University of Arkansas, Fayetteville
ScholarWorks@UARK

Theses and Dissertations

5-2012

Extending the HybridThread SMP Model for Distributed Memory Systems

Eugene Anthony Cartwright III
University of Arkansas, Fayetteville

Follow this and additional works at: http://scholarworks.uark.edu/etd

Part of the Digital Communications and Networking Commons, and the Graphics and Human Computer Interfaces Commons

Recommended Citation
Cartwright, Eugene Anthony III, "Extending the HybridThread SMP Model for Distributed Memory Systems" (2012). *Theses and Dissertations*. 400.
http://scholarworks.uark.edu/etd/400

This Thesis is brought to you for free and open access by ScholarWorks@UARK. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of ScholarWorks@UARK. For more information, please contact scholar@uark.edu, ccmiddle@uark.edu.

Extending the HybridThread SMP Model
for Distributed Memory Systems

Extending the HybridThread SMP Model
for Distributed Memory Systems

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Computer Engineering

By

Eugene Anthony Cartwright, III
University of Arkansas
Bachelor of Science in Computer Engineering, 2009

May 2012
University of Arkansas

Abstract

Memory Hierarchy is of growing importance in system design today. As Moore's Law allows system designers to include more processors within their designs, data locality becomes a priority. Traditional multiprocessor systems on chip (MPSoC) experience difficulty scaling as the quantity of processors increases. This challenge is common behavior of memory accesses in a shared memory environment and causes a decrease in memory bandwidth as processor numbers increase. In order to provide the necessary levels of scalability, the computer architecture community has sought to decentralize memory accesses by distributing memory throughout the system. Distributed memory offers greater bandwidth due to decoupled access paths. Today's million gate Field Programmable Gate Arrays (FPGA) offer an invaluable opportunity to explore this type of memory hierarchy. FPGA vendors such as Xilinx provide dual-ported on-chip memory for decoupled access in addition to configurable sized memories. In this work, a new platform was created around the use of dual-ported SRAMs for distributed memory to explore the possible scalability of this form of memory hierarchy. However, developing distributed memory poses a tremendous challenge: supporting a linear address space that allows wide applicability to be achieved. Many have agreed that a linear address space eases the programmability of a system. Although the abstraction of disjointed memories via underlying architecture and/or new programming presents an advantage in exploring the possibilities of distributed memory, automatic data partitioning and migration remains a considerable challenge. In this research this challenge was dealt with by the inclusion of both a shared memory and distributed memory model. This research is vital because exposing the programmer to the underlying architecture while providing a linear address space results in desired standards of programmability and performance alike. In addition, standard shared memory programming models can be applied allowing the user to enjoy full scalable performance potential.

This thesis is approved for recommendation
to the Graduate Council

Thesis Director:

Dr. David Andrews, Ph.D.

Thesis Committee:

Dr. Miaoqing Huang, Ph.D.

Dr. Christophe Bobda, Ph.D.

Thesis Duplication Release

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

Agreed ____

Eugene Anthony Cartwright, III

Refused ____

Eugene Anthony Cartwright, III

Acknowledgments

I would like to express the deepest appreciation to my mother, Antoinette Campbell, my sister, Christina Campbell, my step-father, Donald Campbell, and my grandmother, Daphne Grissom for their endless encouragement. I would not be the person I am without them in my life.

I would also like to thank my girlfriend Rachel Findley, for her continued support and love. She has truly been a blessing in my life. I also would like to thank Dr. Jason Agron, for his continued mentorship throughout my Master's program.

I would like to show my gratitude to Dr. David Andrews, for providing me the opportunity to learn and conduct research under his authority.

Finally, I would like to extend my thanks to my thesis committee and friends-from-afar, including Anson Moxey and Dontae Jacobs.

Contents

1 Introduction **1**
- 1.1 Thesis Contributions and Organization 4

2 Background **6**
- 2.1 Taxonomy of Computer Architectures 6
- 2.2 Shared Memory Architectures 8
  - 2.2.1 Uniform Memory Access 9
  - 2.2.2 Non-Uniform Memory Access 10
- 2.3 Non-Shared Memory Architectures - Distributed Memory 11
- 2.4 Related Work - Distributed Memory 14

3 System Design **19**
- 3.1 Brief Intro to Hthreads 20
  - 3.1.1 Original Heterogeneous Hthreads System 21
- 3.2 Distributed and Shared Memory Design 22
  - 3.2.1 Configuring Bus/Local Memory Components 22
  - 3.2.2 First Approach - Enabling P2P 25
  - 3.2.3 Second Approach - Split-BRAM 27
  - 3.2.4 Third Approach - Placing the V-HWTI in Private Memory 30
  - 3.2.5 Local Memory Initialization Issues 33
- 3.3 Software Development 35
  - 3.3.1 Heterogeneous Compilation Flow Modifications 36
  - 3.3.2 Hthreads Kernel Modifications 39

4 Results **43**
- 4.1 Defining Scalability 43
- 4.2 Latency for Data Accesses 43
- 4.3 Weak Scalability 47
  - 4.3.1 Matrix Multiply 49
  - 4.3.2 IDEA 51
- 4.4 Strong Scalability 54
- 4.5 Hardware/Software Microkernel vs RPC Scalability 57
  - 4.5.1 Synthetic Test - RPC vs Non-RPC 60

5 Conclusion **71**
- 5.1 Research Contributions 71
- 5.2 Future Work 73
  - 5.2.1 Expanding the Hthreads OS 73
  - 5.2.2 Heterogeneous Compilation Flow 74

References **76**

List of Figures

2.1 SIMD: Single Instruction, Multiple Data . . . 7
2.2 MIMD: Multiple Instruction, Multiple Data . . . 7
2.3 Uniform Memory Access Structure . . . 10
2.4 Non-Uniform Memory Access Structure . . . 12
2.5 Distributed Memory Architecture - local memory is now private . . . 13

3.1 Hthreads SMP System . . . 23
3.2 IPLB configured for P2P . . . 25
3.3 Split-BRAM with multiple V-HWTI's on common bus . . . 28
3.4 Default Linkerscript was modified to report a shorter length for "private_memory" . 29
3.5 Linkerscript accounting for V-HWTI in "private_memory" . . . 31
3.6 V-HWTI placed inside of private memory . . . 31
3.7 Full Split-BRAM System . . . 32
3.8 Host memory map includes all memories . . . 34
3.9 Slaves memory map include all other slave's memory . . . 34
3.10 Final arrangement of Host and Slave processor . . . 35
3.11 Embedding Process [1] . . . 37
3.12 Memory space is preserved between *thread_function and foo2 but size remains the same* . . . 38
3.13 Thread function consumer_thread may be marked as a dependency . . . 39
3.14 Illustrates the additions done to the original heterogeneous compilation flow . . . 40
3.15 Thread Create and Join calls that make use of the DMA . . . 41
3.16 Thread Creation using the DMA controller after a free slave processor is found . . 42

4.1 The thread function used during the Access Latency Test . . . 44

4.2 Work Delay = 250 instructions for 1 thread running 45
4.3 Work Delay = 500 instructions for 1 thread running 46
4.4 Work Delay = 1,000 instructions for 1 thread running 46
4.5 Work Delay = 250 instructions for 6 threads running 47
4.6 Work Delay = 500 instructions for 6 threads running 48
4.7 Work Delay = 1,000 instructions for 6 threads running 48
4.8 Writing/Reading to the Shared Memory 56
4.9 Thread Function code for the RPC Synthetic Test 59
4.10 SMP - RPC vs. Direct Calls 62
4.11 Split-BRAM (Groups of 3) - RPC vs. Direct Calls 62
4.12 Split-BRAM (Groups of 4) - RPC vs. Direct Calls 63
4.13 Split-BRAM (Groups of 8) - RPC vs. Direct Calls 63
4.14 Split-BRAM Speedup differences between Groups of 3 and 4 on a single PLB . . . 64
4.15 Split-BRAM Groups of 3 and SMP: Coarse Grained 66
4.16 Split-BRAM Groups of 3 and SMP: Fine Grained 67
4.17 Split-BRAM Groups of 3 and 4: Coarse Grained 67
4.18 Split-BRAM Groups of 3 and 4: Fine Grained 68
4.19 Split-BRAM Groups of 3 and 8: Coarse Grained 68
4.20 Split-BRAM Groups of 3 and 8: Fine Grained 69
4.21 Split-BRAM Group of 8 and SMP: Coarse Grained 69
4.22 Split-BRAM Group of 3 and SMP: Fine Grained 70

List of Tables

4.1 Matrix Multiply - Thread Running time Averages 51

4.2 Matrix Multiply - Operating System Overhead for Thread Creation 52

4.3 Matrix Multiply - DMA Overhead for Thread's Instructions (4 KB) & Data (1.6 KB) 52

4.4 Matrix Multiply - Total Execution Time 52

4.5 IDEA - Thread Running time Averages 54

4.6 IDEA - Operating System Overhead for Thread Creation 54

4.7 IDEA - DMA Overhead for Thread's Instructions (8 KB) & Data (3.2 KB) 55

4.8 IDEA - Total Execution Time 55

4.9 Total Calculation Time for a 20 *x* 20 Matrix 57

4.10 Total Calculation Time for a 256 *x* 256 Matrix 57

4.11 Total Calculation Time for a 512 *x* 512 Matrix 57

4.12 Total Calculation Time for a 1024 *x* 1024 Matrix 58

4.13 Total Calculation Time for a 2048 *x* 2048 Matrix 58

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.

CPU Central Processing Unit. A hardware component, often referred to as processor or core that processes instructions of a program.

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

DLP Data Level Parallelism. A form of parallelization that emphasizes distribution of data on parallel processingelements. i

FPGA Field Programmable Gate Array. An integrated circuit that can be configured/re-configured by a designer.

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).

HLL High-Level Language.

HW Abbreviation for Hardware.

HWTI Abbreviation for Hardware Thread Interface.

V-HWTI Abbreviation for Virtual Hardware Thread Interface.

ISA Instruction Set Architecture. The instruction set: data types, instructions, etc., native to a particular architecture.

I/O Input/Output.

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

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

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. A method of communication between devices through load and store operations into memory (read and write).

MMU Memory-Management Unit.

MPI Message Passing Interface.

MPSoC Multi-processor System-on-Chip. A system-on-chip having multiple processors.

OS Abbreviation for Operating System.

OTS Off-the-Shelf.

RISC Reduced Instruction Set Computer.

RPC Remote-Procedure Call. A method of IPC that relies on a process being able to execute code in a another address space that where it was coded on.

RTOS Abbreviation for Real-time Operating System.

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.

TLP Task/Thread -Level Parallelism. A form of parallelization that emphasizes on the distribution of threads/tasks across parallel processing elements.

Chapter 1

Introduction

A little over a decade ago Xilinx teamed with IBM, Mentor Graphics, Synopsis, Wind River, and MathWorks to effect a paradigm shift in usage for FPGAs from glue logic and single function accelerators to more holistic systems on chip components. New hybrid CPU/FPGA Platform components emerged which contained diffused processors, multipliers, and distributed SRAM blocks within the FPGA fabric. These Platform FPGAs currently contain over a million LUTs, a sufficient density to turn a single FPGA chip into a complete multiprocessor system on chip (MPSoC). As FPGA's continue to follow Moore's law density levels will allow 100's to 1,000's of mixed type programmable processors as well as custom accelerators to be configured within a single chip. The use of Platform FPGA's as multiprocessor systems on programmable chips (MPSoPCs) opens up the potential to bring the benefits of higher level software centric programming models into the reconfigurable computing arena. This is appealing as it will allow software engineers, and not just hardware designers to enjoy the benefits of reconfigurable computing. The programmability of Platform FPGAs as MPSoPCs is also of interest to the wider computer architecture community as they begin to explore new memory hierarchies, bus interconnects, and distributed run time support for next generation heterogeneous manycore chips.

Current density levels allow MPSoPCs to comfortably support systems with 10's of processors. Researchers and chip manufacturers have been developing infrastructure and programming models to create Symmetric Multiprocessor Architectures (SMP). SMP systems are characterized by their homogeneity: same processors with shared bus structures and a common global memory. Processors rely on local caching to minimize bus contention and long access times associated with global memory across a single shared bus. Xilinx provides infrastructure in the form of the Multi-Port Memory Controller (MPMC) that allows instruction caching of up to seven processors including a shared bus for data accesses to global memory. This helps relieve bus contention for single issue

processors such as the MicroBlaze which may fetch instructions every clock cycle [19]. SMP systems in general are known not to scale well past 10's of processors. For Xilinx based systems in particular, the MPMC does not scale and will not allow connecting greater numbers of processors even though FPGA density may allow it. For systems with greater processor numbers, accesses must be multiplexed across the shared bus, which can result in a degradation of system performance. For these reasons researchers are investigating Distributed Memory (DM) architectures that offer better scalability characteristics. DM systems are characterized by distributed memory hierarchies and more robust interconnect networks.

DM systems break the basic assumption of SMP's centralized linear address space model. This has implications for programmers using languages such as C, and programming models such as POSIX threads (Pthreads) that are built on this model. New programming models such as OpenCL are only beginning to emerge in support for manycore systems that contain heterogeneous processors and multi-tiered distributed memory hierarchies. Interestingly most of these newer models are additional abstraction layers that use Pthreads as a base and provide additional API's for programmers to implicitly manage data transfers across the multi-tiered memory hierarchy.

Hybrid threads (Hthreads) is a Pthread compliant run time system that was originally developed to blur the boundary between threads running in software on a CPU and threads that were translated into hardware accelerators running within the FPGA fabric. Hthreads assumed the global memory address space of the Pthreads multithreaded programming model for SMP systems [3]. Hthreads targeted embedded and distributed real time systems and was organized as a hardware/software co-designed microkernel structure. As part of its developmental progression, Hthreads evolved to support SMP systems with mixes of heterogeneous processor types. With the realization that SMP systems are not scalable it was reasonable to ask if the Hthreads system could be modified and extended to support DM architectures and newly evolving heterogeneous manycore programming models. With the observation that these newly evolving programming models were based on fundamental support provided by Pthreads, it was reasonable to assume that Hthreads could also be extended and serve in this role. In addition to supporting newly evolving programming models we

were also interested in understanding if the fundamental shared memory multithreaded programming model could be transparently modified and augmented with a minimal number of APIs for both instruction and data transfer such that the same program with minimal modifications could run efficiently on both SMP and DM architectures. This would provide a segue for comfortable programming of SMP systems towards next generation scalable DM architectures.

Enabling the same, or slightly modified program to run on both SMP and DM systems also allows some performance comparisons to be made between the two architectures running a common code base. As we move into the manycore era arguments continue over the effectiveness of cache hierarchies. Early distributed memory manycore systems such as the IBM Cell eliminated local caches, replacing them with software managed local scratchpad memories. The malleability of the FPGA would allow us to investigate different configurations of SMP systems using caches, and DM systems with replacement scratchpad memories. Running a common code base would allow fair comparisons of the two systems and prove the programming feasibility when switching between them.

To explore these issues we developed a significant number of SMP and DM architectures within a Xilinx Platform FPGA. While Xilinx provides additional support components for creating small SMP systems, no such infrastructure exists for DM architectures. New approaches were required for creating and grouping processor/local memory pairs together, and configuring standard busses with bridges to create a suitable interconnect network. A significant amount of trial and error exploratory work was required to enable the MicroBlaze slave processors to access instructions and data across their private busses from local BRAM memories, while still allowing the BRAM memories to be accessible across the system busses. Modifications were also made to the Hthreads run time system to seamlessly transfer instructions and data between the different tiers of the memory hierarchy. Direct Memory Access (DMA) IP components were added to the system, with the transfer of instructions and data using DMA hidden under the typical thread_create() API. Additionally, modifications were required to direct each processor to execute code out of local instead of global memory as done in the SMP variant.

After the appropriate architectures were built and modifications to the run time system were made, we conducted a series of experiments to evaluate the performance differences between the SMP and DM architectures running common benchmarks. We also analyzed how grouping different numbers of processors on local busses effected the performance of our DM systems. This provided important insight for building and configuring future systems with 100's to 1,000's of processors. We also analyzed the performance difference between SMP systems with caches and DM systems with scratchpad memories replacing caches. This also yielded interesting results that will be of interest to the wider manycore architecture community. Finally we believe we successfully showed that the standard Pthreads multithreaded programming model could be ever so slightly modified and used effectively for DM systems. This could result in an interesting transition path for current programmers making the switch to next generation DM based manycore systems.

1.1 Thesis Contributions and Organization

This work proposes that by utilizing both distributed memory and shared memory within a HybridThreads system, we can provide *scalable* performance and system design as we increase the processor count over a HybridThreads system employing only the use of a centralized shared memory with caches.

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

- A new HybridThreads system that combines both the distributed memory and shared memory models.
- Integration of transparent transfers of a thread's instructions and data utilizing a Direct Memory Access (DMA) controller.
- A ported HybridThreads system for use on the Virtex ML605 Evaluation board in order to realize such larger systems.

- PowerPC and MicroBlaze host repositories merged into one to allow easy switching across several heterogeneous platforms.

- A new compile flow created, automated, and integrated into the existing heterogeneous compile flow in order to statically or dynamically create, transfer, and retrieve a thread's instructions and data without incurring too much redundancy across transfers.

- An optimized nanokernel and compiled heterogeneous applications that creates smaller binaries and avoids increased bus contention due to unnecessary global memory accesses.

This thesis draws a comparison between the scalability of an HybridThreads system integrating both a shared and distributed memory model compared to one with supporting a cached shared memory model. Chapter 2 introduces basic concepts of these two models and illustrates certain issues of both models by discussing related work. Chapter 3 describes the architecture of both SMP and DM architectures, and highlights some of the design decisions that were made in order to come to the final product. Chapter 4 discusses results that highlight the scalability of the two systems. Also, an additional discussion is added to show the scalability of the Hthreads hardware/software co-designed micokernel. Finally, chapter 5 provides conclusions and possible considerations for future work.

Chapter 2

Background

2.1 Taxonomy of Computer Architectures

Most of today's multiprocessor architectures fall into either the *single instruction, multiple data* (SIMD) or *multiple instruction, multiple data* (MIMD) architectures defined under Flynn's taxonomy [8]. The SIMD classification defines architectures that operate on a single instruction stream, but on multiple similar or dissimilar data streams. As illustrated in Figure 2.1, a single instruction stream resides in a centralized memory, whereas the multiple data streams are accessible through distributed memories. As a result of multiple data streams, these architectures are capable of exploiting *Data-Level Parallelism* (DLP) –a form of parallelism that emphasizes distribution of data amongst parallel processing elements. Examples of SIMD architectures include Vector-Processing Array machines, and off-the-shelf (OTS) Graphics Processing Units (GPUs). SIMD architectures typically require the use of specific programming languages to achieve DLP. MIMD machines offer more flexibility by offering multiple instruction and data streams as shown in Figure 2.2. Instruction streams can be similar or dissimilar, enabling MIMD architectures to execute concurrently different applications. Intel and Advanced Micro Devices (AMD) who create MIMD architectures also include support for SIMD such as the Streaming SIMD Extensions (SSE). MIMD also encourages heterogeneity due to different instruction streams and Thread-Level Parallelism (TLP). TLP is another form of parallelism that focuses on the distribution of threads or tasks amongst parallel processing elements. Additionally, programming is much more comfortable on MIMD architectures as many programming models such as POSIX threads (Pthreads) that support these architectures are built on top of popular programming languages such as C/C++.

MIMD architectures are subdivided into two distinct groups: Shared Memory and Distributed Memory. Both groups differ only by the presence or absence of a common linear address space

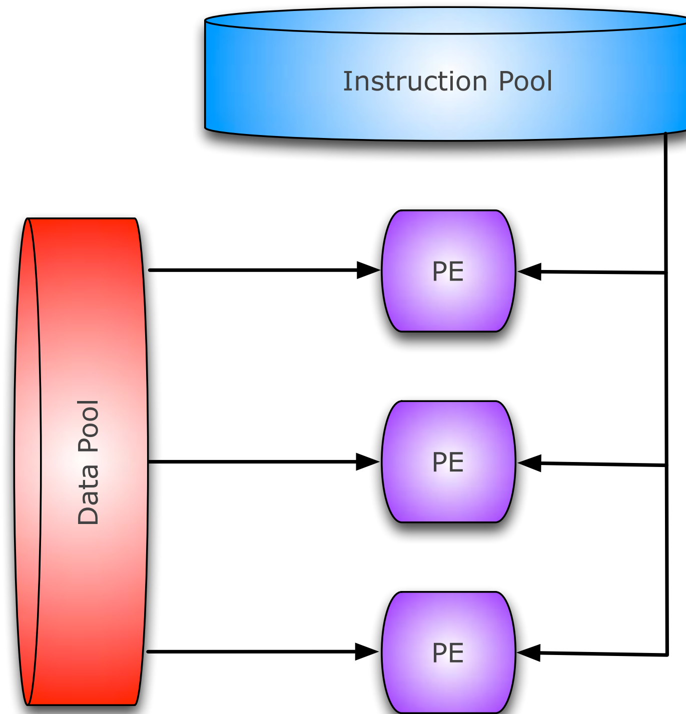

Figure 2.1: SIMD: Single Instruction, Multiple Data

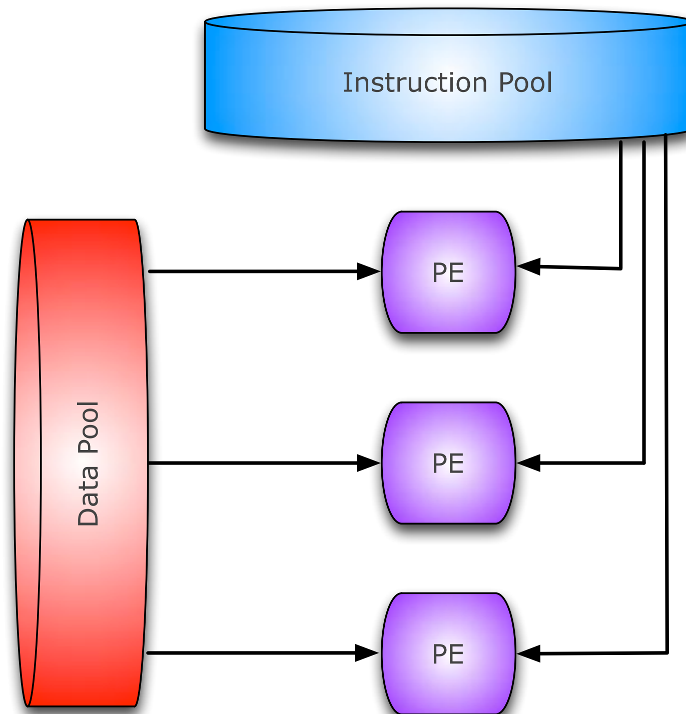

Figure 2.2: MIMD: Multiple Instruction, Multiple Data

between all processors within the system (or between processing nodes). This distinction is important as there exists systems with distributed memory that continue to maintain the common linear address space. Therefore, the following sections are provided to expand more on the similarities and differences between the two groups and also to highlight their advantages/disadvantages.

2.2 Shared Memory Architectures

Shared Memory Multiprocessor Architectures share a common linear address space between all processors. For example, memory location *0xDEADBEEF* resides in the same (physical) place for all processors. Therefore, address regions cannot be duplicated (as can be the case for distributed memory architectures). However, this does not imply that all memory must centralized, but can take a distributed form as well. As a result of being *globally* viewable, memory coherency between processors is automatically maintained as long as data is not changed simultaneously by two or more processors. For example, when processors change data throughout memory, other processors can immediately see this change due to the common shared memory space. When multiple processors attempt to access data at the same memory location, special handling must be done to prevent scenarios of race conditions, or dirty reads. How one handles this and ensures memory consistency across all processors is referred to as memory coherency protocol.

Within a MPSoPC, presenting a common linear address space aids in programming effort and compiler support. Much of the popular compilers such as the GNU compiler, *gcc* all assume a linear address space. Therefore, applications written for shared memory architectures are highly portable across different shared memory systems. It also aids in programming effort for shared memory architectures. Programmers operating over large data structures that would otherwise be very difficult to maintain when sharing data amongst processors can benefit from the implicit communication of shared memory. However, memory access times can differ depending on the structure of shared memory. As mentioned previously, shared memory does not imply a centralized memory but can be distributed throughout the system while maintaining the overlay of a linear address space. Therefore, shared memory architectures can be further categorized as two types:

Uniform Memory Access and *Non-Uniform Memory Access*.

2.2.1 Uniform Memory Access

Uniform Memory Access (UMA) refer to multiprocessor (and uniprocessor) systems that are characterized by their uniform memory access latencies to shared memory. For example, when referencing a particular memory location, A, all processors capable of addressing A will be able to do so in similar timing. As a result of this uniformity, UMA systems generally are homogeneous, or at the minimum include processors of the same Instruction Set Architecture (ISA). UMA systems also are referred to as *Symmetric Multiprocessor* (SMP) Systems due to their uniform access times and their inherent homogeneity. Due to symmetric access times, SMP systems also share similar bus structures. Most modern day Desktop computers, servers, and now even mobile devices are SMP systems. The Intel Core 2 Duo, Intel Xeon, and ARM Cortex-A9 are all examples of including UMA memory architectures [10]. Figure 2.3 shows a generic layout of an SMP system and illustrates its symmetry.

Unfortunately SMP systems do not scale well past a small number of processors [14]. The centralized memory in addition to the rising *Memory Wall* ultimately saturates memory bandwidth. FPGA vendors such as Xilinx and Alterra include multi-channel memory controllers to relieve bus pressure by allowing processors to directly attach to shared memory as done in Figure 2.3. Although these controllers can be made bigger to accommodate increased numbers of processors, this does not present a scalable solution. Furthermore, the multi-channel memory controllers such as Xilinx's Multi-Port Memory Controller (MPMC) attaches to a single piece of memory [20]. This presents a choke point that is quickly revealed when increasing demand for memory through growing processor numbers. Therefore, any performance to be gained by supplying additional processors within the system may run the risk of degrading overall performance. System designers were able to reduce shared memory demand by including caches. Caches are (multiple) local, on-chip memories that aid in shared memory accesses by allowing data to reside much more closer to a processor. Depending on what is being cached, this also introduces expensive cache coherency

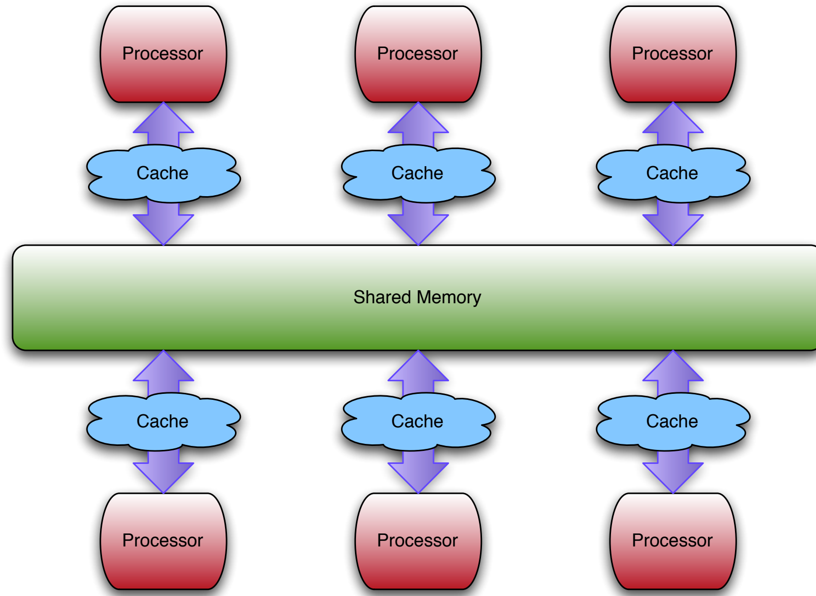

Figure 2.3: Uniform Memory Access Structure

protocols resulting in bigger, more power hungry components.

2.2.2 Non-Uniform Memory Access

Non-Uniform Memory Access (NUMA) refers to a similar idea that there exist a shared linear address space across all processors but no longer centralized. Rather, the address space is mapped across several physically disjointed memories that are each individually attached to processors. These types of memory architectures are referred to as *Distributive Shared Memory* (DSM) architectures. Due to the disjointed memory layout, shared memory access times differ when referencing different parts of memory. For example, memory location A may map onto a processors local memory but memory location Z may map to non-local memory. As a result, memory access times to both A and Z may differ. Examples of DSM systems include the NEC AzusA, Compaq AlphaServer GS80/GS160/GS320, and SGI Origin [4]. Figure 2.4 shows a generic DSM Architecture organization.

Unlike UMA architectures where shared memory access times are deterministic, NUMA architectures are not. However, NUMA architectures such as DSM systems provide two immediate

advantages. First, memory bandwidth is scaled with increasing processor counts. For example, a DSM system with four processors all simultaneously accessing their local memories has 4*x* the memory bandwidth than an SMP system sharing a centralized memory. Second, the linear address space is also scaled with increasing processor counts. That is, adding more processors in a DSM system including their local memories contributes to the shared linear address space additively.

Although one may have the benefit of both increased memory bandwidth and the transparent expansion of the available linear address space, processors can spend most of their time making remote memory accesses. This can happen due to shared data residing entirely into one piece of memory rather than distributed according to frequent use. Caches can help resolve this penalty through exploiting locality. Locality refers to the proximity of data to a processor. For a distributed memory system where access times can vary according to the proximity of memory, locality is an important characteristic to have. Locality can be categorized in two forms: temporal and spatial. Temporal locality refers to the idea of if there is a reference to a memory location, the probability that it will be referenced again in the near future is high. An example of this is referencing a particular memory location such as a counter within a loop. Sometimes processors not only cache specific referenced data within shared memory, but adjacent data to that location as well. This optimistic type of prefetching is referred to as spatial locality and it follows the idea that if there is a reference to a memory location, the probability that adjacent memory locations will be referenced in the near future is high. Besides caching, other ways of achieving locality exist and are further discussed in Section 2.4.

2.3 Non-Shared Memory Architectures - Distributed Memory

Distributed Memory (DM) architectures are similar to the DSM's discussed in 2.2.2. Both hold the concept of a decentralized memory structure. However, what differentiates DSM with DM architectures is that DM architectures do not provide a common linear address space. Instead each processor operates out of its own private memory space as shown in Figure 2.5. As a result of this, one can infer to a multiprocessor DM system as a collection of multiple uniprocessor systems as

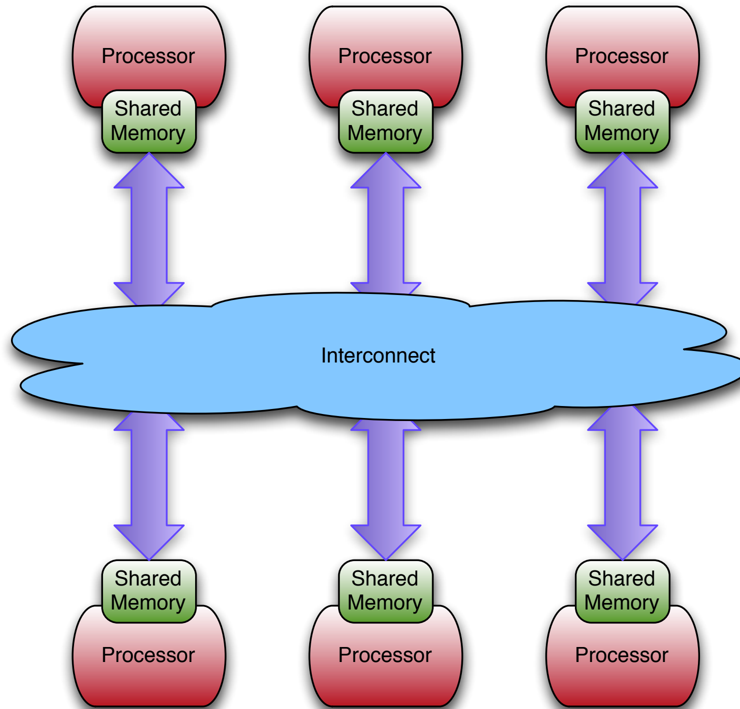

Figure 2.4: Non-Uniform Memory Access Structure

was a common setup in the past in creating very cheap server clusters.

Due to the private memory address spaces, an immediate disadvantage of using them as parallel machines is that they can undergo much communication overhead when needing to share data. As implied in Section 2.2, communication was done through the shared memory space for both SMP and DSM architectures. When a processor updates a memory location, other processors automatically are able to see this change due to the common view of memory. For DM architectures, modified data remain private in a processor's local memory until it is explicitly shared. Sharing data within a DM architecture generally involves the use of messages: a processor prepares a message to be sent, releases the message out on its interconnect, and finally another processor receiving and unpacking the message. The general specification, known as the Message Passing Interface (MPI) arose during the nascent stages of DM architectures to ensure the support of communication when sharing data within a program remain efficient and portable across different MPI-compliant DM systems [9].

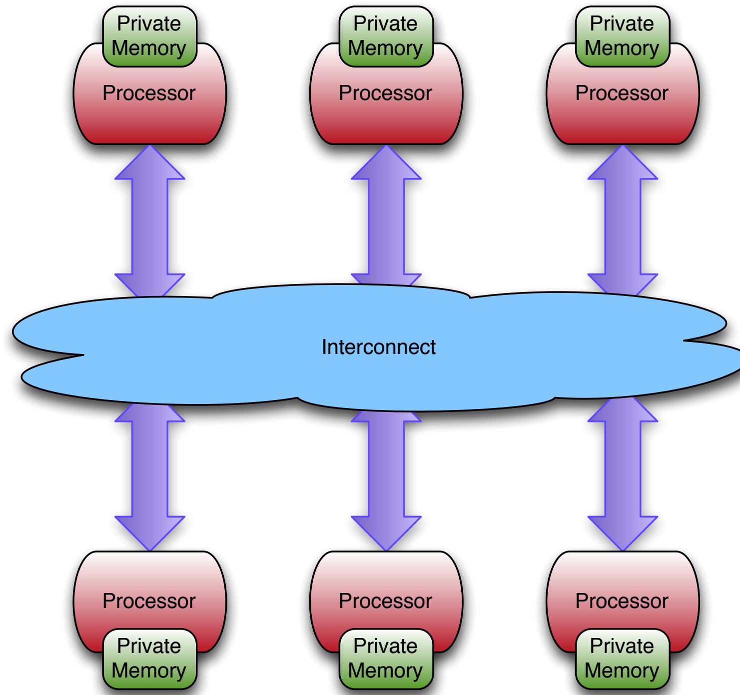

Figure 2.5: Distributed Memory Architecture - local memory is now private

MPI does not address how does one properly distribute a program's given problem amongst all processors. Data partitioning must be done by the programmer to ensure distributed placement of data within processor's private memories. This entails exposing the underlying memory hierarchy to the programmer. However, this can have negative effects because code written for these architectures may require explicit memory addressing to unlock peak performance, reducing application portability. Therefore, in absence of the shared memory model there has been several approaches for abstracting this memory hierarchy. One approach is to provide a programming environment such as NVIDIA's Compute Unified Device Architecture (CUDA), OpenCL or Sequoia [7] to aid in programming effort. Another approach is to provide the programmer either through software or hardware the illusion of a linear address space similar to DSM architectures. Regardless of what approach is taken, ensuring data locality is important to realize for gaining the scalable performance of any distributed memory architecture.

2.4 Related Work - Distributed Memory

Increasing data/instruction locality is critical to enabling performance in distributed memory systems [11, 15, 7]. As a result, much research in this area focuses on two key factors that can influence data locality: placement and remote accesses to data. Memory coherency protocols is also a relevant challenge as it focuses on data locality. Placement of data refers to optimally placing referenced memory regions as close as possible to the requesting processor(s). Without profiling, it is difficult to predict the behavior of the program to identify the best-case static data placement. During runtime, data can be replicated and/or migrated to achieve better processor affinity. However steps must still be taken to mitigate any additional latencies that may result from migration of data to different memory banks. Furthermore if replication is desired, the availability of free local memory space may be limited at the processor level.

Remote accesses refers to a processor referencing a memory location that does not map in the locally attached memory bank. Therefore, decreasing the amount of remote accesses depends on data placement. In the cases when it cannot be circumvented or the latency involved is less than placing the data locally, additional hardware extensions or prefetching are typically used to service these remote accesses. Ultimately, for both data placement and remote access, optimal problem decomposition and algorithmic restructuring cannot be achieved if the underlying memory hierarchy is not understood by the programmer. This is why DSM architectures have been successful according to [15] as the programming cost is cheap and readily allows many to adapt much legacy code written for SMP architectures with minimal effort while inheriting the benefits of scalable parallelism. It is this reason why many DM architectures implement virtual memory.

Virtual memory allows these disjointed memories to be seen as one linear address space, even though they may have non-contiguous address ranges. Chen et. al. in [6] observes that many implementations of DM architectures with hardware support virtual memory contribute the entire address range of one particular memory to the global linear address space. Their argument here is if local memory was divided into shared and private regions, there may be some performance

gains to be achieved. This is because accesses to this part of memory would not suffer any additional latency given by virtual memory translations, as addressing these memory locations would be done by physical addressing resulting in faster access times compared to logical addressing. Furthermore, their research efforts have also allowed dynamic runtime partitioning of these two memory regions for each processor's memory. They argue that different parallel applications contain different *data properties*. Data properties in this sense details how much of that data is *only* read/written to by that processor (private memory) and how much of that data is read/written to by multiple processors (shared memory). This ratio may change when running different applications or possibly during the execution of the same application. Therefore, they allow the programmer to redefine the two memory boundaries during runtime of the system to maximize the storage of private data within the private memory region.

Most of their implementation was done in hardware to ensure fast decision making when determining local or remote accesses for the particular processor. This implementation included a Network-On-Chip (NoC) mesh supporting 16 Processor Nodes: each node containing a processor with instruction/data caches, a local memory, a programmable memory controller, and a Network Interface Controller (NIC). The local memory is a dual-ported memory that allows servicing remote and local memory requests simultaneously if need be. The included programmable memory controller allows for run-time reconfiguration to dynamically change the sizes of both the private and shared memory regions. This, in addition to their virtual-to-physical (V2P) translation logic, is capable of servicing concurrent local and remote requests. This allowed them to achieve a maximum of 34.42% performance gains in some of the benchmarks used.

Vuletiȁ et. al. in [17] similarly implemented hardware accelerators for virtual-to-physical address translations. Coupled with OS support, their efforts included hardware/software mechanisms for transparently interfacing between both user application and co-processor, and co-processor and user-memory space. This approach enables portable and accessible user application code that can take advantage of co-processors, and system independent HDL code for co-processors. The hardware accelerators implemented provide virtual memory support for both processor (user ap-

plication) and reconfigurable device (co-processor). This lessens programming complexity by not requiring programmers to perform explicit memory-to-memory transfers but rather, pointers to data in the mutually viewable address space. Furthermore, virtual memory does not require hardware designers to manage explicit memory addresses when designing co-processors, thus aiding in accelerator portability and modularity. Fortunately, due to the common view of a shared memory space for both applications and co-processors, the communication between the two is done implicitly. Their OS extensions allow the programmer to write high-level functions passing data whereby the OS forwards this data to the co-processor (if available) via the shared memory space. To guarantee memory consistency, the processor must stall during execution of the co-processor. The processor also services page faults for the co-processor for references to virtual memory locations not found in the co-processors Translation Look-aside Buffer (TLB). As discussed in [18], it was soon realized that the idle time for the processor could be used to prefetch data to eliminate page faults. More importantly, prefetching could mitigate any memory-to-memory transfer latencies through online profiling of memory access patterns. Due to this approach, the processor can optimistically fetch consecutive pieces of data, i.e. looping over an array. This technique albeit not a new concept, helped reduced a generous amount of overhead attributed to copy and management time of virtual memory that plagued earlier revisions of this project.

Resource partitioning takes a different approach to that of dynamic scheduling and/or data migration. The approach involves partitioning an optimal number of hardware resources for a particular application (i.e. the number of processors per application, the amount of memory space given/needed, or both). The optimal number of resources is typically determined through first profiling the target application. Resource partitioning can be very beneficial for energy savings as well. In [13], Macii et. al. profiled applications to identify memory access *hot spots*, or addresses that are frequently requested. This allowed a single memory containing the application to be partitioned into smaller, less energy consuming memories by placing frequently referenced addresses (and their associating data) within them. Macii referred to this as address clustering and observed average energy savings of 25%.

Xue et. al. in [26] argued that in multiprocessor environments where there may exist several applications running, partitioning of both computation and storage resources is of equal importance to obtain the best results. Allocating more resources to computations such as providing additional processors to a problem may not provide more speedup past a certain point. This is an inherent limitation of all applications due to some percentage of sequential code/instructions. This type of sequential code manifests in applications that engage in heavy synchronization. Clearly, allocating more memory to an application than is needed will also not net in any further speedup and can negatively impact other applications in need for more memory. Therefore, the ideal memory requirements are also found to avoid this scenario. Based on this research effort, they are able to exploit this flexibility (dynamic partitioning) during runtime whenever an application is introduced, changed, or removed from the system. An arguable disadvantage is that this capability is only achieved through offline profiling.

Computation migration described in [11], is the process of moving the computation to the data unlike data migration which involves moving data to the computation. Computation migration is very similar to *thread migration*, a common technique for load balancing purposes. Hsieh et. al. defines computation migration as *partial* thread migration. The two differ only in that computation migration considers the first few activation frames of a thread. In other words, the context of the currently executing subroutine(s) would only need to be transferred to the processor closest to the data as opposed to the entire thread stack. Computation migration can be beneficial if the size of the data set is large enough to warrant the transfer. Also, it can be advantageous in the case of frequent writes to shared data due to the localized memory accesses that would occur. Cache-coherent systems exhibit data migration tendencies but can suffer in performance if the amount of traffic needed for the memory coherency protocol introduces larger latencies compared to computational migration.

For computation migration to work in [11], they make use of client-server stubs placed at compile time between function calls annotated for computation migration. That is, a processor currently requesting computation migration (client) towards a currently free processor (server),

first contacts the requested processor to initialize a special *continuation procedure* to handle the incoming computation migration. However, an underlying assumption is that the system must be homogeneous. A heterogeneous environment can exist using their techniques, but computation would have to be restricted within clusters of the same ISA. To accommodate this restriction, they propose that multiple ISA specific client-server stubs can be placed between function calls annotated for computation migration. Unfortunately, this introduces bigger applications that must be stationed in close proximity to the processor.

Chapter 3

System Design

A critical issue in designing memory systems is enabling ease of programmability while supporting performance scalability. Sections 2.2 and 2.3 discussed how abstracting physically disjointed memory banks within a shared memory model can provide a simple view of memory for programmers. Under shared memory models, programmers can write code that is both portable and efficient. Application speedup is provided through the underlying hardware, OS, and program decomposition. The underlying hardware and OS can provide and exploit implicit locality for both instructions and data. However, attempting to abstract away too much of the implementation details of complex memory hierarchies can prove to be counter productive to achieving good scalable performance. The architecture proposed in this thesis combines a global shared memory with private memories that can support efficiency through runtime data migration.

The proposed architecture is an extension of earlier SMP systems that evolved from the original Hthreads multithreaded programming model. An important aspect of this work was to seamlessly extend the Hthreads API's over the new architecture to retain continuity for programmers as well as providing support for use of distributed memory. An overview of the original Hthreads run time system and SMP implementations are provided in the next section. The design of our new distributed shared memory systems is then presented. We outline the evolutionary development of these systems, which resulted in our split BRAM organization. We then present modifications to the compilation flow that were made to enable Hthreads programs to be run on our new Split-BRAM systems. Finally, we discuss the modifications made to Hthread's run time system to allow programmers to seamlessly run multithreaded programs developed for SMP systems on our Split BRAM systems.

3.1 Brief Intro to Hthreads

HybridThreads (Hthreads) is a hardware/software co-design microkernel operating system that provides a unifying abstraction for combinations of threads running on processors or as custom hardware circuits. Software threads are threads that execute on the main/host processor(s). Hardware threads originally were defined as threads executing on custom user hardware written in HDL or using HLL-to-HDL translators. The Hthreads system investigated replacing custom hardware circuits with programmable processors to improve design efficiency. Ultimately, the notion of hardware thread was extended to encompass software threads running on general purpose soft processors that replaced the custom circuit hardware thread. The result of this effort was enabling Hthreads to abstract heterogeneous systems, or systems with different processor types.

Both software and hardware threads adhere to the same uniform set of policies that simplifies communication and coordination between threads, and between a thread and the OS. Having key OS services implemented within hardware was critical in achieving desirable ISA-agnostic communication between different types of slave processors. Although the mechanisms for system calls differ, the higher level user visible policies remains the same. The Hardware Thread Interface (HWTI), and the Virtual HWTI (V-HWTI) are abstraction layers for threads residing in custom hardware and slave processors respectively, to seamlessly interface into the Hthreads system [1, 2]. This allows portable HDL or general purpose processors to interface simplistically with Hthreads via several read/write registers. In the case of processors as hardware threads, the V-HWTI operates in co-ordination with a small kernel (4KB) called the Hardware Abstraction Layer (HAL). The HAL transforms Hthread system calls into simple load/store operations that access the Hthread's hardware cores. Using general purpose processors as replacement components for custom hardware circuits offers flexibility and increased designer productivity but at the expense of performance. Although we don't exclude the inclusion of hardware threads, this thesis focuses on memory organizations for slave processor based threads. As such the HWTI is not discussed further. It is worth pointing out that for applications that absolutely require performance associated

with custom circuits, the MicroBlaze used in this work as a slave processor offers 16 Fast Simplex Links (FSL) that can be used to support additional custom accelerators.

The following sections provide a high-level description of the two systems considered in this work. Section 3.1.1 gives an overview of the original heterogeneous Hthreads system. Sections 3.2.2 and 3.2.3 briefly walk through the issues faced and decisions made to achieve scalable distributed shared memory systems. The final version of the system is presented in Section 3.2.4.

3.1.1 Original Heterogeneous Hthreads System

The original heterogeneous Hthreads systems was defined to validate several ideas: runtime support of both software and hardware threads within a heterogeneous platform, how a hardware microkernel can provide ISA-agnostic services that offer better scalability compared to traditional remote procedural call (RPC) mechanisms, and to show support for hardware threads running on both user-defined hardware accelerators and general purpose processors. The original heterogeneous Hthreads system was built on the Xilinx Virtex ML507 evaluation board. This system was later ported to a Xilinx Virtex ML605 evaluation board. The Virtex 5 on the ML507 board contained 11,200 slices and 71,680 logic cells. The Virtex 6 on the ML605 increased slice counts to 37,680 and logic cell counts to 241,152 [23, 24]. A noticeable difference between the two boards is the removal of the PowerPC processor. However this modification did not effect the Hthreads hardware based microkernel cores, or the ability of the system to resolve issues of processor heterogeneity.

The Figure 3.1 shows a high level overview of an earlier SMP structured heterogeneous system. This system includes a host processor and several slave processors that run as *hardware threads*. As explained in 3.1, hardware threads could also be represented as user-defined hardware accelerators. Each processor within the system uses an instruction cache (direct mapped) with a default cache size of 8 KB (cache line size is 4). The data cache is intentionally turned off due to the lack of required cache coherency protocol circuitry [5, 19]. Instead data accesses occur over the Processor Local Bus (PLB) which is considerably slower than the XCL bus [19]. This system supports a

maximum of 6 slave processors and 1 hard/soft host processor. When configured with 6 slave processors, this system consumes about 70% of slices on the ML507 and 35% on the ML605. Due to the SMP structure of slave processors, all memory accesses are symmetric. As a result of each directly connected to shared memory, bigger designs are not possible to build unless the MicroBlaze's IXCL is removed for all of the processors. This is due to the current 8-port limit on the Xilinx Multi-Port Memory Controller [5] and not a limitation of the threads system itself.

3.2 Distributed and Shared Memory Design

To overcome the port limitations of the MPMC a new Hthreads system was built that augmented the shared global memory with multiple local scratch-pad memories using Block RAMs (BRAMs). This hybrid system incorporates the use of distributed memory at the slave processor level while maintaining a global addressable memory usable by other processors throughout the system. Support was then implemented within the OS to transparently and efficiently transfer instructions and associated data to a targeted slave processor's local memory. This new architecture was able to abstract processor heterogeneity at both the slave and host processor level. This was verified on an ML507 board by building two different systems; on which used the PowerPC as the host processor and MicroBlazes as the slave processors, and the second that replaced the PowerPC with a MicroBlaze as the host processor.

3.2.1 Configuring Bus/Local Memory Components

With the availability of ports on the MPMC limited to 8, there was a need for a new system that was not reliant on caches but continue to provide cache-like performance. To add to this, we were also interested in determining how the memory organization would effect an MPSoPC system composed of heterogeneous processors. The Xilinx Embedded Design Kit, v.10.1, only provides two interface options for the MicroBlaze to achieve Xilinx Cache Link (XCL) speeds: the Fast Simplex Link (FSL) and the Local Memory Bus (LMB).

The Instruction and Data XCL interfaces use the FSL links [19]. However, Xilinx extends

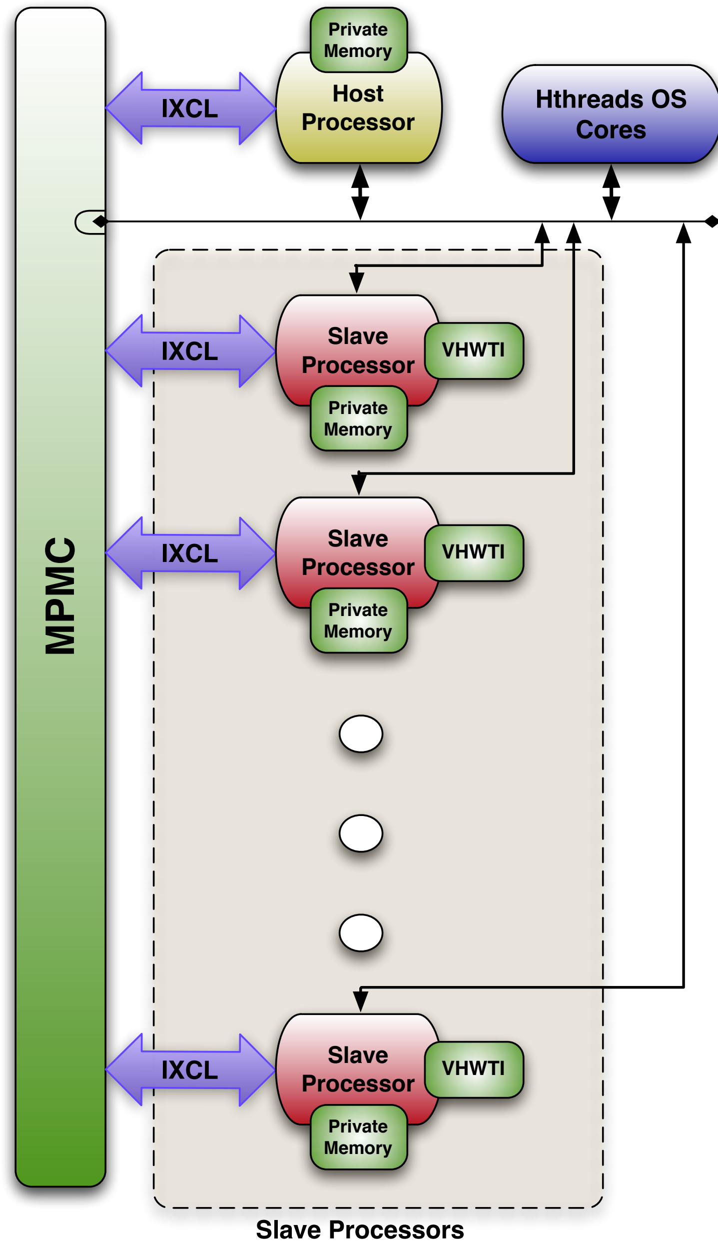

Figure 3.1: Hthreads SMP System

the FSL to allow tight integration between the MicroBlaze and the MPMC for instruction/data fetching to feed directly into the 3 or 5 stage pipeline. Xilinx also provides several application programming interfaces (APIs) for both the FSL and XCL interfaces. For the XCL, this allows the programmer to perform a *limited* number of cache operations such as enabling and disabling the cache, invalidating/flushing out cache lines, updating cacheable address ranges, etc. Similarly, there are several API calls for the FSL interface to allow the user to explicitly transfer/receive data over the MicroBlaze's 16 FSL ports. This presents a difficult challenge when attaching an FSL bus between a MicroBlaze and a BRAM block containing instructions/data. Unlike the MicroBlaze's PLB, LMB, or the On-chip Peripheral Bus (OPB), there are no automated mechanisms to allow instruction/data fetch to occur over the FSL link as well as to be supplied directly into the pipeline. To overcome this problem, data would have to be stored over a memory-mapped I/O (MMIO) device, or the MicroBlaze would have to be connected to the MPMC using the XCL ports.

The second and more promising alternative is to use the Local Memory Bus (LMB). Traditionally, MicroBlaze reference systems reserve this interface to access private memory for fast *single* cycle access to the heap, stack, program instructions, and the interrupt/exception vector table. The LMB uses a very light weight Point-to-Point (P2P) protocol. Conversely the PLB was defined to support many-to-many connections with the use of a bus arbiter to enable multiple bus masters. Therefore, there was a need to have instructions and data located within the private memory to allow execution over the faster LMB interface. The MicroBlaze includes the Harvard Architecture found on its PLB (and the older OPB) interface as well as its LMB interfaces. Thus the LMB provides separate ports for the Data LMB (DLMB) for data access and the Instruction LMB (ILMB) for instruction accesses. Attaching a MicroBlaze to a dual ported BRAM block using both DLMB and ILMB would restrict access to only the attached MicroBlaze. This would result in additional complexity integrated into the nanokernel. Therefore, access to these private memories were still needed to provide instruction and data transfer using a Direct Memory Access (DMA) controller from global memory to the private memories. Only after such data is transferred would the MicroBlaze begin fetching instructions and data across the fast access LMB.

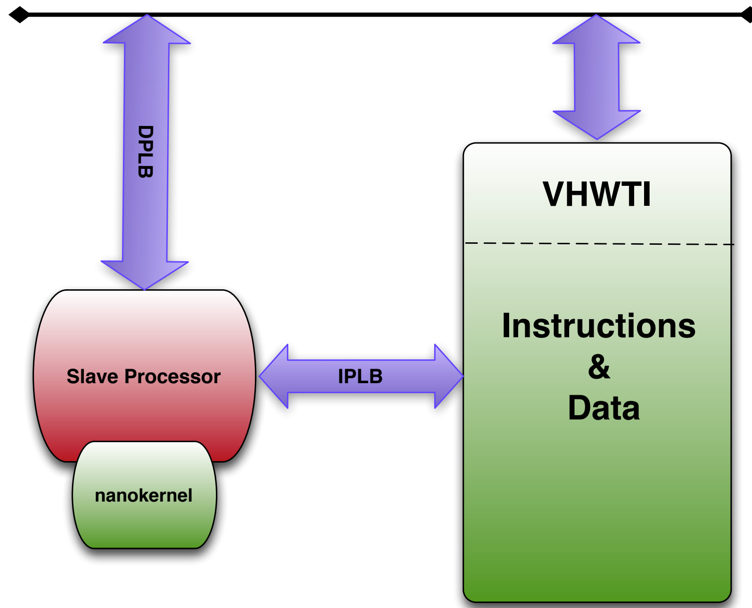

Figure 3.2: IPLB configured for P2P

3.2.2 First Approach - Enabling P2P

Our first attempt explored defining a Point-to-Point (P2P) connection between a single slave and a single master on a PLB. The purpose of this exploration was to use an existing BRAM that is generally attached to the slave processor's bus. This BRAM housed the V-HWTI occupying less than 100 bytes. As per the Xilinx PLB specification, if two master devices exist on a PLB, there is a three cycle delay for priority arbitration. Enabling P2P mode removes the need for priority arbitration, thus reducing area and latency (one clock cycle less) [22]. A slave device responds to all *Valid* PLB signals as the assumption here is there is only one master in P2P mode. As a result of these optimizations, the best case latency for a master being able to *master* the bus is two clock cycles [22, 25]. Whether a slave can combinatorially respond within the same clock cycle (second clock cycle) or not may further increase that latency. In the case of a PLB BRAM memory controller, an additional clock cycle is needed for write and two clock cycles for a read operation.

Figure 3.2 shows how the slave processor was configured to make use of the P2P PLB connection. In this configuration, the processor is able to master two busses simultaneously over both

its Instruction PLB (IPLB) and Data PLB (DPLB) interfaces. However, since the IPLB is directly attached to the slave device (V-HWTI BRAM), it is configured as P2P. The DPLB interface is indirectly connected to the V-HWTI as shown. Initially the small kernel, namely the *nanokernel* referenced earlier in Section 3.1, was placed inside the V-HWTI BRAM to minimize SRAM usage as much as possible. This would allow the removal of the private memory that is attached to the slave processor. Generally for the MicroBlaze, the nanokernel is placed with an address region beginning with *0x0000 0000*. Therefore, this address region would be a part of the MicroBlaze's memory map over the IPLB and DPLB; hence, both ports of the V-HWTI BRAM began with *0x0000 0000*. This presented a problem as it did not allow for a unique address for the V-HWTI. This unique address is needed by the Hthreads OS in order for thread creation. If having more than one MicroBlaze in the system, the address would prove to be too generic to perform MMIO to it. Therefore, the addresses over both ports of the V-HWTI had to be different.

The motivation for using one large BRAM block rather than the traditional V-HWTI BRAM and private memory found in the Hthreads SMP system, was to keep area utilization relatively low as well as on-chip memory. Instantiating smaller memories whereby accumulation, equates to a single large memory block should utilize roughly the same memory resources. However, this approach would also add additional memory controllers ultimately adding more slaves onto the PLB. More slave devices on the bus translates to a bigger PLB arbiter, additional bus paths, bigger multiplexers, etc. In choosing a single BRAM to house both V-HWTI and a thread's instructions and arguments, execution of those instructions did not work reliably. This is due to the non-overlapping address range for the V-HWTI BRAM across the MicroBlaze's DPLB and IPLB ports. Although allowed, Xilinx discourages this practice over the same interface (i.e DPLB and IPLB, ILMB and DLMB). Therefore, a private memory was kept in order to execute the nanokernel within the lower *0x0000 0000* address range while the V-HWTI BRAM was used to store the the V-HWTI registers as well as free space for transferred data and instructions.

This approach was shortly discarded after comparing it's performance with a Hthreads SMP system. Data intensive programs provided a little speedup over the SMP system but for more

Compute intensive programs, this approach was slower. Although there is no contention on the IPLB port, the best case for mastering the P2P bus is 3 clock cycles per instruction; a 2 clock cycle read can be overlapped on the last cycle when mastering the bus. This latency extends to 4 clock cycles when performing data loads; 3 clock cycles to master the bus and 2 clock cycles to read from the BRAM with overlap of last clock cycle of addressing phase [19, 22, 25]. Also worth noting, the MicroBlaze does not support Burst Mode over the IPLB (nor the DPLB). That is, all operations must use single beat writes/reads. Therefore, there is no pipelining of loads/stores to the BRAM requiring arbitration for each data access and the performance would continually decrease due to increased bus contention. Therefore, this platform would not provide scalable parallelism within increased processor counts.

3.2.3 Second Approach - Split-BRAM

The move from the first to the second approach was a radical change, and Figure 3.3 sheds more light on the phrase: *Split-BRAM*. There are three BRAMs that are shown: two of which are directly connected to the slave processor and one indirectly connected to it as well. The two that are directly connected to it are attached via the LMB to the MicroBlaze's DLMB and ILMB ports. As mentioned earlier in Section 3.2.2, both are given the same address range in order to ensure correct functionality. Additionally, the MicroBlaze is attached to a local PLB to allow general access to other memory-mapped devices over its DPLB port. Similarly, the IPLB is also attached in order to allow the execution of instructions outside of the LMB ports address range. As a note, the connection of the IPLB is unnecessary if we are assuming execution of instructions always occurring over the MicroBlaze's LMB ports. The slave processor's V-HWTI is located within a separate BRAM that is attached to the slave's local PLB. This is a very similar placement as done in the Hthreads SMP system. However, this V-HWTI BRAM differs in that it does not store the V-HWTI for a single processor but rather, for all processors located on that particular bus. This does not imply one needs to have multiple processors on a single bus. Not shown, is the V-HWTI BRAM's second port that is attached to the main bus in order to 1) avoid writing and reading

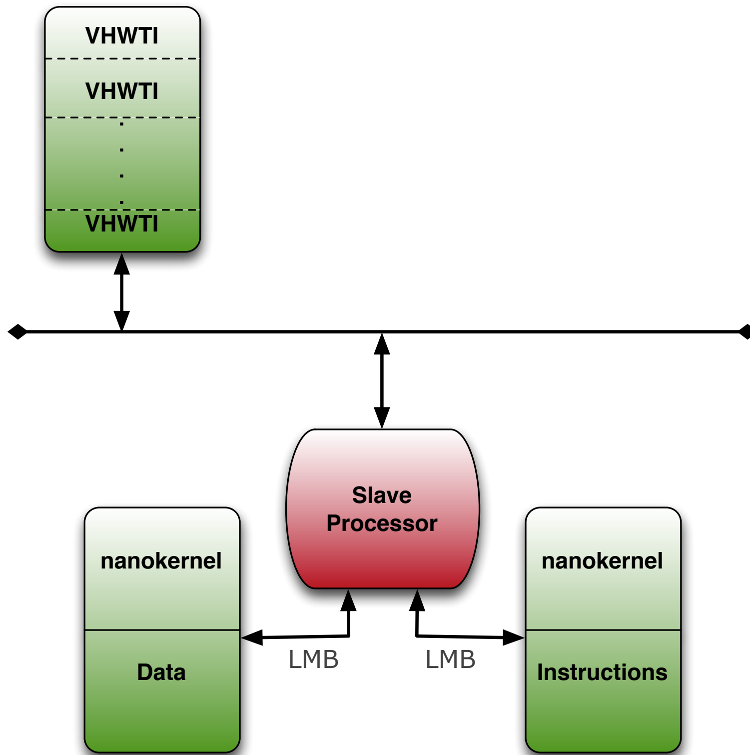

Figure 3.3: Split-BRAM with multiple V-HWTI's on common bus

across too many PLB-to-PLB bridges (expensive for single beat operations over the PLB bus), and 2) to avoid producing more traffic on the slave processor's bus. Similarly, the two BRAMs directly attached to each slave processor allow an extra port each, thus making it possible to place new instructions and data within these BRAMs while being simultaneously accessed over the other port via the LMB by the slave processor. For each of the two BRAMs, this enables the designer to assign a unique address to one port for data migration purposes, and a more generic address (i.e. *0x0000 0000*) to the other port for which the faster LMB can be used.

Examining the two directly attached BRAMs further, both are divided into the nanokernel and Instruction/Data. The nanokernel appearing twice is a side effect of the Xilinx tools for having an overlapped address range across two physically disjoint memories within the same interface (i.e. DLMB and ILMB interface). As mentioned previously, this overlapping is needed to ensure reliable program execution but also presents us a simple uniform address range where data migration

```
00: _STACK_SIZE = DEFINED(_STACK_SIZE) ? _STACK_SIZE : 0x400;
01: _HEAP_SIZE = DEFINED(_HEAP_SIZE) ? _HEAP_SIZE : 0x0;
02:
03: /* Define Memories in the system */
04:
05: MEMORY
06: {
07:    /* Possible length for Private memory == 0x4000 == 16KB */
08:    private_memory : ORIGIN = 0x00000050, LENGTH = 0x00000FB0
09:    shared_memory  : ORIGIN = 0xA0000000, LENGTH = 0x10000000
10: }
```

Figure 3.4: Default Linkerscript was modified to report a shorter length for "private_memory"

for both instruction and data occur within the two BRAMs. As one can gather from Figure 3.3, instructions transferred during runtime are placed within one of the BRAMs that are then fetched over the ILMB interface. Similarly, data is placed in the adjacent BRAM and fetched over the DLMB interface. Currently, this free space has a maximum of 12 KB when using a system with 32 slave processors on the Xilinx ML605 Evaluation board.

The MicroBlaze ABI dictates that the stack is always implemented in the highest address range and grows to a lower address range [19]. Therefore, the default linkerscript had to be modified in order to force the stack not to overlap with the address space reserved for Instructions/Data, but rather placed right before that boundary. That is, the address region where the nanokernel is specified must be told it is of a size that is not equal to the entire possible address range of that physical memory, but rather the ceiling of the size of the nanokernel nearest the 2^n boundary. This was solved by reporting a shorter length of that BRAM block to the linkerscipt as shown in Figure 3.4.

These systems were named with the prefix, *shared_vhwti* due to several slave processors sharing a BRAM block that housed several V-HWTIs. Although the slaves now have fast access to their instructions and data, this system still suffers in performance due to the V-HWTI BRAM that can be shared amongst 2-8 slave processors on one local bus. According to IBM's PLB and Xilinx implementation of it, each master device has its own address, data read and data write path [22, 16, 12]. Slave devices on the PLB also include the three paths as well but are attached to shared, decoupled busses (i.e. shared address path, shared data read path, shared data write path). This

indicates that the PLB can support different parallel transactions across the three. For example, data write and data read can begin sequentially but can execute/complete simultaneously thereafter. Within Hthreads, each slave processor constantly polls its V-HWTI when waiting for a thread to be scheduled onto it. More specifically, each master (processor) on that bus address the same slave device (V-HWTI BRAM), and begin a data read (load word, non-burst read). Again, to master the PLB requires at least three clock cycles of the arbitration cycle or addressing phase. Currently, Xilinx implementation of IBM's PLB bus only allows two deep address pipelining. Therefore, a secondary master can be queued without having to enter the address phase again. However, if there exists three or more masters, these devices have to re-enter the address phase repeatedly each time the primary and secondary spots are already filled on the bus. This potential bottleneck is not only exemplified for hardware thread creation but any blocking OS calls such as mutex lock as processors currently perform a spinlock on its V-HWTI in order to determine if the current thread should resume. One way to overcome this bottleneck is to use interrupts in order to signal slave processors to read the VHWTI; however, this would require additional logic to be implemented into the scheduler and/or require the host to monitor *all* V-HWTIs for each slave processor. However, there exists a easier solution which is explained in the following section.

3.2.4 Third Approach - Placing the V-HWTI in Private Memory

The third approach continues to exploit spatial locality with respect to both instructions and data, but now acquiring that characteristic over the V-HWTI interface. As shown in Figure 3.6, the second and third approaches are very similar with one slight difference: the location of a slave processor's V-HWTI. The shared V-HWTI BRAM is removed from the local bus and each slave processor's V-HWTI is placed within the Data Memory for fast, undisturbed polling of those set of registers. This proximity is very similar to the HWTI described in [2]. The V-HWTI is placed on the data side because no instructions are ever executed from it, hence no data loads would occur over the ILMB interface but over the DLMB interface.

The linkerscript was further modified (Figure 3.5) from the last approach to handle this already

```
00: _STACK_SIZE = DEFINED(_STACK_SIZE) ? _STACK_SIZE : 0x400;
01: _HEAP_SIZE = DEFINED(_HEAP_SIZE) ? _HEAP_SIZE : 0x0;
02:
03: /* Define Memories in the system */
04:
05: MEMORY
06: {
07:    /* Possible length for Private memory == 0x4000 == 16KB */
08:    private_memory : ORIGIN = 0x00000050, LENGTH = 0x00000FB0 - 0x30
09:    shared_memory  : ORIGIN = 0xA0000000, LENGTH = 0x10000000
10:    vhwti          : ORIGIN = 0x00000FD0, LENGTH = 0x30
11: }
```

Figure 3.5: Linkerscript accounting for V-HWTI in ”private_memory”

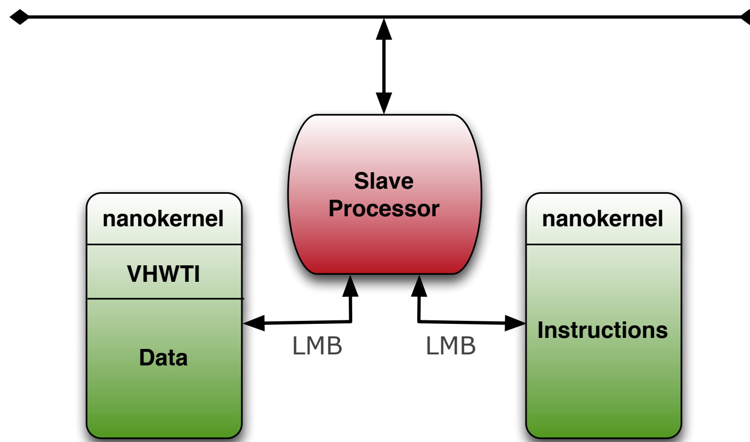

Figure 3.6: V-HWTI placed inside of private memory

subdivided physical memory. Not shown in Figure 3.6 is both of these BRAMs are partitioned into three sections but only the data side is modified during runtime. Since both Data and Instruction LMB ports overlap the same address range but over two BRAM blocks, Xilinx Synthesis Technology (XST) tool maps both BRAMs into 3 subsections: nanokernel, V-HWTI, Free space for Data/Instructions. Fortunately, both the V-HWTI and nanokernel are small enough to fit just below a 2^n boundary, allowing the beginning address of the free Data/Instruction memory space to remain the same from the second approach. Now that polling of the V-HWTI is channelized only through the LMB, this allows the flexibility of more slave processors to be placed on a single PLB without degrading performance due to bus contention. Depending on the application, however, the designer may want to experiment with different sizes of slave processor clustering. The final overview of this system is shown in Figure 3.7.

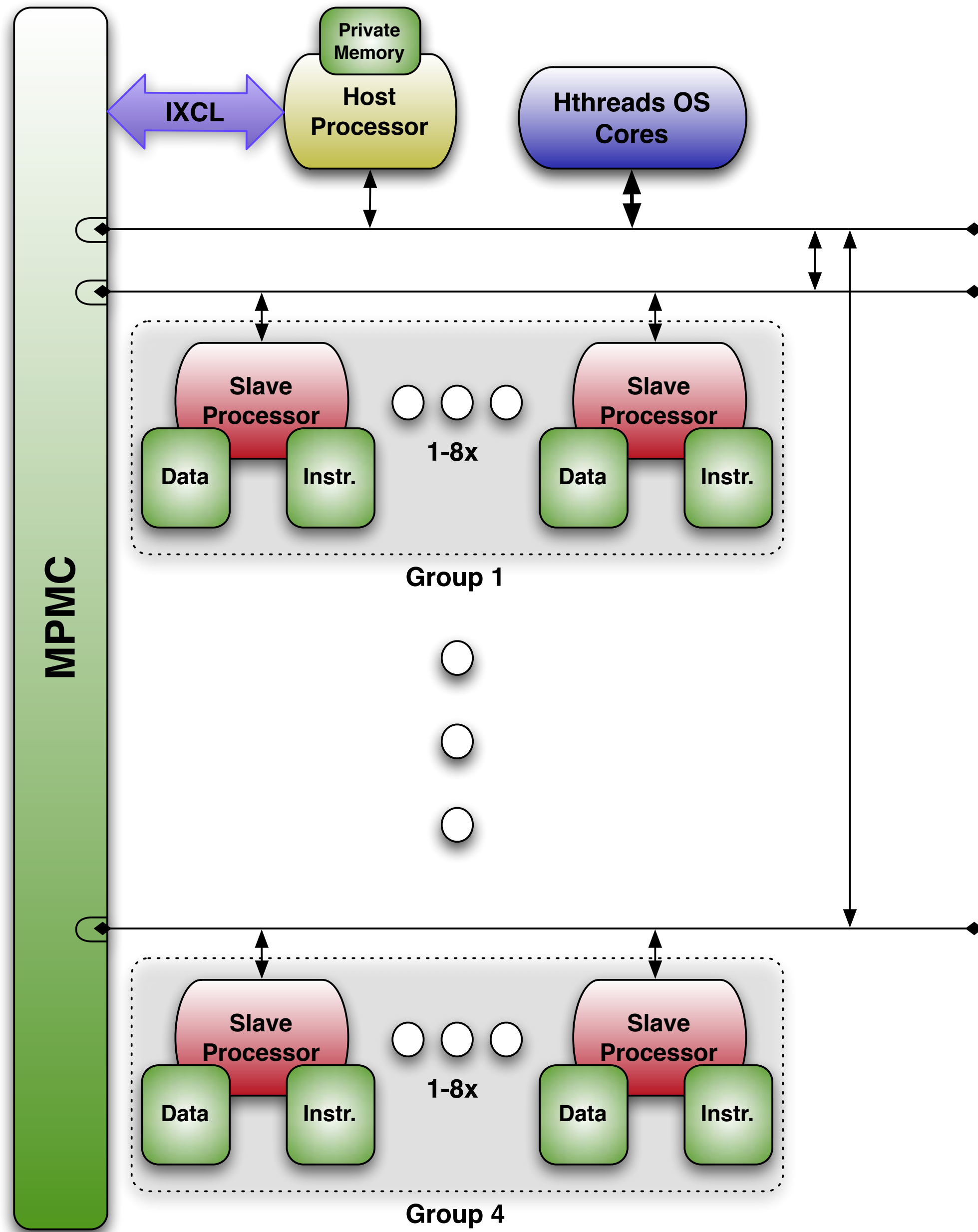

Figure 3.7: Full Split-BRAM System

3.2.5 Local Memory Initialization Issues

A problem faced when building the Split-BRAM systems on both the ML507 and ML605 Evaluation Boards were disappearing BRAM contents attached to the slave processors. To explain, Figure 3.7 shows several processors attached to PLBs. The host processor is attached to a bus (main bus) that allows its address map to span over all slave devices within the system. This address map also includes the two BRAMs (Split-BRAMs) directly attached to each slave processor due to their second ports attached elsewhere in the system to allow data migration during runtime (Section 3.2.3). This is advantageous for several reasons. First, thread creation involves the host initializing a thread's context within the global Thread Context Block (TCB). Additionally, some of this information including a Thread ID (returned from the Thread Manager), and the function address and arguments the thread will execute are also needed to be placed in a targeted V-HWTI for a slave processor. This is done through simple pointer addressing into the two BRAMs for a slave processor. Also, thread status such as currently blocked on a mutex request can be retrieved in the same manner. However, including the Split-BRAMs attached to each slave processor as part of the host processors memory map causes slave processors not to function.

Each slave processor is initialized at boot to execute the nanokernel that was downloaded as part of the programming of the FPGA. Having the host processor *see* inside of the Split-BRAMs causes this initialized data to be erased. When using the Xilinx Microprocessor Debugger (XMD) on the host processor, manually issuing a memory read across the Split-BRAMs revealed all zero's (Figure 3.8). This problem does not just manifest with the memory map of the host processor, but for slave processors as well. For example, when two slave processors are attached to the same bus with their address maps including both of their Split-BRAMs and the host processor does not include their memories in its address map, the first slave processor on the bus operates correctly but the second does not. For the second slave processor, its Split-BRAMs contain all zeros as shown in Figure 3.9.

These problems were solved by removing the Split-BRAMs out of any processor's memory map. Slave processors attached directly to its Split-BRAMs would continue to include it as part of

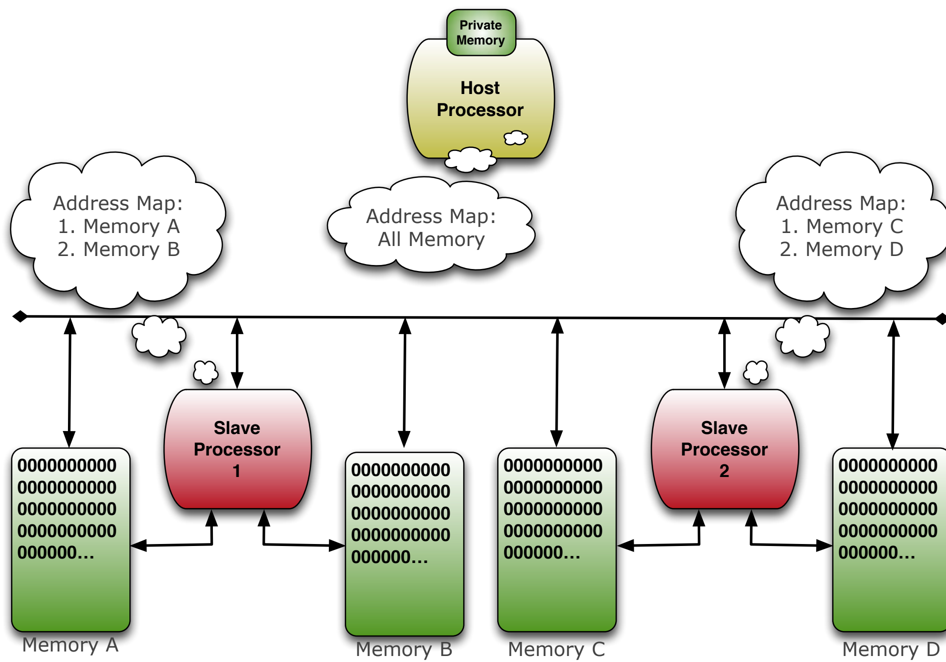

Figure 3.8: Host memory map includes all memories

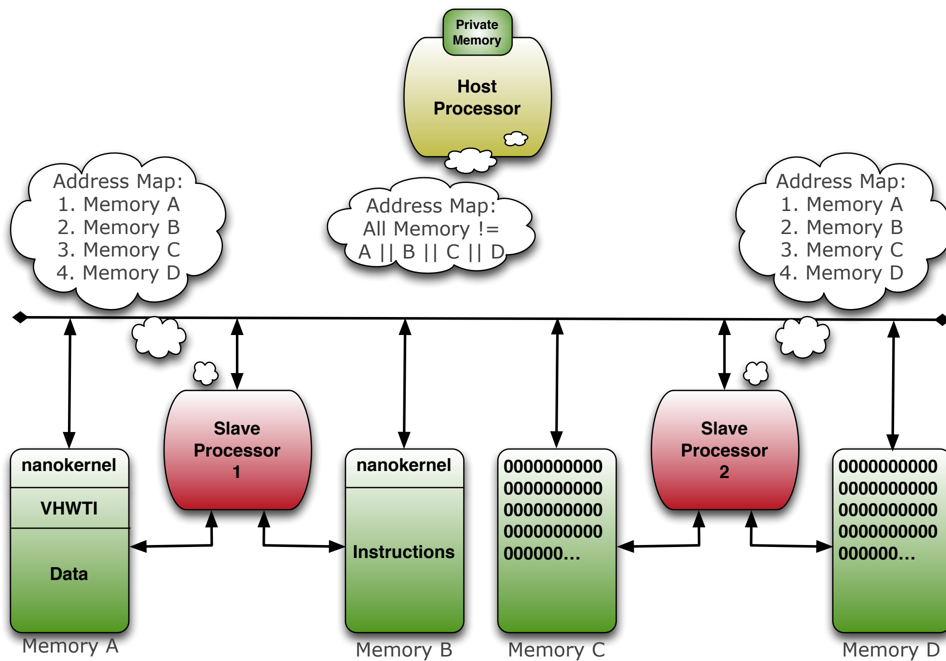

Figure 3.9: Slaves memory map include all other slave's memory

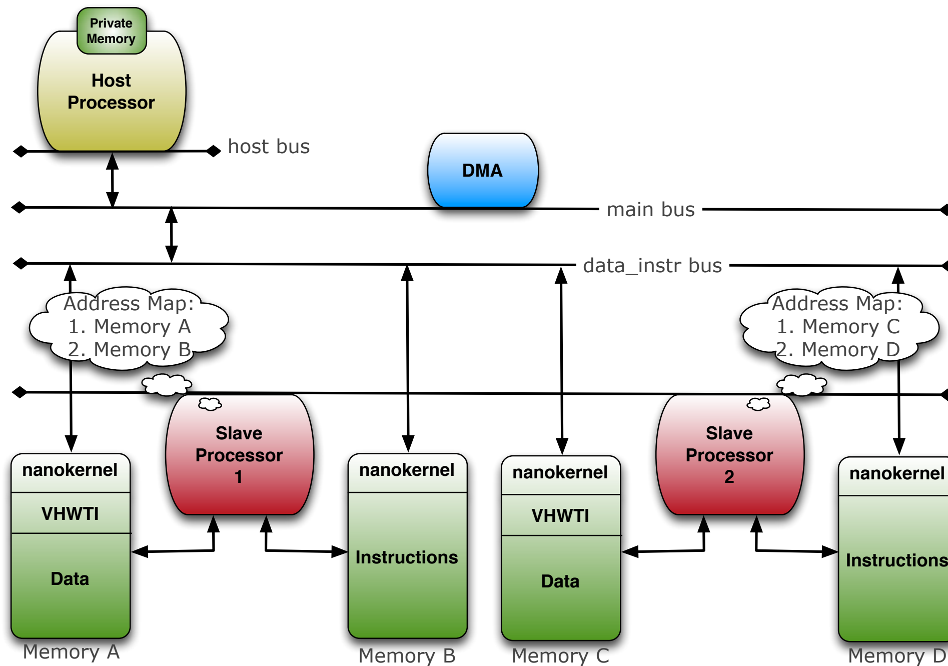

Figure 3.10: Final arrangement of Host and Slave processor

its memory map. This required placing the host processor onto another bus with a bridge between it and the main bus undefining any memory ranges belonging to the Split-BRAMs. Similarly, the second port of each of the Split-BRAMs were attached to a bus labeled as data_instr bus as shown in Figure 3.10. Due to the inability of the host processor directly writing to a slave processor's V-HWTI, the DMA controller was used as a proxy. This allows the OS to perform hardware thread creation as well as provide data migration through the DMA controller.

3.3 Software Development

As said in Section 3.2.1, the purpose for integrating data migration within the OS was to keep the nanokernel executing on heterogeneous slave processors simple. Therefore, the goals for distributed memory support within the Hthreads OS was kept simple: at thread creation time, the OS automatically transfers a hardware thread's instructions and arguments to the appropriate local memory (Split-BRAMs) all the while making it transparent to the programmer by both mitigating transfer latency whenever possible and not requiring explicit memory addressing. One assump-

tion that follows in this implementation is that the instructions and/or its arguments to which is transferred to the the Split-BRAMs do not exceed a size more than 12 KB. Future consideration of software-managed caching and other techniques to accommodate fixed-sized memories is discussed further in Section 5.2.

3.3.1 Heterogeneous Compilation Flow Modifications

Thread creation on a heterogeneous platform is achieved through an automated multi-pass compilation flow. Different ISA specific binaries are produced by compiling the original source file for said targeted architecture(s), flattening the binary to preserve memory layout, embedding the binary within the host processor's binary, and recording the offsets at which point within the embedded binary do *thread functions* begin. As the name implies, thread functions are functions that threads can target and they differentiate themselves from other functions only in their name declaration: thread functions contain the *_thread* prefix within their name declaration. Currently, the possibilities for realizing heterogeneity at the slave processor level includes MicroBlazes and user-defined hardware logic and at the host processor level, the PowerPC and the MicroBlaze. However, this compile flow allows a more diverse set of architectures when made available overtime. The heterogeneous compilation flow was created and automated by Jason Agron, PhD., and uses a combination of python scripts and the *GNU binutils* tools to achieve a single host binary image that includes heterogeneous instructions embedded within [1]. Figure 3.11 gives a high level overview of this compilation flow and shows how a programmer can write a single image source file, but target many different architectures during runtime. Naturally, the benefit of resolving different code bases during compile time (static compilation) outweighs other approaches such as just-in-time compilation (dynamic compilation) and virtualization/binary translation (emulation) since these occur during runtime introducing additional latency to the user's program.

Since the programmer writes a single source file for a heterogeneous system, it may seem plausible to transfer the entire binary image to each of the slave processors for execution. However, this may introduce a lot of unneeded instructions. One can reduce this by selectively transferring

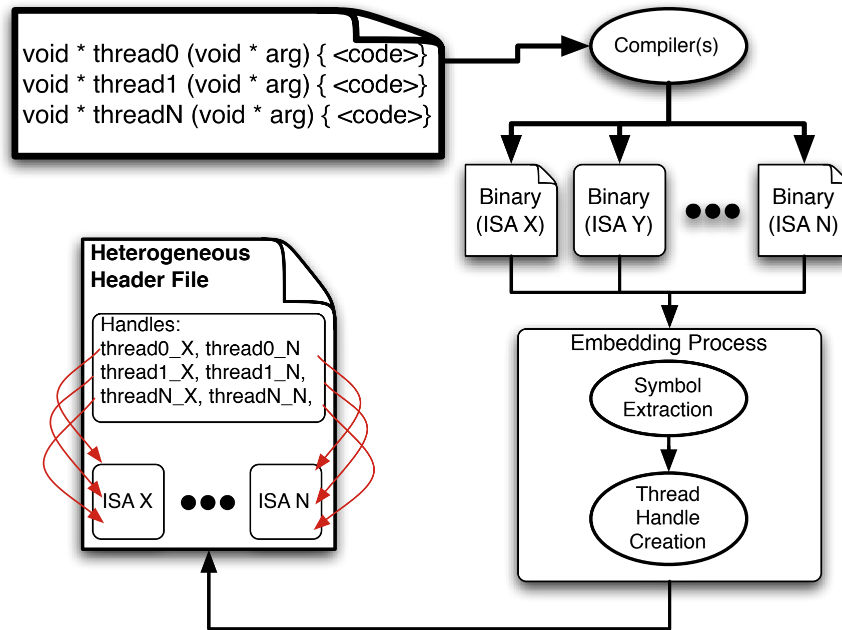

Figure 3.11: Embedding Process [1]

the ISA-specific instructions belonging to the thread function block and any other dependencies such as other functions it calls. Caches work on these same principles of fetching data on a need basis. Implementing a similar mechanism would involve profiling the intended program to capture a trace of instructions and data needed. It would also involve preserving the memory layout through flattening which would not help with removing instructions in between other needed instructions located at low and high address ranges. See Figure 3.12 for an example. The other alternative is at compile time, selectively taking out the targeted thread function to run on the slave processor, the main function (i.e. *int main()*, and any dependencies the thread function needs to create and compile a separate, possibly more concise source file.

To achieve this, additional stages were placed within the original heterogeneous compilation flow. After the original source file is copied into a ISA-specific folder (in this case, a folder labeled *MicroBlaze Hthread_hal*), it is compiled using the gcc compiler with additional debugging information (i.e. *-ggdb, -gdb*) under the targeted architecture. This is the point at which Figure 3.14 begins to show the additional stages. Next, each thread function is found by extracting from the symbol table of the original source file and a new source file is created for each thread function

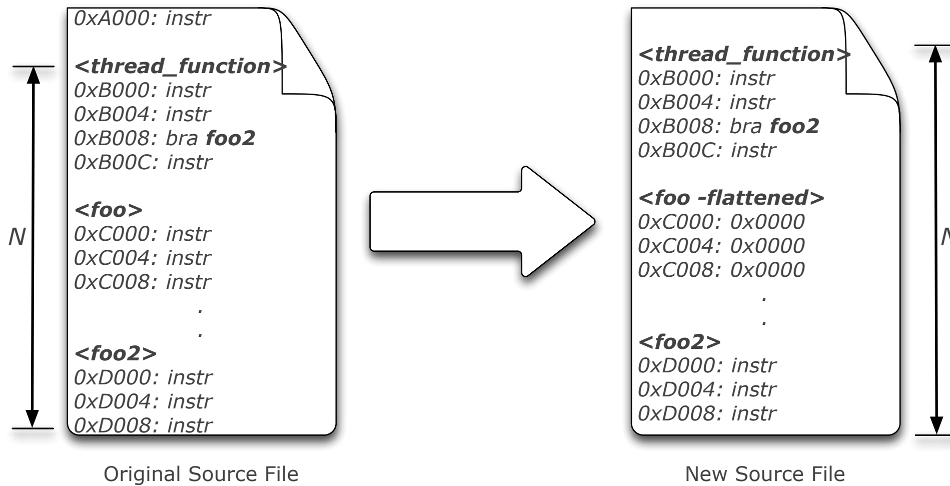

Figure 3.12: Memory space is preserved between *thread_function and foo2 but size remains the same*

to include itself, any dependencies that it may need, and the main function. Using the supplied debugging information and the symbol table from the ELF file in the previous step, one can quickly determine at which line in the file do thread functions occur using the GNU tools *addr2line* and *nm*. Since the C Programming language allows for function declarations to be written as multiple lines as done in Figure 3.15 and the *addr2line* tool reports only the beginning opening bracket (i.e., {) of the thread function, these cases needed to be solved. This was addressed by backtracking to previous line numbers within the original source file to get the absolute starting line number of a thread function declaration and not its start of the function definition/body. Next, one must remove comments from the actual source file. Although thread function locations within the source file can be collected as mentioned above, these locations express the beginning of thread functions and not the end. If comments are not removed beforehand, this step and the process of resolving any dependencies for the particular thread function may produce false positives. This can occur due to function calls and/or closing brackets written within single-line or multi-line comment blocks. Figure 3.13 gives an example where thread function *producer_thread* may incorrectly be resolved as having thread function *consumer_thread* be declared as a dependency in addition to incorrectly determining the point at which *producer_thread* ends on lines 5 and 6.

Once line numbers and dependencies are resolved for each thread function within the original

```
00: void producer_thread (void * arg)
01: {
02:     consume_data = 0xDEADBEEF;
03:
04:     /*********************************
05:     consumer_thread( consume_data )
06:     }
07:     *********************************/
08:
08:     while (more_data < consume_data)
09:     {
10:         more_data = produce_more_data_thread();
11:         .
12:         .
13:         .
14:     }
15: }
```

Figure 3.13: Thread function consumer_thread may be marked as a dependency

source file, a mask is created for each thread function that specifies from the original source file, which lines to write to the new source file (named after the thread function). These specified lines indicate the location of the thread function itself, any dependencies it may need, and the main function. Again, if there exists several thread functions within the original source file, there is an output of several thread function source files as shown in Figure 3.14. The final steps in this process is integrating back into the original compile flow: now that there exist additional source files for each thread function, it must now be compiled and embedded into the host processor's binary image as shown in Figure 3.11. As a result of multiple source files, there will be additional binary images embedded into the host processor's binary as simple character arrays. These data structures in addition to an offset extracted from the symbol table is used to point to the thread functions for a particular ISA. The processor includes this information through an automatically generated header file during the compilation.

3.3.2 Hthreads Kernel Modifications

For DSM architectures, instructions may begin overlaying across one or more memories within the shared memory space and eventually become duplicated by processor's (more local) caches. Therefore, there can exist multiple copies of instructions throughout the system. Depending on the cache properties, this can minimize instruction fetching from remote memories but also al-

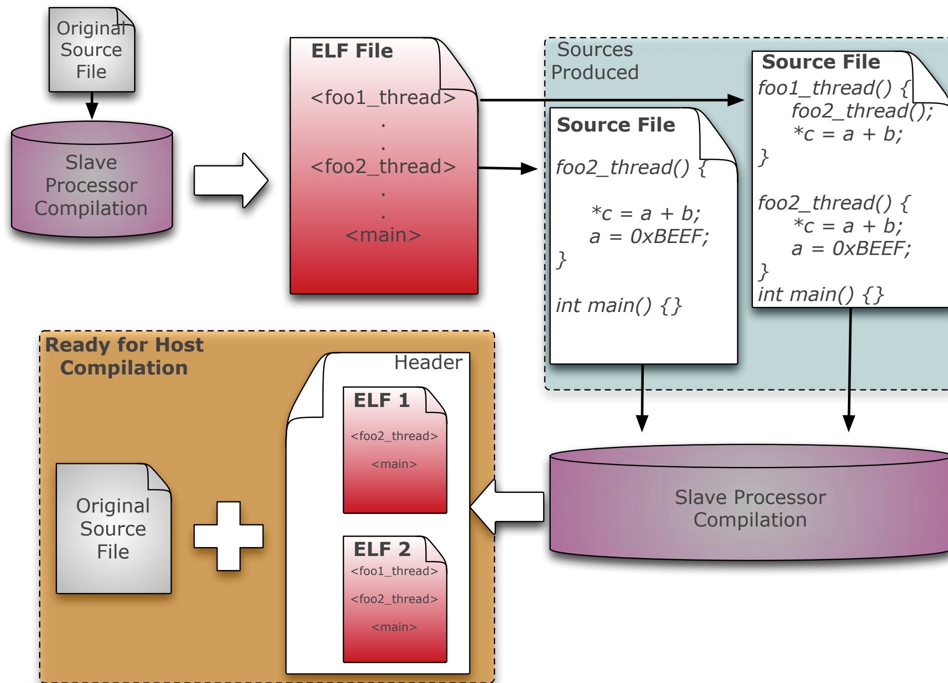

Figure 3.14: Illustrates the additions done to the original heterogeneous compilation flow

lows for data migration on needed instructions only. In order to realize this need-based fetching within a DM architecture that does not employ any caching mechanisms or a shared memory, the OS needs to know ahead of time what instructions slave processors execute. Implementing such functionality resembles very closely to software managed caching, and is beyond the scope of this work. Nevertheless, steps were taken in order to reduce the amount of instructions transferred to the Split-BRAMs during thread creation.

The first thing that was done in order optimize the system in this manner was to avoid retransferring the same instructions if it already exists at the targeted slave processor. As explained in Section 3.3.1, each thread function results in a new source file that contains the thread function, any other functions that may be needed, and possible definitions of data types and constants, etc. This source file's binary is ultimately embedded within the host code and located at a specific address within the shared memory space. When scheduling hardware threads, either dynamically or statically, a table is kept at runtime to keep track of previously transferred instructions to each of the slave processors. That is, a hardware thread's address, the previous start address of the sched-

```
void microblaze_create_DMA(hthread_t * tid, hthread_attr_t * attr,
                           unsigned int function_id, void * arg, unsigned int arg_size,
                           unsignedint dest_offset, unsigned int slave_proc)

void dynamic_create_DMA(hthread_t * tid, hthread_attr_t * attr,
                        unsigned intfunction_id, void * arg, usigned int arg_size,
                        unsigned int dest_offset)

void * hthread_join_DMA(hthread_t tid, void ** retval, void * arg,
                        unsigned int size, unsigned int offset)
```

Figure 3.15: Thread Create and Join calls that make use of the DMA

uled thread function (ISA specific), and a flag to indicate whether that hardware thread's address has been used is all managed by the host processor during runtime. This table is initialized at boot for the host processor and contains entries for all slave processors.

For a thread's arguments, determining if this data has been previously transferred using its start address alone should not be done. Unlike a thread's instructions which is viewed as read-only data, a thread's arguments may change during runtime but remain at the same memory location. Therefore, a decision based on start address of the data may lead to false positives/negatives. To reduce possible redundant transfers of thread arguments, additional parameters are given in a thread create API call allowing the programmer to specify an offset in which data should be placed at within the Split-BRAMs. This offset is needed as the OS always assumes the thread's arguments will be transferred at the beginning of the slave processor's preallocated free space within the Split-BRAMs. Therefore, if one has several contiguous data structures existing within this local memory, it can be specified to overwrite/update certain portions of this preallocated space by using a supplied offset. The two thread creation API calls for static and dynamic scheduling are shown in Figure 3.15.

When joining on hardware threads, there may be a need to transfer results back from the Split-BRAMs to shared memory. This can be done explicitly within the thread itself but would require added complexity and memory space for either the thread function or the nanokernel. To aid the programmer in transferring results back to shared memory, the call *hthread_join_DMA* was created to alleviate the burden of explicit memory addressing. For similar reasons mentioned in the previous paragraph, this call also provides an offset in scenarios where data needed does not

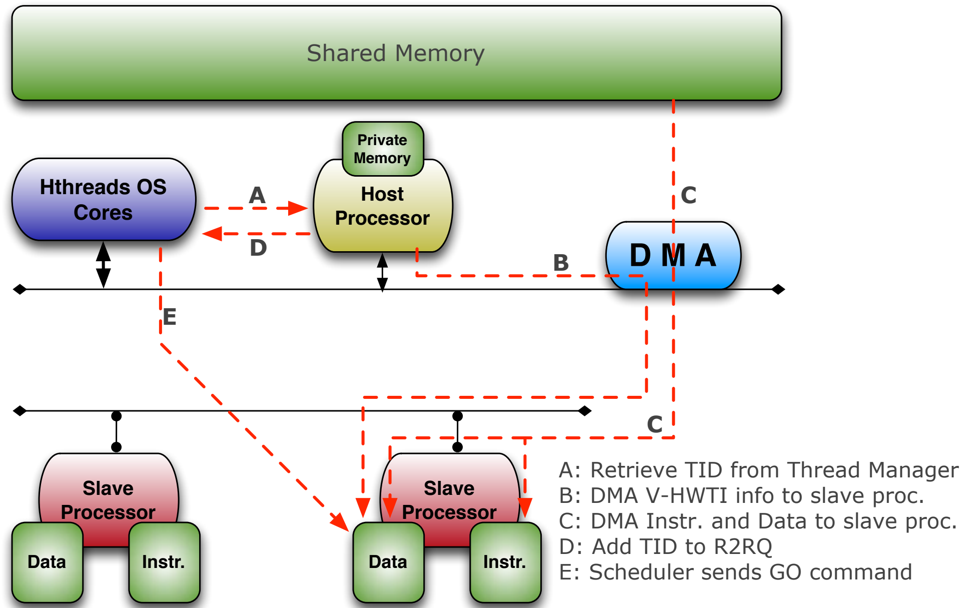

Figure 3.16: Thread Creation using the DMA controller after a free slave processor is found

begin at the start of the preallocated memory space within the Split-BRAMs. This API call and its arguments are also shown above in Figure 3.15.

Figure 3.16 highlights the new scheduling flow under the Split-BRAM system. This flow is the same for both static and dynamic thread creation as it shows the steps after a target slave processor has been identified. The original Hthreads SMP systems identified free slave processors by simply reading the *utilized* register within the processor's V-HWTI. Slave processors would be marked as busy by the host processor at the time of thread creation and marked as free when slave processors executed successfully a thread exit system call. Therefore, there was not a possibility of race conditions. As explained in Section 3.2.5, the host processor can no longer read the contents of these registers directly in order to determine if the slave processor is free or not. Therefore, it makes use of the DMA controller in order to transfer the value of this register to the shared memory space in order to read it.

Chapter 4

Results

4.1 Defining Scalability

Scalability refers to a system's ability to handle either additional, increasing or fixed amounts of workloads in proportion to the number of processors. Scalability is generally measured in two forms: *strong* and *weak*. If the time in which a system can solve a fixed sized problem continually decreases as processors are added and work is divided evenly among them, then the system is displaying strong scalability. Strong scalability can be represented by $T_{exec} = \frac{P_{exec}}{N}$, where T_{exec} is total execution time, P_{exec} is problem execution time for 1 processor, and N is the number of processors. On the other hand, a system displaying weak scalability will execute at a constant time due to fixed sized workloads for N number of processors. That is, $T_{exec} = \frac{N \times P_{exec}}{N} = P_{exec}$. Since weak scalability suggests a constant time for N processors within a system, this metric may clearly show system overhead such as OS overhead, bus contention, and limited memory bandwidth. Strong scalability is a good indicator of an insufficient problem decomposition, available memory bandwidth, and OS overhead. Both types of scalability are considered in the following sections. All tests were done using a Split-BRAM system with 32 slave processors (hardware threads) and an SMP system with 6 slave processors (hardware threads) on the Xilinx ML605 Evaluation Board. For the SMP systems, each processor was allocated its own private bus and performed data accesses to shared memory over a common bus. The Split-BRAM systems contained a variable number of processors on slave processor bus. Both systems were discussed in detail in Chapter 3.

4.2 Latency for Data Accesses

First, the efficiency of transferring both instructions and data to a slave processor's local memory was compared to caching instructions and accessing data over the general bus. The MicroBlaze was

```
00: void * worker_thread (void * arg)
01: {
02:     targ = (int *) arg;
03:
04:     // Grab the delay time
05:     time = *(targ + time_offset);
06:
07:     // Grab Size of Data
08:     data_size = *(targ + size_offset);
09:
10:     // Delay for certain amount of time
11:     delay(time);
12:
13:     // Now do some "work" over the entire data
14:     for ( i = 0; i < data_size; i++)
15:     {
16:         // Read arguments and place on stack
17:         temp = *(targ + i);
18:     }
19: }
```

Figure 4.1: The thread function used during the Access Latency Test

configured to include a direct-mapped cache over both instruction and data memory accesses. Data caches were not used for reasons explained in Chapter 3, but instruction caches were allowed due to the read-share only property that applies to that data. The best case for an Hthreads SMP system that uses instruction caching is that it is able to fit the entire program within its cache. Figure 4.1 shows the code for a small thread function executing over a loop. The code is uniform and should net satisfactory cache hits within the MicroBlaze. One thing to note is that slave processors within the Hthreads SMP systems operate outside of the cacheable range until thread creation. As a result, there is always a true cold start at thread creation. However, in testing, this only occurred once as the tests were executed multiple times in order to calculate averages. In the case of the Split-BRAM system, both instruction and data were transferred to the slave processor for each trial. The feature for determining whether instructions were previously transferred as discussed in Section 3.3.2 was purposely left out. Because the Split-BRAM systems consistently transferred data as well thread function instructions, the SMP systems performed better with small data arguments passed to the thread function. However, as the size of the data arguments increased and as more processors were added into the system, the SMP systems were quickly outperformed by the Split-BRAM systems.

The benchmark ran 1-6 threads with 3 work delays with the work delays increasing the size by 4 bytes. The range of the data size was 12 bytes through 12,000 bytes. The Split-BRAM system

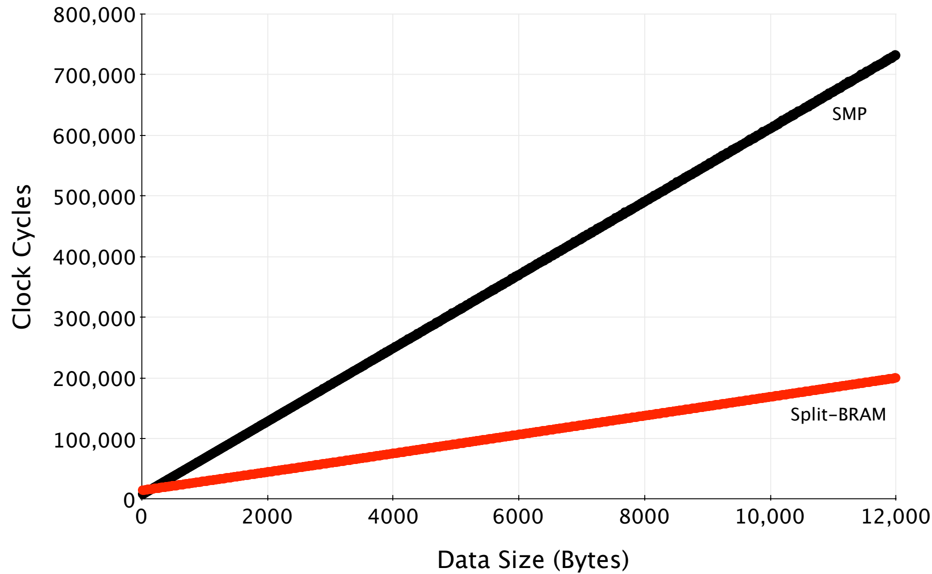

Figure 4.2: Work Delay = 250 instructions for 1 thread running

included a group of 8 slave processors on a single bus; however, only up to 6 processors were tested to directly compare to the SMP system of 6 slave processors. As previously stated, the test for the Split-BRAM included transferring both instruction and data for each thread in every trial. Over 1,000 trials were scheduled for the thread functions in order to calculate an average, causing a disadvantage to the Split-BRAM due to the fact that SMP systems do not require extra time for instruction DMA setup and transfer.

As shown in Figures 4.2, 4.3, and 4.4, the point of intersection between the two systems occurred very quickly for a small data set. This can also be seen through the slopes averaging 60.4 clock cycles per byte on the SMP system and 15.4 clock cycles per byte on the Split-BRAM system for 1 thread, a 4*x* difference. Naturally, when we increased the number of threads to 6, these slopes changed as can be seen in Figures 4.5, 4.6, and 4.7. The rate of growth for these systems then became 100.8 clock cycles per byte for the SMP system and 47.2 clock cycles for the Split-BRAM system, exhibiting a 2*x* difference. In the case of 6 threads, the Split-BRAM system rate of growth is ultimately less than that of the SMP system which has only 1 thread. What is even more interesting is that these results are from a single DMA controller located on the main bus. Also, instructions (2 KB) and data were transferred to each thread sequentially. Therefore, adding

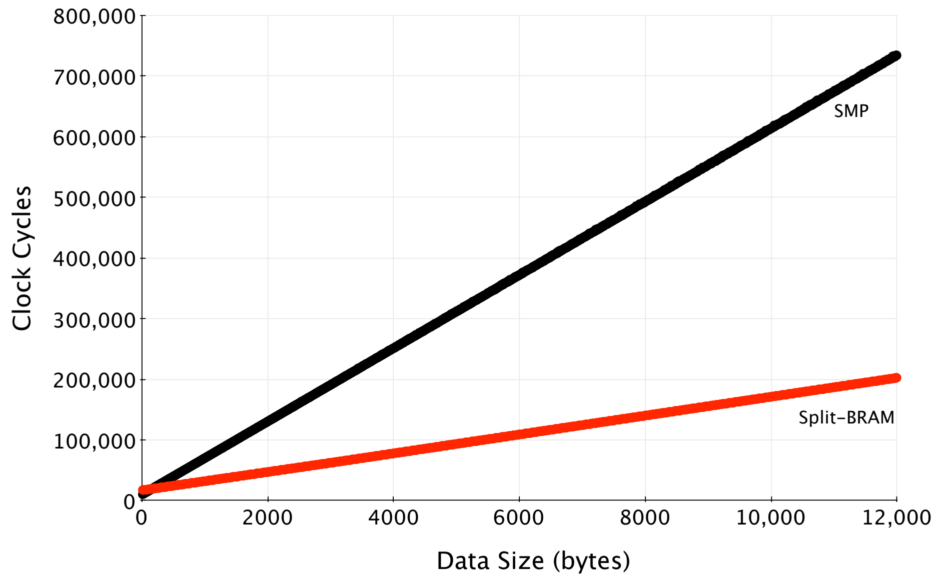

Figure 4.3: Work Delay = 500 instructions for 1 thread running

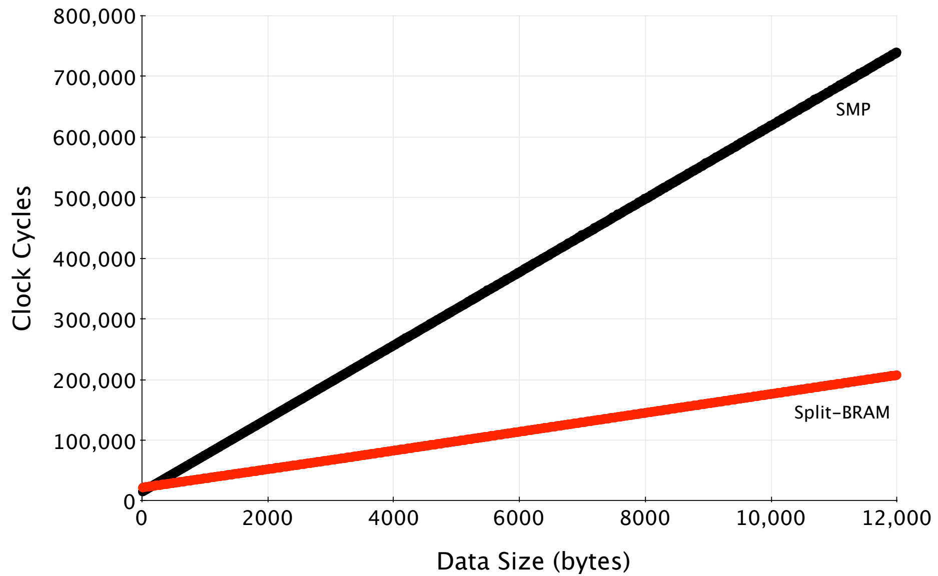

Figure 4.4: Work Delay = 1,000 instructions for 1 thread running

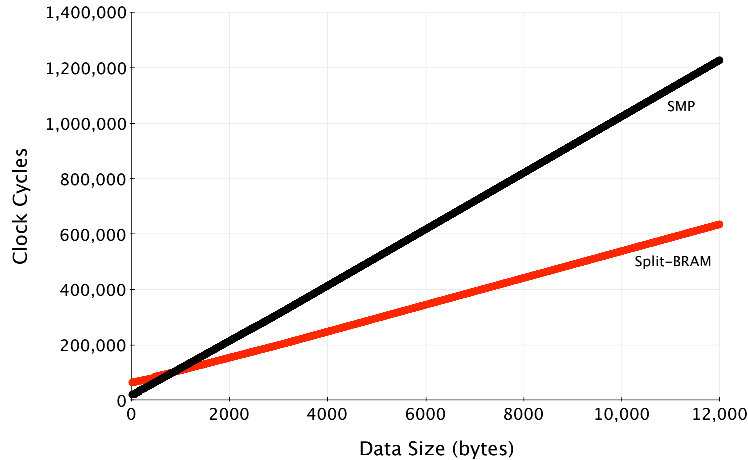

Figure 4.5: Work Delay = 250 instructions for 6 threads running

multiple DMA controllers would improve the performance numbers of the Split-BRAM systems which are already at a disadvantage.

Although the SMP systems were able to cache instructions, providing fast local memory access, data accesses remained a problem which was augmented by adding more processors within the system. This shows the limitations of SMP systems using a shared bus and also highlights the bottleneck experienced in global memory accesses. Due to the locality of both instructions and data over an undisturbed bus, the Split-BRAM system was able to perform much faster, and continued to do so with increasing thread numbers. Where it suffered, as shown in the results, was the data migration setup and transfer times. This should prove to be an interesting test to re-run with the optimized Hthreads kernel modifications.

4.3 Weak Scalability

Two applications were chosen to evaluate the weak scalability of the Split-BRAM system and the SMP system: Matrix Multiply and the International Data Encryption Algorithm (IDEA). Matrix Multiply is a common application used in benchmarks where more data accesses are desired. As a result, the results shown here represent how quickly a slave processor can access the arguments passed to it when the majority of caches are able to be accessed on the SMP system. On the other hand, IDEA portrayed a more instruction intensive application as the algorithm involved several

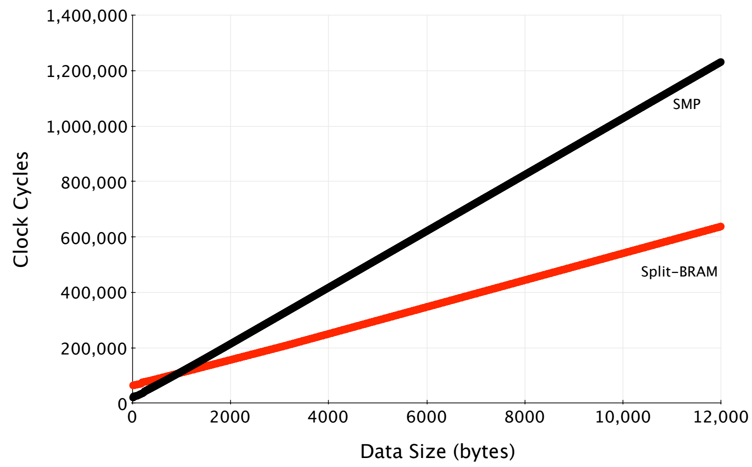

Figure 4.6: Work Delay = 500 instructions for 6 threads running

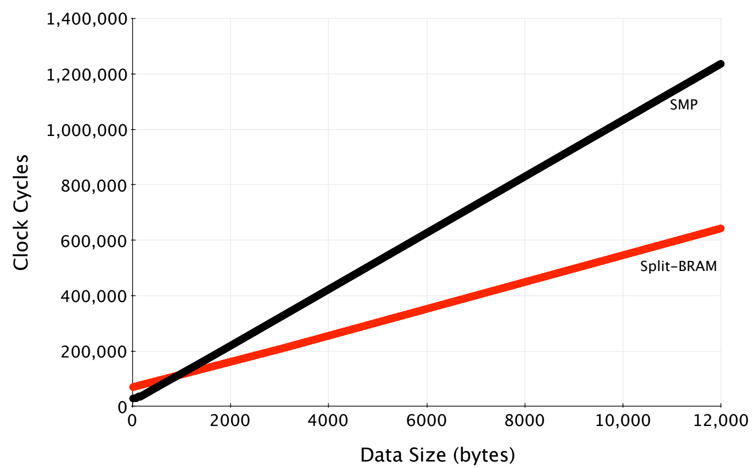

Figure 4.7: Work Delay = 1,000 instructions for 6 threads running

data transformations in order to arrive at a solution. Therefore, the instructions per byte of input data was much higher than Matrix Multiply and may have raised the probability of cache misses. There does not exist any synchronization primitives within both applications. As a result, both of these tests should exhibit raw processor performance, and the time results of adding more processors in the system should not deviate much from running the same tests with just 1 processor. Simply put, results between thread counts should only deviate due to OS overhead and/or bus contention in the case for Matrix Multiply. In the case for IDEA, a more compute intensive application, all three systems should perform similarly. If, however, additional processors are added to the Split-BRAM system, data migration latencies may cause the performance to suffer due to sequential DMA transfers.

To minimize the delay found in SMP systems contending for the bus, a third system was built that is identical to the SMP system discussed in Chapter 3. Architecturally, the two SMP systems were the same. The difference was that the thread's arguments were transferred into free space located in the V-HWTI BRAM for one SMP system, and remained in the shared memory space for the other. This allowed for data to exist within greater proximity and eliminate possible bus contention. Therefore, one of the SMP systems was able to read data over an undisturbed bus and the other remained to fetch its arguments within shared memory attached to a centralized bus (main bus). For both of these programs, there were 4 measurements given: 1) thread execution time, 2) DMA transfer overhead for transferring arguments and/or instructions, 3) Operating System Overhead for thread creation, and 4) Total Execution Time accounting for thread creation and joining on all threads.

4.3.1 Matrix Multiply

Table 4.1 displays the average execution time for threads to complete. These times are recorded within each thread. Therefore, they do not include any OS overhead such as thread_create() or thread_join() which can skew the results due to the sequential creating/joining of threads. As shown, the thread running time for the Split-BRAM system remains constant for 1-32 threads. The

deviation of $1\mu s$ is attributed to the test being downloaded and executed twice on the Split-BRAM systems: the first time for testing 1-6 threads, and a second time for running 8, 16, and 32 threads. Results for the SMP system with its arguments transferred to a local memory (the V-HWTI BRAM) remained constant as well. This shows that this system exhibits very good weak scalability, but as previously discussed, this can only be verified for 1-6 threads. For the SMP system that held its arguments in main memory, the contention of bus access in addition to the inherent latencies of the off chip global memory began to show in the results. As more processors were added, the contention increased as well as the data access latencies.

Table 4.2, reveals differences amongst the three systems in OS Overhead. Both SMP systems produced similar results for running 1 thread; however, the two increasingly differentiated when scheduling additional threads. The reason for this, yet again, was bus contention. The host processor sets up several arguments in the V-HWTI for hardware thread creation purposes. Therefore, there were several single writes done over the busses for each thread. The slave processors within the SMP systems with data located in the V-HWTI BRAM did not contribute any bus traffic on the main bus other than doing a single store word to the thread manager for thread exiting. In contrast, the slave processors within the SMP system with its data in shared memory add contention to the main bus for each of its single read accesses towards its data arguments. The size of the data for each slave processor in this test was a 20 x 20 integer array, where an integer in this system represents 32 bits = 1 word. The amount of data read accesses that would occur to compute this sized matrix is $400 \; x \; 40 = 16{,}000$. Therefore, it can be inferred that the results for this system would vary greatly for a bigger argument size.

The Split-BRAM system began to show additional OS overhead compared to the SMP system with local data, and also for the SMP system for 1-5 threads. The greatest cause of higher latency times was the fact that there were DMA transfers being implemented within the OS. In both of the SMP systems, whenever the host processor performed a thread create, it simply wrote to the V-HWTI using a pointer whereas the Split-BRAM systems made use of the DMA controller. It wrote only 9 words to the V-HWTI BRAM. As previously explored in Section 4.2, executing over

Thread Running Time (Average)			
Thread #	**SMP (μs)**	**SMP w/ Local Data (μs)**	**Split-BRAM (μs)**
1	1,393,731	743,382	645,141
2	1,411,951	743,382	645,141
3	1,444,141	743,381	645,141
4	1,484,001	743,382	645,141
5	1,536,517	743,382	645,141
6	1,607,171	743,382	645,140
8	N/A	N/A	645,140
16	N/A	N/A	645,140
32	N/A	N/A	645,140

Table 4.1: Matrix Multiply - Thread Running time Averages

a few amount of bytes is more feasible with pointers than simply doing the same with a DMA transfer that requires configuring the DMA, starting, and then polling it for completion.

Table 4.3 includes times for DMA Overhead of data and/or instructions. As you can see, the two systems shown here only included the Split-BRAM system and the SMP system with its data local in the V-HWTI BRAM. For the Split-BRAM system, the times captured the transfer of instructions (4 KB) and data to and from the the slave processor's local BRAM (1,600 bytes each way, 3,200 bytes total). Therefore, the 2*x* difference between the two systems is to be expected.

Finally, Table 4.4 concludes the results for this benchmark. As seen in the previous tables, the small deviations from simply running 1 thread to *N* threads revealed OS overhead in both the Split-BRAM and the SMP with local data systems. For the SMP system, the results mainly take into account bus contention. Both the Split-BRAM and the SMP with local data systems revealed good weak scalability. Split-BRAM completed much faster (1.15*x* faster for 1 thread) despite having to incur nearly a 2*x* penalty for data migration of both instructions and data. For 32 threads, the Split-BRAM deviates only by 2% compared to running 1 thread.

4.3.2 IDEA

IDEA is a more instruction intensive application. Thus, the ratio of instruction fetch to data fetch is much higher. This is immediately seen in Table 4.5. As before, all timing done here occurred within the thread function. Both Split-BRAM and SMP with local data exhibited constant results when increasing the processor count. For the SMP system accessing data within global memory,

OS Overhead (Average)			
Thread #	**SMP (μs)**	**SMP w/ Local Data (μs)**	**Split-BRAM (μs)**
1	77.8	79.5	262.0
2	125.6	99.8	450.0
3	212.2	134.2	592.2
4	237.2	178.4	779.3
5	907.9	199.6	966.9
6	4591.2	220.5	1111.3
8	N/A	N/A	1477.6
16	N/A	N/A	2846.4
32	N/A	N/A	5617.8

Table 4.2: Matrix Multiply - Operating System Overhead for Thread Creation

DMA Overhead (Averages)		
Thread #	**SMP w/ Local Data (μs)**	**Split-BRAM (μs)**
1	117.7	243.9
2	229.6	477.3
3	341.9	710.4
4	453.8	943.3
5	565.8	1177.7
6	677.5	1409.9
8	N/A	1869.5
16	N/A	3727.8
32	N/A	7447.1

Table 4.3: Matrix Multiply - DMA Overhead for Thread's Instructions (4 KB) & Data (1.6 KB)

Total Time (Average)			
Thread #	**SMP (μs)**	**SMP w/ Local Data (μs)**	**Split-BRAM (μs)**
1	1,393,808	743,579	645,647
2	1,412,077	743,711	646,068
3	1,444,353	743,858	646,443
4	1,484,283	744,014	646,863
5	1,537,425	744,147	647,285
6	1,611,763	744,280	647,662
8	N/A	N/A	648,487
16	N/A	N/A	651,715
32	N/A	N/A	658,205

Table 4.4: Matrix Multiply - Total Execution Time

its thread execution times remained fairly constant as well despite having to contend with the bus from other threads. The reasoning for this is how the reads were done within the thread function: the function read 5 *unsigned short* variables, performed computation for some time, and then wrote 5 variables of the same type to global memory. Because of the sparsity of reads/writes to global memory, contention to it was kept at minimum. Therefore, the difference in execution time observed between the two SMP systems can be attributed to the shorter unperturbed path the SMP with local data system took.

Similarly, in Table 4.2, Table 4.6 shows similar results in OS overhead between the Split-BRAM system and the SMP system with local data. However, the difference between the two SMP systems was changed. The host processor, no longer underwent as much bus contention as previously realized within the Matrix Multiply test. Therefore, what is observed for OS overhead displayed similar results between both SMP systems, eliminating any further cause to total time differences towards this metric.

The size of the instructions for this test equated to 8,000 bytes. The data size for threads was 3,200 bytes, equating to 6,400 bytes transferred for both directions. This represented a total DMA transfer of 6,400 bytes for the SMP system with local data and $6,400 + 8,000 = 14,400$ bytes for the Split-BRAM system. This translated to a 2.25*x* increase for the amount of data needing to be transferred to the Split-BRAM system. Table 4.7 shows very similar increases in time: for 1 thread, the Split-BRAM has a 2.36*x* increase in DMA latency and a 2.33*x* at 6 threads. The difference between 2.25*x* increase of total data transferred to what was actually observed can be attributed to the DMA setup time which requires several single writes for configuration and starting the device.

Table 4.8 concludes the results for this test and offers similar patterns that were seen in Table 4.5. Unlike the previous test, all three systems exhibited good weak scalability. The difference in total times amongst the three systems was minute and was smallest between SMP with local data and the Split-BRAM systems. However, the Split-BRAM system continued to complete faster than the other two, despite the penalty obtained by transferring 2.25*x* more data. As said in Section

Thread Running Time (Average)			
Thread #	**SMP (μs)**	**SMP w/ Local Data (μs)**	**Split-BRAM (μs)**
1	677,707	650,715	644,572
2	677,714	650,715	644,572
3	677,728	650,715	644,572
4	677,752	650,715	644,572
5	677,775	650,715	644,572
6	677,793	650,716	644,752
8	N/A	N/A	644,571
16	N/A	N/A	644,571
32	N/A	N/A	644,571

Table 4.5: IDEA - Thread Running time Averages

OS Overhead (Average)			
Thread #	**SMP (μs)**	**SMP w/ Local Data (μs)**	**Split-BRAM (μs)**
1	74.3	74.4	270.1
2	95.1	93.9	431.7
3	143.6	152.5	601.1
4	163.0	172.7	790.4
5	191.6	191.1	961.2
6	225.5	241.0	1119.0
8	N/A	N/A	1481.1
16	N/A	N/A	2852.9
32	N/A	N/A	5631.4

Table 4.6: IDEA - Operating System Overhead for Thread Creation

4.3, introducing additional processors within the Split-BRAM system for this application increased data migration latencies as indicated at 16+ threads. In order to mitigate these latencies, parallel DMA transfers were used by distributing multiple DMA controllers throughout the system.

4.4 Strong Scalability

In order to evaluate the strong scalability of both SMP and Split-BRAM systems, matrix multiply was chosen again. Unlike the previous matrix multiply test in Section 4.3 that performed computation on fixed sized matrices of 20 *x* 20, this matrix multiply application allowed computation of larger sized, uniform or non-uniform matrices. Having bigger matrices allowed a fair amount of work to be divided amongst slave processors, especially in the bigger Split-BRAM system. As a result, this ultimately translated to increased amounts of reads/writes to global memory for the SMP system. The SMP system with data locally stored was omitted for this test. For the Split-

DMA Overhead (Averages)		
Thread #	**SMP w/ Local Data (μs)**	**Split-BRAM (μs)**
1	112.9	266.5
2	222.9	522.2
3	333.0	776.5
4	443.1	1031.7
5	553.1	1288.7
6	663.3	1543.6
8	N/A	2054.5
16	N/A	4098.7
32	N/A	8187.8

Table 4.7: IDEA - DMA Overhead for Thread's Instructions (8 KB) & Data (3.2 KB)

Total Time (Average)			
Thread #	**SMP (μs)**	**SMP w/ Local Data (μs)**	**Split-BRAM (μs)**
1	677,781	650,903	645,108
2	677,809	651,032	645,526
3	677,872	651,201	645,949
4	677,915	651,331	646,394
5	677,967	651,460	646,822
6	678,019	651,620	647,234
8	N/A	N/A	648,107
16	N/A	N/A	651,523
32	N/A	N/A	658,391

Table 4.8: IDEA - Total Execution Time

BRAM system, it also introduced more DMA traffic within the system. The reasoning for this was the small 12 KB of free space that was available within the Split-BRAMs. However, as was shown in Section 4.2, the Split-BRAM system was capable of outperforming the SMP system when there existed many data memory accesses to operate over. This was due to the single beat memory accesses the MicroBlaze is currently limited to whereas the DMA controller took advantage of burst reads/writes, sending data elements in a pipeline fashion over the bus. As shown in Figure 4.8, the problem was exacerbated when any PLB-to-PLB bridge slave processors crossed while doing single beat reads, doubling the traditional read time further [21]. Therefore, the Split-BRAM system displayed better total calculation time as the number of processors grew to accommodate larger matrices.

This test included a comparison of the Split-BRAM system and the Hthreads SMP system accessing data in global memory. Tables 4.10, 4.11, 4.12, and 4.13 all show a halving on the Split-BRAM systems every time the thread count was doubled, exhibiting a near linear speedup

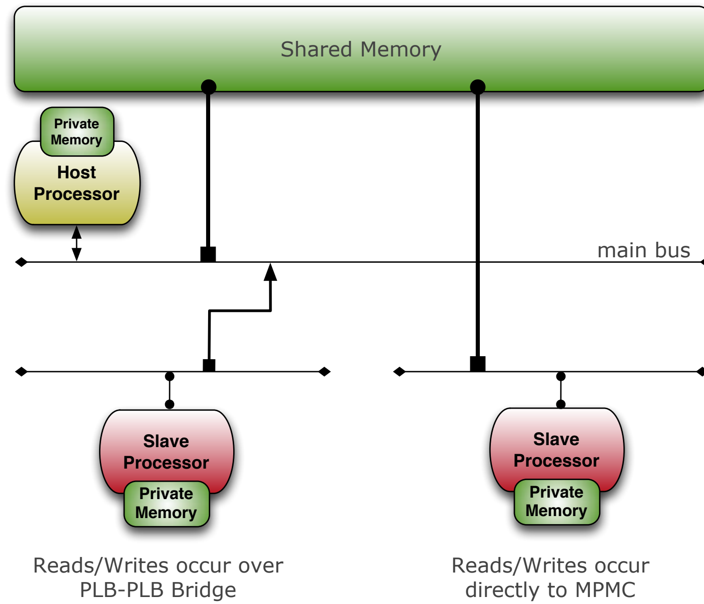

Figure 4.8: Writing/Reading to the Shared Memory

up to 32 processors. The SMP system also shows a halving but is capped at 2 threads. From there, the speedup achieved when providing additional threads into the SMP system began to plateau quickly. The time differences at each number of threads was also large. For 1 thread computing a 2048 *x* 2048 matrix, the SMP system was 17*x* slower than the Split-BRAM system. Increasing the thread count to 6 revealed that this number grows to 33*x*. When computing a smaller 20 *x* 20 matrix, having additional threads in the Split-BRAM system continued to provide better speedup, albeit not 33*x* at 6 processors over the SMP system. Table 4.9 shows SMP having been 17*x* slower compared to the Split-BRAM system. This is similar to what was seen in the bigger matrices. At 6 threads, however, the SMP was only 18*x*. There can be two reasons for this behavior. The first being contention for the timer. Each slave processor had a local 32-bit timer on its local bus to ensure fast access to timing readings, and to minimize extra latencies due to bus mastering. For the Split-BRAM system, there were three slave processors assigned to a bus whereas SMP limits this grouping to 1. Section 4.5 discusses more about this chosen processor grouping. Since the data set (2 20-element matrices at a time) was reduced, the probability that slave processors would read the local timer simultaneously was higher on the Split-BRAM system. The second reasoning is

Total Calculation Time - 20 *x* 20 (Averages)		
Thread #	**SMP (*ms*)**	**Split-BRAM (*ms*)**
1	18.919	1.120
2	9.509	0.560
4	4.793	0.280
6	3.806	0.224
8	N/A	0.168
16	N/A	0.112
32	N/A	0.056

Table 4.9: Total Calculation Time for a 20 *x* 20 Matrix

Total Calculation Time - 256 *x* 256 (Averages)		
Thread #	**SMP (*s*)**	**Split-BRAM (*s*)**
1	36.38	2.194
2	19.66	1.097
4	12.83	0.549
6	11.26	0.369
8	N/A	0.274
16	N/A	0.137
32	N/A	0.069

Table 4.10: Total Calculation Time for a 256 *x* 256 Matrix

the decreased traffic for an SMP system over the smaller data set. Nevertheless, the Split-BRAM system continued to provide better performance and, more importantly, strong scalability towards increasing thread counts for larger matrices.

4.5 Hardware/Software Microkernel vs RPC Scalability

An additional benchmark was tested in order to verify the scalability of the Hthreads OS on a new platform board with the availability of more processors. The benchmark RPC Synthetic Test, was previously used to evaluate an Hthreads SMP system on the Xilinx ML507 Evaluation board using

Total Calculation Time - 512 *x* 512 (Averages)		
Thread #	**SMP (*s*)**	**Split-BRAM (*s*)**
1	292.6	17.50
2	157.2	8.750
4	104.2	4.375
6	93.93	2.940
8	N/A	2.188
16	N/A	1.094
32	N/A	0.547

Table 4.11: Total Calculation Time for a 512 *x* 512 Matrix

Total Calculation Time - 1024 *x* 1024 (Averages)		
Thread #	**SMP (*s*)**	**Split-BRAM (*s*)**
1	2370.6	139.8
2	1277.9	69.90
4	840.3	34.95
6	766.1	23.35
8	N/A	17.48
16	N/A	8.737
32	N/A	4.369

Table 4.12: Total Calculation Time for a 1024 *x* 1024 Matrix

Total Calculation Time - 2048 *x* 2048 (Averages)		
Thread #	**SMP (*s*)**	**Split-BRAM (*s*)**
1	18929.5	1118.4
2	10141.9	559.2
4	6741.1	279.6
6	6141.8	186.8
8	N/A	139.8
16	N/A	69.9
32	N/A	34.9

Table 4.13: Total Calculation Time for a 2048 *x* 2048 Matrix

a PowerPC and 6 MicroBlazes [1]. This benchmark highlighted the scalability of Remote Procedural Calls (RPC) versus direct system calls to hardware components. The test began by creating threads that execute a variable number of instructions, known as the work delay, and performed a mutex lock and unlock to mimic thread synchronization. Work delay varied between 250, 500 and 1,000 instructions and each were defined as fine-grained, medium-grained, and coarse-grained, respectively. Figure 4.9 shows the pseudo code for the thread function for both RPC and non-RPC (direct calls) functionality.

RPC is a form of inter-processor communication (IPC) that defines the methods used for communication amongst processors in the system. Message Passing is also a form of IPC. RPC is defined as executing a subroutine in another address space without explicitly specifying the transfer within the code. An example of a subroutine executing in another address space is performing a system call handled by another processor operating in a different address space. This is the type of RPC calls that were included in this test with a platform containing 1 host and 32 slave MicroBlazes. RPC mechanisms are quite common today, especially in pure software Operating Systems where data may be exchanged from threads running in user space to threads running in kernel

```
00: void * worker_thread (void * arg)
01: {
02:
03:     for (i = 0; i < iterations; i++)
04:     {
05:         // Do some "work"
06:         delay ( delay_time );
07
08:         // Simulate thread synchronization
09:         mutex_lock( &mutex );
10:         delay(delay_time / DIVISOR );
11:         mutex_unlock( &mutex );
12:     }
13: }
```

Figure 4.9: Thread Function code for the RPC Synthetic Test

space. Also, heterogeneous platforms such as the IBM Cell operates under this same mechanism as it can simplify the different ISA code bases for system calls between processing elements, and allows a standardized interface to the OS. For Hthreads, this is the purpose of the Hardware Abstraction Layer (HAL). The HAL takes high-level API calls and transforms them to simple MMIO calls. As long as processors can accommodate simple load/store instructions, the HAL is able to provide this layer of abstraction to the programmer within a heterogeneous environment. This also results in minimal overhead compared to the more expensive RPC approach. RPCs require slave processors to set up a message, send it to the host processor where it then unpacks it and serves the request, and finally send the data back to the slave processor. This quickly generates a lot of traffic, and also leads to a centralized bottleneck -the host processor. The HAL is less expensive as system calls are ultimately translated into atomic operations to the Hthreads kernel. Additionally, since major components of the Hthreads kernel are implemented in hardware, servicing these requests are also fast. Therefore, the direct calls continue to provide linear speedup on the bigger Split-BRAM systems. Any cause for delay would be attributed to the bus bandwidth of the system. When creating bigger systems (100's of processors), it may be beneficial to distribute the microkernel as well.

Threads running the RPC Synthetic benchmark do not share any data whatsoever. Therefore, the results here do not reflect any contention for a shared variable. Contention occurred only for bus access when communicating for OS services. This presented an opportunity to evaluate

another metric of the Split-BRAM system: the ideal number of slave processors attached to one bus. Originally, the Split-BRAM system attached 8 processors to a bus in order to reduce PLB-to-PLB bridges as well as reduce the number of master devices that ultimately attach to the main bus. As an early test for this metric, the groups of 8 Split-BRAM system was used to verify if the results changed when threads were not scheduled sequentially on one bus but rather scheduled 1 thread per bus before scheduling another on the same bus. What was found was scheduling three threads on one bus provided no notable difference compared to scheduling three threads on separate busses. This observation changed however, when increasing the processors per bus to 4. As a result, three Split-BRAM systems were created and included in the following tests: groups of 3, 4, and 8 processors per bus.

4.5.1 Synthetic Test - RPC vs Non-RPC

The first set of figures presented show speedup relative to 1 thread for RPC vs Non-RPC (direct) calls. Figures 4.10, 4.11, 4.12, and 4.13 all show similar patterns for RPC. It is immediately shown in Figure 4.10, that the RPC approach was not scalable. For a system with only 6 slave processors, the speedup of adding more processors began to plateau earlier as we decreased the work delay within the thread function. This is due to the much larger overhead incurred for RPC invocation. Included in this overhead is 1) context switching between several software threads in order to service hardware threads RPC requests, 2) serialization of those requests due to the host processor as a centralized bottleneck resulting in more bus collisions, and 3) passing of large data structures in order to communicate with the host for correct OS call operations. This led to indeterminable and sometimes erratic results as shown in Figures 4.11, 4.12, and 4.13. On the other hand, direct calls showed more promising speedups on the hardware-based microkernel. Nearly a linear speedup of (5.5*x*) was achieved on the SMP system at 6 threads using a coarse-grained workload in Figure 4.10. Similarly, Figure 4.11 shows a speedup of 5.9*x* at 6 threads, 15.4*x* at 16 threads, and 27.5*x* at 32 threads for the groups of 3 Split-BRAM based system.

Focusing on the three Split-BRAM systems for 1-6 threads, one can see that for direct calls,

there was a point at which speedup plateaued. This is observed in both the groups of 4 and groups of 8 systems, and began to appear when increasing the thread count from 3 to 4. This was not observed however, in the groups of 3 system. This indicated that the point at which that particular bus has maxed out its bandwidth occurs when 4 or more slave processors are attached to it. The speedup of running 3 threads on one bus for the groups of 3 configuration was similar to the speedups of running 4, 5, 6, 7 and 8 threads on the other configurations. Groups of 4 or 8 processor configuration achieved linear-like speedups equal to those of the groups of 3 configuration by scheduling only 3 threads per bus. For example, for the groups of 8 processor configuration, this translated to 4 busses totaling 32 processors. Therefore, scheduling 3 threads at a time on each bus would translate to a maximum of 12 concurrent threads running. Similarly, for the groups of 4 processor configuration, this would result in 8 busses allowing a maximum of 24 threads.

The reason for this plateau has to do with the PLB and the PLB slave devices (PLB-to-PLB Bridge). Although the PLB masters or slaves have their own address, data read, and data write paths, there still occurs possible serialization of requests. An example of this is the times when PLB masters are requesting in the same direction, or both are doing a data read. Additionally, the PLB slave device can cause further latency (1 or more clock cycles) if it cannot respond on the third clock cycle of the address phase done once a slave processor masters the bus. IBM's PLB specification outlines the possible use of an *n-deep* address pipelining, that allows masters to address and be queued up *n-deep* [22, 16] and avoid having to restart the address phase. Xilinx currently offers only a 2-deep address pipelining. Any additional requests made in the same direction are simply blocked, thus introducing additional latencies. This would explain why 3 slave processors on one bus peaks maximum throughput on that bus. For tests such as Matrix-Multiply and IDEA in Section 4.3, there was not much demand for bus bandwidth on the Split-BRAM systems due to exploitation of locality over the LMB.

Although speedups are of good use for indicating how well the Hthreads microkernel scales when adding more processing elements, it does not show how well it fairs against other systems. Comparing speedups between different systems can lead to unrealistic conclusions as a system

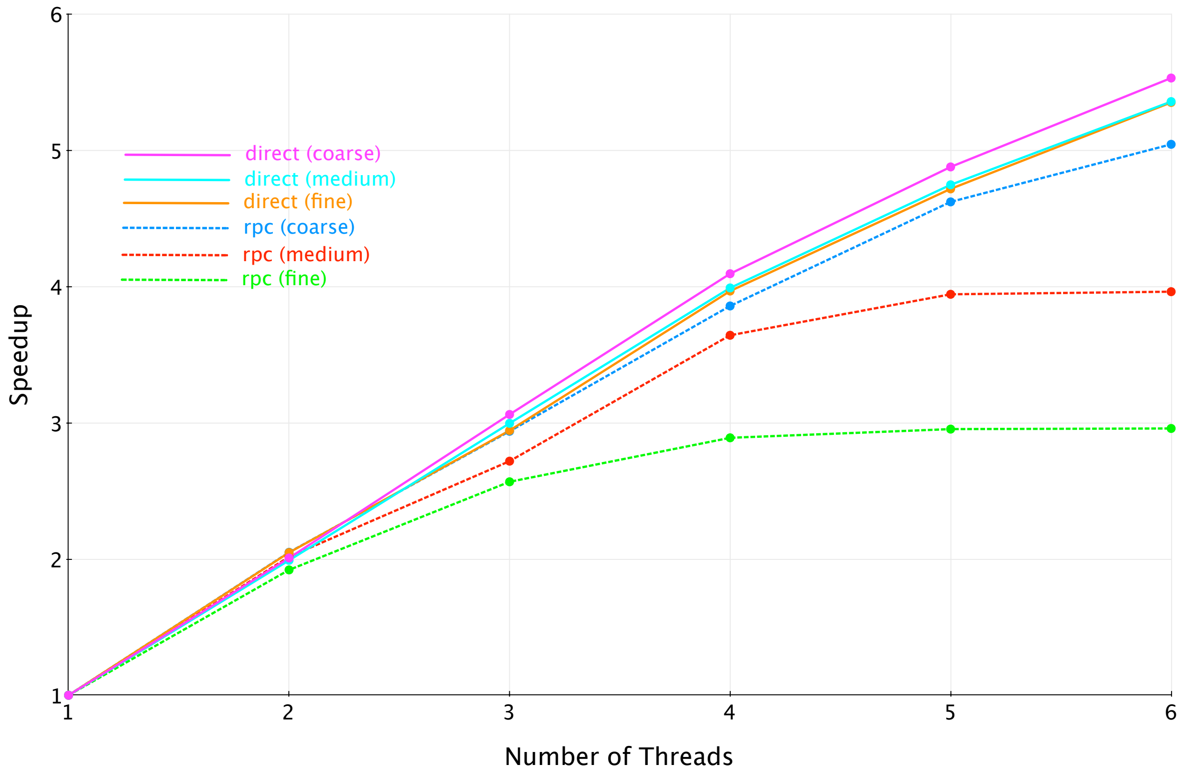

Figure 4.10: SMP - RPC vs. Direct Calls

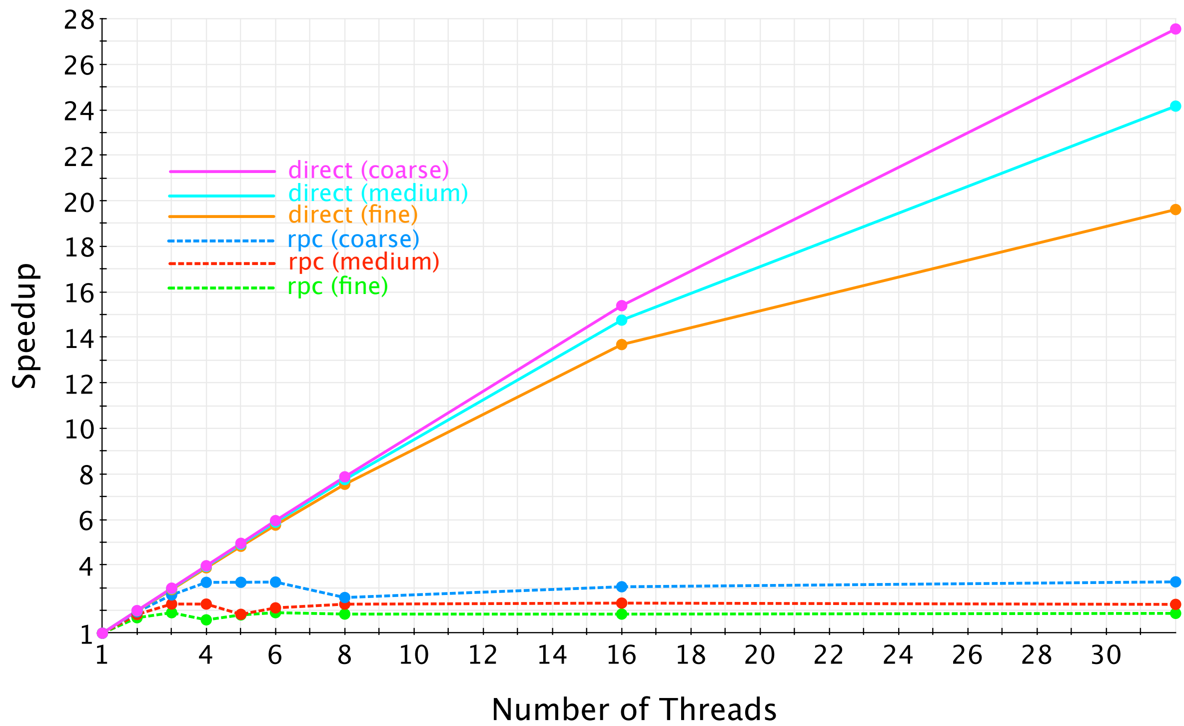

Figure 4.11: Split-BRAM (Groups of 3) - RPC vs. Direct Calls

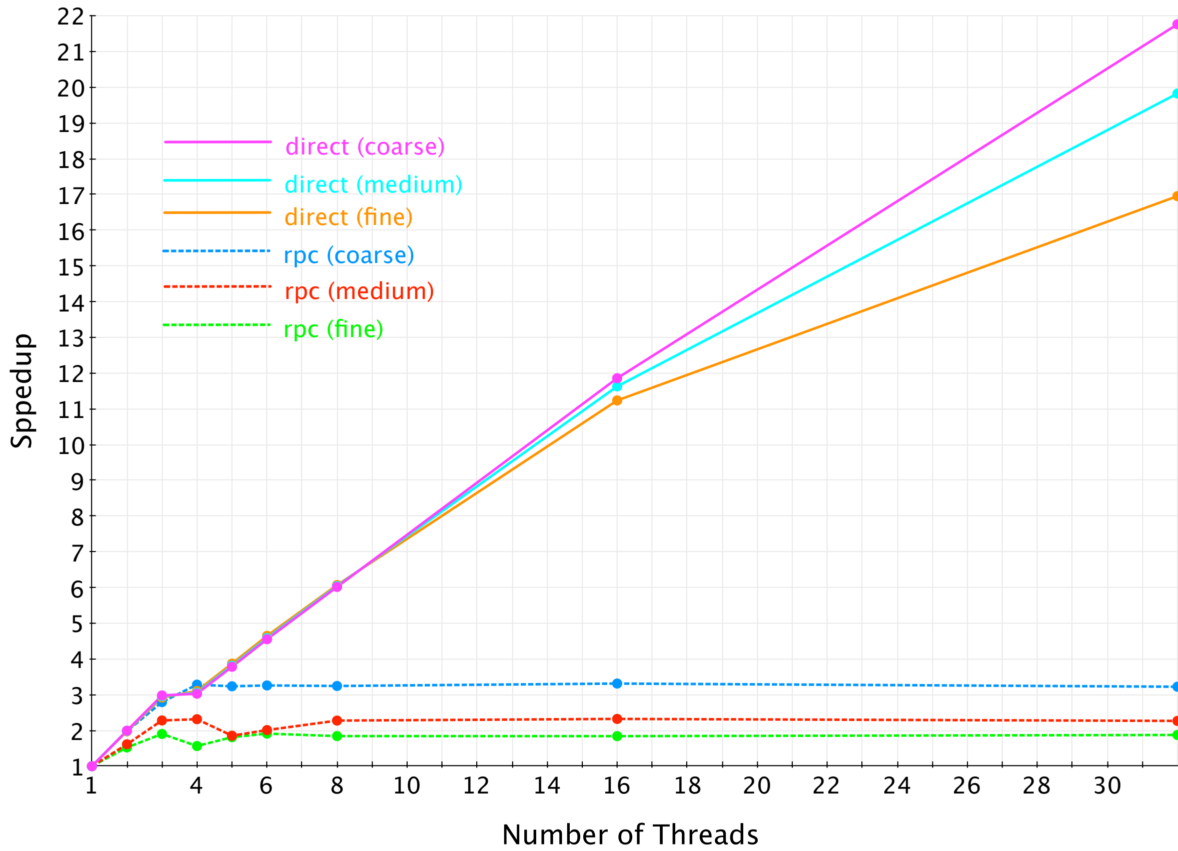

Figure 4.12: Split-BRAM (Groups of 4) - RPC vs. Direct Calls

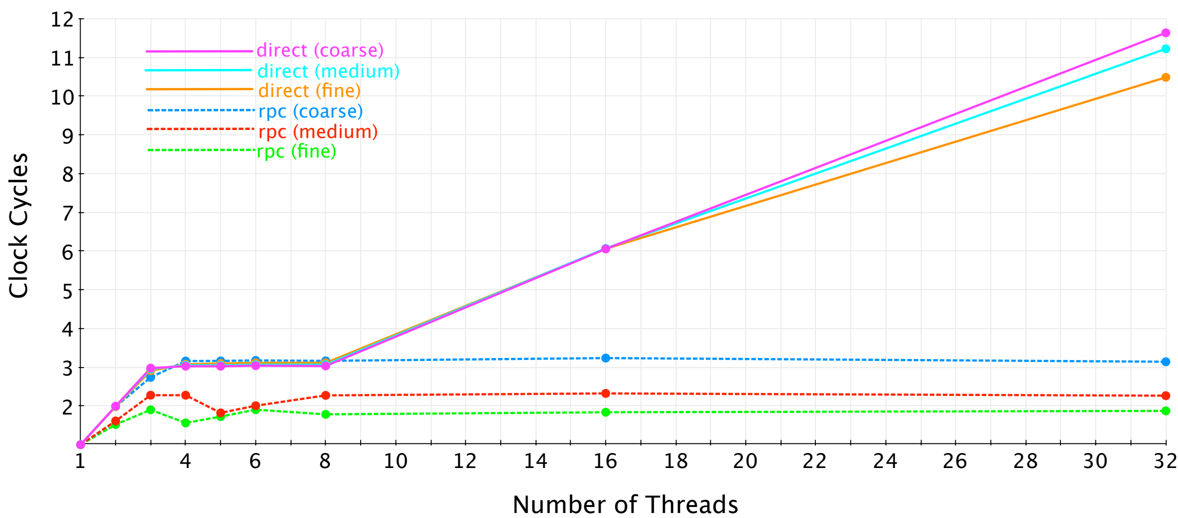

Figure 4.13: Split-BRAM (Groups of 8) - RPC vs. Direct Calls

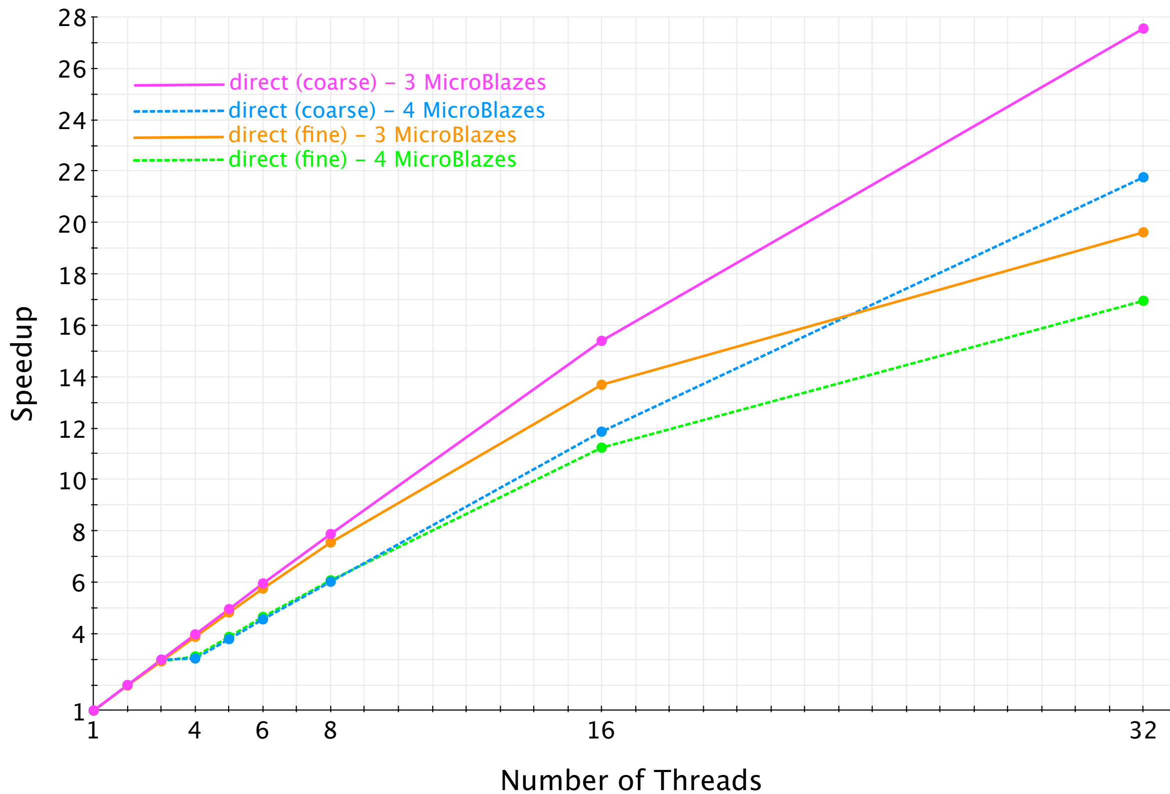

Figure 4.14: Split-BRAM Speedup differences between Groups of 3 and 4 on a single PLB

may appear to have better latency due to the relativity of 1 thread on the same system. That is, a system can display poor scalability as the case for 4 threads in Figure 4.12 or 4.13, but actual execution times may be faster in comparison to other more scalable solutions. For example, Figure 4.14 shows the comparison of speedups for both fine-grained and coarse-grained on the groups of 3 and groups of 4 configuration. At 32 threads, fine-grained on the groups of 3 configuration appears to performs worst than the coarse-grained groups of 4 configuration in terms of latency. In reality, their total execution times were 2,300,000 and 7,700,000 for groups of 3 and groups of 4, respectively. Therefore, comparisons of latency differences between the systems are provided.

The first comparison of total execution times between the Split-BRAM (groups of 3) and the SMP systems is shown in Figure 4.15 and 4.16. The difference in times for both coarse-grained and fine-grained was large. For example, for 6 threads with coarse-grained workload, the completion time for the Split-BRAM system was 2,800,000 clock cycles and 9,200,000 for the SMP systems.

This represents over a 3.2x difference in latency. Similarly for a fine-grained work load, the completion time of 700,000 for the Split-BRAM system and 2,600,000 for the SMP system yielded an increase latency difference of 3.7x. In addition, the total time at 1 thread for the Split-BRAM was 4,500,000 whereas the total time at 3 threads for the SMP system was 4,900,000 representing a latency difference of 1.08x. This shows that although both systems illustrated earlier near-linear speedups, there is a large difference between total execution time between the two.

Figures 4.17 and 4.18 show the comparison of the groups of 3 and 4 processor configuration on the Split-BRAM. In this comparison, differences between the two systems on both workloads were less drastic. For 1-3 threads with coarse-grained workload, total times were equal across the two systems. At a thread count of 4, the two began to show differences in total times. For the groups of 4 configuration, the speedup achieved at 3 and 4 threads was 3.0x: the performance remained the same as we added the 4th thread on the same bus. However, the groups of 3 configuration experienced a 3.0x speedup at 3 threads, and a 4.0x speedup at 4 threads. These two observations are also reflected in Figures 4.11 and 4.12. As a result of this extra contention on a shared bus for the groups of 4, differences in latencies increased as we continually added more threads in the system. However, the amplitude of those differences began to shrink as we moved to 100% slave processor utilization. At 100% slave processor utilization, the contention on individual groups busses were hidden with other dominating latencies introduced at this point. Such latencies included the bus contention across the main bus due to all OS service requests being funneled onto one bus.

The final comparison was done over the groups of 3 and 8 configuration. The groups of 8 configuration was the most area efficient, but at an expense shown both in Figures 4.20 and 4.19. The groups of 8 continued to have the same total time for threads 3-8. This was the point of it plateauing as shown in Figure 4.13 and occurred for both coarse-grain and fine-grain work loads. Previously, it was shown that there were decreasing difference in latencies between groups of 3 and 4 configurations beginning to appear as the number of threads increased. At 1-32 threads, this did not appear to hold for this comparison. The reasoning for such behavior was due to the higher capacity of bus contention on the groups of 8 configuration. The amount of contention

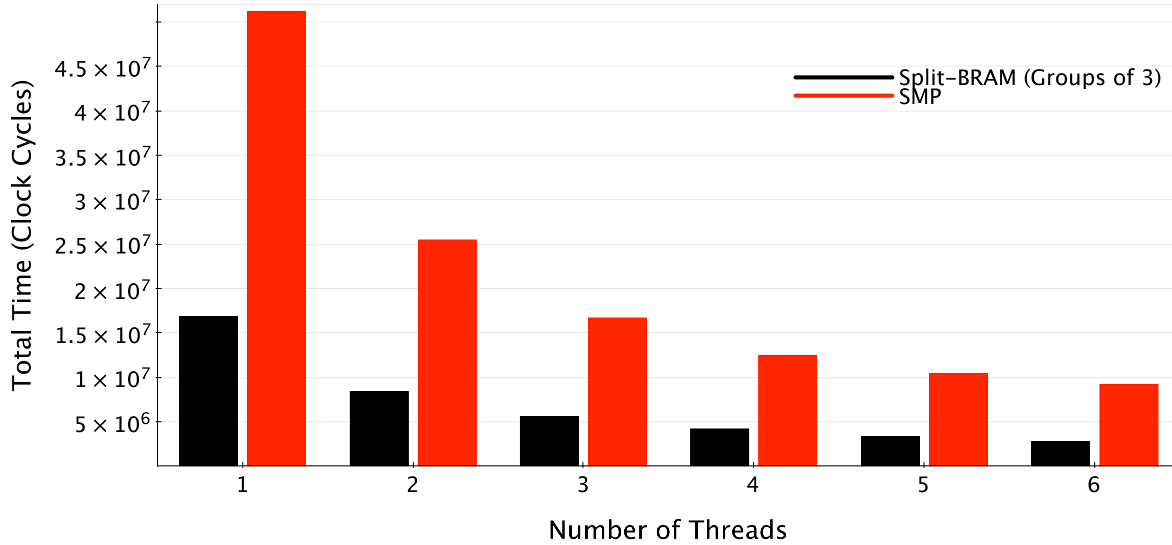

Figure 4.15: Split-BRAM Groups of 3 and SMP: Coarse Grained

within this system continued to be the dominant overhead even for both coarse-grain and fine-grain workloads. Due to the PLB only allowing two-deep address pipelining, there were more slave addressing collisions experienced here per bus compared to the groups of 3 configuration. Therefore, when both systems were at the point of 100% slave processor utilization, the groups of 3 configuration displayed better performance than the groups of 8.

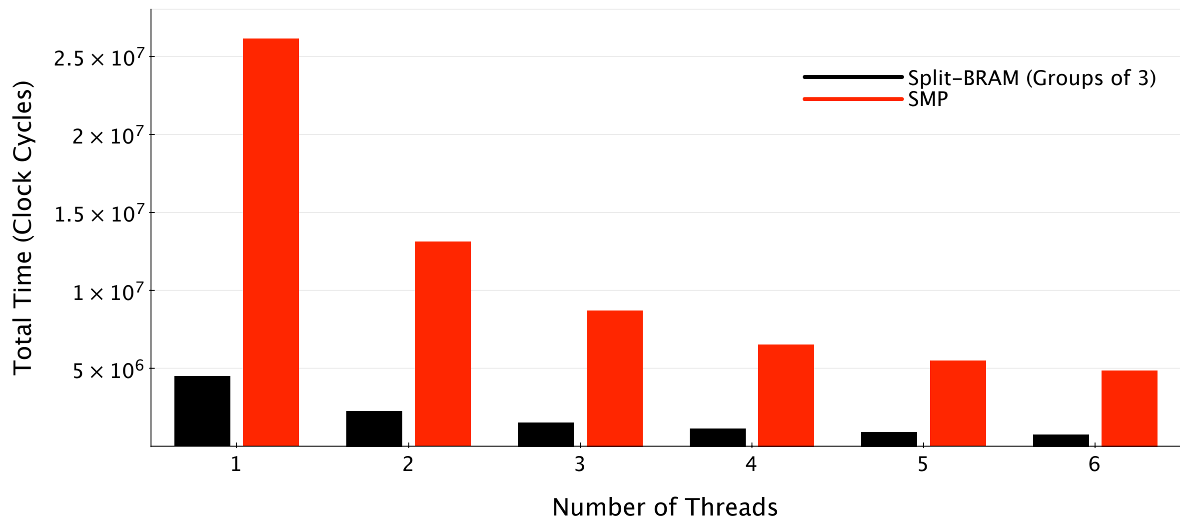

Figure 4.16: Split-BRAM Groups of 3 and SMP: Fine Grained

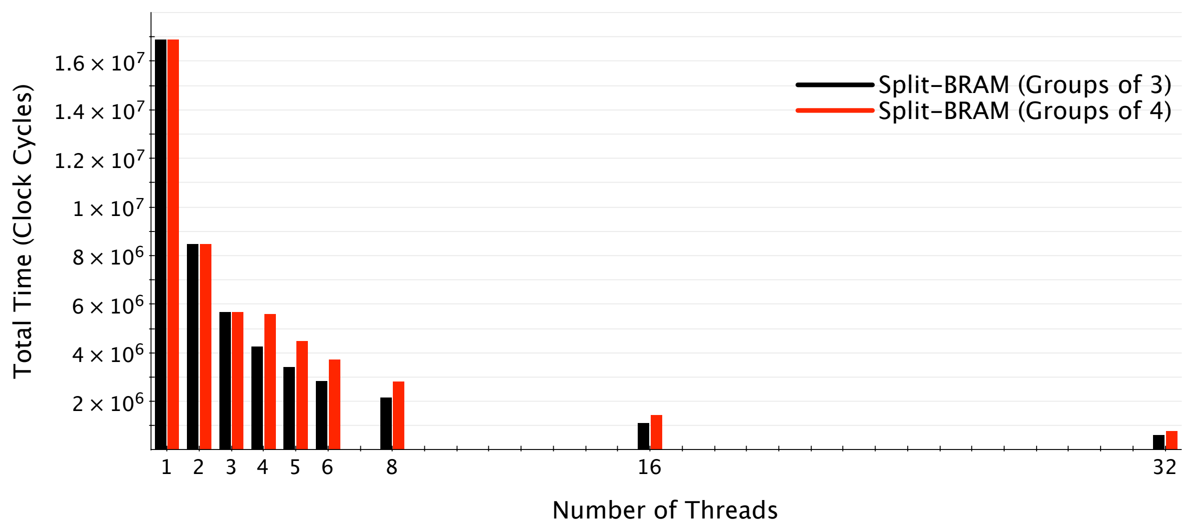

Figure 4.17: Split-BRAM Groups of 3 and 4: Coarse Grained

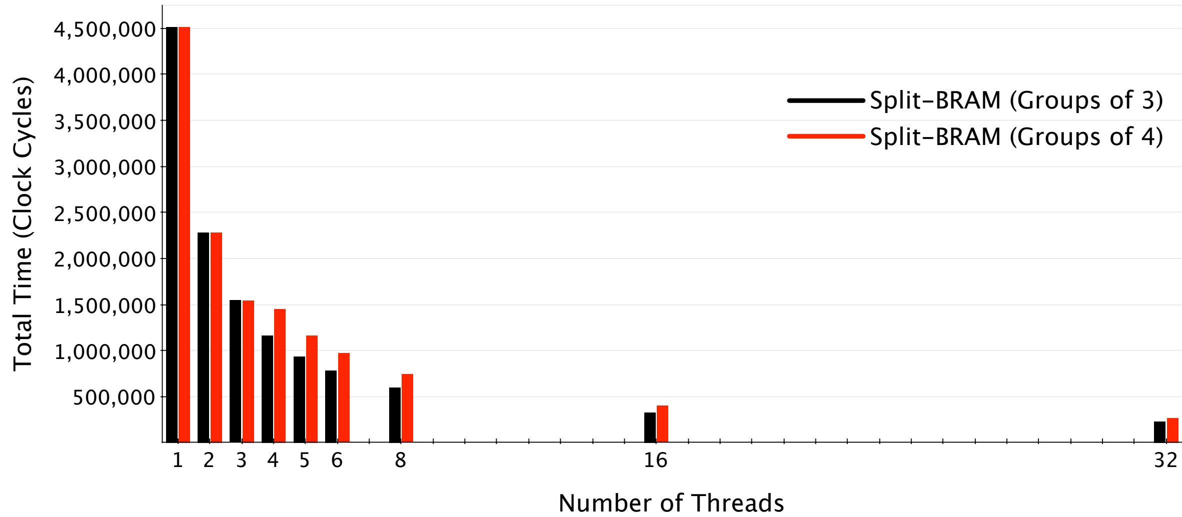

Figure 4.18: Split-BRAM Groups of 3 and 4: Fine Grained

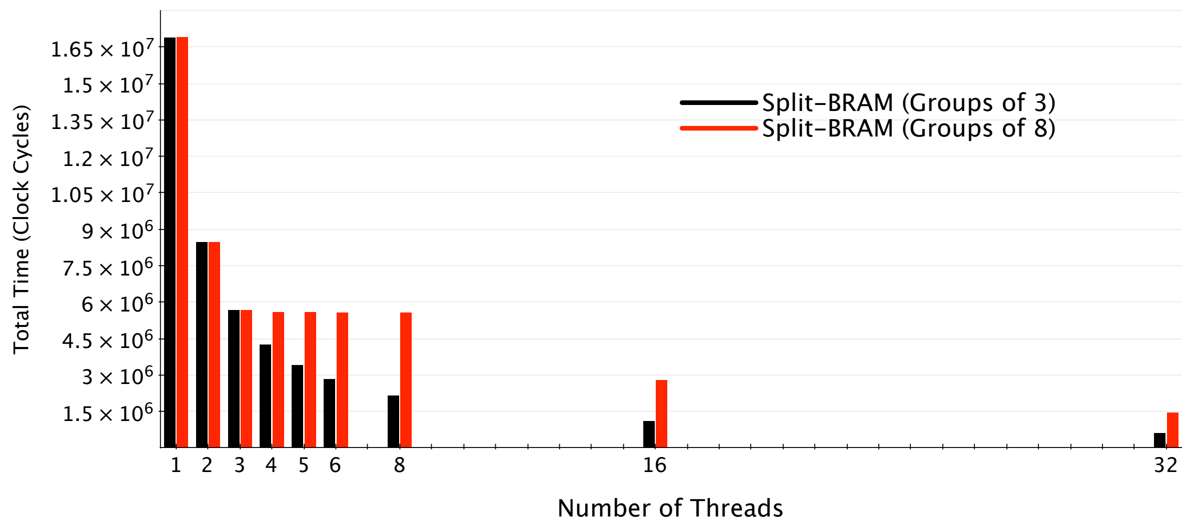

Figure 4.19: Split-BRAM Groups of 3 and 8: Coarse Grained

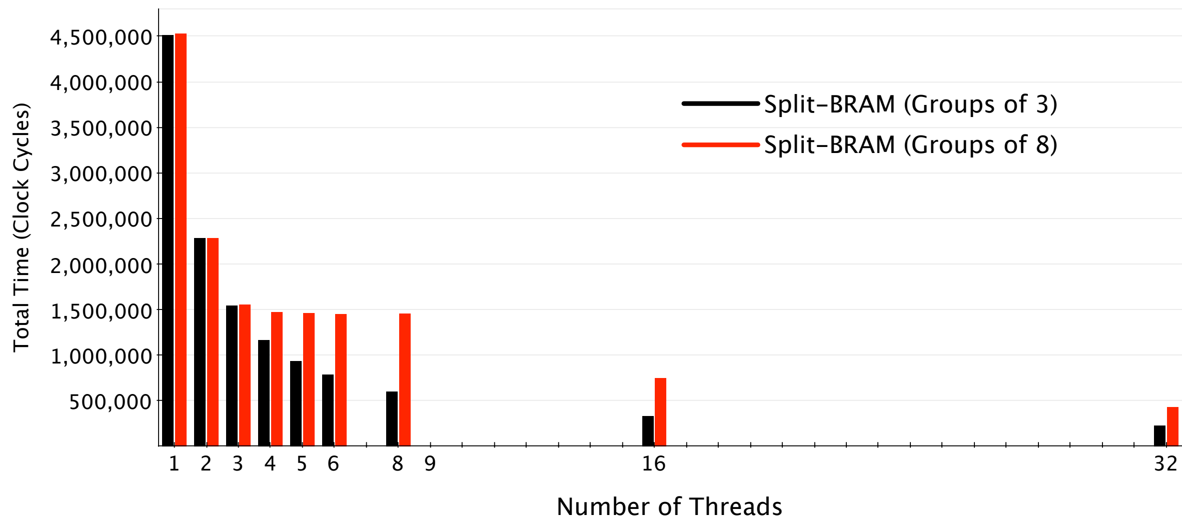

Figure 4.20: Split-BRAM Groups of 3 and 8: Fine Grained

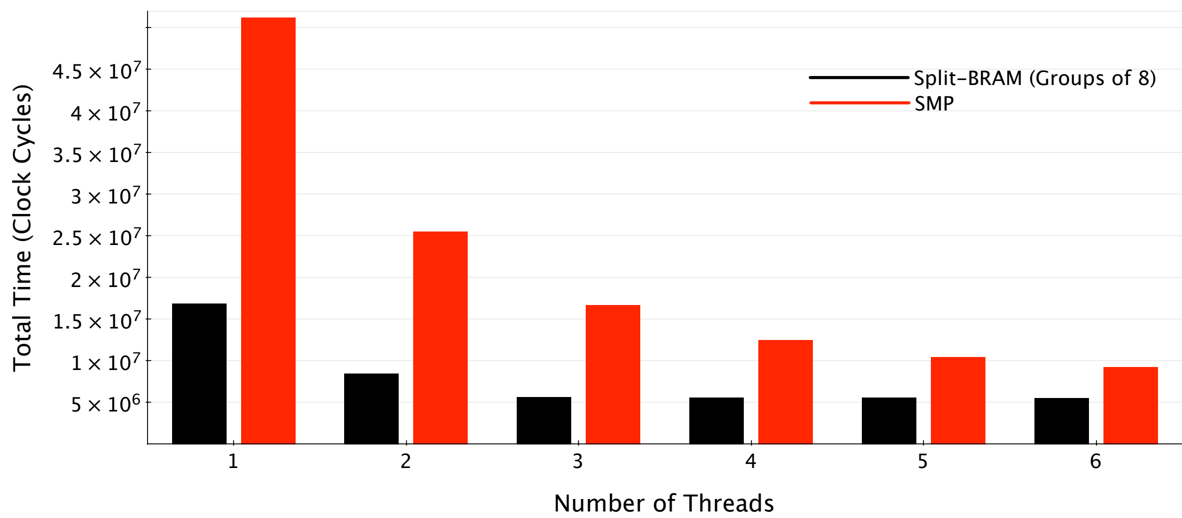

Figure 4.21: Split-BRAM Group of 8 and SMP: Coarse Grained

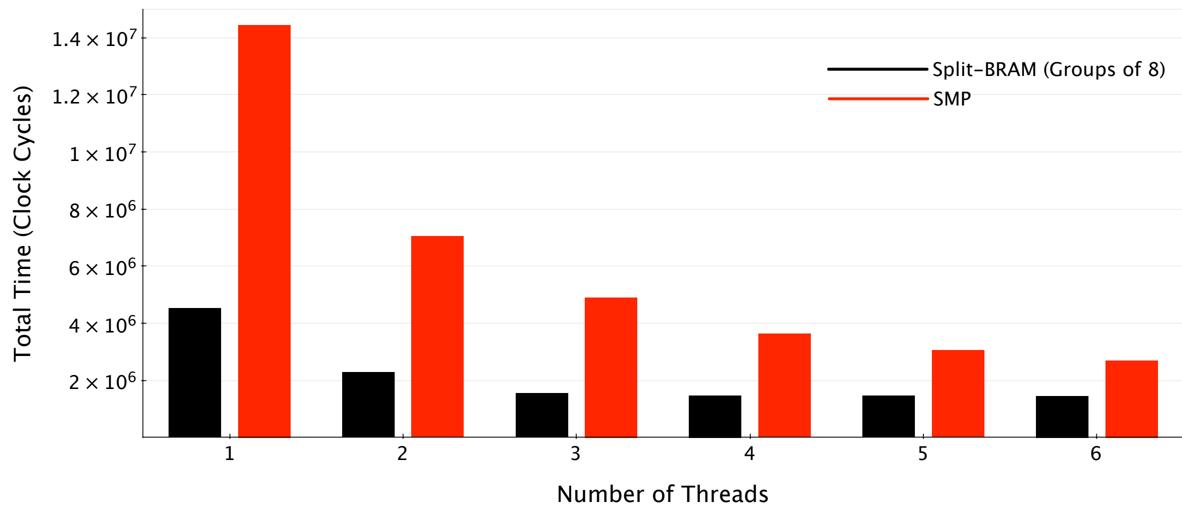

Figure 4.22: Split-BRAM Group of 3 and SMP: Fine Grained

Chapter 5

Conclusion

SMP architectures are known not to scale beyond a few processors due to their centralized memory hierarchy and common bus structure(s). The resulting bottleneck can be remedied through caches, reducing the amount of memory accesses to the common memory. However, caching mechanisms can be expensive in both area and performance for frequent shared-write memory accesses. Distributed memory aims to exploit similar spatial locality with less complexity. Distributed Memory Architectures decentralize the common memory typical within SMP systems, and allow instructions and data to be distributed across local, private memories. The potential memory bandwidth (and address space) over these disjointed memories grows proportionately to the number of processors within the system. Despite the improved scalability of this memory hierarchy, many prefer the shared memory model due to easier programmability and portability. In order for distributed memory systems to be adopted by the software engineering community, they must be able to accommodate some form of linear address space and continue to offer scalable performance without requiring much programming effort.

5.1 Research Contributions

The purpose of this research was to include a linear address space as well as fast, disjointed local memories in order to offer scalable performance on a platform easily targetable across both SMP and distributed memory code bases. The end result consisted of a modular design platform, capable of supporting a 32+ MicroBlaze processor system while continuing to provide cache-like speeds and scalable parallelism. This platform would allow programmers to be more satisfied with the shared memory model as it is maintained in its entirety. The infrastructure for scalable parallelism through distributed memory is provided on top of this model, allowing users to easily adapt shared-memory code to take advantage of distributed memory. Therefore, some of the contributions of

this work include:

- A new HybridThreads system that combines both the distributed memory and shared memory models.
- Integration of transparent transfers of a thread's instructions and data utilizing a Direct Memory Access (DMA) controller.
- A ported HybridThreads system for use on the Virtex ML605 Evaluation board in order to realize such larger systems.
- PowerPC and MicroBlaze host repositories merged into one to allow easy switching across several heterogeneous platforms.
- A new compile flow created, automated, and integrated into the existing heterogeneous compile flow in order to statically or dynamically create, transfer, and retrieve a thread's instructions and data without incurring too much redundancy across transfers.
- An optimized nanokernel and compiled heterogeneous applications that creates smaller binaries and avoids increased bus contention due to unnecessary global memory accesses.

The contributions of this work include the adaptation of a hybrid shared/distributed memory architecture within a hardware microkernel, multiprocessor system. The system allows hardware threads to be scheduled dynamically or statically across different heterogeneous slave processors and provides support for threads to be executed either from a shared memory or a local private memory space, or both. Steps were taken to minimize DMA transfer latency whenever possible within thread creation and join, as well as host processor/OS interactivity among hardware threads. Compilation of hardware thread source files were optimized to not include unneeded thread function code. As the results enforce, this system provides data migration for both instruction and arguments and executes faster over SMP systems employing instruction caches despite the additional transfer latencies. Additionally, both weak and strong scalability have been shown to be beyond acceptable from 1-32 processors within this new platform.

5.2 Future Work

With a new platform capable of building bigger multiprocessor systems and a scalable hardware/-software microkernel, it presents a great exploration of additional research as well as proving an excellent learning tool. Although it has been shown by this work that distributed memory provides a scalable memory hierarchy within a multiprocessor system, how one takes advantage of the memory hierarchy can produce varying results. This can be seen by much of the research efforts out there as there does not exist an all-in-one solution. This can be said about other design challenges within MPSoPCs.

5.2.1 Expanding the Hthreads OS

Being a software/hardware microkernel, Hthreads allows allows great modularities to exist. A possible consideration for future work is Hthreads hardware OS core distribution. Currently, OS cores such as the Synchronization Manager is centralized albeit providing very fast access and response times. As additional processors are added to the system there may be a need to distribute them as similarly done with the memory hierarchy in order to avoid potential bottlenecks associated with centralized MMIO.

Future work may also include the combination of multiple host processors within both the Hthread's SMP and DM systems. This would enable multiple similar or dissimilar applications to exist on the host processors and allow the ability to schedule hardware threads within a slave processor pool. This applies very similarly to modern day desktops whereby there exists a homogeneous multicore processor capable of handing off work to an accelerator such as an FPGA or GPU device. Hthreads has the infrastructure to allow host processors of different ISAs to exist due to the ISA agnostic hardware cores. Therefore, this can provide much more opportunities in areas such as load balancing and dynamic reconfiguration. Additionally, having two or more processors would be beneficial for data migration as it allows concurrency in data transfers to occur. This is not an issue now as transfer latencies were shown to be minuscule compared to actual execution

times of different programs. However, adding additional processors and performing data migration and thread setup for each one sequentially will impact the system's weak scalability eventually.

The limited free space for each slave processor in this new platform is also a limiting factor that deserves future research. Keeping some notion of a shared linear address space is still important in order to ensure easier programming effort and code portability. Today, much of the distributed memory systems use virtual memory in order to expand a linear address space across different and non-contiguous address space memories. However, presenting a linear address space would require rethought of data locality and redundancy to minimize slave processors from executing both instruction and data from a remote address.

5.2.2 Heterogeneous Compilation Flow

Currently, one of the disadvantages of the modified heterogeneous compilation flow is that it does not resolve dependencies within non-thread functions, namely helper functions. This can result in compilation errors where programs such as the IDEA application used in Chapter 4 contain dependencies within helper functions (i.e. helper functions calling other helper functions). This can easily be solved with the use of a dependency tree for each function of the source file, allowing thread functions to continue resolving dependencies as they do now but refer to the dependency tree when writing out the new source file.

A good portion of the binary that is transferred to a slave processor contains code that is never used or executed. As an example, slave processors already execute the nanokernel which waits in an indefinite loop until it is told to bootstrap a function outside of the nanokernel's address range, and then returns to the nanokernel. Therefore, slave processors already have an existing stack space for the nanokernel. When compiling for the MicroBlaze, the higher address regions of an ELF file are empty to correspond to the reserved stack space. The stack space, currently 1KB, can also be shaved off during the embedding stages of the compilation flow. Similarly, other automatic insertions of instructions that are compiled for every program include the interrupt/exception tables. As per the MicroBlaze ABI, this exist in the lower address range of *0x00 - 0x50* and presents 80

bytes that can be shaved off from the embedding process. A more challenging task is reducing included object library files whereby programs calling a single function within the library can safely remove other functions within the library. A possible approach of clustering *needed* functions only can include static code inlining as well as custom *sections* placement but may be limited by the capabilities of the MicroBlaze's C compiler.

References

[1] Jason Agron. *Hardware Microkernels - A Novel Method for Constructing Operating Systems for Heterogeneous Multi-Core Platforms*. Ph.D. Dissertation, University of Arkansas, August 2010.

[2] Erik Konrad Anderson. *Abstracting the Hardware / Software Boundary through a Standard System Support Layer and Architecture*. Ph.D. Dissertation, University of Kansas, May 2007.

[3] D. Andrews, D. Niehaus, R. Jidin, M. Finley, W. Peck, M. Frisbie, J. Ortiz, Ed Komp, and P. Ashenden. Programming models for hybrid fpga-cpu computational components: a missing link. *Micro, IEEE*, 24(4):42 – 53, july-aug. 2004.

[4] Fumio Aono and Masayuki Kimura. The azusa 16-way itanium server. *IEEE Micro*, 20(5):54–60, September 2000.

[5] Vasanth Asokan. Xilinx White Paper 262: Designing Multiprocessor Systems in Platform Studio. http://www.xilinx.com/support/documentation/white_papers/wp262.pdf. Last accessed April 27, 2012.

[6] Xiaowen Chen, Zhonghai Lu, A. Jantsch, and Shuming Chen. Run-time partitioning of hybrid distributed shared memory on multi-core network-on-chips. In *Parallel Architectures, Algorithms and Programming (PAAP), 2010 Third International Symposium on*, pages 39 –46, dec. 2010.

[7] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. Sequoia: programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[8] Michael J. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers*, C-21:948–960, 1972.

[9] Message Passing Interface Forum. Document for a standard message-passing interface. Technical report, University of Tenessee, Nov 1993. Available on www.netlib.org.

[10] ARM Holdings. The arm cortex-a9 processors. White paper V2.0, sep 2009. Available online on http://www.arm.com.

[11] Wilson C. Hsieh, Paul Wang, and William E. Weihl. Computation migration: enhancing locality for distributed-memory parallel systems. In *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '93, pages 239–248, New York, NY, USA, 1993. ACM.

[12] IBM. 128-Bit Processor Local Bus Architecture Specifications (Version 4.7). https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/3BBB27E5BCC165BA87256A2B0064FFB4/$file/PlbBus_as_01_pub.pdf. Last accessed April 27, 2012.

[13] A. Macii, E. Macii, and M. Poncino. Improving the efficiency of memory partitioning by address clustering. In *Design, Automation and Test in Europe Conference and Exhibition, 2003*, pages 18 – 23, 2003.

[14] E.J. Marinissen, B. Prince, D. Keltel-Schulz, and Y. Zorian. Challenges in embedded memory design and test. In *Design, Automation and Test in Europe, 2005. Proceedings*, pages 722 – 727 Vol. 2, march 2005.

[15] J. Protic, M. Tomasevic, and V. Milutinovic. A survey of distributed shared memory systems. In *Proceedings of the 28th Hawaii International Conference on System Sciences*, pages 74–, Washington, DC, USA, 1995. IEEE Computer Society.

[16] Andrew Schmidt. *Quantifying Effective Memory Bandwidth on Platform FPGAs*. Master's Thesis, University of Kansas, April 2007.

[17] M. Vuletic, L. Pozzi, and P. Ienne. Virtual memory window for application-specific reconfigurable coprocessors. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 14(8):910 –915, aug. 2006.

[18] Miljan Vuletic, Laura Pozzi, and Paolo Ienne. Dynamic prefetching in the virtual memory window of portable reconfigurable coprocessors. In *In Proceedings of the 14th International Conference on Field-Programmable Logic and Applications*, pages 596–605, 2004.

[19] Xilinx. MicroBlaze Processor Reference Guide. http://www.xilinx.com/support/documentation/sw_manuals/mb_ref_guide.pdf. Last accessed April 27, 2012.

[20] Xilinx. Multi-Port Memory Controller LogiCore Datasheet. http://www.xilinx.com/support/documentation/ip_documentation/mpmc.pdf. Last accessed April 27, 2012.

[21] Xilinx. PLBv46 to PLBv46 Bridge. http://www.xilinx.com/support/documentation/ip_documentation/plbv46_plbv46_bridge.pdf. Last accessed April 27, 2012.

[22] Xilinx. Processor Local Bus v4.6. http://www.xilinx.com/support/documentation/ip_documentation/ds531.pdf. Last accessed April 27, 2012.

[23] Xilinx. Virtex-5 FPGA Family Brochure. http://www.xilinx.com/publications/prod_mktg/Virtex_family_brochure.pdf. Last accessed April 27, 2012.

[24] Xilinx. Virtex-6 FPGA Product Table. http://www.xilinx.com/publications/prod_mktg/Virtex6_Product_Table.pdf. Last accessed April 27, 2012.

[25] Xilinx. XPS Block RAM Interface Controller. http://www.xilinx.com/support/documentation/ip_documentation/xps_bram_if_cntlr.pdf. Last accessed April 27, 2012.

[26] Liping Xue, Ozcan ozturk, Feihui Li, Mahmut Kandemir, and I. Kolcu. Dynamic partitioning of processing and memory resources in embedded mpsoc architectures. In *Proceedings of the conference on Design, automation and test in Europe: Proceedings*, DATE '06, pages 690–695, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.