calls to
while(1)
{
    // Busy wait until targeted
    wait_for_go_command();

    // Grab thread argument and function pointer
    arg = get_vhwti_arg();
    func = get_vhwti_func_ptr();

    // Invoke thread
    bootstrap_thread(func, arg);
}

Figure 5.1: Pseudocode for HAL Bootloop Process

initialize the V-HWTI data structures and for blocking/unblocking threads. The memory-map of the V-HWTI is shown in Table 5.1. When a heterogeneous processor is idle, the HAL acts as a small kernel that waits for a thread to be created on the heterogeneous processor. The pseudocode of the HAL bootloop can be seen in Figure 5.1. When a thread is created and destined to run on a heterogeneous processor, the scheduler issues a signal command to the processor’s V-HWTI. The HAL then unpacks the thread argument and function pointer from the V-HWTI, and bootstraps the thread for execution. The thread then begins executing on the target processor. Once a thread finishes execution, the bootloop readies itself for the next resulting invocation.

The HAL’s linkable high-level APIs allow heterogeneous threads to directly make system calls without the use of an RPC mechanism. The HAL is written entirely in C and is quite portable. Each high-level API is backed by code that first converts a system call into an encoded, memory-mapped I/O command. The encoded command is then invoked, and its return value interpreted. Non-blocking system calls translate return values into user-level return values before returning control back to the calling thread. Blocking system calls interpret return values and make a decision to either continue running or to block the calling thread. If a thread should block, then it does so by waiting for a signal to enter the V-HWTI. Upon signaling, the thread unblocks and continues execution on the heterogeneous processor. An example protocol stack traversal for a system call request from a heterogeneous processor can be seen in Figure 5.2. The process begins with the invoca-
Figure 5.2: Protocol Stack for a Heterogeneous System Call: Translating User-Level APIs to HW/SW Interactions
tion of a user-level API synchronization API: `hthread_mutex_lock`. The user-level API acts as a wrapper around the system call handler, a centralized function that decodes, dispatches, and interprets the results for each system call. The system call handler decodes the system call, and dispatches the specialized handling function, which in this case is the `_syscall_mutex_lock` function. The specialized handler function implements the call’s functionality, and in this case performs all of the necessary interactions with the hthreads hardware platform. The handler performs two calls that interact with lower-level hardware features: (1) it invokes the `_current_thread` function, and (2) it invokes `_mutex_acquire`.

The `_current_thread` function returns the thread identifier of the calling thread. For heterogeneous threads, the thread identifier (TID) is located at offset 0x0 within the V-HWTI of the active processor. The TID of the calling thread is utilized by the `_mutex_acquire` function, as all mutex operations must be associated with a calling thread for identification for thread blocking and mutex ownership. The `_mutex_acquire` call forms an encoded command that contains the base address of the hthreads synchronization manager, the mutex lock opcode, the identifier of the mutex, and the identifier of the calling thread using the `_mutex_cmd` function. It then uses this encoded command by invoking the `read_reg` function, a function that performs a single-beat read (one MMIO transaction) that interacts with the internal state machine of the hthreads synchronization manager. The state machine will acknowledge this read with the status of the mutex lock operation, and return the result to software. The subsequent code in the `_mutex_acquire` function can block the calling thread if the status indicates that the mutex was already owned by another thread, thus returning control to other ready-to-run threads by invoking the scheduler through the `_run_sched` function. However, if the status indicates that the mutex is now owned by the calling thread then control is simply returned back to the caller.

The current version of the HAL is a linkable library, and is therefore directly callable from application-level code. The use of a trap mechanism to invoke system calls can be used, but would require the use of architecture-specific code to implement the trapping
mechanisms. In turn this would reduce the portability of the HAL itself, as system call invocation mechanisms invariably require the use of ISA-specific assembly code. The use of a library-style HAL allows for the code to remain highly portable between CPU architectures and it eliminates the overhead associated with trap-and-emulate style system call structures. The software layers that comprise the HAL serve to encapsulate the details of the hthreads microkernel cores in a form suitable for traditional software programmers. The system call handler and helper functions can be extended to provide higher-level services that do not depend on the existence of hardware-based OS cores; thus allowing the hthreads OS framework to be extended in the same way that a traditional software-only OS is built. The HAL is compiled and linked with each heterogeneous thread in order to produce executables capable of invoking operating system services. Processors with distinct ISAs have distinct HALs, however each is compiled from a common set of source, modulo architecture-specific changes including how to retrieve unique processor identifiers. The HAL can be considered hardware-dependent software (HdS) as different binary versions must exist for each type of processor in a heterogeneous multi-core platform [72, 43]. Each HAL serves as a lightweight (shallow) protocol stack, implementing the necessary processor-specific functionality for invoking the common system services provided by the hthreads microkernel cores.

5.3 HAL + Cores = Heterogeneous OS

In traditional monolithic systems, the operating system can be though of as a deep, vertically-organized software stack, containing many layered levels of software between the application and hardware [91]. In homogeneous multi-core systems, portions of the operating system communicate with one another transparently as the entire software stack from top to bottom is shared and perfectly ISA compatible. The presence of multiple ISAs in heterogeneous multi-core systems presents problems when trying to integrate distinct software stacks. Processors with different ISAs cannot share the same kernel code, and sharing ker-
nel data structures requires synchronization in order to prevent races and inconsistencies. Differences in ISA can prevent processors from using compatible sets of low-level atomic operations, including those discussed in Section 2.1.2. This results in scenarios where processors cannot synchronize with one another, and therefore cannot natively share kernel data structures. In general, several benefits can be had by migrating portions of the traditional software stack into hardware:

- Hardware-based OS services accessible through memory-mapped I/O
  - Services can be parallelized and streamlined, leading to lower OS overhead.
  - OS processing occurs in parallel with application execution – no need for periodic interrupts.
  - ISA-neutral interfaces allow heterogeneous processors to directly share OS services and data structures.
– OS processing packaged in a single MMIO (load/store) operation – equivalent to a single instruction fetch.

• Shrinking of the software stack.
  – Reduction in the amount of code that needs to be ported to all processors.
  – Less layer traversal, more efficient OS framework.

Migration of core OS functionality into hardware not only shrinks the size of the traditional software stack, but it also provides a common set of services that can be directly used by a variety of heterogeneous processing units through memory-mapped I/O. Each processor’s HAL functions as a thin software library; providing functional APIs that can be used to invoke OS services. The HAL transforms these APIs into a uniform set of memory-mapped I/O commands; invoking the common run-time services provided by the hthreads OS IP cores as shown in Figure 5.3. The hardware-based services are ISA-neutral, making them a perfect fit for heterogeneous systems. Direct access to OS services completely circumvents the need to provide RPC access to heterogeneous slave processors. Additionally, the hardware-based services have been parallelized and streamlined, which has resulted in a set of horizontally-integrated services that are very fast and efficient – leading to an OS for heterogeneous multi-core systems which has both low overhead and low jitter.
Chapter 6

Design Flow

Heterogeneous systems require modifications to be made to existing design flows. One major challenge is to create a design flow that allows a user to work from a single application with portions of the code destined for execution on processors with different ISAs. This is similar to the goal of single system image (SSI) efforts for distributed shared memory (DSM) systems. Currently, heterogeneous systems require the definition of separate, or at least partitioned, applications for each type of processing element. This change can be circumvented through the use of an interpreter or virtual machine environment, however this can lead to execution inefficiencies due to interpreter overhead [49]. To avoid this overhead, a specialized compilation technique similar to those used by IBM’s Cell and Intel’s EXOCHI [121, 47, 28] was used to embed heterogeneous binaries into a single executable image. This approach has many benefits, one of which, allows a developer to look at an application as a single, coherent program rather than a set of several programs that contain implicit interactions. Additionally, the technique leverages native code execution without the need for complex just-in-time compilation frameworks.

6.1 Related Work

Heterogeneous computing platforms require specialized compilation processes designed to target the individual instruction sets of each processor type. Traditional homogeneous compilation processes produce a traditional executable file; a binary representation of the instruction stream for a specific type of processor. Each instruction has an ISA-specific encoding that perfectly matches a decode stage in a given processor, allowing a single compiler to hide ISA-level intricacies from the programmer. In heterogeneous systems, compilation flows must produce fat binary executables that contain multiple instruction streams, each
of which is encoded in a specific ISA [73]. Additionally, the distinct instruction streams
must be linked so that references to the same data object refer to the same address in mem-
ory. Unfortunately, most executable formats do not specify how to resolve references for
common data and code objects between different processor architectures [67]. This often
bars the use of standard compilation and linking utilities, as they are unable to work across
different architectural spaces [66].

To word around this limitation, many heterogeneous design flows have created aug-
mented compilation tools that are able to handle cross-linking heterogeneous binaries [66,
121, 73, 114, 96]. However, modifying modern compilers and linking utilities is an ex-
ceedingly difficult process; as compilation frameworks are extremely large and complex.
Additionally, maintenance and extension of heterogeneous systems not only implies, but
requires the modification and maintenance of multiple compilers and binary utilities. As
an example, many projects have created customized extensions to standard executable-
linkable file formats (ELFs) to allow for cross-architecture linking and embedding of bi-
nary files [66, 73, 121]. These extensions require the modification of compilers to produce
ELF files that conform to the new extensions. The extensions defined by both IBM’s CE-
SOF [66, 67] and Joglar’s HELF [73] aim to create standardized executable formats that
enable heterogeneous compilation flows while simultaneously preserving backwards com-
patibility. Other heterogeneous compilation flows have strived to leave compilation tools
as is, by creating new helper utilities that are able to link and embed heterogeneous bina-
ries by using intermediate files that are encoded in high-level languages. As an example,
both NVIDIA’s heterogeneous compilation drivers (nvcc) [96] and Arachne’s preprocess-
ing compiler [42] produce intermediate files that are encoded in traditional library header
files that can be readily linked and embedded using normal compilation tools. This allevi-
ates the need to modify low-level compilation, linking, and binary utilities – thus making it
easier to extend and port heterogeneous system software infrastructure. This is especially
advantageous, as heterogeneous platforms are being created at increasingly faster rates, and
incorporating processors with different and constantly evolving ISAs.

6.2 Multiple, Architecture-Specific Executables

The heterogeneous design flow allows developers to pass multithreaded program code, and library support functions through a compiler for a target architecture in order to create a binary executable suitable for use in a heterogeneous, multi-ISA, environment. This process can be performed manually by the programmer, or can be automated using a specialized compiler and build system [28]. Within hthreads, all heterogeneous threads are defined in separate files so that they can be easily compiled for a variety of target architectures. All executables are then embedded into a single heterogeneous executable using command line tools. Programmers are also free to use cross-platform compilation techniques through the use of directives such as #ifdefs. This enables programmers to exercise finer-grained control throughout the process of heterogeneous compilation.

6.3 Embedding Executables

The embedding process takes an executable file, or ELF (executable and linkable format), of one architecture and embeds it into the ELF of a different architecture [47]. The heterogeneous executable is flattened to preserve the memory layout of instructions and data. This is required as sections in an ELF file may not be contiguous. As an example, consider the a portion of the ELF file shown in Figure 6.1. The two sections, _start and _vector_sw_exception, are non-contiguous as one starts at address 0 and the other at address 8. Proper program execution requires that these instructions are separated by one word (8 bytes), however if the ELF file is embedded before begin flattened then the instructions will be mistakenly placed next to one another as shown in Figure 6.2. The flattening process transforms the ELF into a loadable binary form by padding all non-contiguous sections with zeros. This ensures correct instruction and data placement without the need for a dynamic code loader. The correct placement of the instructions of the ELF in Figure 6.1 are shown
Next, symbol table information, including the addresses of thread start functions must be extracted as the standard GNU compilers used do not have heterogeneous linking capabilities. The extraction process makes use of standard binary utilities (binutils [54]) including GNU's nm, objcopy, and objdump. Finally, threads are not first-class objects in C, as thread start functions are represented as function pointers [42, 24]. Therefore each embedded ELF must contain a set of translated function pointers, called thread handles, that correspond to the heterogeneous versions of the embedded threads. These handles correspond to offsets into each compiled binary, allowing a programmer to reference a function in an architecture-specific manner.

A linkable C-header file is produced after flattening and symbol extraction. The file contains a pre-initialized array to hold the binary version of the flattened ELF. All thread handles are added to the file, where each thread handle is a pointer into the embedded ELF. A program can link against this C-header file and use the thread handles to create heterogeneous threads from within a single application. The flow works by embedding heterogeneous binaries into the data section of a host executable, where the host executable targets the processor(s) on which the application will begin execution. The symbol table
Figure 6.4: Symbol Table Extraction Process and Generation of Thread Handles
The extraction phase is illustrated in Figure 6.4, whereas the entire heterogeneous compilation and embedding process is shown in Figure 6.5. The flow is unique in that it does not require the modification of the individual compilers used, and currently only depends on the ability of each compiler to generate position-independent code (PIC). The embedding process depends on the generation of position-independent code as each embedded executable is relocated within the global memory space when included in the final heterogeneous executable. Support for more advanced linking technologies, including the use of system-wide global variables, may require additional internal compiler and linker modifications [66].

### 6.4 Heterogeneous Compilation Utilities

A set of new utilities have been created in order to make the heterogeneous compilation process intuitive for end-user programmers. The new utilities enable the construction of automated or semi-automated build systems for heterogeneous systems. The process of
creating heterogeneous embedded header files has been encapsulated within a new tool that can be added to traditional software build systems. The tool transforms an ELF file from a given CPU architecture into an embeddable header file containing both the embedded binary, and any desired thread function handles. The resulting header files can be linked in, and compiled with traditional compilation tools, and the embedded handles can be used to spawn heterogeneous threads at run-time.

The tools are written to be portable amongst different platforms and CPU architectures. Currently, the tool is implemented in Python [102] and contains hooks for altering the CPU-specific binary utilities that are invoked. The hooks include definitions for:

- Input Architecture Type
- Embedded Object Names
- Object Copy Tools (GNU objcopy)
- Symbol Table Tools (GNU nm)
- Code Formatting and Hex Dump Tools (xxd)

Several different versions of the embedding tools have been implemented. The basic version requires the user to provide the names of the desired functions to create thread handles for. This allows for full programmer control over the naming conventions used when generating thread handles as well as which functions are actually translated into thread handles. The more advanced version attempts to automatically create a set of thread handles on behalf of the user. Currently, each symbol table entry in the text section of the heterogeneous binary that does not begin with a leading underscore is transformed into a thread handle. This technique tends to generate more thread handles than there are actual threads, as it defines handles for standard user-defined functions. Advanced thread selection functions have been created that allow a programmer to specify search keys that can be used to detect thread functions. Compilation passes that make use of function signature information can also be constructed, but this would require source analysis tools, lexers, and
parsers. Additionally, modifications to the compiler could enable the use of directives or pragmas, that could be used to signify which functions are actually user-level threads [42].
Chapter 7

Thesis Evaluation

The thesis of this work is a hardware-based microkernel operating system can provide both efficient and uniform services in heterogeneous multi-core environments, in ways that are not readily achievable in software-only operating systems. The evaluation of the thesis statement was broken into three broad sets of experiments. The first focused on efficiency, the second uniformity, and the third how a hardware microkernel can restore the operating system to serve as a unified virtual machine. Unique tests were created for each set of experiments, each attempting to provide a focused evaluation of the objective of the experiment. Each series of tests sought fair and unbiased comparisons between the proposed hardware microkernel approach and a standard monolithic kernel. Obtaining fair comparisons required implementing both the hardware microkernel and a representative monolithic kernel on the same hardware platform. By using the same compiler and development software, all biases have been removed that can result from different clock frequencies, architecture configurations, and compiler optimizations that occur when using different compilers. The experimental platform that hosted the monolithic kernel was implemented using a modified kernel running on a master node that allowed access from slave nodes through a remote procedure call (RPC) mechanism. The test platform is further detailed in Section 7.2, and the development environment used to target it is described in Chapter 6.

7.1 Evaluation Criteria

7.1.1 Defining Efficiency

The efficiency experiments evaluated system call latency (overhead) and variability (jitter) of operating system calls without bias from the application code [8, 9]. Synthetic benchmarks were created for these evaluations. Additional tests were run to observe how effi-
ciency affects application scaling as the number of processors and threads increased. The results from the efficiency comparisons are presented in Section 7.3.

7.1.2 Defining Uniformity

The uniformity experiments evaluated if a single uniform set of services could be provided across heterogeneous CPUs. The first set of tests contained individual APIs that accessed operating system services from different heterogeneous CPUs. These tests were functional and exercised all available hthreads operating system APIs. A second set used the APIs in mini-benchmark applications to exercise the typical parallel programming patterns of fork/join, producer/consumer, pipeline, and irregular parallelism [8, 5, 7, 4, 6]. These two test sets allowed us to observe if the hardware-based microkernel could provide uniform access to OS services in a heterogeneous platform featuring processors with different ISAs. These tests also demonstrated the ability of a hardware-based microkernel to support a uniform multithreaded parallel programming model. These results from the uniformity tests are presented in Section 7.4.

7.2 Evaluation Platform Description

7.2.1 Common Hardware Platform

All experimental results are actual run-time information from program execution on a Xilinx ML507 development board [8, 9]. The ML507 contains a Virtex-5 FXT70 FPGA that contained an embedded, hard core IBM PowerPC440 CPU, sea of reconfigurable logic gates, embedded Block RAMs, and other diffused IP. The Virtex-5 FXT70 FPGA consists of 11,200 logic slices, and approximately 71,680 logic cells [129]. The FPGA architecture is defined in terms of configurable logic blocks (CLBs), where each CLB consists of 2 slices. An FPGA slice itself consists of four 6-input lookup tables (LUTs), and four 1-bit flip-flops [129].

The heterogeneous system created for the following tests consists of different configu-
rations of MicroBlaze [126] and PowerPC processors as well as the specialized hardware cores that make up the hthreads run-time system. Each system contains five specialized hardware OS cores (scheduler, thread manager, synchronization manager, condition variable manager, and CPU-bypass Interrupt Scheduler), as well as one PowerPC CPU, and one to six MicroBlaze soft-core CPUs in various configurations as shown in Figure 7.1. The MicroBlaze is unique in that it is a configurable processor that features an extensible instruction set. This configurability allows for the ISA of the MicroBlaze to be augmented by the designer in order to vary the overall heterogeneity level of the system.

The experimental platform also contained the Xilinx Multi-Port Memory Controller (MPMC), allowing a large shared memory that was equally accessible to all system components [127]. The MPMC contains a total of eight configurable memory ports. The PowerPC processor requires a single MPMC port, while each MicroBlaze processor is able to utilize two MPMC ports; one port each for instruction (IXCL) and data (DXCL) access [126]. Cache coherency is not supported by the Xilinx implementations of the MicroBlaze and PowerPC, so all data caches are disabled to guarantee the coherency of shared memory accesses [19, 128, 126]. Without data cache access, the MicroBlaze can be configured to use only a single MPMC port (for I-cache access) while all data accesses can be multiplexed.
through a single PLB-based MPMC connection. This allows up to 7 processors to be connected to the MPMC. This limitation is solely a result of the MPMC architecture [127], and is not a limitation of the hthreads run-time system nor the heterogeneous design flow. Larger systems composed of more than 7 processors were constructed by multiplexing both instruction and data accesses on MPMC memory ports. However, on-chip testing revealed that this architecture caused too much contention for shared memory resources, leading to system lock up even with fair, round-robin bus-arbitration protocols in place. Thus results are only presented for systems containing up to 6 MicroBlaze and one PowerPC processor.

7.2.2 Remote Procedure Call Configuration

The common hardware platform served as the framework for the hardware microkernel and the RPC model. Software was created to configure the operating system with a generic RPC invocation mechanism that performed the following steps:

1. Thread on Slave CPU places request info in memory.

2. Slave thread signals condition variable.

3. Delegate thread on Master awakens, reads memory.

4. Delegate issues OS request on Master.

5. Delegate places result in memory.

6. Delegate thread signals with condition variable.

These steps are graphically shown in Figure 7.2 for better understanding. The first step, **RPC-SETUP** shown in Figure 7.2(a), sets up the RPC-specific data structure with the parameters associated with a request. The second step, **RPC-SIGNAL** shown in Figure 7.2(b), signals the host processor via a standard condition variable. The third step, **RPC-AWAKE** which is implicit in the figure, occurs when the operating system context switches to the desired RPC delegate thread. The delay between **RPC-SIGNAL** and **RPC-AWAKE** includes time
Figure 7.2: Processing Steps for RPC Mechanism

(a) Setting up RPC argument

(b) Signaling Master

(c) Retrieve Arguments, Invoke OS

(d) OS Returns, Marshal Results

(e) Signaling Slave

(f) Retrieve Result, Resume Execution
for unblocking a waiting thread, running the scheduler to pick this thread to run, and a context switch to this thread. The URPC mechanism was implemented to guarantee the RPC delegate thread woke up as soon as possible to eliminate biasing in the scheduler. This was achieved by assigning the delegate threads highest priority of all threads in the system. In the fourth step, each RPC delegate thread decoded the requested service in the RPC-specific data structure as shown in Figure 7.2(c). Once decoded, the fifth step, or RPC-EXECUTE, was undertaken by the delegate thread. The delegate thread invoked the system call on behalf of the requester and gathered the return values produced by the system call as shown in Figure 7.2(c). The sixth step, RPC-RETURN shown in Figure 7.2(d), populated the RPC-specific data structure with any and all system call return values. The seventh step, RPC-ACK shown in Figure 7.2(e), used the RPC condition variable to signal the requesting thread the system call was completed. The slave thread, as shown in Figure 7.2(f), then retrieved the result from the RPC data structure and resumed execution.

### 7.3 Efficiency Experiments

The following experiments were created to compare/contrast the efficiency of direct access and RPC-based OS frameworks for heterogeneous systems. There is little to no data concerning how the architecture of a heterogeneous OS affects application-level performance, and these benchmarks have been designed to aid in discerning differences in operating system efficiency in terms of: system call latency, system call jitter, and overall application scalability [8, 9].

These experiments consider operating system service execution time as well as invocation time, thus enabling the quantification of the effects of contention on the efficiency of an operating system services. This is quite important, as contention management has become an extremely important factor as the number of processors and threads continues to increase.
Figure 7.3: Synchronization Latency Histograms

(a) Synchronization Latency, 1 Slave Thread

RPC → Mean = 8765, StDev = 443, Min = 8433, Max = 12352
Direct → Mean = 408, StDev = 31, Min = 341, Max = 458

(b) Synchronization Latency, 3 Slave Threads

RPC → Mean = 11090, StDev = 1848, Min = 8423, Max = 14305
Direct → Mean = 401, StDev = 36, Min = 329, Max = 462

(c) Synchronization Latency, 6 Slave Threads

RPC → Mean = 12249, StDev = 1010, Min = 9980, Max = 14385
Direct → Mean = 401, StDev = 35, Min = 329, Max = 468
Figure 7.4: Thread Create Latency Histograms

(a) Thread Create Latency, 1 Slave Thread

(b) Thread Create Latency, 3 Slave Threads

(c) Thread Create Latency, 6 Slave Threads
7.3.1 System Call Latency

The first experiment allowed for quantitative comparison of system call latencies between the RPC and hardware microkernel configurations. To summarize, a single RPC invocation required several sets of condition variable signal/wait pairs, incurred the overhead of the OS scheduler, context switching, and possible preemption of already running threads. The system call latency also had to include the overhead of the call itself, and the time taken to marshal data into and out of RPC-specific data structures. The inclusion of invocation overhead along with service latency results in an accurate portrayal of actual system call delays experienced within application programs.

Results

The PThreads-based synthetic benchmark used for this evaluation exercised only thread creation and synchronization APIs [8, 9]. The execution times of application code was minimized to allow observation of operating system API latencies. All measurements taken were cycle accurate, using on-chip free running counters. Figures 7.3 and 7.4 show histogram results of system call delays under the following application configurations: (a) a single active thread requesting OS services, (b) three active threads, and (c) six active threads [8, 9]. The average, maximum and minimum number of clock cycles are shown for each configuration. Additionally, mean latencies are shown for all possible configurations in Table 7.1. The histograms shown in Figures 7.3 and 7.4 are highlighted with bounding boxes to illustrate minimum and maximum histogram bin locations. Samples from direct-access and RPC-access calls do not overlap in any of the tests.

Analysis

The results show the reduced latency times achieved by direct access to our microkernel cores when compared to the RPC approach [8, 9]. The average latencies for a direct synchronization operation are 408 clock cycles compared to 8765 for a single slave processor
requesting services via RPC. The average latencies for a direct thread create operation are 1547 compared to 9866 via an RPC. In both types of system calls, as the number of processors increases, the mean for the microkernel remains essentially constant; displaying little to no jitter [8, 9]. The RPC approach exhibited significant variance due to serialization and contention for both the common bus and the services provided by the delegate threads executing on the master processor. Our analysis does not separate the two. However, no significant variance due to bus contention was observed in the direct-access cases. This is primarily caused by a decreased probability of contention as the service times for each operation lie in the range of tens of clock cycles. Specifically a synchronization request across the bus is a single 8 clock cycle load operation (3 cycles for bus arbitration, 5 cycles for synchronization manager processing). As a reference, a single load from off-chip memory requires approximately 17 to 31 clock cycles for SRAM and DDR RAM respectively. The serialization of RPC requests to the master CPU increases the probability of collisions, and results in longer and more variable delays as the number of competing processors is increased. The tight performance envelopes of the microkernel is particularly beneficial for systems with hard real-time performance requirements. Overall, the direct-access system calls are much more efficient, scalable, and have much more predictable timing characteristics than the RPC-based calls. This result supports the claim that a hardware-based microkernel can provide efficiency in ways not readily achievable in a software-only operating system [8, 9].

As a reference, the average latency of a NULL RPC system call with a single active thread in the system was 7250 clock cycles. A large component of this time is attributed to the context switch that occurs when an RPC delegate thread is run. A context switch takes approximately 3000 clock cycles on average. For comparative illustration, inter-processor communication in Intel’s EXOCHI system [121], which takes advantage of their proposed Multiple Instruction Stream Processor (MISP) ISA extensions, is estimated to take approximately 5000 clock cycles when implemented in microcode and 500 to 1000 cycles
Table 7.1: Mean System Call Latencies (Clock Cycles)

<table>
<thead>
<tr>
<th>Test</th>
<th>1 PE</th>
<th>2 PE’s</th>
<th>4 PE’s</th>
<th>6 PE’s</th>
</tr>
</thead>
<tbody>
<tr>
<td>RPC Synch</td>
<td>8765</td>
<td>12258</td>
<td>12245</td>
<td>12249</td>
</tr>
<tr>
<td>Direct Synch</td>
<td>408</td>
<td>399</td>
<td>402</td>
<td>401</td>
</tr>
<tr>
<td>Speedup</td>
<td>21</td>
<td>30</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>RPC Create</td>
<td>9866</td>
<td>13680</td>
<td>13659</td>
<td>13632</td>
</tr>
<tr>
<td>Direct Create</td>
<td>1547</td>
<td>1583</td>
<td>1579</td>
<td>1579</td>
</tr>
<tr>
<td>Speedup</td>
<td>6</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

if optimized with hardware-assist circuitry [57, 58]. These estimates appear a little lower than that observed measured timings within our experimental system. However, they are based on simulation and emulation estimates in fully cache coherent systems and not actual run time measurements. The processors used in the experimental system are not cache-coherent. Therefore, the RPC overhead included the overhead of communicating through a high latency, off-chip memory hierarchy. While a lower latency RPC mechanism may be achievable, the generalized effects of serialization and contention overhead are inherent to the RPC model and cannot be avoided. The direct-access microkernel approach alleviates the system of the RPC bottleneck altogether.

7.3.2 Application Scalability

The second benchmark allowed the quantitative comparison of how efficiency can effect an application’s ability to scale and achieve increased performance as the number of processors increases. The scalability of the application running on the hardware microkernel should be enhanced when contention for shared resources are decreased through distribution. Conversely, greater contention should result when the number of processors requesting service on a single shared resource is increased. Increased contention should have a negative effect on scalability The synthetic divide-and-conquer workload, outlined by the pseudocode shown in Figure 7.6, was used to evaluate contention and scalability. The benchmark allowed the observation of system call overhead and contention for shared resources, and how shared versus distributed resources affect achievable speedup when the number of proces-
Figure 7.5: Scalability Test: Direct Vs. RPC

```c
void * worker_thread (void * arg) {
    for (x = 0; x < iterations; x++) {
        time_delay (work_delay_count);
        hthread_mutex_lock (mutex);
        time_delay (crit_delay_count);
        hthread_mutex_unlock (mutex);
    }
}

int main () {
    time_start = timer_get ();
    for (t = 0; t < num_cpus; t++) {
        hthread_create (&tid[t], .., worker_thread);
    }
    for (t = 0; t < num_cpus; t++) {
        hthread_join (tid[t], ..);
    }
    time_stop = timer_get ();
    time_elapsed = time_stop - time_start;
}
```

Figure 7.6: Pseudocode - Scalability Test
sors is increased.

**Results**

The benchmark was executed in three different workload scenarios: coarse-grained, medium-grained, and fine-grained, by changing the relative work delay counts. The fine-grained delay was set to represent the execution of approximately 250 simple assembly instructions, medium-grained to 500, and coarse-grained to 1,000 assembly instructions. The goal of this benchmark is to demonstrate how changes in the size, or granularity, of work can affect the overall speedup achievable in a heterogeneous system in the face of both direct and RPC-based system calls. Figure 7.5 shows the obtainable speedup for the RPC-based approach tailing off well below ideal linear speedup in all three workload scenarios.

**Analysis**

As the workload moves towards a finer-grained level of parallelism, the effects of overhead and contention become more dominant. This effect has also been measured in traditional microkernel-based systems [83]. With direct access to the microkernel, the achieved speedups continue to scale at a nearly linear rate. The difference between the direct and RPC-based curves in Figure 7.5 is due to changes in the ratio between the amount of user-level computation time, and the amount of overhead incurred while invoking system calls. As the granularity of work becomes finer, the ratio of system call overhead to work increases, and achievable speedup diminishes. These results show that the operating system itself can prevent seamless scaling of an application into smaller and greater numbers of threads to match the increasing number of processors predicted by Moore’s law on next generation platform MPSoPCs. This should be concerning as it suggests that RPC models should not be used in systems with large numbers of processors when performance is a concern. This benchmark demonstrates that the operating system itself must follow the tenets of scalability: reduce contention and provide efficient parallel services. The hthreads
hardware-based microkernel does just this, by building OS mechanisms that provide fast and uniform access to system services. This example highlights the importance of Amdahl’s Law [13] in any parallel program – both application-level and system-level.

### 7.4 Uniformity Experiments

The primary goal of the following tests is for functional verification of the hthreads operating system, and to demonstrate the ability of the new hthreads system to support the execution of typical multithreaded programs in a heterogeneous environment. These tests verified the ability to compile and execute traditional multithreaded code on a heterogeneous multi-core system. Each test exercised a particular parallel programming pattern, and makes use of multithreaded primitives including thread creation, synchronization, and inter-thread signaling. Uniformity is upheld by the hardware-based microkernel by allowing the compilation and execution of threaded code across heterogeneous processor boundaries. Additionally, performance data is given for each test to help illustrate the performance benefits that come with the use of a uniform hardware-based microkernel.

#### 7.4.1 Parallel $\pi$ Approximation

One of the more popular forms of parallel programming, popularized by both OpenMP and PThreads, is the fork/join model of parallelizing code. This model is often used to target performance-intensive loops, and is able to produce speedup by executing different iterations of the loop in parallel. This model differs from the typical loop unrolling methodology used by optimizing compilers, as unrolling merely attempts to eliminate branching overheads between loop iterations. The fork/join model seeks to truly execute loop iterations in parallel across multiple processing elements. Use of this parallel programming pattern in a heterogeneous environment is supported by the hardware-based microkernel as thread creation, synchronization, and inter-thread signaling is supported across ISA boundaries.

This example shown here is an iterative approximation of $\pi$ that distributes loop itera-
typedef struct {
    int inc;
    int intervals;
    int start;
    double * pi;
    hthread_mutex_t * pi_mutex;
} targ_t; // Thread argument type

void * pi_thread(void * arg) {
    targ_t *targ = (targ_t *)(arg);
    register double local_sum = 0;
    register double width = 1.0/targ->intervals;

    // Accumulate local sum
    for(i=targ->start; i<targ->intervals; i+=targ->inc){
        register double tvar = (i + 0.5) * width;
        local_sum += 4.0 / (1.0 + tvar * tvar);
    }
    local_sum *= width;

    // Atomic update of shared pi
    hthread_mutex_lock(targ->pi_mutex);
    *(targ->pi) += local_sum;
    hthread_mutex_unlock(targ->pi_mutex);

    return (void*)0;
}

Figure 7.7: π Approximation Thread

tions across parallel threads. Although simple, the algorithm highlights the system’s ability
to use unaltered synchronization operations within threads running on processors with dif-
ferent ISAs. In this example, each thread must atomically update the shared variable pi
through the use of mutex primitives as shown in Figure 7.7. The underlying hthreads OS
and the heterogeneous design flow enable efficient and transparent synchronization between
heterogeneous processors in a traditional fork/join programming pattern that is illustrated
in Figure 7.8. This example is tested in two configurations:

- Direct Access - All synchronization operations are built on direct-access OS APIs.
- RPC Access - All synchronization operations are built on RPC-OS APIs forwarded
to the host.

Results

Each worker thread in this example performs many iterations of local computation be-
fore committing its results back to the global pi variable, thus leading to a high ratio of
computation-to-synchronization. The relatively low amount of synchronization suggests that direct- and RPC-based access scenarios should perform in a similar manner. Overall execution time and speedup results for the test (using 1,000 intervals) are shown in Figure 7.9 [5, 8].

**Analysis**

In both RPC and direct-access cases, increasing the number of worker threads (and thus decreasing the granularity of parallelism) leads to a reduction in execution time as shown in Figure 7.9(a). This reduction corresponds to linear speedup seen in Figure 7.9(b). There is a slight difference in the two curves that can be accounted for by the increased latency of synchronization operations in RPC mode. However, the infrequent use of synchronization operations (once per thread) does not cause a significant reduction in performance in the RPC scenario. If OS requests were used with a higher frequency, RPC-based access can lead to greater contention and serialization of OS services. In turn, this can lead to overall application slowdown – thus limiting the scalability of parallel programs that make use of RPC-based services.
Figure 7.9: Pi Approximation Scalability: Direct Vs. RPC

(a) Overall Execution Time

(b) Overall Speedup
7.4.2 Mailbox-based Communication

This experiment highlights the advantage of abstraction and uniform OS-level APIs by creating a set of higher-level communication primitives in the form of mailboxes. Both native and heterogeneous threads use the same set of mailbox APIs, eliminating the need to make code change when migrating threads across across heterogeneous boundaries [6, 5]. This test constructs a circular pipeline composed of parallel threads as shown in Figure 7.10. An example of a pipeline stage thread is shown in Figure 7.11. The pipeline consists of $k$ stages, where $k$ is the number of worker threads plus one (the main thread). Figure 7.12 illustrates the overall execution time, as well as average latency per element of a typical $k$-stage pipeline. Given $N$ elements are to pass through a pipeline, the first element to pass through will take $k$ time units, and all other elements will begin to fall through 1 time unit after another. As the number of elements passing through the pipeline becomes sufficiently large, the average latency per element approaches 1. This means that as the number of pipeline stages in the example is increased, the average latency per element should remain constant. The pipeline example is tested in two configurations:

- Direct Access - All mailbox operations are built on direct-access OS APIs.
- RPC Access - All mailbox operations are built on RPC-OS APIs forwarded to the host.

Results

Each pipeline stage continually performs a mailbox read, a single arithmetic operation, followed by a mailbox write. The ratio of computation to communication is quite low in
typedef struct targ{
    mailbox_t mb_input;
    mailbox_t mb_output;
} targ_t;

void* mbox_thread( void *data ) {
    targ_t * targ = (targ_t *) data;
    // Infinitely loop, read, modify write (mailboxes)
    while (1) {
        void *ptr = mailbox_read(&targ->mb_input);
        //pipeline stage code
        mailbox_write(&targ->mb_output, (void*)result);
    }
    return NULL;
}

Figure 7.11: Pipeline Thread

\[
\begin{align*}
    t_{exec} &= k + (n - 1) \\
    \frac{t_{exec}}{n} &= \frac{n}{k + (n - 1)} \\
    \lim_{n \to \infty} \frac{t_{exec}}{n} &= \frac{n - 1}{n} \\
    &\approx 1
\end{align*}
\]

where:
- \(k\) = Pipeline Stages
- \(n\) = Data Elements
- \(t_{exec}\) = Overall Latency
- \(\frac{t_{exec}}{n}\) = Latency/Element

Figure 7.12: Pipeline Scalability Equation
Figure 7.13: Mailbox Scalability: Direct Vs. RPC

(a) Average Latency Per Element

(b) Overall Speedup
this example as the majority of work done by each pipeline stage is primarily dedicated towards mailbox operations. Therefore, a disparity should be seen between direct- and RPC-based access as synchronization and communication time dominates thread execution time. Both the average latency per data element and overall application speedup for the pipeline example are shown in Figure 7.13.

Analysis

In Figure 7.13(a), average latency per element in the direct-access case follows the pipeline equation as expected, staying essentially constant as the number of pipeline stages is increased. However, the RPC-based latency scales nearly linearly as all mailbox operations are being serialized on the host processor. If these trends are looked at from a scalability perspective as illustrated in Figure 7.13(b), RPC-based access causes severe slowdown as the number of pipeline stages is increased. Direct-access performance stays relatively constant, with only a slight degradation that is primarily caused by contention for shared memory resources used for mailbox operations as the amount of local computation performed by each pipeline stage is so low.

Overall, this example uses both native- and heterogeneous-threads working together to move data through the pipeline; illustrating how heterogeneous threads cooperate, share data, and synchronize under a common run-time system and a uniform set of APIs. These tests show that increasingly complex communication constructs (mailboxes and bounded buffers) can be built using traditional OS primitives, and still remain usable across heterogeneous processor boundaries with the help of a heterogeneous operating system framework [5]. Additionally, evaluating this application under different OS invocation mechanisms (Direct and RPC) demonstrates the effect of OS structure on program performance and scalability.
7.4.3 Parallel IDEA Encryption

This example is built to highlight the ability for the hthreads framework to support heterogeneous processors equipped with application-specific hardware acceleration units [8]. This test exercises the block-based IDEA encryption algorithm on a large set of data. The encryption process is multithreaded, parallelizing the encryption process by creating threads to encrypt different sets of data blocks. The hardware platform for this example has been specialized by creating two separate clock-domains for each processor type, and by adding IDEA hardware acceleration units to each of the available MicroBlaze processors. A fast (125 MHz) clock domain is used for the PowerPC and the memory sub-system, while a slow (62.5 MHz) clock rate is used for the MicroBlaze processors and acceleration units. Each IDEA hardware accelerator unit is attached to the MicroBlaze’s fast-simplex link (FSL) interface, a native co-processor interface accessible through assembly instructions and C-macros. The acceleration units have been implemented in FSMLang, a domain-specific language for describing FSM-based computations [3]. Encryption threads come in three forms:

1. Native threads.

2. Heterogeneous (MicroBlaze-based) threads.

3. HW-accelerated heterogeneous threads.

Results

All threads have been coded in standard C using hthreads APIs, however the hardware-assisted thread makes use of MicroBlaze-specific macros to allow access to the FSL-based hardware acceleration units. Each thread implementation can be chosen dynamically at runtime by selecting the desired thread handle from the embedded heterogeneous header file. The choices in thread type create a large performance-space for this application, ranging
from little-to-no speedup using native multithreading all the way to an order of magnitude speedup while utilizing the heterogeneous threads that make use of the IDEA hardware acceleration units. Speedup results for encrypting 768 blocks in a 7-processor system composed of one PowerPC processor, and six IDEA-accelerator equipped MicroBlaze processors is shown in Figure 7.14 [8].

**Analysis**

The all native thread case does not experience any speedup as all of the worker threads are merely time-multiplexing on a single CPU. In fact, the slight performance degradation seen can be attributed to context switch overhead as the number of worker threads is increased. The heterogeneous case (no acceleration unit) illustrates that the single-threaded performance of the MicroBlaze is half that of the PowerPC, which is expected as the MicroBlaze is running at half the clock rate of the PowerPC. However, by targeting multiple MicroBlaze processors in unison, overall speedup can be achieved. In the hardware-accelerated
heterogeneous thread case, the MicroBlaze single-threaded performance is dominant, even while running at half the clock rate of the PowerPC. When multiple accelerated threads are used, a performance increase of nearly 9x is achieved.

This experiment demonstrates the ability of the hthreads operating system to cope with processors with different, even extensible ISAs. Additionally, it illustrates how differences in ISA, even for a single processor architecture, can lead to dramatic performance increases if application code is capable of taking advantage of the instruction set features. The hardware-based microkernel hides the details associated with interactions between processors of different ISAs, thus enabling the utilization of different processing cores in a transparent and uniform way [8].

7.5  Tying It All Together: Real Examples

The following examples illustrate the extensibility of the hthreads system at the software level. Although the core functionality of the hthreads OS is implemented in hardware for efficiency and heterogeneity reasons, additional service layers can be built on top of the base services just as in a traditional microkernel. All such services eventually target the core primitives provided by the hthreads OS, thus enabling the higher-level services to interoperate over heterogeneous processor boundaries. This is a unique feature of a hardware-based microkernel as it allows higher-level software (both system- and application-level) to leverage the ISA-agnostic services supported provided by the underlying hthreads system.

7.5.1  Thread Pools and Work Queues

Frameworks for higher-level forms of parallel processing comprised of pools of worker threads can be built on top of a heterogeneous multithreading environment [8]. Work-queues are a parallel programming pattern that consists of a pool of worker threads and a mechanism for submitting jobs to be executed by the pool of worker threads as illustrated in Figure 7.15. This patterns makes use of long-living threads that simply put themselves to
sleep when not executing submitted jobs, thus eliminating the need to repeatedly incur the overhead of thread creation and destruction. Each worker thread steals work (jobs) from the queue as they arrive, ensuring that threads will remain busy if adequate amounts of work exist in the queue. This example is unique in that a thread pool constructed on top of the hthreads OS can span heterogeneous processing boundaries all while using a single POSIX-based programming model. This example exercises the ability to pass abstract data types between heterogeneous processing cores as well as the ability to transparently synchronize and signal threads executing on different types of processors.

![Figure 7.15: Thread Pool (Work Queue) Framework](Image)

The work-queue model provides a job submission API, however the model does not provide a job query API, as all job queries can be implemented through traditional shared memory objects. As an example, the argument structure for a job can be modified to include a boolean flag to represent task completion. Before the job is submitted to the work-queue, the boolean flag can be initialized to false. Once the job completes, the task can set the boolean flag to true, and the task that submitted the job to the work-queue is free to poll this value at any time to check on the status of the job. Higher-level job querying mechanisms can be built on top of mutex synchronization objects, condition variables, and barriers.

The work-queue model is based on work-stealing and allows for dynamic load bal-
ancing between heterogeneous processors. As the work-queue is filled, jobs are taken by free processors in a first-come, first-serve basis. If a certain type of CPU is processing jobs at a faster rate than others, then it will become free at a higher rate, thus allowing it to consume more work-queue jobs. Conversely, a slower CPU’s job intake rate will be much lower, therefore it will take up a smaller fraction of the available workload. When the work-queue is empty, the pool of worker threads will return to idle. A programmer can implement difference balancing schemes by creating multiple work-queues that make use of different processing resources. For instance, an MP3 player application may create two parallel work-queues; submitting audio processing jobs to CPUs outfitted with SIMD vector units, while all GUI-based tasks are submitted to a work-queue that is only associated with traditional out-of-order execution RISC CPUs. The hardware-based microkernel allows work-queue models to span heterogeneous boundaries transparently, thus allowing programmers to create work-stealing structures without having to manually partition jobs according to architecture-specific performance features.

7.5.2 Image Processing with Heterogeneous Device Drivers

The purpose of this test is to demonstrate that the heterogeneous version of the hthreads OS retains the ability to support typical embedded applications including the offload of image processing tasks as well as graphics drivers and I/O [8]. This example is a simple inter-locked image processing pipeline that consists of three threads:

1. Input Thread
2. Processing Thread
3. Output Thread

The input thread is responsible for loading an image into the off-chip memory (frame-buffer) of the FPGA. Currently, this benchmark uses a fixed image, however the input thread
Figure 7.16: Laplace Edge Detection - Before and After

(a) Original Image

(b) Processed Image

Figure 7.16: Laplace Edge Detection - Before and After
could be made to pull in streaming video frames if desired. The processing thread is responsible for transforming the original input image, and placing the resulting image into a location accessible by the output thread. Currently, four simple image transforms have been implemented within the processing thread framework: color image threshold, overlaid intensity histogram, 3x3 laplace edge detection, and image negation. An example of the original image used and the laplace transformed image are shown in Figure 7.16. The output thread is responsible for displaying the processed image on an external video monitor through a DVI/VGA interface. The video output process is analogous to the work done by a video device driver.

All three of the threads are able to transparently synchronize with one another using standard mutex objects, even if they execute on processors with different ISAs. The image processing test is executed in three different configurations:

1. **AN** - All Native, all three threads execute on the PowerPC CPU.

2. **HP** - Heterogeneous Processing, the processing thread executes on a MicroBlaze CPU.

3. **HPO** - Heterogeneous Processing and Output, both the processing thread and output threads execute on MicroBlaze CPUs.

Each of the three configurations aim to show that a uniform heterogeneous multithreading environment allows for a variety of design-space trade-offs when building parallel systems composed of different types of processors. In this example, image processing and video output tasks can be easily offloaded onto heterogeneous processors without having to make any changes to thread-level or device-driver code. The ability for all processors to directly access OS services allows each of the threads to share common sets of OS-level synchronization objects. This alleviates the need for programmers to manually partition code for different processing architectures, and allows for enhanced inter- and intra-platform portability on heterogeneous MPSoCs.
7.5.3 Dynamic Thread Dispatch

Full user-level control of the placement of tasks in a heterogeneous system is useful, but it can also be cumbersome to all but the most expert parallel programmers. Facilities to help developers manage heterogeneous processing resources can make the job of programming these systems much easier; however the way in which these resources can and should be managed may be application-specific. Scheduling and management decisions as to when and how heterogeneous processing cores are used can be offloaded to a configurable thread dispatch function. This function, or set of functions, can either be defined by the OS developer or through user-defined policies. To support such functions, the system must present an interface and a set of support mechanisms for achieving policy goals.

As an example, the system may keep track of all available thread handles for each processor type in a heterogeneous multi-core system, while also tracking various state metrics for each individual processing element. The dispatch function can be defined to consider the various state metrics when deciding which processing element to execute a task on. The dispatch function’s decision as to which processor architecture to utilize can be fed into the thread handle table to look up the proper heterogeneous code pointer, which can then be forwarded to the OS for thread creation as shown in Figure 7.17.

Once the specific policy has been defined, it can be encapsulated inside a user-level API that can be used to invoke the decision function. This has the positive effect of giving the system the freedom to decide where a task will execute. This also alleviates the need for manual user-level control, and helps abstract application-level decisions in the form of a thread dispatch or scheduling policy. Additionally, the use of a higher-level API, as opposed to a hard-coded processor-selection technique, allows the underlying dispatch function to be altered without having to manually re-work application-level code.

An example of using a binary processor utilization metric can illustrate the utility of this approach. A system-level table is constructed automatically by the heterogeneous compiler that contains a vector of thread handles for each function identifier. The compiler is
Figure 7.17: System-Level Table of Thread Handles used for Dynamic Thread Dispatch
// Table entry type: generic function ID + vector of handles
typedef struct {
    unsigned int func_id;
    void * handles[MAX_HANDLES_PER_ENTRY];
} thread_entry_t;

// Table type: array of table entries
typedef struct {
    thread_entry_t table[MAX_ENTRIES_PER_TABLE];
} thread_table_t;

// Application-specific thread table
thread_table_t global_table = { ... };

// Dynamic dispatch API: configurable architecture/processor selection policy
unsigned int dynamic_dispatch(hthread_t * tid, hthread_attr_t * attr, unsigned int func_id, void * arg) {
    // Select architecture
    arch = <policy_code>;

    // Lookup thread handle
    handle = lookup_handle(global_table, func_id, arch);

    // Create thread for chosen architecture
    setup_attr(attr, arch);
    ret = hthread_create(tid, attr, handle, arg);
    return ret;
}
Begin Dispatch
Parameters:
Attributes,
Thread ID,
Thread Argument,
Function Identifier

System/Arch Query
Scan available CPU resources. Pick a resource based on built-in policy.

Thread Table Query
Lookup thread table entry for chosen architecture and Function ID.

Dispatch Thread
Initialize thread attributes, and launch thread on chosen CPU resource.

Figure 7.19: Control Flow of Dispatch Function
also modified so that it automatically generates a stub function in which dynamic thread dispatch policies can be defined as shown in Figure 7.18. The parameters of the dynamic dispatch function are very similar to that of a traditional thread creation API, except that a generic function identifier is used in place of a traditional function pointer; thus enabling the dispatch function itself to find the architecture and handle of interest. An abstraction of the control flow of the dispatch function is shown in Figure 7.19. In this example, the V-HWTI is extended to contain an additional field used to indicate processor utilization as:

- **USED** - CPU is being utilized.
- **FREE** - CPU is idle.

The stub function is replaced with a decision function that considers the utilization of heterogeneous processing units in order to decide where a thread will execute. The function has a built in preference for MicroBlaze processors, and attempts to launch the thread on one of the MicroBlaze cores if one is available. If no MicroBlaze is available, then the function chooses to launch the thread on the PowerPC. As MicroBlaze threads are launched, their V-HWTI utilization field is set to **USED**, and upon thread termination, the utilization field is set back to **FREE**. Utilization information is dynamically updated by each heterogeneous CPU, and is queried each time the dynamic dispatch function is invoked. The utilization-based dispatch function is shown in Figure 7.20

Although this type of utilization policy is quite simple, it effectively alleviates programmers from having to manually assign tasks to processors, and aids in fully utilizing all of the computational components available in a heterogeneous multi-core system. More complex policies can be created to take into account factors that consider processor load, power, hot-spots, inter-thread relationships, security concerns, and weighted processor affinities. Each metric can be calculated locally and then published to each processors’ V-HWTI. Then, any run-time function can query the V-HWTI metrics in order to make intelligent decisions during thread creation and dispatch.
unsigned int utilization_based_dispatch{
    hthread_t * tid,
    hthread_attr_t * attr,
    unsigned int func_id,
    void * arg
} {
    unsigned int ret;
    void * func;

    // ---- Look for any free heterogeneous processors ----
    int i = 0;
    int found = NOT_FOUND;
    for (i = 0; i < NUM_AVAILABLE_HETERO_CPUS; i++) {
        if (_hwti_get_utilized_flag(base_array[i]) == FLAG_HWTI_FREE) {
            // Mark CPU that was found, and break out of loop
            found = i;
            break;
        }
    }

    // ---- Create a native thread if no hetero CPUs are free ----
    if (found == NOT_FOUND) {
        // Create a native thread
        func = lookup_handle(&global_thread_table, func_id, TYPE_PPC);
        if (func == NOT_INITIALIZED) {
            ret = ERROR;
        } else {
            printf("Creating Native Thread!
");
            ret = hthread_create(tid, attr, func, arg);
        }
    }

    // ---- Otherwise create a hetero thread ----
    else { // Create a heterogeneous thread
        func = lookup_handle(&global_thread_table, func_id, TYPE_MB);
        if (func == NOT_INITIALIZED) {
            ret = ERROR;
        } else {
            // Create thread hetero using V-HWTI[found]
            printf("Creating Hetero Thread (CPU #%d)!\n", found);
            hthread_attr_init(attr);
            hthread_attr_sethardware(attr, (void*)base_array[found] );
            ret = hthread_create(tid, attr, func, arg);
        }
    }

    return ret;
}
A standard multithreaded program from the Lawrence Livermore National Labs (LLNL) POSIX Threads tutorial was chosen to demonstrate the utility of the modified compilation flow and dynamic dispatch functionality [24]. The example chosen is *dotprod*, a multi-threaded implementation of a dot-product kernel that operates on two individual vectors of data [25]. The original program was lightly modified so that it no longer utilizes global variables for thread arguments and mutex synchronization objects. Instead, these global variables are integrated into the thread argument structure, and passed explicitly to each child thread. The program was also modified so that the total vector length could be adjusted independently of the number of executing threads. The threaded code does not require any architectural changes to allow it to work in the heterogeneous environment as the OS provides uniform APIs to any processor, regardless of differences in ISA. The thread argument data structure and thread bodies of both the original code and the hthreads version are shown in Figure 7.21 and 7.22.

Next, the program was modified to utilize the dynamic dispatch routines created by the heterogeneous compilation flow. The dot-product invocation code was duplicated, making two full dot-product calculations on the vector data. The first pass utilizes heterogeneous threading, while the second pass is done using only native threads. Instrumentation points were then added to the two passes that allow for the two kernel execution passes to be timed. The code, as shown in Figure 7.23, was then passed through the heterogeneous compilation flow. This flow analyzes the application code and extracts the code associated with user-
typedef struct
{
    double *a;
    double *b;
    double * sum;
    int veclen;
} DOTDATA;

// Global variables
DOTDATA dotstr;
pthread_t callThtd[NUMTHRDS];
pthread_mutex_t mutexsum;

void * dotprod_thread (void *arg)
{
    int i, start, end, len;
    long offset;
    double mysum, *x, *y;
    offset = (long)arg;

    // Unpack arguments
    len = dotstr.veclen;
    start = offset*len;
    end = start + len;
    x = dotstr.a;
    y = dotstr.b;

    // Perform dot-product with local variables
    mysum = 0;
    for (i = start; i < end; i++)
    {
        mysum += (x[i] * y[i]);
    }

    // Atomic update of total sum
    pthread_mutex_lock (& mutexsum);
    dotstr.sum += mysum;
    pthread_mutex_unlock (& mutexsum);

    pthread_exit ((void*)0);
}

Figure 7.21: LLNL Dot-Product Thread and Argument Structure (Original Version) [24, 25]
typedef struct {
double *a;
double *b;
double * sum;
int veclen;
int offset;
hthread_mutex_t * mutexsum;
} DOTDATA;

void * dotprod_thread (void *arg) {
DOTDATA * targ = (DOTDATA *) arg;

int i, start, end, len;
int offset;
double mysum, *x, *y;
offset = (long) targ->offset;

// Unpack arguments
len = targ->veclen;
start = offset * len;
end = start + len;
x = targ->a;
y = targ->b;

// Perform dot-product with local variables
mysum = 0;
for (i = start; i < end; i++) {
    mysum += (x[i] * y[i]);
}

// Atomic update of total sum
hthread_mutex_lock (targ->mutexsum);
*(targ->sum) += mysum;
hthread_mutex_unlock (targ->mutexsum);

return NULL;
}

Figure 7.22: LLNL Dot-Product Thread and Argument Structure (hthreads Version) [24]
defined threads. The threaded code is then compiled for both the MicroBlaze and PowerPC processors, and is then re-linked to form a single heterogeneous executable. The modified flow generates an application-specific thread handle table, and dynamic dispatch function. Finally, the flow emits a heterogeneous-ready executable file (ELF) that can be downloaded and executed on the hthreads multi-core platform. The resulting executable was tested with a fixed vector length, while varying the number of worker threads from one (single-core) to seven (6 MicroBlaze CPUs and 1 PowerPC). The results gathered from this experiment are shown in Table 7.2. By utilizing the dynamic dispatch API, the program is able to transparently advantage of the available heterogeneous processing resources. This results in near linear speedups, as shown in Table 7.2, with minimal effort on behalf of the programmer. The hardware-microkernel’s efficient services and uniform programming model enable the seamless use of traditional multithreaded programs in heterogeneous multi-core environments. Additionally, the adherence to standard POSIX-compatible programming models enables higher levels of portability, even with a hardware-based microkernel.
7.6 Summary

Overall these experiments demonstrate that a hardware-based microkernel can be used as the basis for supporting traditional multithreaded programming in a heterogeneous multi-core system. Furthermore, the structure of the hardware-based microkernel enables uniform access to efficient services; resulting in an extremely scalable system as compared to traditional RPC-based methods. Not only is overall system call latency improved by an order of magnitude in a direct access microkernel, but the effect of system call latency on overall application scalability has been quantified as well. The results clearly show that while RPC methods are flexible, they have inherent performance drawbacks that can limit a programmer’s ability to parallelize their application. The direct-access microkernel structure provided by hthreads minimizes both OS service and invocation overhead by providing ISA-neutral interfaces to streamlined OS services. The ISA-neutral interface facilitates direct access in a heterogeneous environment, as any processor (regardless of ISA) is able to directly access services through memory-mapped I/O – a feature common to all CPU components.

The ISA-neutral services provided by hthreads are uniform, enabling access to a common set of OS services from heterogeneous processors, regardless of differences in ISA. This enables the use of a uniform parallel programming model, that allows multithreaded code to be compiled for each type of available processing unit. This not only simplifies the programming model by eliminating architecture- and platform-specific non-uniformities, but it also maintains portability between heterogeneous and homogeneous computing platforms.

The higher-level framework experiments illustrate that the hardware-based OS services can be extended just as traditional microkernel services can be built. New services can either be built upon user-level software layers that eventually map to OS primitives, or the hardware-based services themselves can be augmented or extended. Enhanced management, scheduling, and dispatch routines can be built on top of the hthreads platform in
order to facilitate automatic management of parallelism within application programs. The thread pool model allows for work-stealing constructs to be built on top of heterogeneous processing cores with no extra effort on behalf of a programmer. Additionally, the compiler-assisted configurable dispatch functions allow for the definition of custom scheduling and dispatch functions. The image processing example illustrates the ease in which legacy code and drivers can be made to work in a heterogeneous system with the help of an operating system built with heterogeneity in mind. Finally, a traditional POSIX-based multithreaded program is run through the modified compilation flow. The porting process did not involve any code partitioning nor re-factoring, and yielded linear speedups in a heterogeneous multi-core environment.
Chapter 8

Conclusion

Operating systems form the core of today’s computer infrastructure as they are the target platform for nearly all high-level languages, programming models, and application frameworks. The manycore revolution has already begun to transform computing hardware, violating many of the assumptions made by typical operating systems frameworks. Systems are no longer composed of single CPUs, or homogeneous SMP systems. Today, systems are composed of heterogeneous processing units, non-uniform memory hierarchies and cache structures, hardware acceleration units, and specialized interconnects. This has the effect of leaking architectural details into both the programming model and compilation flow required to target such system; resulting in systems that are difficult to program, and programs which are non-portable.

Additionally, many of the current adaptation techniques used to modify traditional OSes to work in heterogeneous systems introduce non-uniform access to system-level services. While this approach is flexible and allows the use of legacy operating systems, it does have serious performance drawbacks in terms of service request latency, contention, and ultimately scalability. Furthermore, current systems reflect many of the non-uniformities in the underlying architecture directly into the programming model; thus forcing programmers to adhere to architecture-specific details that not only make programming heterogeneous architectures difficult, but also results in extremely non-portable programs.

8.1 Research Contributions

This research work aims at establishing a new operating system framework geared towards maintaining both efficiency and uniformity in heterogeneous systems. The resulting framework consists of a hardware-based microkernel that supports direct access to a set of dis-
tributed OS services, regardless of differences in ISA. Additionally, the OS services themselves have been streamlined to offer extremely low latency operations. The pairing of fast OS services with lightweight, ISA-agnostic invocation mechanisms results in an OS that is both scalable and heterogeneous-friendly. Additionally, the uniformity of the operating system allows for standard parallel programming techniques, such as multithreading, thus providing developers with a programming model infrastructure that is not only familiar, but also portable amongst different platforms.

As processor technology progresses farther into the domain of parallel and multi-core systems, the need for efficient run-time systems will become paramount. Fabrication breakthroughs and optimizing compilers no longer provide the performance boost required to speedup serial programs, and parallel programs are only as scalable as their underlying run-time and operating systems. The key to enabling widespread use of heterogeneous multi-core platforms is the production of familiar programming environments that provide a scalable infrastructure capable of exploiting heterogeneous computational units. Without scalable operating systems, the peak performance of multi-core architectures will remain un-tapped. Additionally, unfamiliar and non-portable programming models will lead to the production of systems that are difficult to program as well as to port code to. The research contained within this dissertation strives to achieve both of the aforementioned goals: scalability and a familiar, uniform parallel programming model. Some of the major contributions of this work include:

- Heterogeneous Compilation - Allowing a single executable to represent and hold interacting binaries of different ISAs without modification of individual compilers and linkers.
- Heterogeneous Parallelization - Allowing a single, parallel application to span heterogeneous resources dynamically while using a standard multithreaded programming model.
- Heterogeneous Communication - Sharing data between processors of different ISAs.
• Heterogeneous Synchronization - Direct synchronization between processors of different ISAs.

• Heterogeneous Integration - Allowing a heterogeneous computing system to be managed under a single OS image.

• OS Uniformity - Creation of an OS framework built on upon the notion of ISA-neutral interfaces, allowing any CPU to directly access OS services. Resulting in a system that demonstrates an order of magnitude lower latency and jitter than a comparable RPC-based OS.

The heterogeneous coordination layer, including the HAL and V-HWTI, enable traditional multithreaded programming styles on heterogeneous MPSoC platforms containing different ISAs. The heterogeneous compilation flow aids programmers when targeting such systems by automatically compiling and linking programs to form heterogeneous binary executable files. The hthreads hardware cores provide ultra-efficient OS services packaged in an ISA-neutral manner, which allows for direct access from processors with different ISAs. Together, the HAL and cores provide a framework that is capable of providing scalable OS services that minimize latency and jitter by an order of magnitude.

8.2 Future Work

In general, computer architecture is in a time of great flux as we move further into the multicore era. As such, there are no true standard architectures on which general platform-level assumptions can be made. Questions such as how many cores will be on next-generation CPUs, or how many different types of cores will exist on a single chip are currently unanswered. However, it is clear that the number of processing cores is going to increase exponentially, the interconnect between cores will become more exotic, and heterogeneous mixed-ISA systems will become the norm. As such, heterogeneous multi-core platforms will continue to be a rich area for future research; especially in the area of adapting OS
and run-time frameworks for next-generation architectures. The following sections outline some of the major foreseeable research thrusts in this area:

### 8.2.1 Heterogeneity Detection and Migration

The panacea of heterogeneous multi-core OS frameworks is an intelligent framework capable of detecting platform-level heterogeneous characteristics such as the ISA capabilities of each processor, interconnect and memory hierarchy topology, and performance characteristics of the system. Future OS frameworks may manage an internal database of platform characteristics in order to enhance traditional management and scheduling duties with platform-optimized decisions. This type of intelligence can allow for the OS to decide exactly where a given task should run based on its individual characteristics. Additionally, with adequate high-level information, the OS could dynamically migrate tasks amongst heterogeneous processors according to both application and environmental demands. The repository could also be used to hold data extracted from the already existing system-level performance counters being added to new processor architectures.

Building a system-level repository for architectural information is essential to providing enhanced services in heterogeneous multi-core platforms. Investigation into the contents and structure of this repository could serve as an excellent extension to the existing hthreads HW/SW co-designed system. As the repository will be accessed by different types of processors it should be built in an ISA-neutral fashion. However, a pure hardware implementation of such a repository may be too limiting if the amount of data to be collected is quite large. Additionally, what would the access frequency of such a repository look like? Should the repository be distributed in order to manage contention in multi- and many-core systems, or would a centralized implementation suffice. Additionally, how should the information in the repository be combined with traditional OS policies and mechanisms. Would the repository be involved in all traditional OS services, or would a new set of services be created to utilize this information.
Another important question revolves around user-level control of system resources. Currently, most heterogeneous OSes allow a programmer to specify where a task will execute. Should this ability to specify execution location be removed from programmers’ control, or is it necessary in order to create highly-optimized applications even in the face of intelligent OS services? Additionally, one must also quantify how dynamic repository-based policies would affect the predictability and scalability of systems – especially as many modern applications (such as media and network processing) require at least soft real-time behavior.

### 8.2.2 Heterogeneous Compiler Technology

Adapting traditional, relatively static homogeneous compiler technology to work in the dynamic area of heterogeneous computing platforms is a ripe area of research. Pushing high-level semantic information such as threads and tasks farther into the compilation process is especially important in heterogeneous compilation frameworks, as this information is needed at run-time by a heterogeneous OS. Knowing what type of information is important to the OS may be platform-specific, especially in the case of architectures that may have highly specialized cores. The boundary between compiler and run-time has already began to blur with the advent of parallel processing frameworks. This boundary is sure to become even less well-defined as heterogeneous systems begin to incorporate higher levels of application- and platform-specific details at both compile- and run-time.

The compilation flow outlined in this work is specifically designed to allow for high-level information to be passed directly from the compiler to the run-time system. Information such as thread handle tables and dispatch functions are defined by the compiler and exported to the run-time system. However, what types of higher-level information should be extracted and explored? Details such as whether or not a thread would benefit from executing on one type of processing unit or another, or inter-thread communication patterns and data sharing? There are many possibilities, many of which require or would benefit from static analysis or interactive compilation.
8.2.3 Platform Flexibility and Degree of Heterogeneity

Further research into more advanced prototypes of hthreads systems is also an interesting area of research. Incorporation of new SIMD soft-IP processing cores into the hthreads platform would enable higher levels of thread specialization [130, 131], but requires a mature compiler flow to accompany each new processing core. Additionally, modern soft-core CPU offerings from Intel [122, 106] could also be incorporated in order to validate the hthreads approach alongside industry-standard processor architectures.

The platform used throughout this research utilizes two different processor architectures: a standard IBM PowerPC processor as well as a configurable Xilinx soft-core MicroBlaze processor [128, 126]. While the ISA of the PowerPC is static, the ISA of the MicroBlaze can be specialized through standard IP additions such as floating point units, or via custom or semi-custom acceleration units attached to it’s FSL coprocessor interface. This flexibility aids in validating the hthreads approach as the OS is able to handle a variety of ISA definitions, however further specialization through the integration of new processor architectures would bring additional value in terms of degree of achievable heterogeneity, as well as system prototyping for non-reconfigurable and non-embedded domains.
References


on Multicore Platforms: Theory and Practice (XRTS) held in conjunction with the 38th International Conference on Parallel Processing (ICPP), September 2009.


[29] Luca Benini, Michele Lombardi, Michela Milano, and Martino Ruggiero. A Constraint Programming Approach for Allocation and Scheduling on the CELL Broad-


[37] Julia Chen, Philo Juang, Kevin Ko, Gilberto Contreras, David Penry, Ram Rangan, Adam Stoler, Li-Shiuan Peh, and Margaret Martonosi. Hardware-Modulated Par-


163


[117] Kritchalach Thitikamol and Peter J. Keleher. Thread Migration and Load-Balancing in Heterogeneous Environments. In *LCR ’00: Selected Papers from the 5th Inter-


