Hardware Support for Productive
Partitioned Global Address Space (PGAS) Programming

by Olivier Serres

M. Eng. in Computer Science, September 2007,
University of Technology of Belfort-Montbéliard, France

MRes. in Mechatronics, January 2008,
University of Technology of Belfort-Montbéliard, France

A Dissertation submitted to

The Faculty of
The School of Engineering and Applied Science
of The George Washington University
in partial satisfaction of the requirements
for the degree of Doctor of Philosophy

January 31, 2016

Dissertation directed by

Tarek El-Ghazawi
Professor of Engineering and Applied Science

The School of Engineering and Applied Science of The George Washington University certifies that Olivier Serres has passed the Final Examination for the degree of Doctor of Philosophy as of December 18, 2015. This is the final and approved form of the dissertation.

Hardware Support for Productive
Partitioned Global Address Space (PGAS) Programming

Olivier Serres

Dissertation Research Committee:

Tarek El-Ghazawi, Professor of Engineering and Applied Science, Dissertation Director

Suresh Subramaniam, Professor of Engineering and Applied Science, Committee Member

Vikram Krishna Narayana, Assistant Research Professor of Engineering and Applied Science, Committee Member

Tian Lan, Assistant Professor of Electrical and Computer Engineering, Committee Member

Jaafar Gaber, Professor of Computer Science at University of Technology of Belfort-Montbéliard, Committee Member

William W. Carlson, Researcher at IDA Center for Computing Sciences, Committee Member

Tyler Simon, Researcher at DoD, Committee Member

© Copyright 2015 by Olivier Serres
All rights reserved

I would like to dedicate this dissertation to my

mother and father,

brother,

grandparents,

and all families and friends

who have given me their support

throughout this journey.

Acknowledgments

This Ph.D. program would not have been possible for me without the guidance, friendship and support of so many people who in one way or another contributed to the completion of this program.

First and foremost, I am very grateful to my advisor Prof. Tarek El-Ghazawi, who has supported me throughout my thesis with his patience and knowledge. As the founding director of the High Performance Computing Laboratory (HPCL) at The George Washington University (GWU), co-director of the NSF Industry/University Center for High-Performance Reconfigurable Computing (CHREC) and director of IMPACT (The Institute for Massively Parallel Applications and Computing Technologies), Prof. El-Ghazawi gave me many opportunities to work on many interesting projects, with many institutions (including the Arctic Region Supercomputing Center (ARSC) and the National Institute of Standards and Technology - NIST) and also allowed me to experience all the different aspects of the academic life. In HPCL, I could also become familiar with many state-of-the-art supercomputers. I am certain that the skills and experiences acquired at HPCL are invaluable assets for my future career.

I would also like to thank my committee members Prof. Suresh Subramaniam, Prof. Vikram K. Narayana, Prof. Tian Lan, Prof. Jaafar Gaber, Dr. William Carlson and Dr. Tyler Simon for their time and extremely valuable feedback.

I also would like to thank the people of the Arctic Region Supercomputing Center (ARSC). Their warm welcome and technical guidance were exceptional. I want to thank Dr. Frank Williams and Dr. Greg Newby for making my summer internships there a great experience.

I also learned a great deal while working at the National Institute of Standards and Technology (NIST). I want to thank everybody from the Multimodal Information Group, especially Dr. Martial Michel and Dr. Mark Przybocki.

I am very thankful to the members of the HPCL group, a place I called home for so many years. They have contributed immensely to my personal and professional development at GWU. Therefore, I would like to express my gratitude to all current and past group members: Dr. Vikram Narayana, Dr. Saumil Merchant, Dr. Joseph Schneible, Dr. Esam El-Araby, Dr. Miaoqing Huang, Dr. Proshanta Saha, Dr. Suboh Suboh, Dr. Lubomir Riha, Dr. Mohamed Taher, Dr. Ivan Gonzales, Dr. Mohamed Bakhouya, Dr. Harald Simmler, Dr. Zeki Bozkus, Dr. Abdullah Kayi, Dr. Asilah Hamid, Dr. Abdullah Al-Dahlawi, Dr. Gangotri Dey, Dr. Ruben Tikidzhi-Khamburyan, Ahmad Anbar, Maria Malik, Mohamed Abou-Ellail, Preetham Nosum, Yiyi Yao, Kun Xi, Samy Al-Bahra, Gabriel Yessin, Engin Kayraklioglu, David Newsom, Diksha Sharma, Krunal Puri and Jiaxin Peng. It was a great pleasure to work with all of them.

I would also like to deeply thank a few people for being very good friends. Their help and patience were absolutely essential in this journey: Abdullah Kayi who gave me great support and many good ideas, David Newsom who was a great collaborator and an important source of motivation, Vikram Narayana for his technical expertise and our many long conversations, Ahmad Anbar for his help on many projects, Jaafar Gaber who guided me along the way and flew all the way from France for my defenses, and Mingfeng Zhang who helped me through the last steps of my dissertation.

Finally, I would like to dedicate this dissertation to my family because it would not have happened without their infallible support.

Abstract

Hardware Support for Productive Partitioned Global Address Space (PGAS) Programming

In order to exploit the increasing number of transistors, and due to the limitations of frequency scaling, the number of cores inside a chip keeps growing. As many-core chips become ubiquitous, there is a greater need for a more productive and efficient parallel programming model. The easy-to-use, but locality-agnostic, shared memory model (e.g. OpenMP) is unable to efficiently exploit memory locality in systems with Non-Uniform Memory Access (NUMA) and Non-Uniform Cache-Access (NUCA) effects. The locality-aware, but explicit, message-passing model (e.g. MPI1) does not provide a productive development environment due to its two-sided communication and a distributed (and isolated) memory model.

The Partitioned Global Address Space (PGAS) programming model strikes a balance between those two extremes via a global address space that is provided for ease-of-use, but is partitioned for locality awareness. The user-friendly PGAS memory model, however, comes at a performance cost, due to the needed address mapping, which can hinder its potential for performance. To mitigate this overhead and achieve full performance, compiler optimizations may be applied, but are often insufficient. Alternatively, manual optimizations can be applied but they are quite cumbersome and, as such, are unproductive. As a result, the overall benefit of PGAS has been severely limited. In this dissertation, we improved both the productivity and performance of PGAS by introducing a novel hardware support. This PGAS hardware support efficiently handles the complex PGAS mapping and communication without the intervention of an application developer. By introducing the new hardware at the micro-architecture level, fine grain and low latency local shared memory accesses are supported. The hardware is also made available through an ISA extension, so that it can easily be exploited by PGAS compilers to efficiently access and traverse

the PGAS memory space. The automatic code generation eliminates the need for hand-tuning, and thus simultaneously improve both the performance and productivity of PGAS languages. This research also introduces and evaluates the possibility for the hardware support to handle a variety of PGAS languages.

Results are obtained on two different system implementations: the first is based on the well-adopted full system simulator Gem5, which allows the precise evaluation of the performance gain. Two prototype compilers supporting the new hardware are created for experimentation by extending the Berkeley Unified Parallel C (UPC) compiler and the Cray Chapel compiler. This allows unmodified code to use the new instructions without any user intervention, thereby creating a productive programming environment. The second proof-of-concept implementation is a hardware prototype based on the multi-core Leon3 softcore processor running on a Virtex-6 FPGA. This allowed us to not only verify the feasibility of the implementation but also to evaluate the cost of the new hardware and its instructions.

This research has shown very promising results. With benchmarks in UPC and Chapel including the NAS Parallel Benchmarks implemented in UPC, a speedup of up to 5.5x is demonstrated when using the hardware support with unmodified codes. Unmodified code performance using this hardware was shown to also surpass the performance of manually optimized UPC code in some of the cases by up to 10%. With Chapel, we obtained measurable speed-ups of up to 19x. Additionally, the hardware prototype demonstrated that only a very small area increase is needed.

Table of Contents

Dedication . . . . . **iv**

Acknowledgments . . . . . **v**

Abstract . . . . . **vii**

Table of Contents . . . . . **ix**

List of Figures . . . . . **xii**

List of Tables . . . . . **xiv**

List of Acronyms . . . . . **xv**

1 Introduction **1**

1.1 Background and Motivations **1**

1.2 Partitioned Global Address Space Programming Model **2**

1.2.1 Unified Parallel C 6

1.2.2 Chapel 11

1.3 Shared Address Overheads **13**

1.3.1 UPC 13

1.3.2 Chapel 14

1.4 Problem Statement and Objectives **16**

1.5 Scope and Limitations **17**

1.6 Organization of this Dissertation **17**

2 Related Work **19**

2.1 PGAS Productivity or Performance **19**

2.1.1 Productivity 19

2.1.2 Performance 21

2.2 Optimizations to Reduce Shared Access Overheads **22**

2.2.1 Software Optimizations 22

2.2.1.1 Local Shared 22

2.2.1.2 Remote Shared . . . 24
2.2.2 Hardware Support . . . 25
2.3 Summary of Related Work . . . 26

3 PGAS Hardware Support Design . . . 28
3.1 Memory Mapping and Shared Space Traversal . . . 28
3.1.1 UPC . . . 28
3.1.2 Chapel . . . 30
3.2 Hardware Design for UPC . . . 30
3.2.1 UPC Memory Model Overheads . . . 30
3.2.2 The UPC Memory Model Hardware Support . . . 33
3.2.3 Instruction Set Extension . . . 35
3.3 Generalization of the Hardware Design for Chapel . . . 36

4 Performance and Productivity Study . . . 38
4.1 Full-System Simulation . . . 38
4.2 Compilation Support for Productivity . . . 40
4.2.1 UPC . . . 41
4.2.2 Chapel . . . 42
4.3 Experimental Toolset . . . 47
4.4 Performance Results . . . 49
4.4.1 Gem5 Memory Models . . . 49
4.4.2 UPC . . . 49
4.4.2.1 Atomic Model . . . 54
4.4.2.2 Timing Model . . . 63
4.4.2.3 Detailed Model . . . 63
4.4.3 Chapel . . . 64
4.4.3.1 Atomic Model . . . 66
4.4.3.2 Timing Model . . . 67
4.4.4 Summary of Findings . . . 69

5 Area and Timing Study . . . 71
5.1 Hardware Support Implementation on FPGAs . . . 71
5.1.1 Platform for Multi-core Research . . . 71
5.1.1.1 Related Work on Reconfigurable Processors . . . 73

5.1.1.2 Leon3 . . . 74
5.1.1.3 Coprocessor Interface . . . 75
5.1.1.4 Multi-core Self-Reconfigurable System . . . 79
5.1.1.5 Linux Kernel Driver . . . 80
5.1.1.6 Platform Evaluation . . . 81
5.1.2 FPGA Implementation Details of the PGAS Hardware . . . 85
5.2 Performance Evaluation . . . 88
5.3 Performance Sensitivity to Instruction Latencies . . . 90
5.4 Chip Area Dedicated to the Hardware Support . . . 93

6 Conclusions and Future Directions . . . 96
6.1 Summary and Conclusions . . . 96
6.2 Future Directions . . . 97
6.2.1 Other Data Distributions . . . 97
6.2.2 Extended Network Interface . . . 99

List of Figures

1.1 Architecture of the TILE64 Chip . . . 3
1.2 Comparison of the Main Parallel Programming Models . . . 4
1.3 UPC Memory Model . . . 6
1.4 UPC Memory Layout for `arrayA`, some shared pointers are also represented . . . 6
1.5 Vector Summation in UPC - Memory Layout . . . 9
1.6 Chapel Code Example: STREAM Scale . . . 12
1.7 Memory Access Cost for a 64-bit Write (on a dual quad-core AMD cluster) . . . 13
1.8 Multi-core vs. Many-core (Comparison of the sequential C version with a single UPC thread) . . . 15
1.9 Local Access Overhead in Chapel . . . 16

3.1 Shared Pointer Incrementing Algorithm . . . 29
3.2 Chapel Block Distribution: dsiAccess . . . 29
3.3 Shared Pointer Incrementing Flowchart - Conceptual View . . . 31
3.4 Shared Pointer Incrementing - Hardware Implementation . . . 32
3.5 UPC Memory Layout for `shared [4] int arrayA[32];` . . . 34

4.1 Format of the New Alpha Instructions . . . 39
4.2 Gem5 Atomic Model: NAS Parallel Benchmark - EP class W . . . 50
4.3 Gem5 Atomic Model: NAS Parallel Benchmark - CG class W . . . 50
4.4 Gem5 Atomic Model: NAS Parallel Benchmark - FT class W . . . 51
4.5 Gem5 Atomic Model: NAS Parallel Benchmark - IS class W . . . 51
4.6 Gem5 Atomic Model: NAS Parallel Benchmark - MG class W . . . 52
4.7 Gem5 Atomic Model: Matrix Multiplication 512x512 . . . 52
4.8 Gem5 Atomic Model: Sobel benchmark 4096x4096 . . . 53
4.9 Gem5 Atomic Model: Heat benchmark 64x64x64 . . . 53
4.10 Gem5 Timing Model: NAS Parallel Benchmark - CG class W . . . 55
4.11 Gem5 Timing Model: NAS Parallel Benchmark - FT class W . . . 56
4.12 Gem5 Timing Model: NAS Parallel Benchmark - IS class W . . . 56
4.13 Gem5 Timing Model: NAS Parallel Benchmark - MG class W . . . 57

4.14 Gem5 Timing Model: Matrix Multiplication 512x512 57
4.15 Gem5 Timing Model: Sobel benchmark 4096x4096 58
4.16 Gem5 Timing Model: Heat benchmark 64x64x64 58
4.17 Gem5 Detailed Model: NAS Parallel Benchmark - CG class W 59
4.18 Gem5 Detailed Model: NAS Parallel Benchmark - FT class W 60
4.19 Gem5 Detailed Model: NAS Parallel Benchmark - IS class W 60
4.20 Gem5 Detailed Model: NAS Parallel Benchmark - MG class W 61
4.21 Gem5 Detailed Model: Matrix Multiplication 512x512 61
4.22 Gem5 Detailed Model: Sobel benchmark 4096x4096 62
4.23 Gem5 Detailed Model: Heat benchmark 64x64x64 62
4.24 Gem5 Atomic Model: Chapel STREAM Scale - 4M 64
4.25 Gem5 Atomic Model: Chapel Matrix Multiplication 128x128 65
4.26 Gem5 Atomic Model: Chapel Random Access 65
4.27 Gem5 Atomic Model: Chapel 3D Heat Diffusion 64x64x64 66
4.28 Gem5 Timing Model: Chapel STREAM Scale - 4M 67
4.29 Gem5 Timing Model: Chapel Matrix Multiplication 128x128 68
4.30 Gem5 Timing Model: Chapel Random Access 68
4.31 Gem5 Timing Model: Chapel Heat Diffusion 64x64x64 69

5.1 4 Core Leon3 SMP with PGAS Support 72
5.2 The Leon3 and the Proposed Coprocessor Pipelines. (Also represented, the pipeline timing for the load and store operations between the two units.) 75
5.3 Reduced VHDL Interface for the Coprocessor 77
5.4 Leon3 SMP with Reconfigurable Coprocessors 79
5.5 Placement of a 4-Core SMP Leon3 Design with Reconfigurable Coprocessors (the four squares represent the reconfigurable areas) 83
5.6 Scalability of the Modified Stream Test, (showing the contention on the AMBA bus) 84
5.7 The 7-stage Leon3 Pipeline Extended with PGAS Support 86
5.8 Leon3 with PGAS HW Support - Benchmarks 89

6.1 Distributions based on a Bit Select Function 98
6.2 Bit Select Function Hardware 98
6.3 Many-core Hardware Support for PGAS 99

List of Tables

2.1 Classification for the Related Work on Reducing the Shared Access Overheads . . . 23
2.2 Summary of Notable Related Work in Reducing the Overheads for Accessing the PGAS Space . . . 26

4.1 Instructions Added to the Alpha ISA . . . 39

5.1 The SPARC Instructions Reserved for the Coprocessor . . . 78
5.2 Area Cost for Different Configurations of Leon3 - Slice Resources . . . 81
5.3 Area Cost for Different Configurations of Leon3 - BRAM and DSPs . . . 82
5.4 Area Cost per Additional Feature to the Configuration of Leon3 . . . 82
5.5 Modified Stream Benchmark Performance Result (MB/s) . . . 84
5.6 Implementation time for a 4 core SMP design . . . 85
5.7 Leon3 Configuration . . . 87
5.8 PGAS Hardware Support SPARC V8 ISA Extension . . . 87
5.9 PGAS Instruction Stall Cycles Sensitivity Study - CG . . . 90
5.10 PGAS Instruction Stall Cycles Sensitivity Study - FT . . . 90
5.11 PGAS Instruction Stall Cycles Sensitivity Study - IS . . . 91
5.12 PGAS Instruction Stall Cycles Sensitivity Study - MG . . . 91
5.13 PGAS Instruction Stall Cycles Sensitivity Study - MM . . . 92
5.14 PGAS Instruction Stall Cycles Sensitivity Study - Sobel . . . 92
5.15 PGAS Instruction Stall Cycles Sensitivity Study - Heat . . . 93
5.16 Area Cost Evaluation for the PGAS Hardware Support . . . 93
5.17 Evaluation of the Transistor Count for the Base Leon3 Core . . . 94
5.18 Evaluation of the Transistor Count for a Leon3 Core with the PGAS Hardware Support . . . 94

List of Acronyms

AHB	Advanced Microcontroller Bus Architecture (AMBA) High-Performance Bus
ASIC	Application-Specific Integrated Circuit
BRAM	Block Random Access Memory
BWX	Byte Word Extensions (Alpha instruction set extension)
CB	Coprocessor Branch
ccNUMA	cache-coherent Non Uniform Memory Access
CIX	Count Extensions (Alpha instruction set extension)
CMP	Chip Multiprocessor
CPU	Central Processing Unit
DDR2	Double Data Rate Synchronous Dynamic Random-Access Memory interface
DRAM	Dynamic Random-Access Memory
FIX	Floating-Point Extensions (Alpha instruction set extension)
FPGA	Field-Programmable Gate Array
FPU	Floating Point Unit
GAS	Global-Address Space
GASNet	Global-Address Space Networking, a low-level networking layer for GAS HPC communication
GCC	GNU Compiler Collection
gem5	Modular discrete event driven computer system simulator platform
HDL	Hardware Description Language
HPCC	HPC Challenge
HW	Hardware
ICAP	Internal Configuration Access Port (allows for FPGA self-reconfiguration)
ILP	Instruction Level Parallelism
IOCTL	Input/Output Control device (a Linux interface)
IP	Intellectual Property
JTAG	Joint Test Action Group (standard for a dedicated debug port)
LUT	Look Up Table
MMU	Memory Management Unit
MOSFET	Metal-Oxide-Semiconductor Field-Effect Transistor
MPI	Message Passing Interface
MVI	Motion Video Instructions (Alpha instruction set extension)

NAS	NASA Advanced Supercomputing Division
NPB	NAS Parallel Benchmarks
NUCA	Non Uniform Cache Access
NUMA	Non Uniform Memory Access
OpenMP	Open Multi-Processing
PGAS	Partitioned Global Address Space
PSHM	inter-Process SHared Memory
RDMA	Remote Direct Memory Access
SGE	Sun Grid Engine
SLOC	Source Lines Of Code
SMP	Symmetric Multiprocessor
TLB	Translation Lookaside Buffer
TLP	Thread Level Parallelism
UDP	User Datagram Protocol
UPC	Unified Parallel C
VHDL	VHSIC Hardware Description Language

Chapter 1 - Introduction

1.1 Background and Motivations

In 1965, it was observed by Gordon E. Moore that the number of transistors in an integrated circuit had been doubling every two years [1]. The well-known Moore's Law describing the exponential growth in transistor count is still in effect today. Growth in performance has also been aided by Dennard scaling[1], which states that as the transistors get smaller, their power density remains a constant while accommodating higher frequencies (i.e., the power usage is proportional to the chip area) [2]. However, this scaling ceased to hold true since around 2005, leading to power constraints that considerably limit performance improvements from frequency scaling. Additionally, increasing transistor counts do not yield commensurate gains in performance. This is captured by Pollack's rule, which states that a doubling in silicon real-estate (at a given technology node) is accompanied by only a $\sqrt{2}$ times increase in single-processor performance [3]. Micro-architectural innovations such as improvements to out-of-order execution, faster caches and better branch predictors are constantly being explored by the industry. Nevertheless, as we get closer to the exploitable ILP limits, these techniques have provided diminishing returns in performance. Furthermore, a growing processor-memory gap due to large DRAM access times has not helped.

Computer architects have moved toward SMP systems, then multi-core designs (system with multiple cores per CPUs) and toward many-core CPUs in order to efficiently exploit the growing number of transistors. This mitigates the effects of memory stalls and allows the exploitation of higher levels of parallelism. The multi-core revolution has quickly permeated all classes of computing, from embedded to high-performance systems. Single-core chips are now the exception rather than the norm in desktop computers, and even embedded

[1] also known as MOSFET scaling

systems such as cell-phones may contain octa-core chips. For high-performance computing, many-core chips such as the Intel Xeon Phi (61 cores, and 4 concurrent threads running per core) provide the bulk of the computing power for the fastest supercomputer[2].

The main difficulty now lies in effectively exploiting those extra cores. Although automatic parallelization of code is a hot research topic, it is not clear if compilers will ever be efficient at parallelizing real computer programs. Parallel programming is also notoriously difficult. On top of all the difficulties of writing computer programs, the programmer needs to correctly distribute the computation across all cores and orchestrate the communications for maximum performance. In the more complicated cases, the programs may need to be redesigned with new algorithms tailored for parallel computing.

The parallel programming difficulty is particularly apparent for multi- and many- core systems. For such systems, deep memory hierarchies are common. This creates Non-Uniform Memory Access (NUMA) and Non-Uniform Cache Access (NUCA) effects, complicating the programmer's task. For example, the TILE64 chip [4] (Figure 1.1) is composed of 64 cores interconnected through a grid of 5 on-chip mesh networks [5]. Four memory controllers are used to access the memory, creating a NUMA system on chip. On top of that, the Level 2 cache uses a homing mechanism: each memory page is assigned a home cache and during a cache miss the data is first requested to the home cache, creating NUCA effects.

The growing complexity of parallel systems and the ever-present need for shorter time-to-solution has created an imperative need for more productive programing models.

1.2 Partitioned Global Address Space Programming Model

Considering parallel programming models, there have always been trade-offs between programmability and performance. For instance, the two most widely accepted parallel

[2]The Tianhe-2 is the world's most powerful supercomputer (ranked #1 of the TOP500 since June 2013). It includes 48,000 Xeon Phi 31S1P providing 48 PFlops of raw power.

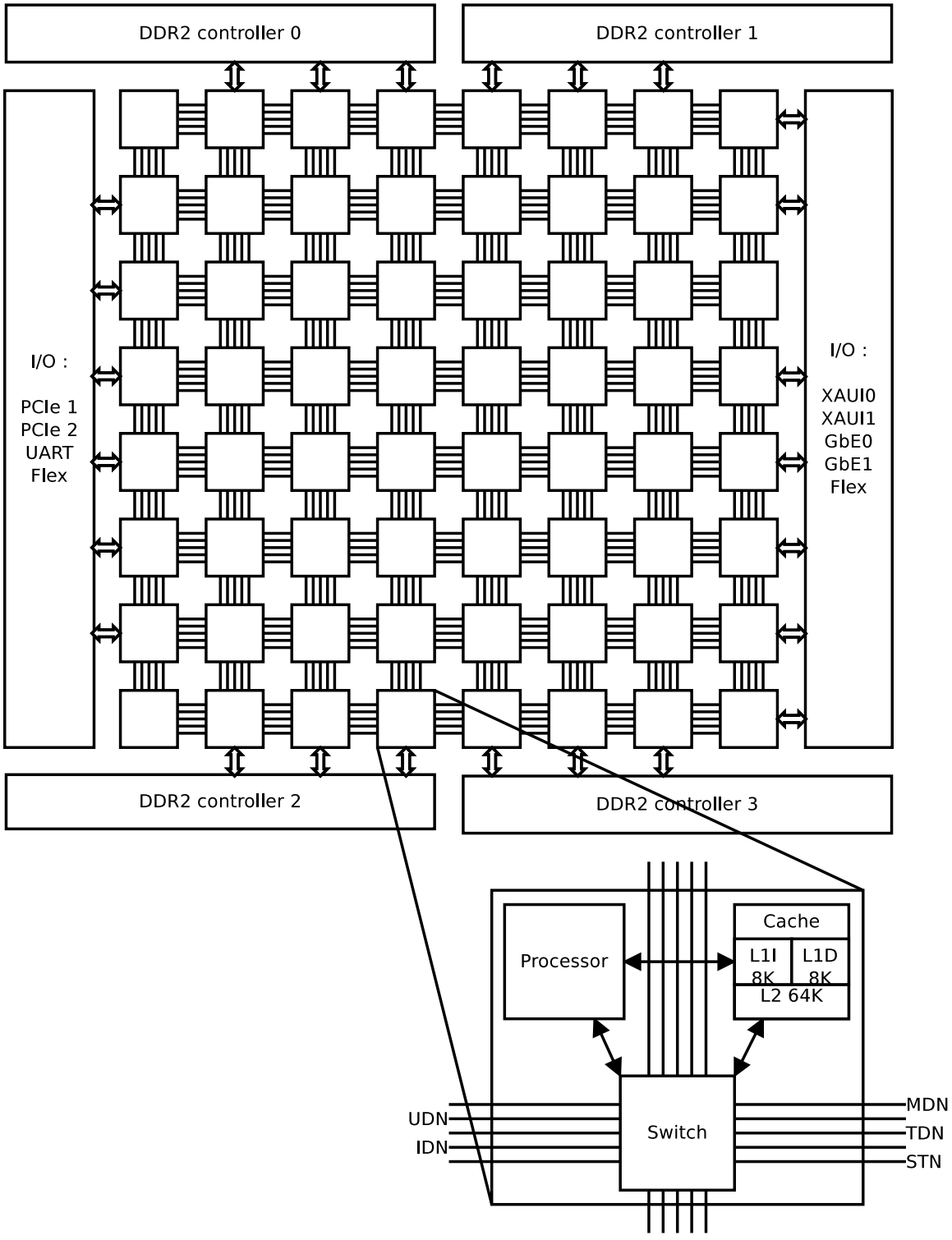


Figure 1.1: Architecture of the TILE64 Chip

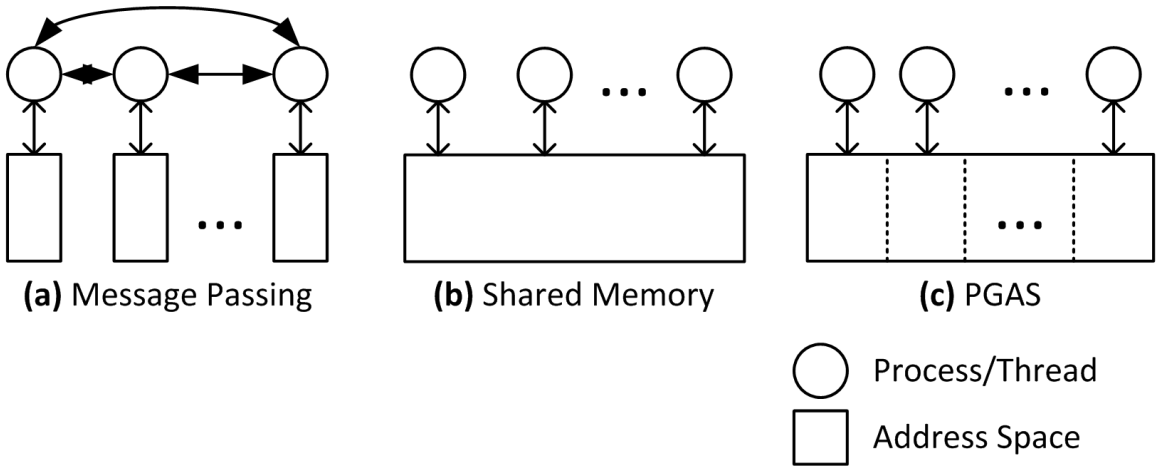


Figure 1.2: Comparison of the Main Parallel Programming Models

programming models, shared memory and message passing, have both their own advantages and disadvantages.

The shared memory model gives the programmer an easy-to-program shared view of the memory (Figure 1.2.b), that allows one-sided communication. At the same time, the shared memory model does not have any notion of data locality. As a result, tasks may frequently work on remote data, due to a lack of information available to the programmer in terms of which data are local and which are remote to each task. This can lead to a severe degradation in performance.

On the other hand, the message passing model (Figure 1.2.a) allows the programmers to fully exploit the data locality to achieve high performance at the cost of productivity. As this model allows only explicit two-sided communication, it is the programmer's responsibility to explicitly express the data movements, both at the sender and receiver tasks.

The Partitioned Global Address Space (PGAS) programming model (Figure 1.2.c) combines the desirable characteristics from both the aforementioned models. The PGAS term was coined by Tarek El-Ghazawi and Katherine Yelick in 2002 [6]. It represents languages with a Global Address Space (GAS) giving the programmer an easy to use shared view of the memory, thus offering a productive environment. The GAS is also partitioned which

allows a good exploitation of the locality of the data, for performance.

Multiple languages that conform to the PGAS model are available, including: Unified Parallel C [7], Co-Array Fortran [8], Chapel [9], X10 [10], etc.

However, this partitioned, yet global view of the memory requires a complex address mapping from the programmer's view of memory to the actual physical layout of the memory. This mismatch necessitates two major elements (1) a new way of representing shared addresses in the global view, and (2) the translation between the global representation and the virtual address representation. For example, in the Unified Parallel C (UPC) language, three different fields are necessary to represent a shared address. This creates a significant overhead within the runtime system to translate this representation to a regular memory address. This forces users to manually optimize their codes, greatly increasing the programming complexity, which, in turn, reduces the overall productivity gain. Manual optimizations to improve performance generally include complex pointers and MPI-like messaging calls, which significantly degrades the PGAS productivity.

To solve this issue, we propose a specific hardware support to handle the complex PGAS address mapping tasks. The hardware support is directly introduced in the processor core via new instructions. A PGAS compiler can directly make use of such instructions to efficiently translate from the shared address representation to the actual memory address on the target machine. This eliminates the need for manual tuning of the code by closing the performance gap between shared pointer addressing and private pointer addressing.

The work in this thesis is applicable to most PGAS implementations. But, as a case study in this dissertation, we chose to use two different PGAS programming languages: UPC (Unified Parallel C) and Chapel. They are presented in detail in the following sections.

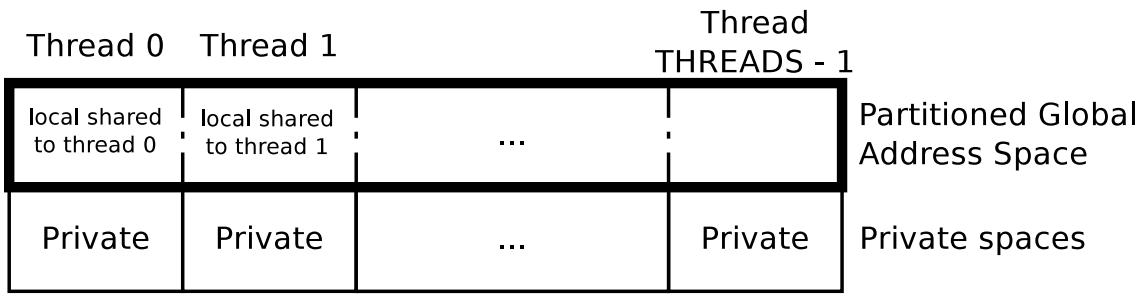


Figure 1.3: UPC Memory Model

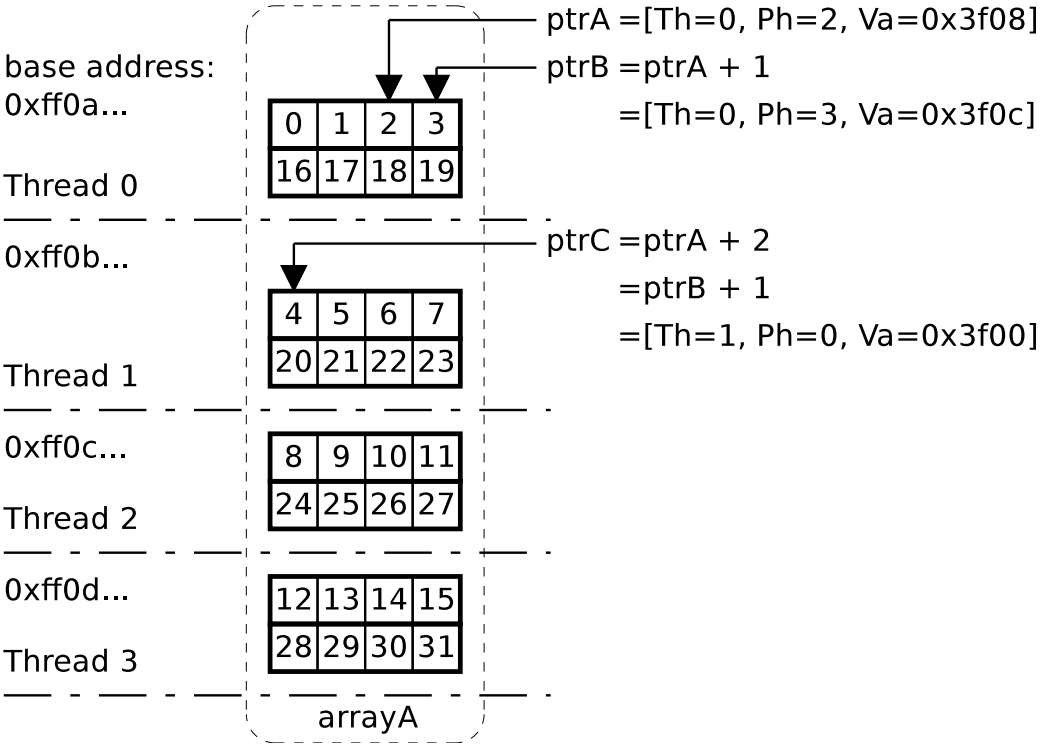


Figure 1.4: UPC Memory Layout for `arrayA`, some shared pointers are also represented

1.2.1 Unified Parallel C

Unified Parallel C (UPC) is a parallel extension of ISO C 99 programming language that follows the PGAS model. The first specifications of UPC, version 1.0, were introduced by El-Ghazawi, Carlson, and Draper in 2001. Today, UPC is available on nearly all high-performance computing platforms through many compiler implementations (GNU UPC,

IBM XL UPC, HP UPC, Berkeley UPC, Cray UPC, etc).

It realizes the PGAS memory model by providing a shared memory view across the system that can be accessed by any thread, yet each thread has an affinity to a part of the shared memory, the local shared space. In addition, each thread has a private space that is accessible only by that thread. Accessing the private space has low overhead and that allows for the best performance. Figure 1.3 provides an overview of the UPC memory model. The language also provides all the facilities needed for parallel programming. These include locks, memory consistency constructs, barriers, and collective operations, to name a few. Accesses to either the shared or the private memory space are syntactically identical and done through simple variable accesses or assignments.

The distribution of shared data across the different threads is controlled by a block size specified by the user for a given array. Elements are distributed in group of *block size* elements in a round robin fashion. Thus, the blocking factor gives the programmer a mechanism to control the locality of the data distribution in the shared space. For example, Figure 1.4 shows how the elements of the following array are distributed across four UPC threads with a block size of 4 elements. Each thread has its own contiguous address space starting at *base address*.

```
shared [4] int arrayA[32];
```

In order to address such arrays, a UPC shared pointer is used. Shared pointers are similar to C pointers, but are able to traverse shared arrays in the logical order that is manifested by the UPC memory model. They must provide a mapping from the logical array order to the actual physical location of the data in the system across the whole shared space. The shared pointer fields allow the compiler and the run-time system to perform the mapping between the shared space and the physical location.

A simple Unified Parallel C (UPC) vector summation example is shown in listing 1.1

Listing 1.1: Vector Summation in UPC

```
shared [4] double data[16];
// Local shared partial sums
shared [1] double partialSum[4];

// Initialize partial sums
partialSum[MYTHREAD] = 0;

// shared pointer privatization
double* privateSum = (double *) &partialSum[MYTHREAD];

// Compute local sums
upc_forall (int i=0; i<16; i++; &data[i])
    *privateSum += data[i];

upc_barrier;

// Reduce at thread 0
if (MYTHREAD == 0){
    double totalSum = 0.;
    for (int i=0; i<4; i++)
        totalSum += partialSum[i];
}
```

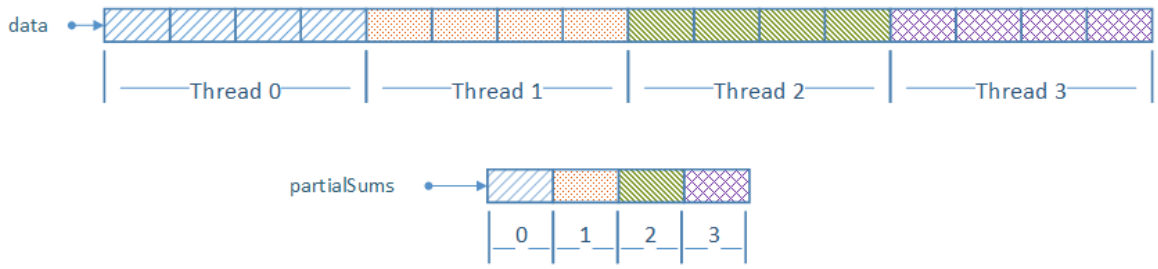


Figure 1.5: Vector Summation in UPC - Memory Layout

to demonstrate these features.

It follows a Single Program Multiple Data (SPMD) execution model in which a specified number of identically coded thread instances are executed in parallel. While the code base for each thread is identical, each thread may execute against a different portion of the data space.

The code sample in Listing 1.1 uses a vector summation example to illustrate sources of overheads in PGAS languages, and opportunities for optimization. This code can be simplified by using UPC collective operations, but in this form, it highlights the basics of the UPC language syntax. Lines 1 and 3 show an example of the declaration of shared memory arrays in UPC. The `shared` keyword is used to mark shared variables. The memory layout of shared arrays is determined by the *blocking factor* a value appearing in brackets in front of the array name, as illustrated in Figure 1.5. In the present example, the runtime system will distribute four sequential elements of the array `data` to each of the four UPC threads for a total of 16 elements. The `upc_forall` construct is used for parallel workload distribution. It resembles a regular C `for` statement with an additional *affinity* parameter that controls the way the runtime system will distribute a particular iteration of the `upc_forall` loop to a specific thread or set of threads. The key concept is that specific iterations of the `upc_forall` loop will execute against the portion(s) of the data set(s) with which they have affinity. Note that there is a dependency between how the shared memory variables are declared and how the affinity of the `upc_forall` is specified to

insure optimal performance.

The `upc_barrier` keyword is used to enforce inter-thread synchronization, by forcing all threads to block until all threads reach the barrier statement. Lines 17 to 22 are only performed by thread 0, to combine partial results obtained from all threads. Line 9 depicts *privatization*, a common hand-made code optimization which seeks to obtain a private pointer of a shared variable. Using the `privateSum` pointer, instead of `partialSum` shared pointer, will help to avoid more performance expensive shared data accesses. In line 13, each thread will do a private access to `privateSum` pointer and a shared access to `data` shared array. The privatization optimization can also be applied to `data` array, but for the sake of simplicity was not included in this example.

PGAS languages tend to incur a significant performance overhead when accessing shared data. From a PGAS compiler/runtime perspective, there are two different address spaces: the private address space and the shared address space. The private address space corresponds to the native view of memory provided by the system. The distributed shared address space view is not usually provided by the operating system or hardware, and a PGAS compiler/runtime system must emulate the shared view of the memory distributed across the several SMP nodes of a compute cluster.

When accessing private data, a compiler can generate a normal C language pointer, representing a virtual address against which to issue platform standard `load` and `store` instructions. Programs can manipulate the pointer with simple additions and subtractions. In contrast, a PGAS compiler must manipulate a more complex (and thus more performance costly) shared pointer to access even local data. As shared data might be stored locally or on a remote node relative to the locality of the calling thread, the PGAS generated code also needs to check if the shared data access requires remote communications. Once this is determined, the run-time system computes the virtual address required to perform the actual memory access.

In a previous study, it was shown that a simple local access (including the shared pointer structure manipulation) can generate up to 20 `load` and 6 `store` instructions [11]. In the absence of any hardware support, the compiler either needs to optimize all of the shared memory accesses, which is non-trivial in view of the potential data layout permutations, or the code developer needs to manually optimize the code (by casting local pointers to private pointers when it safe to do so, for example) which dramatically reduces the productivity and maintainability of PGAS language based codes.

A shared pointer is usually composed of three fields: **thread:** thread affinity of the pointed data, **virtual address:** address of the current element in the local space, **phase:** position inside the current block. This allows the programmer to traverse the array in its logical order. Current implementations of UPC usually use 64 bits to represent a shared pointer. Even if a compiler uses its own internal representation, the UPC specification [7] provides functions to access their implied values (`upc_threadof`, `upc_phaseof`, `upc_addrfieldof` and `upc_resetphase`). Figure 1.4 also presents a few examples of shared pointers (`ptrA`, `ptrB` and `ptrC`).

1.2.2 Chapel

The Chapel programming language is a more modern parallel programming language. The development started in 2003 by Cray Inc. under the DARPA High Productivity System program. The language was first released in December 2006. Chapel is now an open source project under the BSD license [12].

Chapel is a completely new language which is inspired by many other languages such as ZPL, High Performance Fortran, Java, and C++. It is designed from the ground up for parallel programming. It provides many ways to exploit parallelism.

Firstly, it implements the PGAS model and provides a global view with a strong sense of locality through the use of *locales*. Secondly, these *locales* can support multiple threads.

```
1  proc main() {
2      const ProblemSpace: domain(1) dmapped
3           Block(boundingBox={0..#N}) = {0..#N};
4      var A, B, C: [ProblemSpace] elemType;
5      const alpha = 0.1;
6
7      // A = B + alpha * C
8      forall i in ProblemSpace{
9          A[i] = B[i] + alpha * C[i];
10     }
11 }
```

Figure 1.6: Chapel Code Example: STREAM Scale

Future extensions are being developed to have hierarchical *locales*. Finally, Chapel can exploit dynamic parallelism through a powerful tasking layer. New tasks can be dynamically created locally and remotely. Tasks can also be moved across the system.

Chapel is also based on a *multiresolution design*. This allows the user to program at multiple levels. The higher levels provide great productivity but the user is able to go down, closer to the hardware, in order to have more control. For example, a user is able to implement their own distribution, called *domain map*, for a specific problem. In this work, this feature will be used to provide access to the hardware support. Users can also specify iterators that can exploit parallelism and locality [13].

Figure 1.6 presents an example of Chapel code. Line 2 defines a distributed domain for a 1D array. It then distributes the indices from 0 to N-1 using the `Block` distribution, which evenly distributes the indices from the `boundingBox` across all the *locales*. Arrays A, B, and C are then allocated using this specific distribution.

On Line 8, the `forall` statement will distribute the iteration of the loop according to the locality of the `ProblemSpace` distribution. The loop could also be expressed using the short-hand statement seen in the comment on line 7.

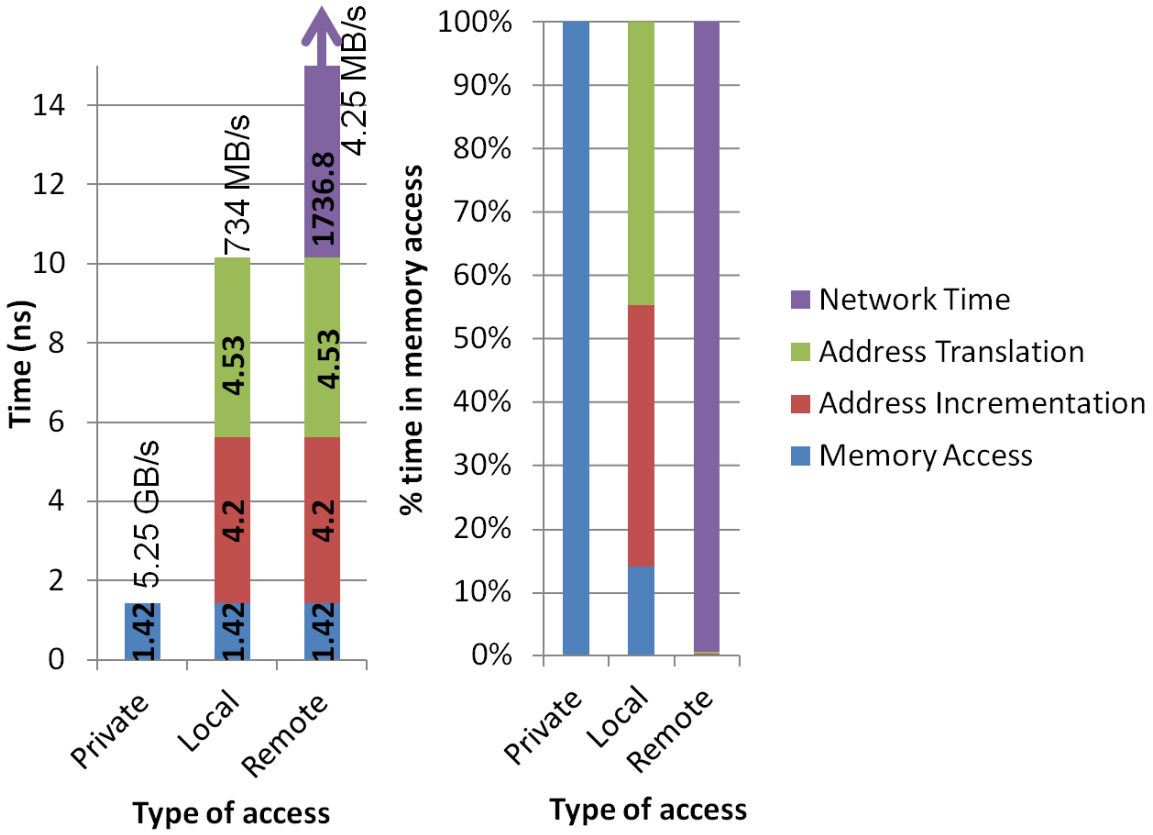


Figure 1.7: Memory Access Cost for a 64-bit Write (on a dual quad-core AMD cluster)

1.3 Shared Address Overheads

In this section, we evaluate the overheads of the PGAS model, both for the UPC and the Chapel languages.

1.3.1 UPC

In order to uncover the overheads of the PGAS memory model, using UPC as an example, a set of micro-benchmarks have been designed. These micro-benchmarks evaluate the cost of different aspects of the shared address manipulations. In order to get precise timing, the benchmark goes over a vector of 4 million elements and the average access times are recorded. The test is performed on a cluster of dual quad-core AMD Opteron 2354 chips running at 2.2 GHz interconnected by a 40 Gbit/s QDR InfiniBand network. 16 threads are

used across two nodes, but only one thread is writing at a time. The Berkeley UPC 2.18.0 compiler with the GASNet InfiniBand conduit was used.

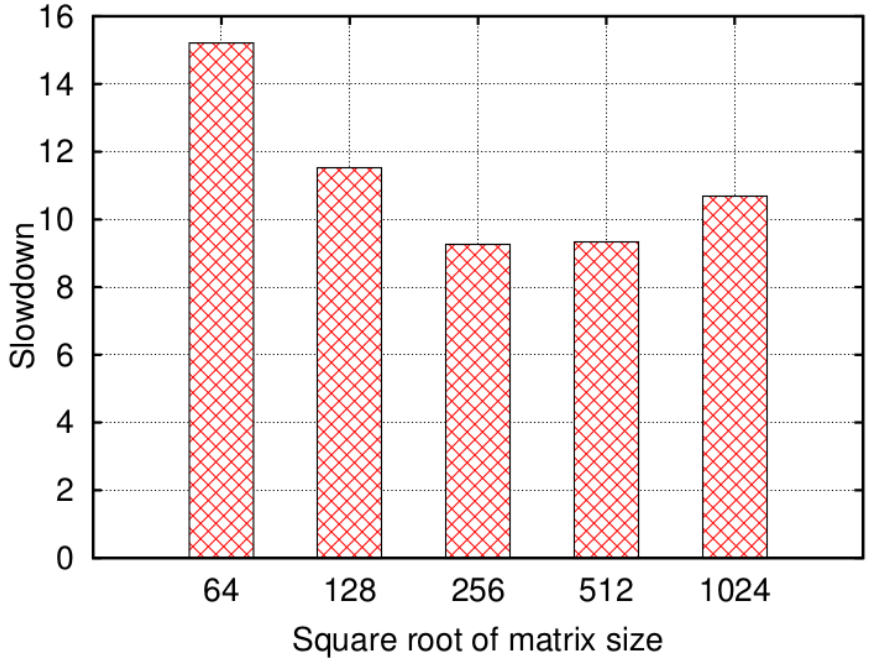
Figure 1.7 presents the results for a 64-bit write. We can see from the results that the address translation and address incrementation overheads of PGAS are significant. *Address Incrementation* represents the time spent updating the complex shared pointer to traverse the shared array. *Address Translation* is the time spent converting the pointer to a normal address before the access. This is especially prevalent in local accesses, where 86% of the time is spent in overhead.

In a different study [14], we show that the overheads also appear to be more prevalent on many-core platform as the Tilera Tile64 processor. On Figure 1.8, we show the overheads of UPC compared to C code on some simple matrix multiplication code on two different platforms: a dual-socket quad-core Nehalem Intel Xeon E5520 and a 64-core Tile64 chip. This is due to two reasons: 1. the simpler in-order cores are less able to mask the address translation instructions, 2. the many core create small blocks to be distributed increasing the address translation work and reducing opportunities for optimizations.

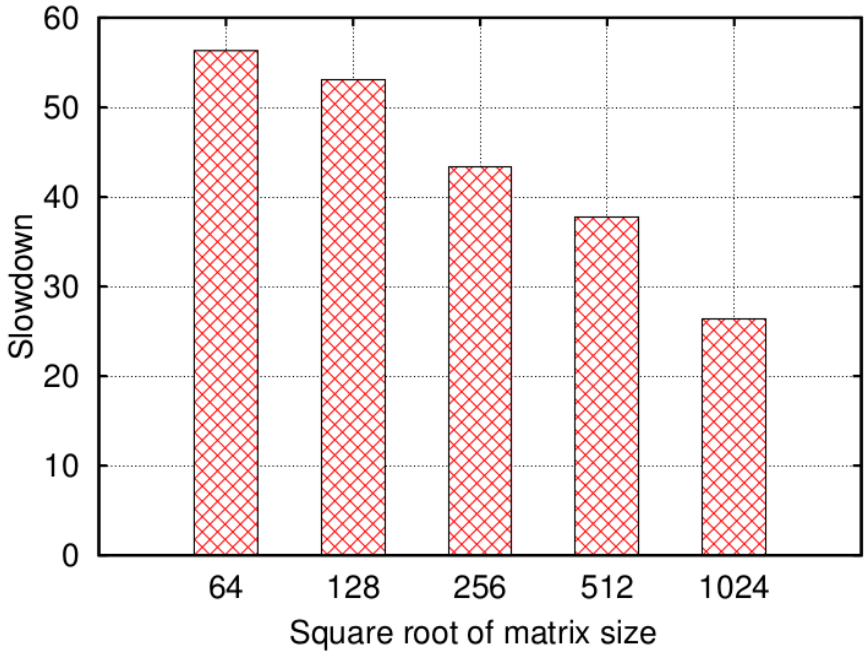
1.3.2 Chapel

Figure 1.9 shows the overhead to access the local shared space in Chapel. Local represents a local array, Local Shared the local portion of a shared array, and Local Shared Opt. represents accesses to a local portion of an array while manually specifying the access as `local`. The results were obtained on George, a Cray XK7 system at the George Washington University.

We can note that the compiler even with the `--fast` optimization flag is not able to fully optimize Local Shared accesses.



(a) On the Nehalem platform



(b) On the TILE64 platform

Figure 1.8: Multi-core vs. Many-core (Comparison of the sequential C version with a single UPC thread)

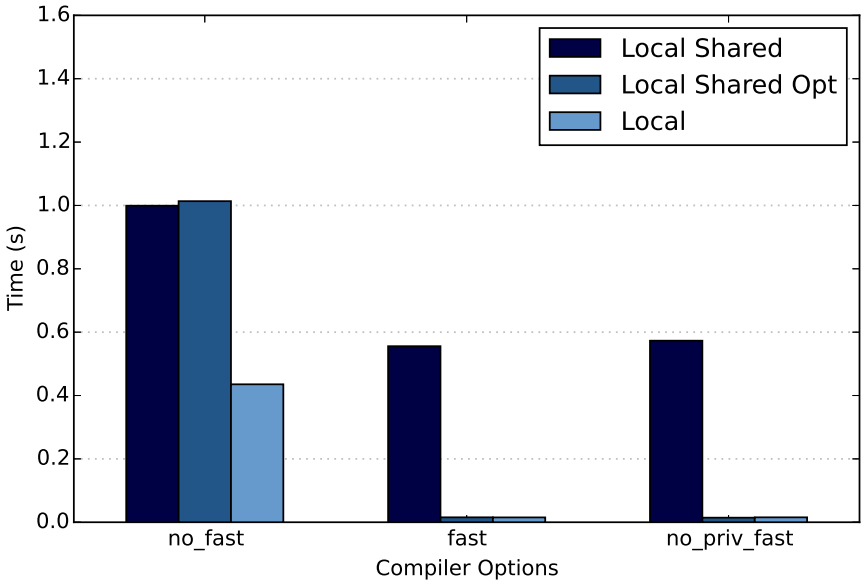


Figure 1.9: Local Access Overhead in Chapel

1.4 Problem Statement and Objectives

The goal of this research is to enhance productivity and performance of PGAS languages by providing an effective hardware support. Given a PGAS programming model implementation and a multi/many-core environment or a distributed system, it is necessary to:

- Establish a hardware/software co-design framework for exploring hardware support for PGAS compilers
- Introduce an efficient hardware support mechanism to simultaneously enable both the productivity and high-performance of PGAS
- Avail the hardware as an extension of the ISA and provide the compiler support for ease of use
- Empirically analyze and characterize the performance with and without the proposed

hardware support

- Prototype the hardware to evaluate the area cost and the feasibility of the proposed hardware support

1.5 Scope and Limitations

In order to limit the scope of the research, we target only the PGAS programming language's data mapping and access across the system. Other features of PGAS languages, such as collective operation libraries and Input/Output libraries, are not considered.

Additionally, in the prototype implementation, we chose to optimize for the common cases. Supporting all cases would be very costly in terms of implementation time and it would create other issues such as a high area usage on chip. For example, with the current prototype implementation, the number of threads supported is limited to a power of 2. It is important to note that when some specific parameters are not supported in hardware, it is always possible to fall back on a pure software implementation.

It is also not possible to support all the PGAS languages, however to show the ubiquity of our method, we first started with an UPC implementation and then show how the hardware could be extended with minor changes to support a more modern PGAS language, Chapel.

1.6 Organization of this Dissertation

The rest of this dissertation is organized as follows. First, related work is presented in Chapter 2: studies showing the productivity and the performance of PGAS languages are presented. Previous work highlighting and mitigating the main sources of overhead in PGAS languages is also presented.

Chapter 3 presents the proposed hardware design to resolve the issues highlighted in this chapter and in the related work. The details of the memory mapping and the shared

address space traversal are first presented both for UPC and Chapel, then an Instruction Set Extension is proposed to support hardware. Finally, the hardware design itself is shown.

In Chapter 4, we describe the full-system simulation used for evaluating the performance and the productivity of the proposed approach. Our Gem5 based full-system simulator stack is described and the specifics of the ISA extension for Alpha are discussed. Then our compiler support for the new hardware support is detailed both for the UPC and the Chapel language. The performance gains obtained with the hardware support and the extended compilers are then presented and analyzed.

In Chapter 5, a prototype implementation of the PGAS hardware support is demonstrated. For this a multi-core prototyping platform using FPGAs was first designed and then used to study the chip area overheads and verify that the proposed hardware can indeed be implemented. A sensitivity study with respect to the instruction timing is also presented, to inform the chip makers of the trade-off possible between the instruction timing and the final performance of the hardware support.

Finally, this dissertation is concluded in Chapter 6, along with a discussion of future research directions such as the new opportunities for network interface level optimizations enabled by this work.

Chapter 2 - Related Work

In this chapter, we will first review the related work concerning the PGAS productivity and performance. Then, we will review previous works that explore techniques to reduce the shared access overheads of the PGAS model.

2.1 PGAS Productivity or Performance

Previous studies have analyzed both the productivity advantage and the performance of PGAS languages under different levels of manual code optimizations and on a variety of systems. Many factors affect the performance, but as presented in the previous chapter the PGAS shared accesses present an important overhead. Especially for local accesses, for which, on a simple example, we have determined that up to 88% was spent in overhead (See Section 1.3 and Figure 1.7). Many of those studies have also shown that the overhead is principally due to the shared pointer arithmetic and the address translation. Those are the main performance impediments in PGAS codes.

2.1.1 Productivity

One of the main PGAS language, UPC, offers fundamental productivity advantages. In [15], Cantonnet et al. demonstrated a clear advantage in terms of productivity when using UPC compared to MPI. Their experiments based on the 5 NAS Parallel Benchmark kernels and 3 synthetic benchmarks have shown that UPC exhibits consistent improvements over MPI in terms of number of lines of code, number of characters, and conceptual effort to write the same program. Ebcioglu et al. [16] performed a 4.5 day study on 27 subjects, to compare the productivity of parallel programming languages. In this study, they compared the time to reach correct output for a biological string machine problem when using several parallel languages including: C+MPI, UPC and X10 [10]. A greater number of subjects would have made the conclusion of the study more definitive, as the few number

of users used can skew the results. But in general, the originally selected problem was not approachable using message passing and easily approachable with PGAS. For a simplified and more embarrassingly parallel version of the problem, however, users spent a bit more time with UPC compared to MPI. But the X10 group was much more productive took on average half of the time. The study also showed that the use of a PGAS language greatly reduces the time spent parallelizing the code.

Co-array Fortran [8] is a PGAS extension to Fortran which has been integrated into the Fortran 2008 standard. A 2011 evaluation of Co-array Fortran with the CGPOP miniapp[1] [17] shows a clear productivity advantage as the SLOC (Source Lines of Code) is reduced by up to 16%. However, for the 1D version, a buffering optimization was added which slightly increased the size of the code.

More recent PGAS languages like Chapel and X10 [18] continue to improve on the productivity by reducing their heritage from legacy languages like C and Fortran. UPC and Co-array Fortran are nearly supersets of C and Fortran which allow them to offer a great compatibility with, and portability from, legacy codes.

Chapel is not directly built on an existing sequential language as the authors believe that doing so would encourage users to write in a sequential style. [9]. The HPC Challenge benchmarks which comprise STREAM Triad, Random Access, FFT and HPL were implemented in Chapel [19, 20]. The Chapel implementations presented a dramatic reduction in the SLOC by a factor ranging between 3.7x to 50x. However, the comparison is not completely fair as the Chapel implementations are not fully complete, as they implement only one of multiple algorithm variations from the reference implementations. In another study [21], on a molecular dynamic code, the Chapel version saves 40% SLOC compared to the serial version in C.

X10 [10, 22] is based on Java and is extended with concepts to support PGAS-like

[1] CGPOP miniapp models the Conjugate Gradient routine of a Parallel Ocean Program simulation.

places and distributed multi-dimensional arrays. The authors implemented the Java Grande benchmark suite comprised of crypt, lufact, series, sor sparsematmult, moltdyn, montecarlo and raytracer [23]. Using X10, an increase in SLOC of 7% was measured, while using the Java threading and MPI libraries resulted in a 45% SLOC increase.

UPC++ [24] is a recent PGAS language that extend the C++ language, it provides an object oriented language, with a good set of feature for parallelism, including multi-dimensional shared arrays, asynchronous remote function calls and the possibility of specifying task graphs. It is implemented solely as a library for C++, which trade-off productivity due to a more complex syntax for portability and interoperability. The authors were able to quickly port pre-existing complex C++ application like Embree and LULESH.

2.1.2 Performance

Along with the productivity studies, many efforts were performed in the direction of evaluating the potential of achieving performance using PGAS.

In [25], the authors evaluated the performance of three different UPC compilers (Berkeley UPC, MuPC and Intrepid UPC, now GCC UPC) on 3 different platforms: a Linux x86 cluster, an AlphaServer SC and a Cray T3E. For all compilers and platforms, a significant overhead is noted for the access of the local space compared to the private memory accesses. The authors also note that ”Hand-tuning optimization techniques such as remote reference prefetching and pointer-to-shared privatization are vital for performance.”

Multiple studies [26, 27, 28] have shown that UPC can provide performance that is comparable to MPI once the code has been fully optimized.

UPC code can also achieve, in specific cases, better performance than MPI. Notably,for the NAS Parallel Benchmark kernel FT, it was possible to obtain better scaling by exploiting the one-sided nature of PGAS and a better overlap of the communication [29, 30].

Concerning Chapel, a preliminary study [21] of the Chapel overheads shows a potential

for many improvements. Although, it is important to note that the compiler is a work-in-progress prototype that is not fully optimized yet. For read and write references to local arrays, there is a 33% overhead. For parallel data types, iterating over pre-defined domains or ranges is 2.7x slower. Array references to distributed arrays are 12x-13x slower for 1D/2D arrays and 27-29x slower for 3D-arrays. Another study [31], shows that in some specific cases with Chapel, it possible to gain up to 35x speedup, by specifically marking some accesses as local.

2.2 Optimizations to Reduce Shared Access Overheads

As described in Section 1.3, the partitioned global view of the memory comes at a cost. This section will present the previous studies characterizing this cost and the research that aims to mitigate those overheads.

In [32], El-Ghazawi et al. clearly demonstrated the overhead of the PGAS shared memory model. They proposed a framework to assess the compilers and the runtime system's capabilities to optimize away such overheads.

The following section describes specific optimizations from the literature. They are classified on two different axes depending on whether they mainly target local or remote accesses and if they are software or hardware based (See Table 2.1 for a representation of the two axes; this table is subsequently completed in Table 2.2 with the notable related work).

2.2.1 Software Optimizations

2.2.1.1 Local Shared

One of the principal ways to mitigate the overheads due to PGAS is to allow the user to manually optimize their code. For example, in UPC [7], one can manually convert a pointer to local shared into a private pointer (which is a normal C language pointer). This allows

Table 2.1: Classification for the Related Work on Reducing the Shared Access Overheads

	Local Shared	Remote Shared
Software Optimizations		
Hardware Support		

for backward compatibility (with C libraries for example) and it allows the user to optimize critical parts of the code at a low level. The main disadvantage is that only the local portion of the memory is accessible which such pointer. Also, the productivity is clearly reduced as the manually optimized code becomes more complex and difficult to develop and maintain.

In order to solve those issues, different compiler optimizations have been researched including optimization techniques such as lookup tables. In [33], Cantonnet *et al.* took the approach of pre-computing tables with address-translations. The address translation code is reduced to the cost of a table look-up. Unfortunately, storing the complete table is often infeasible due to its memory requirements. This is solved by introducing reduced-tables which only includes the translation for the first element of each block. Additional arithmetic operations are required to translate other addresses, trading some computation time for memory. Although the overhead is clearly reduced, it is still significant compared to the use of private pointers. The work was extended to efficiently support multi-dimensional [14] and the overheads and gains were analyzed on both classical multi-core processors and many-core systems.

Alternative representations for shared pointers have also been implemented. In specific cases, a simpler representation can be adequate for shared addresses. For example, to address a shared array with a block-size of one or infinity, storing the phase information

inside the pointer is useless. The Berkeley UPC compiler uses phaseless pointers [34] in those cases. Phaseless pointers have reduced overhead as the phase field does not need to be computed; however it is still present and significant.

In [35], the authors highlight the inefficiencies of the UPC address translation. The major bottleneck for performance identified by the authors is the address translation and the associated complex arithmetic overheads involved. To remedy this problem, the authors introduce the block-major layout. In the block-major layout the arrays, on a physical shared memory system, are stored contiguously as in C regardless of their UPC block-size. This makes the cost associated with shared references to the array to be comparable to that in C. The limitation of that method is that it only targets shared memory systems and it is also not possible anymore to derive a private pointer from the optimized shared pointers (due to the incompatible arrays layout). The optimization is automatically disabled for arrays using private pointers.

More advanced compiler optimizations use linear memory access descriptors (LMAD) [36] to reduce the amount of shared address translations. While this provides good results for applications exhibiting regular memory access patterns, it still has a performance overhead due to the inspector/executor loop and scheduling which can, in some cases, reduce the performance by an order of magnitude compared to hand-optimized code.

2.2.1.2 Remote Shared

Many high-performance networks implement Remote Direct Memory Access (RDMA). RDMA allows performing zero-copy data transfers between compute nodes. The network adapter can directly read and write the memory without involving the CPU. This clearly reduces overheads for network transfers. However this is not specific to PGAS and it provides no help in performing the PGAS mapping.

In [37], the authors present an 8 byte packed representation which has a reduced over-

head compared to bit structures to manipulate shared addresses. Also, a Shared Variable Directory which caches the RDMA addresses is introduced. It reduces the latency of the address translation for remote accesses.

In [38], *symmetric heap mirroring* is presented to reduce the performance cost of threads accessing elements in a shared memory space. While reduced, the authors note that substantial overhead still remains.

2.2.2 Hardware Support

Multiple systems have implemented hardware support for shared memory across a system. For example, the T3D supercomputer used a 'Support Circuitry' chip located between the processor and the local memory [39]. This chip, on top of providing functionality like message passing and synchronization, allowed the processor to access any memory location across the machine. In [40], a network engine especially designed for PGAS languages is proposed. It allows network communication between nodes by mapping other node's memory space across the network and providing a relaxed memory consistency model best suited for PGAS. Results were only presented in terms of read/write throughput and transaction rates as no PGAS applications or benchmarks were tested. This approach is limited, as it focuses solely on the network interface for PGAS languages and it does not take into account the shared space addressing. Combining both an efficient addressing and an efficient network interface is needed to provide an effective support for PGAS.

More interestingly, the T3E [41, 42, 43] improved on that by providing the E-registers and a 'centrifuge' hardware allowing the network interface to perform some mapping for arrays. Using 4 special registers (index, mask, base address, stride and addend), it provides a good support for the data layout of PGAS languages at the level of the network interface. However, this approach has multiple drawbacks that need to be addressed. It uses a great number of registers, the registers are memory-mapped, and hence relatively slow to ac-

cess. The hardware is located outside of the processor chip, close to the network interface, making it useless for improving the performance of the very frequent local accesses.

2.3 Summary of Related Work

Table 2.2: Summary of Notable Related Work in Reducing the Overheads for Accessing the PGAS Space

	Local Shared	Remote Shared
Software Optimizations	Chen *et al.*[34] UPC Consortium [7] Cantonnet *et al.*[33] * Zhao *et al.*[35] Serres *et al.*[14] * Alvanosl *et al.*[36]	Chen *et al.*[34] Farreras *et al.* [37] Dalton *et al.* [38]
Hardware Support	*Work is lacking.*	Arpaci *et al.*[39] Scott *et al.*[41] Carlson *et al.*[43] Mueller *et al.*[42] Fröning *et al.*[40]

** Work done at GWU*

The related work presented here clearly shows that the PGAS programming model can offer either productivity or performance. Thanks to the locality awareness, modern architectures with deep hierarchies can be efficiently exploited. The global view and the use of one-sided communication constructs are characteristics that lead to a productive language.

It appears that the biggest impediment to the adoption of the PGAS programming model is the inefficiency of the data mapping across the system. This forces the user to choose between a simple and elegant but inefficient code or a manually optimized and difficult to maintain code.

Table 2.2 summarizes the survey of previous works targeting the overheads of accessing shared memory under the PGAS model. It appears that there is no previous work that

tried to optimize local accesses using hardware support. As local accesses are often the most common type of access, it is important that the overheads for those are reduced to a minimum. Therefore, one of the focuses of this dissertation is to reduce the overheads of local accesses with a novel hardware support.

Chapter 3 - PGAS Hardware Support Design

This chapter presents the general PGAS Hardware Support Design. Section 3.1 describes how the shared arrays are mapped in memory for both UPC and Chapel. Section 3.2 describes the general hardware support needed to efficiently support the UPC language. Section 3.3 describes how the hardware is extended for the Chapel language.

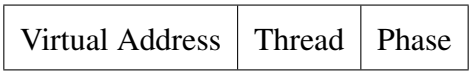
3.1 Memory Mapping and Shared Space Traversal

In this section, we explore the main source of overheads due to the PGAS model.

3.1.1 UPC

To implement the PGAS address space, the data of an application need to be distributed across the threads based on their locality. Therefore, it is necessary for PGAS compilers to have a mechanism to convert the logical view of the space presented to the user to the real, physical location of the data. This is often performed by using a complex, shared pointer which is able to correctly traverse the PGAS space.

With the Unified Parallel C language, in Section 1.2.1 we have seen that the shared pointers are usually composed of three fields: *phase*, *thread* and *virtual address*. A shared address is typically encoded using 64 bits.



In this implementation, arrays are distributed in a round robin fashion by blocks of *block-size* elements. The *phase* field contains the index of the element in the current block. *thread* contains the locality of the element pointed; i.e. which thread owns the current block. Finally, *virtual address* contains the address of the element relative to the base address of the current thread. Figure 3.1 shows the steps involved in incrementing a shared pointer in UPC. All of these operations can result in significant overhead since they need to be carried out for every element that is accessed within an array.

Require: blocksize, elemsize, increment, numthreads, shptr
Ensure: nshptr

```
1       phinc = shptr.phase + increment
2       thinc = phinc / blocksize
3       nshptr.phase = phinc % blocksize
4       blockinc = (shptr.thread + thinc) / numthreads;
5       nshptr.thread = (shptr.thread + thinc) % numthreads;
6       eaddrinc = (nshptr.phase - shptr.phase) + \
7                  blockinc * blocksize;
8       nshptr.va = shptr.va + eaddrinc * elemsize;
```

Figure 3.1: Shared Pointer Incrementing Algorithm

```
1  proc Block.targetLocsIdx(ind: rank*idxType) {
2    var result: rank*int;
3    for param i in 1..rank do
4      result(i) = max(0, min(
5                      (targetLocDom.dim(i).length-1):int,
6                      (((ind(i) - boundingBox.dim(i).low) *
7                      targetLocDom.dim(i).length:idxType) /
8                      boundingBox.dim(i).length):int)
9                    );
10
11   return if rank == 1 then result(1) else result;
12 }
13
14 // Access the elements at the index i
15 // (i can be a tupple for multi-dimensional arrays)
16 proc BlockArr.dsiAccess(i: rank*idxType) ref {
17   local {
18     if myLocArr != nil && myLocArr.locDom.member(i) then
19       return myLocArr.this(i);
20   }
21   if doRADOpt {
22     // Code for the RAD optimization removed...
23   }
24   return locArr(dom.dist.targetLocsIdx(i))(i);
25 }
```

Figure 3.2: Chapel Block Distribution: dsiAccess

3.1.2 Chapel

Chapel has a much more flexible distribution mechanism, as the users are able to write their own distributions. Figure 3.2 shows the code used to access an array element when using the most common (and efficient) Block distribution. Unfortunately, more flexibility can also mean more execution overhead.

The `dsiAccess` function gets as a parameter the indices of the element researched (as a tuple). If we are in a `local` block, either the user has specified or the compiler has inferred that the element accessed is local. The element is then retrieved from myLocArr. If we are not in a local block, the function targetLocsIdx is called in order to determine in which local array the element is located.

This can easily create significant overheads, especially when the backend compiler is not able to optimize all the code generated.

3.2 Hardware Design for UPC

In this section we discuss the general hardware support proposed for the UPC language.

3.2.1 UPC Memory Model Overheads

Other implementations and representations for the shared pointers are possible. In this implementation, the *virtual address* is kept updated to always point to the correct element. Another possible implementation could, for example, keep the *virtual address* pointing to the beginning of the current block, whereas the phase indicates the position within the block.

The final address of an element could be computed just before performing an access. This scheme would reduce the amount of work needed for incrementing a pointer and would increase the work required for the address translation when accessing an element. The scheme chosen for this work presents many advantages. First, it is applicable to all

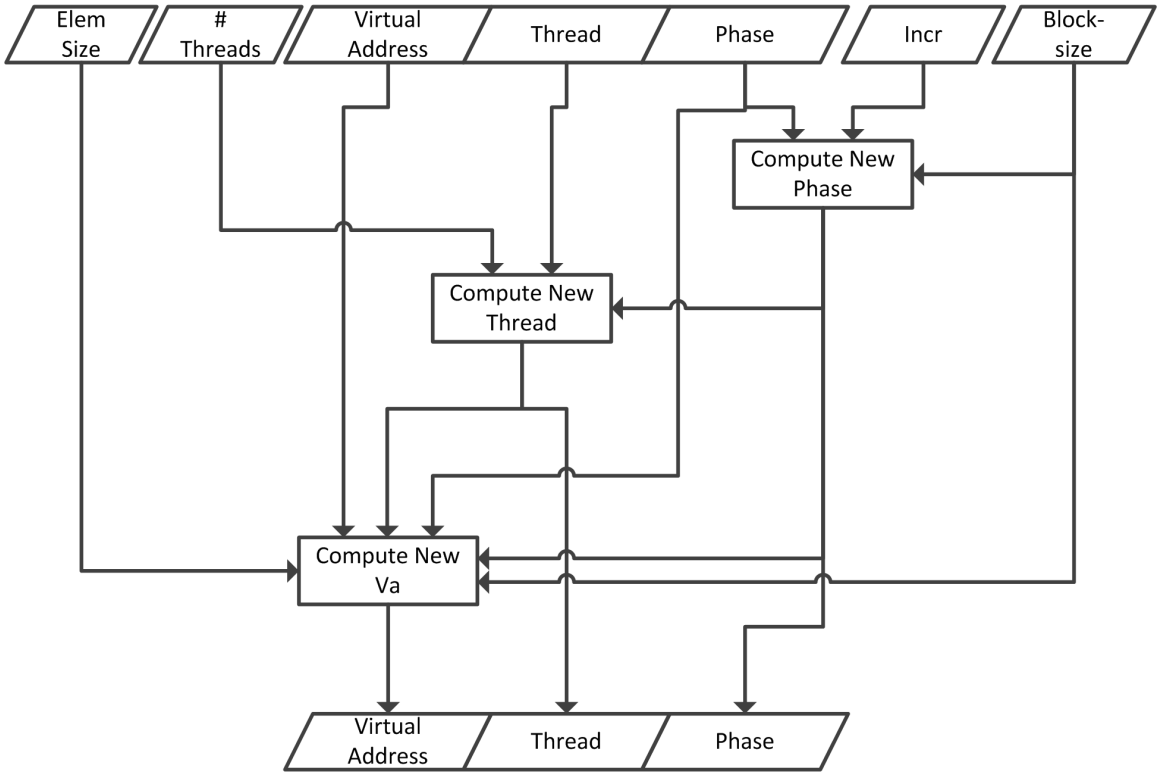


Figure 3.3: Shared Pointer Incrementing Flowchart - Conceptual View

current compiler implementations of UPC. Secondly, the low overhead allows performing shared memory accesses without increasing the time needed compared to a standard memory access. Lastly, the address incrementation can be easily pipelined and it is easier for the compiler to move around address incrementations and hide their latencies.

The shared pointer manipulations are currently performed in software. Incrementing a shared pointer consists of updating the three fields of the pointer to point to a different element in the shared array.

This can be done, for example, with the algorithm presented in Figure 3.1. The algorithm increments `shptr` to the new pointer `nshptr`. This is a particularly complex operation involving additions, subtractions, multiplications and divisions. Flowcharts describing the algorithms are also presented in Figure 3.3 and Figure 3.4.

When performed in software, the algorithm requires temporary registers, which can

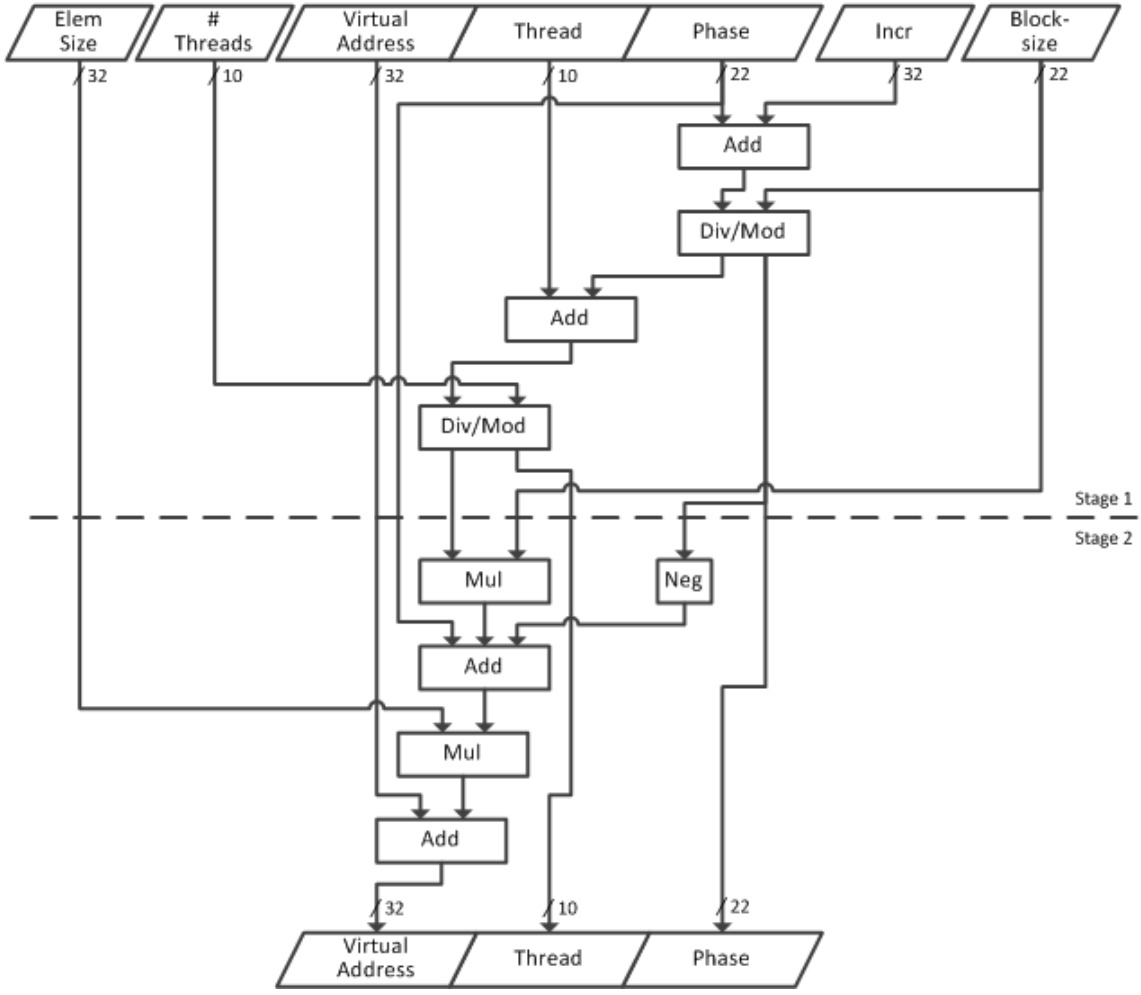


Figure 3.4: Shared Pointer Incrementing Flowchart - Hardware Implementation, multiplications, divisions and modulos are implemented as shifting and masking. The pipeline stage division represented is the one used in our Leon3 implementation.

increase register spill, adding extra pressure on the cache sub-system. It is to be noted that to increment a shared pointer, extra information not encoded in the shared pointer is needed. This includes the array block size, the element size (e.g. 4 bytes for int) and the number of running threads.

The shared pointer manipulations may be optimized by compilers in some specific cases. For example, for shared pointers with a block size of 1 or infinity, the phase field of the pointer is optimized. The shared pointer incrementation may also be optimized when

all the accesses are local. However, this is not always feasible due to the complexity of the code being compiled, due to dependencies on other results, or due to the use of functions from a different pre-compiled unit.

When an element is accessed, the shared pointer needs to be converted to a system virtual address[1] and then to a physical address so that the processor can perform the access. The system virtual address is obtained by adding the virtual address field of the shared pointer to the base address of the current thread. Even though this is not a compute-intensive operation, it is done for each shared accesses and therefore greatly increases the access time required for each element.

3.2.2 The UPC Memory Model Hardware Support

The hardware proposed here adds a specific support for shared pointers. This closes the performance gap between the shared space addressing and the private space addressing. At least two different types of operations need to be optimized in order to get an efficient addressing of the shared space:

- incrementing the shared pointers to correctly traverse arrays in their logical ordering,
- translating the shared addresses to the final physical addresses, allowing for a direct access to the shared arrays' elements.

Other operations, like testing for the locality of a shared pointer, can also benefit from the hardware support. This permits us to quickly call a communication sub-routine if the data is located outside the current node.

The algorithm to increment a shared pointer presented in Figure 3.1 can be efficiently pipelined and implemented in hardware. This is especially true when considering the most

[1] System virtual address refers to a virtual address of the process that can be exploited by the system MMU. In the rest of this work virtual address refers to the virtual address stored in the virtual address field of the shared pointer which is a distinct address relative to the UPC thread base address.

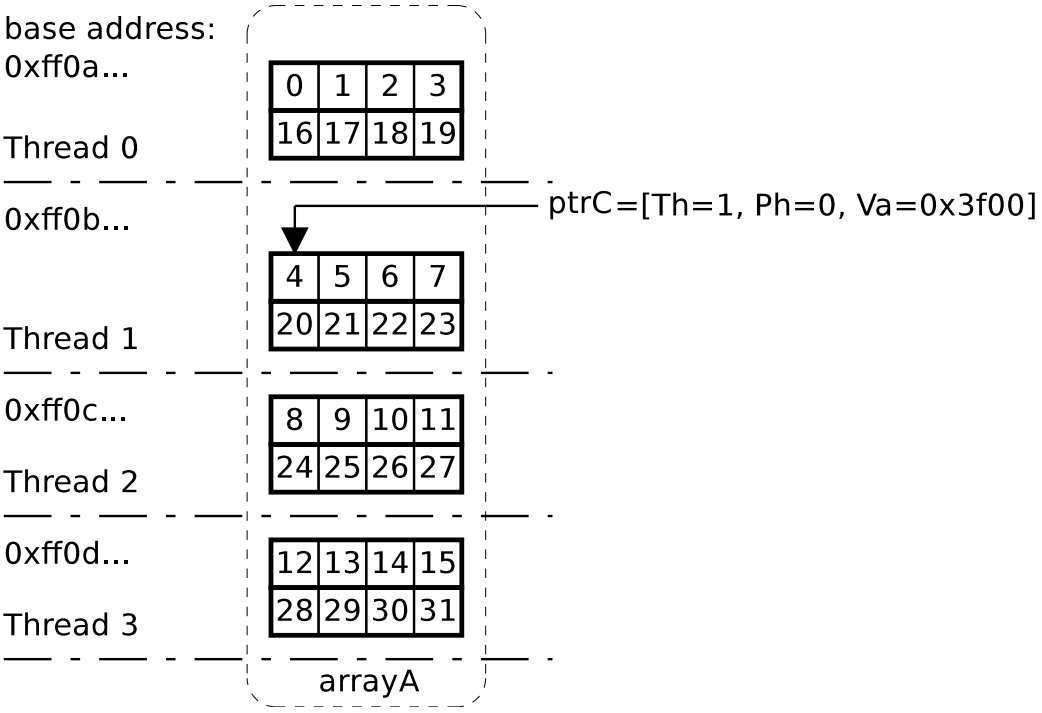


Figure 3.5: UPC Memory Layout for `shared [4] int arrayA[32];`

common case of having `numthreads`, `blocksize` and `elemsize` as powers of 2. This assumption is used in our implementations, allowing us to replace the divisions and multiplications with simple shifting and masking. At the same time, it does not restrict the user as the compiler can fall back on the software implementation for the cases not supported by the hardware.

When using a shared pointer to access data, the physical location should be computed. This is done by adding the *base address* of the thread specified by the pointer to the virtual address field contained in the pointer. The system virtual address will then be transformed to the final physical address using the conventional translation lookaside buffer (TLB) and the Memory Management Unit (MMU). Going through the virtual address is necessary to maintain the system security by providing memory isolation between processes.

At least two different implementations are possible for the address translation. The thread address spaces can start at regular intervals, allowing the *base address* to be simply

computed from the thread number (similarly to [40]), or a lookup table can be used to retrieve the *base address*.

The first method is more restrictive in the addresses possibly used but more scalable as it does not require storing a complete table for the base addresses. We used the second one in our prototype implementation for simplicity. For example, for `ptrC` of Figure 3.5, the system virtual address of the element would be computed by retrieving the base address of thread 1 and adding the virtual address from the shared pointer: $0xff0b00000000 + 0x3f00 = 0xff0b00003f00$.

3.2.3 Instruction Set Extension

We avail the new hardware to compilers via new instructions dedicated to manipulate pointers, and to load/store using a shared pointer as an address. New instructions are preferable to other mechanisms as they allow for pipelining with other instructions and provide very low latency. The low latency is essential to increase the performance of local shared accesses. This solves one of the main drawbacks described in previous related work [41, 42, 43].

Three new classes of instructions are essential for an efficient support of the PGAS address space (not counting the variants for the access size or mode): load/store from/to a shared address and shared address increment.

The 64-bit shared addresses can be stored in the normal processor registers, the other needed information like the block size or the element size can be directly encoded into the instructions. The increment value can be an immediate value or coming from a register; using a register allows to easily address arrays through indexing (e.g. `array[i]`).

In the following implementations, we also use a special register to store the number of threads for the currently running program.

3.3 Generalization of the Hardware Design for Chapel

As seen in Section 3.1.2, Chapel offers a great variety of distributions and the user is able to add his/her own distributions. This makes the creation of a general PGAS hardware support almost equivalent to developing hardware support for Chapel.

Therefore, we chose to extend the hardware support designed for UPC to Chapel. More complex distributions, can then be implemented on top of this (for example, an array starting at a non-zero index can be implemented by creating a domain map that would shift all the indices before using the hardware instructions; a similar software scheme is already used for such arrays).

Due to the dynamic nature of Chapel, it was important to have instructions able to target a variable block-size. In order to do this, the block-size has to be passed as a parameter to the address incrementing hardware.

Unfortunately, in most architectures, instructions receive two registers as input and one as output. As one of the input registers is used for the shared address, and the other as an index, it is not possible to simply pass the block-size by register. The solution to this issue is to pass the block-size encoded with the index.

As we limit ourselves to power-of-2 blocksize, we can encode all the required block-size using 5 bits (a 32-bit block-size) or 6 bits (64-bit block-size). This leaves at least, 58 bits for the index, which is more than enough to address even the biggest arrays (for an array of 64 bits integer, this would be enough to address an array bigger than 2 Exabyte (=2*10^{18} Bytes).

The instruction is more general and can also be exploited with UPC, without tangible overheads as long as the compiler is able to perform the following transformation intended to remove bit-wise or operation inside loops, `pgas_shared_inc_vbs` and `pgas_shared_store` are intrinsics that compile directly to assembly instructions:

```
// Non optimized code
uint64_t bs_enc=3<<59;
for(int i=start; i<end; i++){
    pgas_shared_inc_vbs(array_shaddr,
        i | bs_enc, elem_shaddr);
    pgas_shared_store(elem_shaddr, 0);
}

//Optimized code
uint64_t bs_enc=3<<59;
for(int i=start | bs_enc; i<end | bs_enc; i++){
    pgas_shared_inc_vbs(array_shaddr, i, elem_shaddr);
    pgas_shared_store(elem_shaddr, 0);
}
```

Another particularity of Chapel is to have *locales* supporting multiple threads. This is easily handled by programming the base address table and the number of threads special register accordingly.

The hardware support can also be extended with other special instructions to handle some other distributions, for example an instruction allowing to transform an index from a 2D blocking distribution to a 1D distribution would greatly improve the scope of the array directly supported.

Chapter 4 - Performance and Productivity Study

In this chapter, details of our empirical studies analyzing the performance and the productivity of our proposed approach are provided.

For a more accurate analysis, we utilized a full-system simulator (Gem5) that also enabled us to investigate various hardware design choices. First, our Gem5 based full system simulator stack is described and the details of our proposed ISA extensions for Alpha are discussed. Then our compiler solution for the new hardware support is detailed both for the UPC and the Chapel language. The performance gains obtained with the hardware support and the extended compilers are then presented and analyzed at the end.

4.1 Full-System Simulation

Gem5 is a full-system simulator for system architectural research. It allows us to realistically evaluate the performance characteristics of the hardware support.

Gem5 [46] has multiple advantages for this work: it supports a great variety of architectures, cache configurations, and CPU models allowing for trade-offs between speed of simulation and accuracy. Gem5 has also been shown to be accurate in modeling real systems [47].

In order to simulate our instruction set extensions, we selected the Alpha architecture as it supports full-system simulation with up to 64 cores using GNU/Linux. Gem5 simulates 64-bit Alpha 21264 processors, which includes the BWX, CIX, FIX and MVI extensions defined in the Alpha ISA. The architecture provides 32 integer registers (R0-R31) and 32 floating point registers (F0-F31).

We used the custom BigTsunami architecture variant in order to support up to 64 cores. The Linux kernel version 2.6.27.62 patched for BigTsunami is used, programs are compiled with the Berkeley UPC 2.14.2 compiler and GCC version 4.3.2.

The *Classic* Gem5 memory model is used; each core is configured with a 32 kB L1

Table 4.1: Instructions Added to the Alpha ISA

Shared Address Loads	
pgas_ldbu	Load Byte Unsigned (8 bits)
pgas_ldwu	Load Word Unsigned (16 bits)
pgas_ldl	Load Long Unsigned (32 bits)
pgas_ldq	Load Quad Unsigned (64 bits)
pgas_lds	Load S_float (32 bits, float)
pgas_ldt	Load T_float (64 bits, double)
Shared Address Stores	
pgas_stb	Store Byte Unsigned (8 bits)
pgas_stw	Store Word Unsigned (16 bits)
pgas_stl	Store Long Unsigned (32 bits)
pgas_stq	Store Quad Unsigned (64 bits)
pgas_sts	Store S_float (32 bits, float)
pgas_stt	Store T_float (64 bits, double)
Shared Address Incrementing	
pgas_inc_imm	Address increment, immediate
pgas_inc_reg	Address increment, register
pgas_inc_reg_vbs	Address increment, register with variable block-size
Initialization	
set_threads	Initialize the 'threads' register
set_base_address	Set the base address look-up table

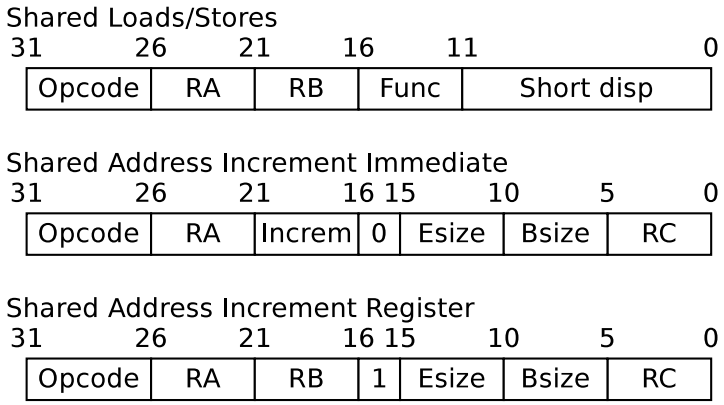


Figure 4.1: Format of the New Alpha Instructions

code and data cache, and a shared L2 cache of 4 MB. A frequency of 2 GHz is used for the system.

The Alpha instruction set is extended with our instructions shown in Table 4.1. Fig-

ure 4.1 presents the instruction formats used. The 64-bit integer registers (R0-R31) are also used to store the shared addresses.

For loads and stores, `RA` and `RB` represent the source and destination registers, respectively. `Opcode` is an unused opcode from the Alpha instruction set. `Func` defines which type of load or store is going to be performed. `Short disp` is a displacement added to the resulting virtual address, after the shared pointer has been translated to the system virtual address. `Short disp` is particularly useful in order to access the different members of a data structure.

For the shared address incrementing instructions, `RA` and `RC` represent the source and destination registers, respectively. `RB` is used in the register version of the instruction and specifies the increment register. Any increment value can be used when using a register. `Esize`, `Bsize` and `Increm` are 5-bit encoded immediate values for the element size, block size and the increment, respectively. They can represent any 32 bits value in which only one bit is set (1, 2, 4, 8, ...).

The `pgas_inc_reg_vbs` is the instruction added to support Chapel. The block-size is encoded is the higher bits of the incrementing register `RB`.

The new address increment instructions use the same timing and functional units as the Alpha integer instructions. The shared load/store instructions use the same timing as the usual load/store instructions. It will be shown, through our FPGA prototyping (Section 5.1), that it is indeed possible to implement those instructions with such a timing.

4.2 Compilation Support for Productivity

It is imperative that the new hardware support maintains the productivity of PGAS languages, therefore the compiler should be able to automatically perform the address mapping by issuing the new instructions. With this in mind, we now describe how this can be achieved with UPC and Chapel compilers as an example.

4.2.1 UPC

In order to maintain the productivity advantage of UPC, the new instructions need to be issued automatically, without any user intervention. Therefore, a prototype compiler for the hardware support was realized based on the Berkeley UPC source-to-source compiler. The source-to-source Berkeley UPC compiler takes a UPC program as input and generates C code which is then compiled with a C compiler.

In order to build our prototype compiler, we first had to modify the Berkeley compiler to remove the phase-less shared pointer optimization. This optimization uses a specific format which reduces the overheads for pointers having a phase of 1 or infinity. The specific format does not provide any advantage when hardware support is present and it generates incompatible shared pointers. Then, the compiler was modified to issue hardware instructions, when possible, for shared pointer operations like incrementations, loads and stores. When it is not possible to issue hardware instructions, the compiler falls back to perform the operation using software routines.

During the code generation, a few simple optimizations are also performed: immediates can be used to avoid using an extra register to store the incrementation value. Also, it is possible to perform non-power of 2 incrementations by combining multiple hardware instructions; When two bits are set in the increment value, the compiler performs the operation with two hardware incrementations. For example, to increment a shared pointer by 5, the address is first incremented by 4 and then the address is incremented again by 1.

The compiler also inserts the required code for the hardware initialization. The initialization happens just before the user's `main` function. Each UPC thread, is assigned to a specific core and the special registers of that core are set with the correct number of threads and the base address.

The rest of the toolchain, like the assembler, was also modified to recognize the new

instructions. The prototype compiler is able to generate machine code, taking advantage of the hardware support from unmodified UPC codes, without the need of the user intervention. This is essential for productivity.

4.2.2 Chapel

The prototype compiler is based on the Cray Chapel version 1.11.0. In order to be able support the Alpha processor architecture, the Cray Chapel compiler was cross-compiled. Chapel is compiled with the GASNet UDP conduit and with PSHM support. The support for GASNet with PSHM having been removed from the last releases, it was manually added again. Chapel is compiled with the fast segment support, LLVM support and uses FIFO as the threading mechanism. The optimizations are also enabled.

The Chapel runtime was slightly modified to fix simple issues with the Alpha architecture (the number of CPU was wrongly detected). Code to run multiple locales on a shared memory system was also added. It correctly informs the runtime of the number of processors available for a given locale and also places the newly created Chapel threads on the correct processors (in the function `thread_begin`).

Most of the code for the hardware support was then written in Chapel itself with both inline C and external C linkage.

In order to have a symmetric allocation, a new simple memory allocator `HardwareHeap` was written as a new Chapel class. When the class is instanced, it creates a new shared memory segment and registers it with the hardware (using the `set_base_address` instruction). The class provides a function allowing the allocation of memory across all the locales with a specified block-size:

```
1 // Allocate size bytes spread across the locales with
2 // bsize blocks. The address returned is aligned on
3 // 4KB page
```

```
4 proc allocate(size: uint, bsize: uint): uint{
5     ...
6     return offset; // Shared memory segment offset
7 }
```

In order to allow the usage of the hardware support within Chapel, we created a new distribution. This shows the flexibility of Chapel and how the *multi-resolution* paradigm can be exploited to provide new powerful distribution or features.

For this we created a new domain map distribution `HWBlock`, which will support arrays benefiting from the hardware support :

```
1  class HWBlock : BaseDist {
2    var hwHeap: HardwareHeap;
3
4    var boundingBox: domain(rank, idxType);
5    var targetLocDom: domain(rank);
6    var targetLocales: [targetLocDom] locale;
7    /* ... */
8
9    proc HWBlock(boundingBox: domain, targetLocales:
10                 [] locale=Locales, /* ... */
11                 param rank = boundingBox.rank,
12                 type idxType = boundingBox.idxType,
13                 hwHeap: HardwareHeap) {
```

We can see that the domain map has specific `hwHeap` arguments which specify the heap on which the new arrays will be allocated.

When the user creates an array using this domain, the class `HWBlockArr` will be used:

```
1  class HWBlockArr: BaseArr {
2    type eltType;
3    type idxType;
4    param rank: int;
5
6    /* Specific fields for the hardware support */
7    var hwHeap: HardwareHeap;
8    var sharedPtr: uint;
9    var elemsize: uint;
10   var blocksize_enc: uint;
11   var dimshift: uint * rank;
12
13   /* ... */
14 }
```

On top of the usual Chapel members like the element type of the array (`eltType`), this class has some extra members used to provide the hardware support. `sharedPtr` is the base address of the array. `blocksize_enc` is a precomputed encoded version of the block-size that will then be combined with the index with a bit-wise or before a shared address incrementation.

To provide access to the elements of an array Chapel usually uses the `dsiAccess` function. This is not adequate for our implementation as this function returns a reference. We extended the language to provide direct accessors[1] that allow directly reading or writing a certain element.

```
1  proc dsiMyGet(i: idxType): eltType {
```

[1] accessors are methods that let the user read and write the value of an instance variable. (also known as setters and getters).

```
2     select eltType{
3       when real(64) do
4         return shared_inc_hw_vbs_load_double(this.sharedPtr,
5                 i:uint | this.blocksize_enc);
6       when uint(64) do
7         return shared_inc_hw_vbs_load_uint64(this.sharedPtr,
8                 i:uint | this.blocksize_enc);
9       when int(64) do
10        return shared_inc_hw_vbs_load_int64(this.sharedPtr,
11                i:uint | this.blocksize_enc);
12      when complex do
13        return shared_inc_hw_vbs_load_complex(
14            this.sharedPtr, i:uint | this.blocksize_enc);
15    }
16  }
17  proc dsiMySet(i: idxType, val: real(64)){
18    shared_inc_hw_vbs_store_double(this.sharedPtr,
19                    i:uint | this.blocksize_enc, val);
20  }
21  proc dsiMySet(i: idxType, val: uint(64)){
22    shared_inc_hw_vbs_store_uint64(this.sharedPtr,
23                    i:uint | this.blocksize_enc, val);
24  }
25  proc dsiMySet(i: idxType, val: complex){
26    shared_inc_hw_vbs_store_complex(this.sharedPtr,
27                    i:uint | this.blocksize_enc, val);
```

```
28 }
```

The code also shows how the type system of Chapel is used to choose the good load and store operation. The type `eltType` is inferred at compile time, and the select operation is also done during compilation, as such there is no runtime overhead for the type selection mechanism.

Chapel supports inline C code when LLVM is enabled but unfortunately, it is not compatible with inline assembly, this is why the above function calls an external C function, similar to:

```
1  #define ALPHA_INST_SHARED_POINTER_INC_SAFE_PGAS_REG_VBS(
2    sharedregdest, sharedreg src, increment, esize) \
3      __asm__ __volatile__ ( \
4        "pgas_addrinc_reg_vbs %0,%1,%2," #esize ";" \
5        :"=r"(sharedregdest) \
6        :"r"(sharedregsrc), "r"(increment));
7
8  #define HW_PGAS_STT(shaddr, data) \
9    __asm__("pgas_stt %0,0(%1);" \
10     :: "f" (data), "r" (shaddr) :"memory");
11
12 void shared_inc_hw_vbs_store_double( \
13     uint64_t sh_ptr, uint64_t vbs_i, double val){
14   ALPHA_INST_SHARED_POINTER_INC_SAFE_PGAS_REG_VBS(
15     sh_ptr, sh_ptr, vbs_i, ELEM_SIZE_8);
16   HW_PGAS_STT(sh_ptr, val);
17 }
```

The usage of external C functions unfortunately removes the opportunity for some optimizations as the registers used to pass the parameters to the function are arbitrarily fixed in advance. This overhead was not present in the UPC compiler. This overhead can be eliminated in the future with the support of inline assembly in Chapel.

We show how one can easily take advantage of the hardware support, when applicable, by making use of the new domain map distribution :

```
1 var hwHeap = new HardwareHeap(128);
2
3 proc main() {
4     const ProblemSpace: domain(1) dmapped HWBlock(
5         boundingBox={0..#N}, hwHeap=hwHeap) = {0..#N};
6     var A, B, C: [ProblemSpace] elemType;
```

4.3 Experimental Toolset

For this research work, a great number of particularly long experiments (simulations) had to be executed. A series of programs for managing those experiments were created.

They allow to create a set of experiments to run (called instances). Those instances are created automatically using both the description of a specific benchmark (defined by a Python class) and the description of a specific platform.

A benchmark description contains the different parameters that can be varied for a specific benchmark (like the data size to use, the different levels of optimization that can be used, etc...). It also contains functions to verify the validity of those parameters, recipes to compile the benchmark, to generate the command line to run, and finally functions to parse and verify the results of a benchmark.

A platform description similarly defines a specific platform on which the experiment will be run. For example, the `gem5_alpha_upc_berkeley` contains parameters such

as the Gem5 CPU model to use, the sizes of the caches to be used (L1 and L2), compiler parameters (e.g. level of optimization), static or dynamic threading, and whether the hardware support is to be used or not. The Python class also contains all the code necessary to set the environment variables for a specific compiler, to compiler the simulator itself, to create the script that the simulator will use, etc. The class also parses the results obtained after a run, and checks for errors (e.g. M5 panic exception, invalid instructions, ...). Some statistics such as the number of PGAS load and store executed on each core, the memory usage are extracted.

A configuration file is then used to define the experiments: which benchmarks should be used in which platform, where to store the results, and which disk image the simulator will use.

A script is then used to run the instances on the clusters It will first load all the previous results obtained for the instances defined in the configuration. When not enough results are present and if the specific instance is not currently running, it will compile the benchmark (or get the executable from a cache) and create the file necessary for the experiments. Then jobs will be submitted to the clusters. Multiple job schedulers are supported like Slurm and Sun Grid Engine (SGE). Multiple tasks can also be grouped together in one job. This allows the use of multiple cores even though the simulator does not support multi-threading.

Finally, a series of scripts are used to extract the results like run-times. They can perform some statistics like extracting the median, or computing the standard deviation. They generate graphs (using GNUPlot) and tables.

In total, there are more than $10,000$ lines of Python3 code used to manage the experiments.

4.4 Performance Results

In this section, the performance results obtained with the Gem5 simulator and our prototype compilers are presented.

4.4.1 Gem5 Memory Models

Gem5 provides multiple CPU models that cover different architecture implementations and provide a trade-off between the speed and accuracy of a simulation.

In this work, we used 3 different CPU models:

Atomic: A simple single Instruction per Clock (IPC) model.

Timing: includes the simulation of the cache hierarchy and is a reasonable approximation of the performance of an in-order processor like the Leon3, Xeon Phi or the Tile64.

Detailed: Introduces out-of-order processor cores, in addition to simulation of the cache hierarchy.

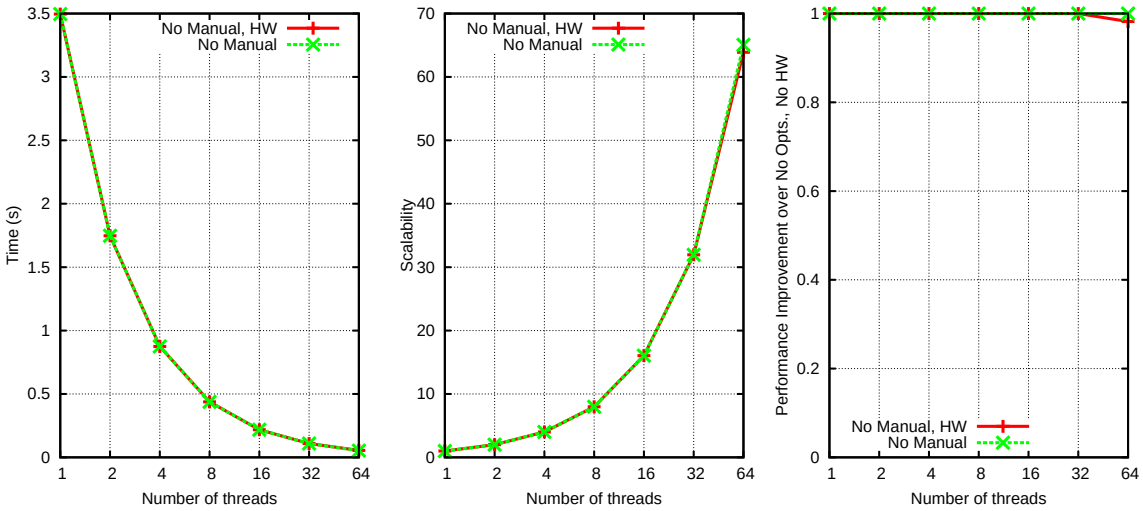
The number of cores presented for the detailed runs are limited as the simulator running time becomes very long; multiple days are needed for a detailed run.

4.4.2 UPC

In order to evaluate the performance improvement of the PGAS hardware support, five kernels from the NAS Parallel Benchmarks [48, 49, 50], and three synthetic benchmarks, were used:

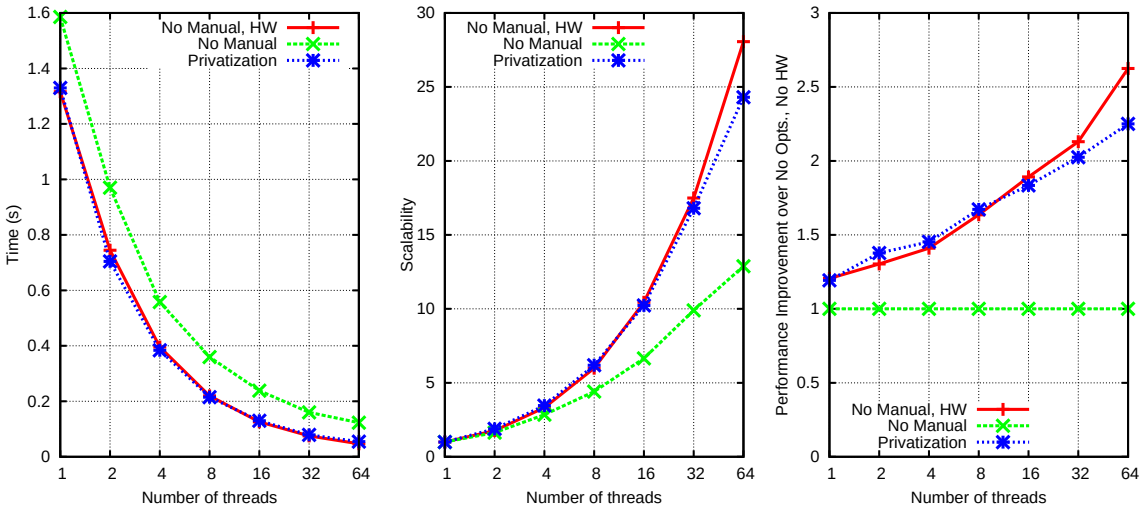
EP Embarrassingly Parallel: Generates pairs of Gaussian random variates.

CG Conjugate Gradient: Approximately computes the small eigenvalues of a symmetric positive matrix. Exhibits long-range communication.



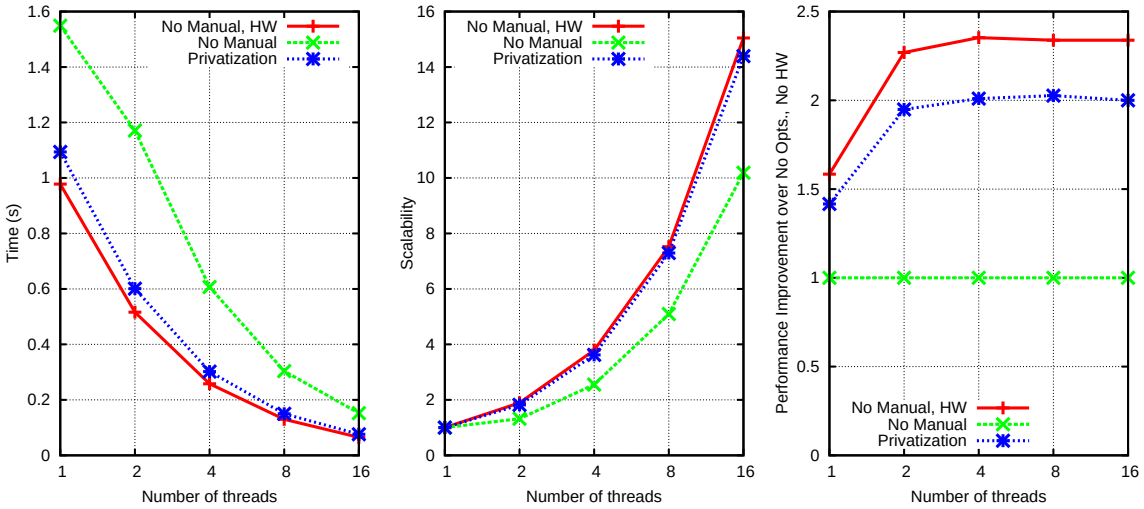
(a) Time **(b)** Scalability **(c)** Improvement

Figure 4.2: Gem5 Atomic Model: NAS Parallel Benchmark - EP class W



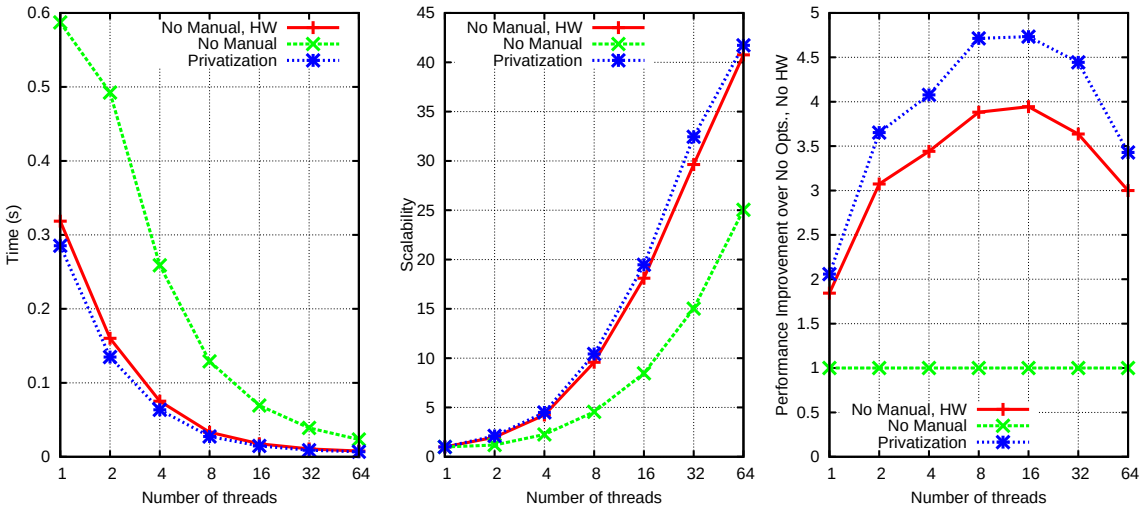
(a) Time **(b)** Scalability **(c)** Improvement

Figure 4.3: Gem5 Atomic Model: NAS Parallel Benchmark - CG class W



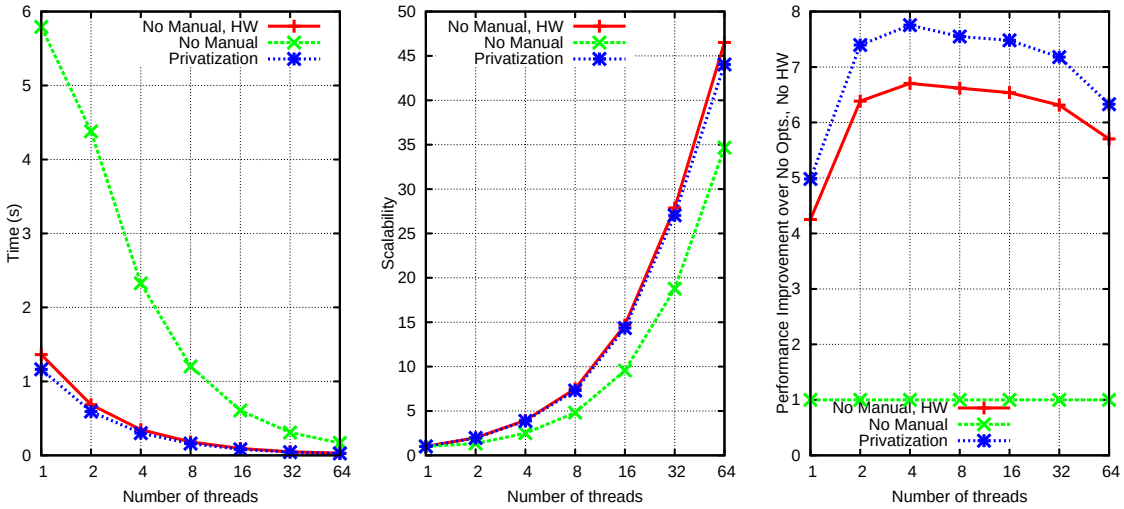
(a) Time **(b)** Scalability **(c)** Improvement

Figure 4.4: Gem5 Atomic Model: NAS Parallel Benchmark - FT class W



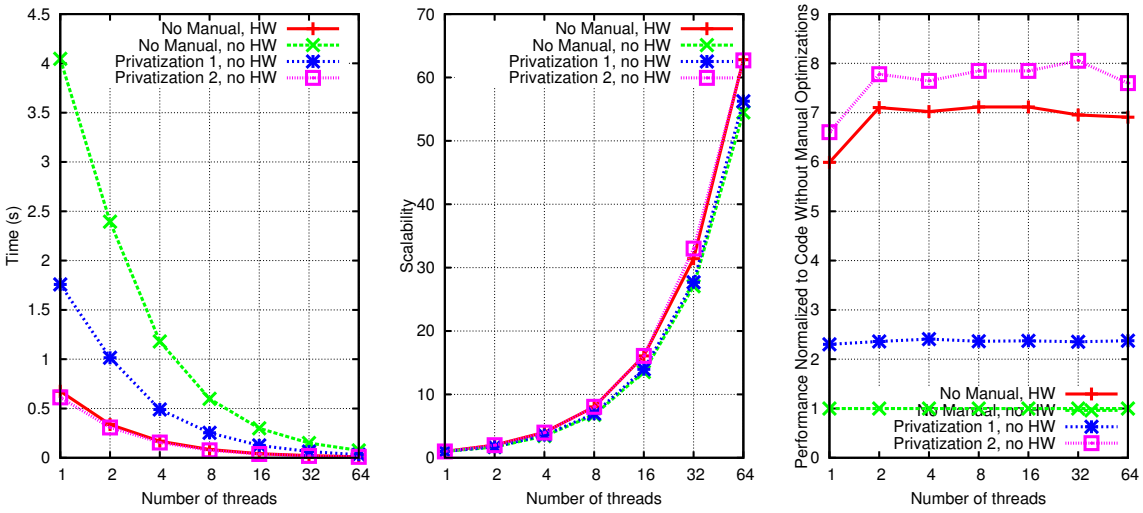
(a) Time **(b)** Scalability **(c)** Improvement

Figure 4.5: Gem5 Atomic Model: NAS Parallel Benchmark - IS class W



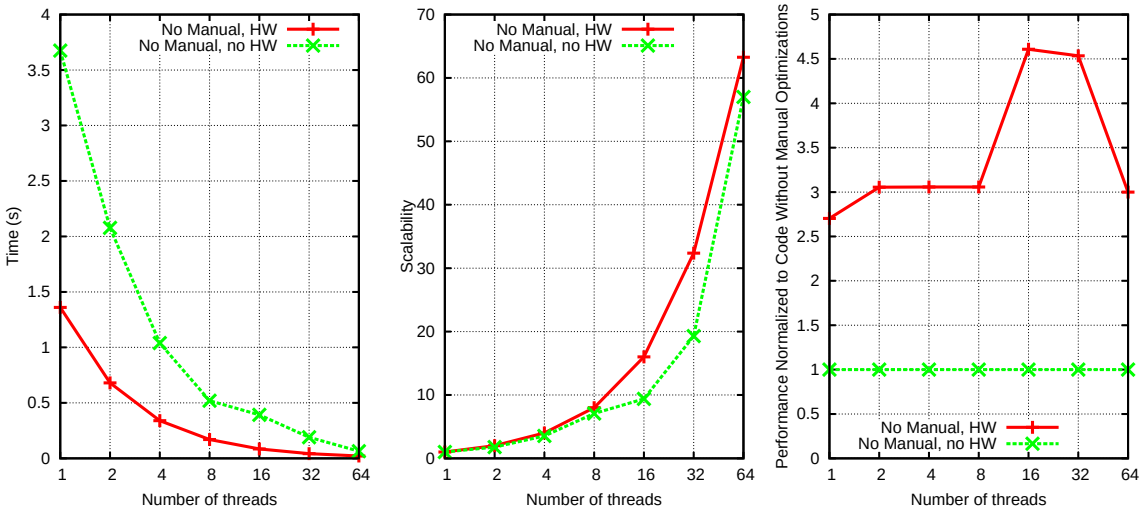
(a) Time **(b)** Scalability **(c)** Improvement

Figure 4.6: Gem5 Atomic Model: NAS Parallel Benchmark - MG class W



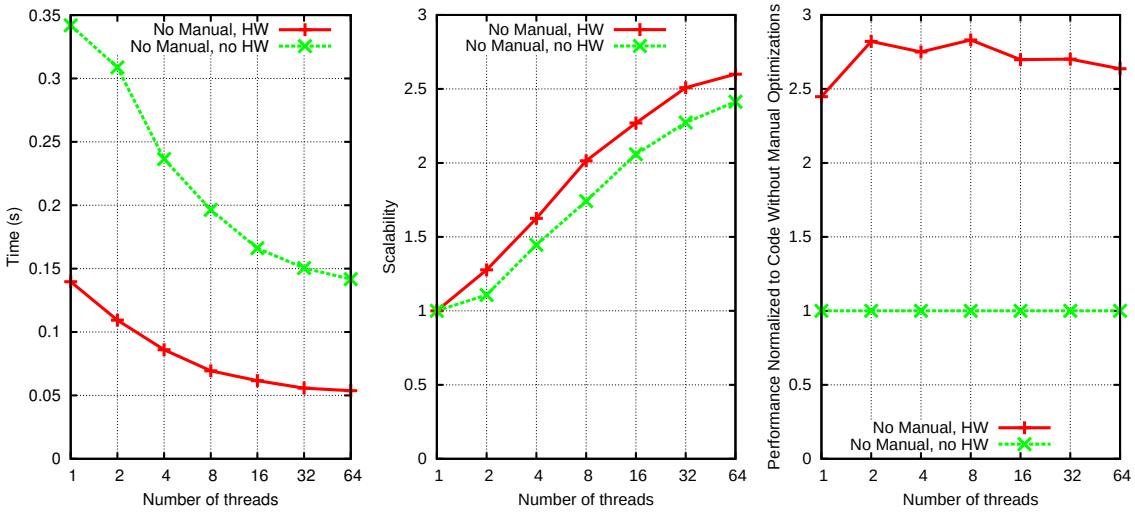
(a) Time **(b)** Scalability **(c)** Improvement

Figure 4.7: Gem5 Atomic Model: Matrix Multiplication 512x512



(a) Time **(b)** Scalability **(c)** Improvement

Figure 4.8: Gem5 Atomic Model: Sobel benchmark 4096x4096



(a) Time **(b)** Scalability **(c)** Improvement

Figure 4.9: Gem5 Atomic Model: Heat benchmark 64x64x64

FT Fast Fourier Transform: Solves a partial differential equation with a discrete 3D Fast Fourier Transform (FFT).

IS Integer Sort: Bucket sort of small integers.

MG Multi-Grid: 3D Poisson equation solving using a V-cycle multigrid method.

MM Matrix Multiplication 512x512 elements.

SOBEL Sobel edge detection benchmark on a 4096x4096 image.

HEAT 3D Heat Diffusion (64x64x64) synthetic benchmark. It uses a finite element methods with a 3D stencil to compute the heat diffusion in a small solid cube.

We used two versions of the benchmarks: a non-optimized version (referred as *No manual*)) and a manually privatized version (referred as *Privatized*), the most common hand-tuning operation done by PGAS programmers. The privatized version was manually optimized to replace UPC shared pointers with private pointers [51]. All version were compiled with all the compiler optimizations enabled. We used the hardware support with the *No manual* version and we refer to it as *No manual, HW*.

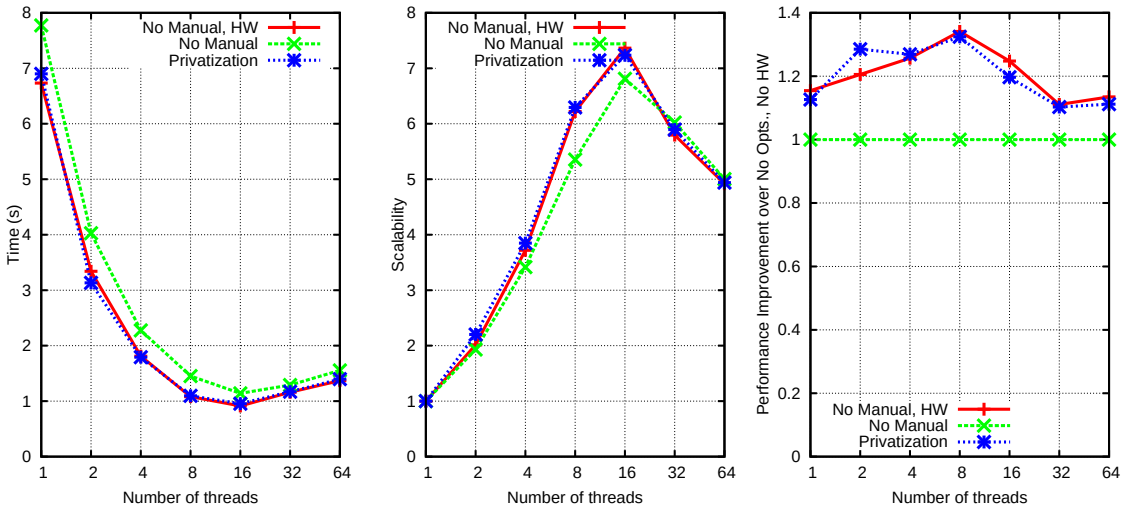
Given the length of time needed for multi-core simulations, only the relatively small W (workstation) class of the NAS Parallel Benchmarks was used.

4.4.2.1 Atomic Model

Figures 4.2-4.8 present the results when using the *atomic* model. The *atomic* model of Gem5 is relatively fast and we were able to run the benchmarks with up to 64 cores (the upper limit of the BigTsunami architecture). The execution time of the *atomic* model is directly proportional to the number of dynamic instructions.

Figure 4.2 shows that the hardware support does not provide any performance improvement for NAS EP. This makes intuitive sense as EP does not use shared pointers in its main

loops due to its embarrassingly parallel nature.

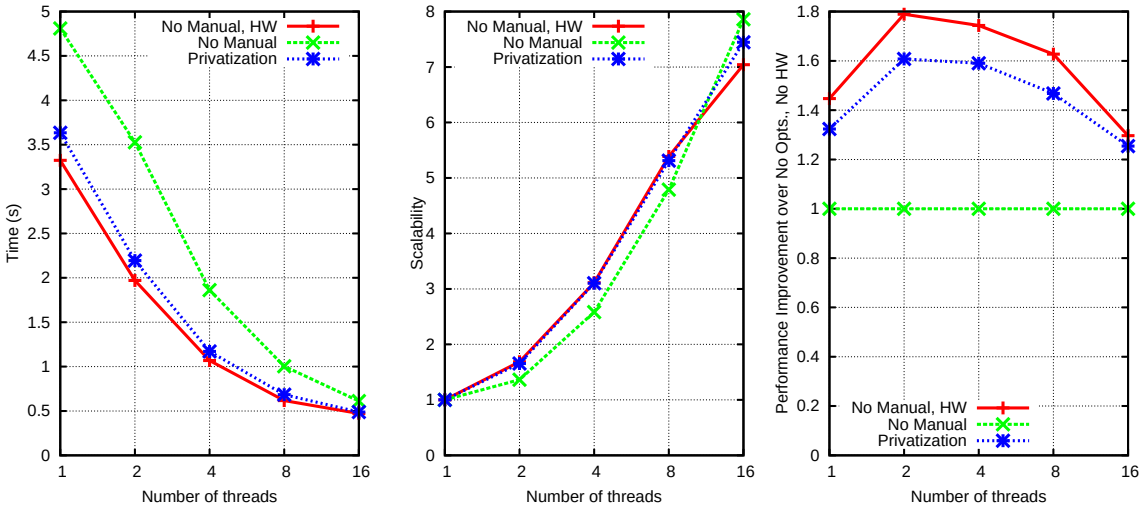


(a) Time **(b)** Scalability **(c)** Improvement

Figure 4.10: Gem5 Timing Model: NAS Parallel Benchmark - CG class W

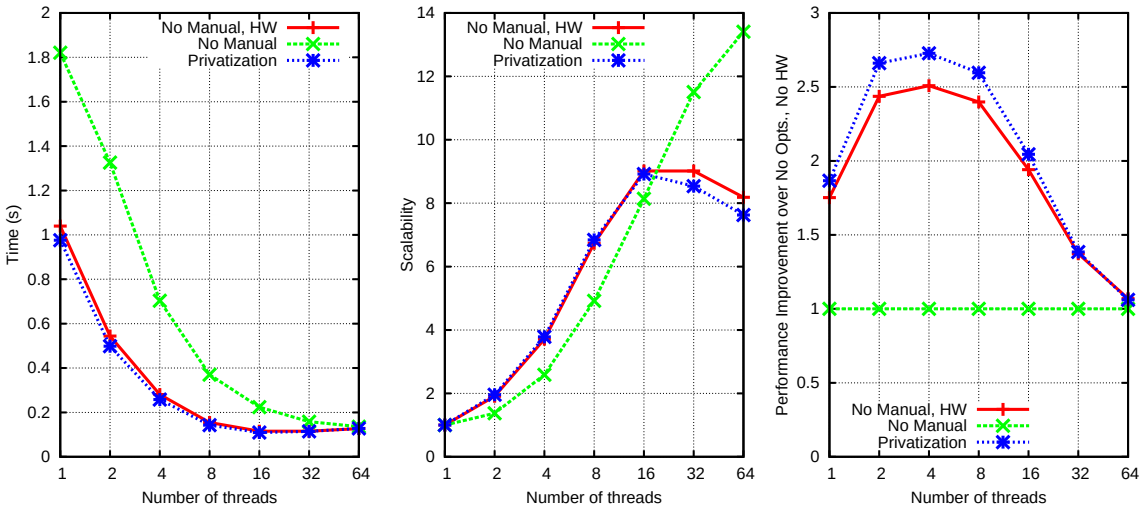
For the CG kernel, not all the shared address incrementation instances were compiled using the new hardware instructions. There were a total of 309 shared pointer incrementation instances in the code. 20 of them used a non-power of 2 element size (the arrays `w` and `w_tmp` have an element size of 56016). These instances had to be implemented with the normal (default) software incrementing algorithms. The code could have been changed to use an element size of 65536 (64K) for extra performance, but was left unmodified to preserve the integrity of the comparison. All the other shared pointer manipulations, including 236 loads/stores, were implemented using the hardware instructions. For the CG kernel (Figure 4.3) the *No Manual, HW* variant ran 2.6x faster than the *No Manual* variant. The *No Manual, HW* is 17% faster than the *Privatization* (manually optimized) variant.

All the shared pointer manipulations in the FT kernel were compiled with hardware instructions (79 address incrementations, and 47 load/store instructions). The FT kernel could only use up to 16 threads because of the geometric limitations of the data distribution of the W Class. The FT kernel performance (Figure 4.4) was improved 2.3x without manual



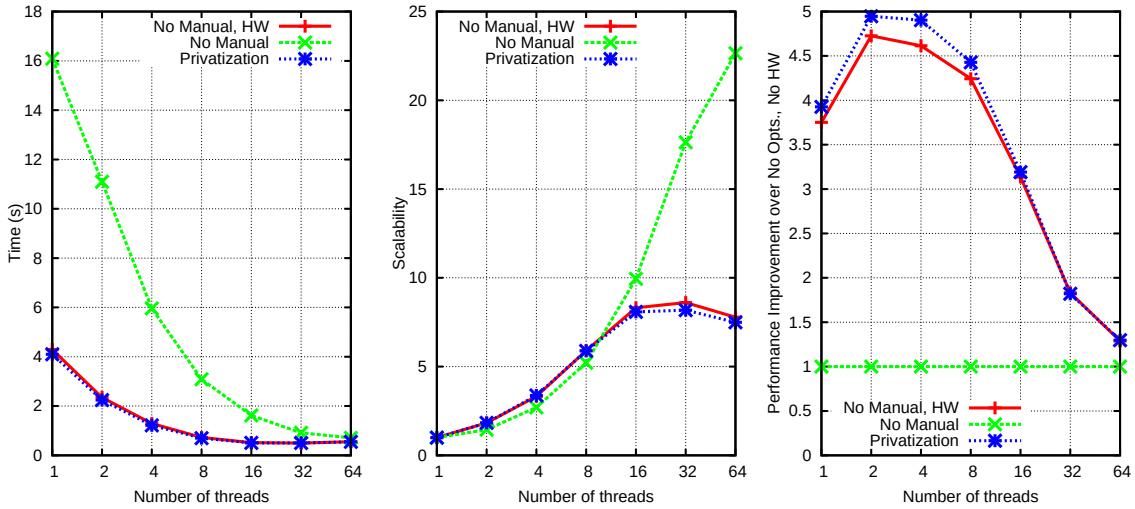
(a) Time **(b)** Scalability **(c)** Improvement

Figure 4.11: Gem5 Timing Model: NAS Parallel Benchmark - FT class W



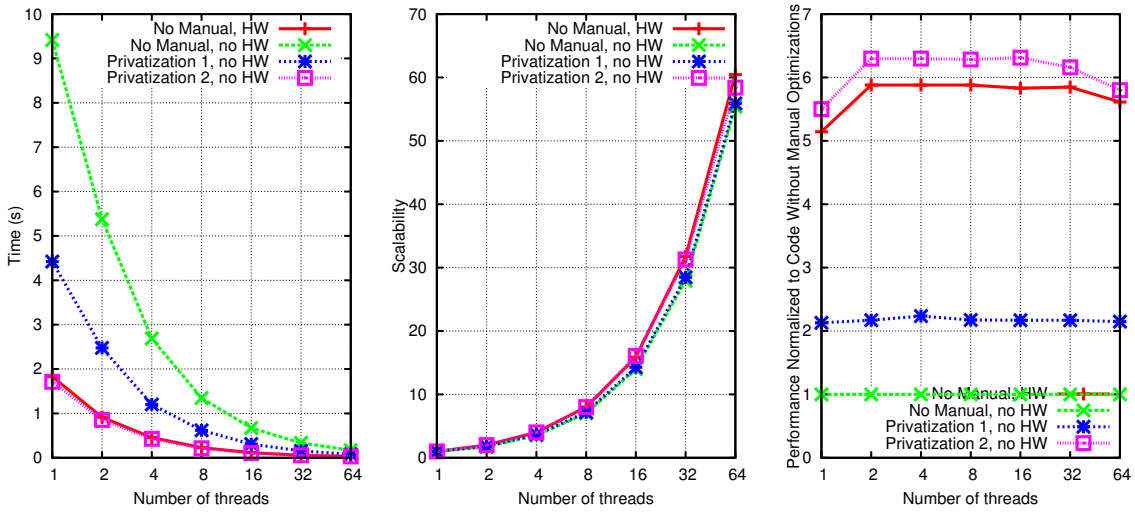
(a) Time **(b)** Scalability **(c)** Improvement

Figure 4.12: Gem5 Timing Model: NAS Parallel Benchmark - IS class W



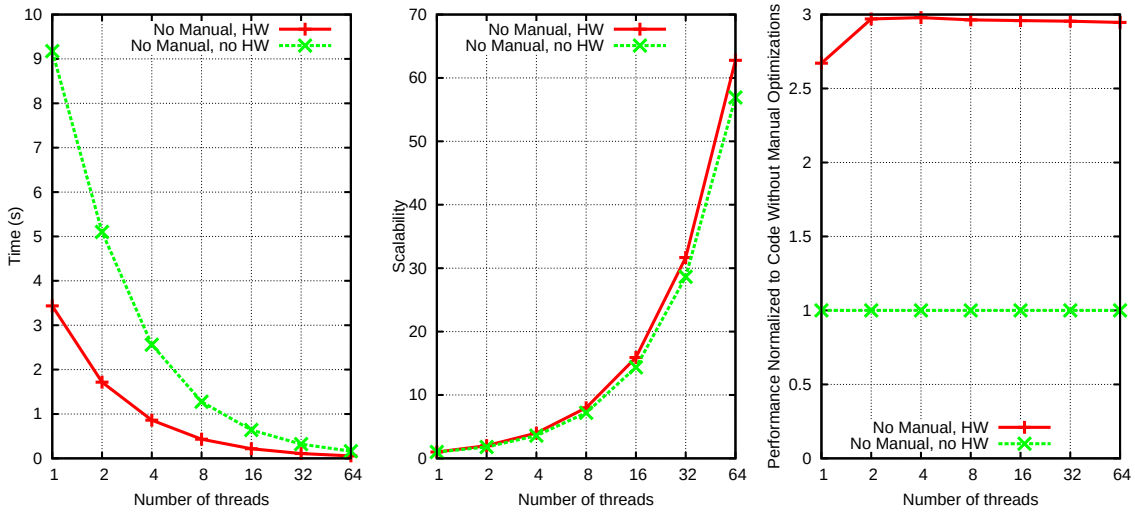
(a) Time **(b)** Scalability **(c)** Improvement

Figure 4.13: Gem5 Timing Model: NAS Parallel Benchmark - MG class W



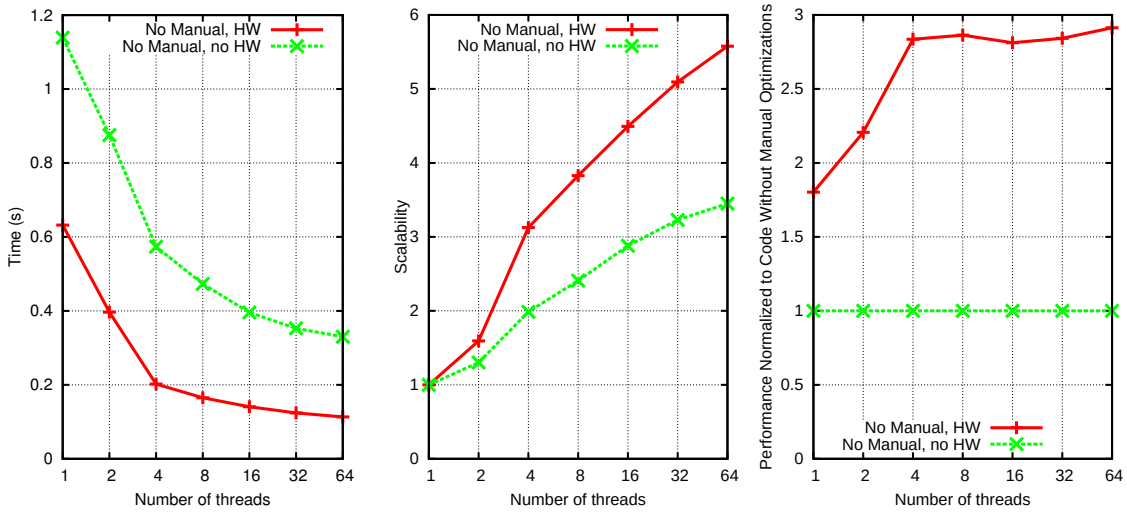
(a) Time **(b)** Scalability **(c)** Improvement

Figure 4.14: Gem5 Timing Model: Matrix Multiplication 512x512



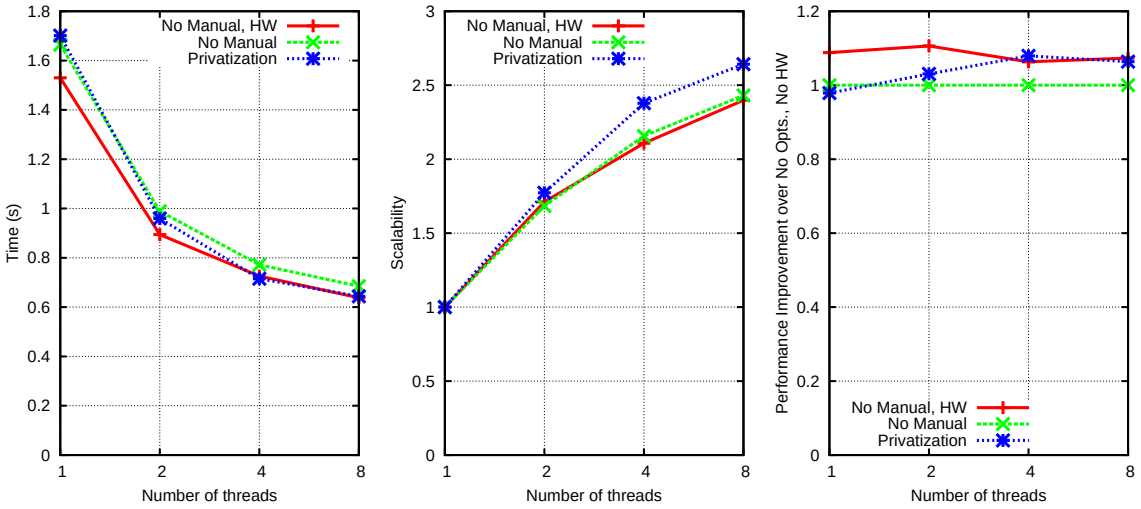
(a) Time **(b)** Scalability **(c)** Improvement

Figure 4.15: Gem5 Timing Model: Sobel benchmark 4096x4096



(a) Time **(b)** Scalability **(c)** Improvement

Figure 4.16: Gem5 Timing Model: Heat benchmark 64x64x64



(a) Time **(b)** Scalability **(c)** Improvement

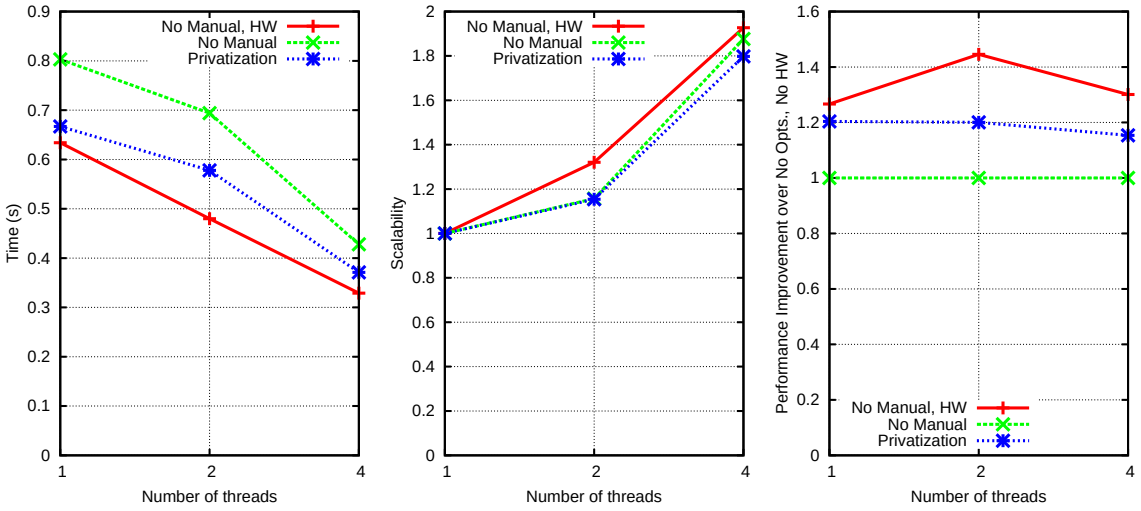
Figure 4.17: Gem5 Detailed Model: NAS Parallel Benchmark - CG class W

optimizations. Similar to CG, the *No Manual, HW* is 17% faster than the *Privatization* variant.

For IS (Figure 4.5), the *No Manual, HW* variant performed 3x better than the *No Manual* variant, but the *No Manual, HW* variant is still 13% slower than the *Privatization* variant. This might be due to missed optimization opportunities during the C compilation. The `asm` statements representing the PGAS store instructions are: 1) marked as *volatile* and, 2) flagged to indicate that they might have written a value anywhere in the memory space. These qualifiers prevent the GCC compiler from moving the store instructions and forces it to reload the data stored in registers.

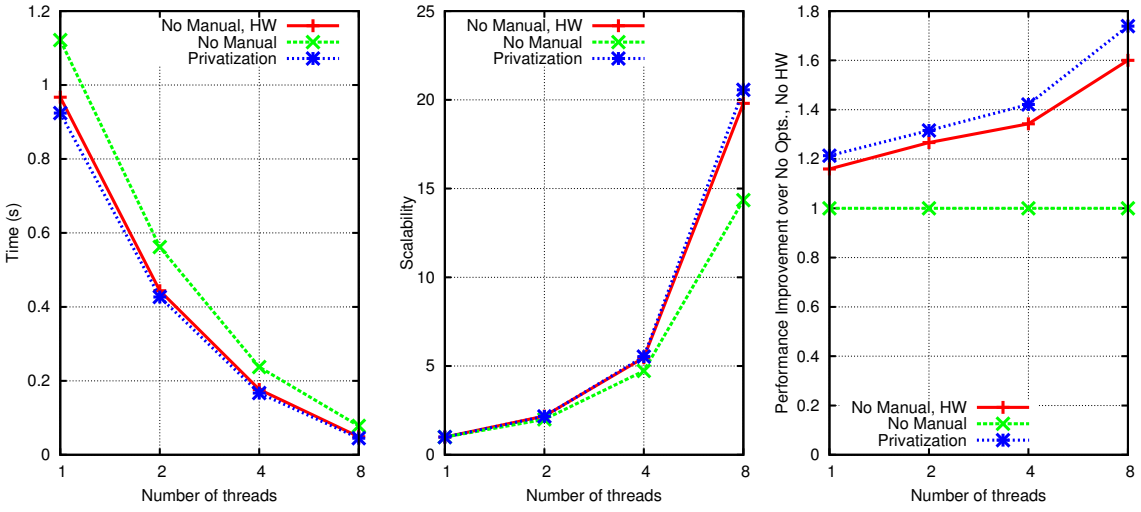
The MG kernel *No Manual* variant performance is improved by 5.5x (Figure 4.6), but it runs 10% slower than the *Privatization* variant.

For the Matrix Multiplication kernel (Figure 4.7), two different levels of manual optimizations were performed: *privatization 1* uses private pointers to access one of the matrices and, *privatization 2* uses a non-standard UPC extension to access all the matrices with private pointers, even when the matrix elements are not local to the calling thread. All



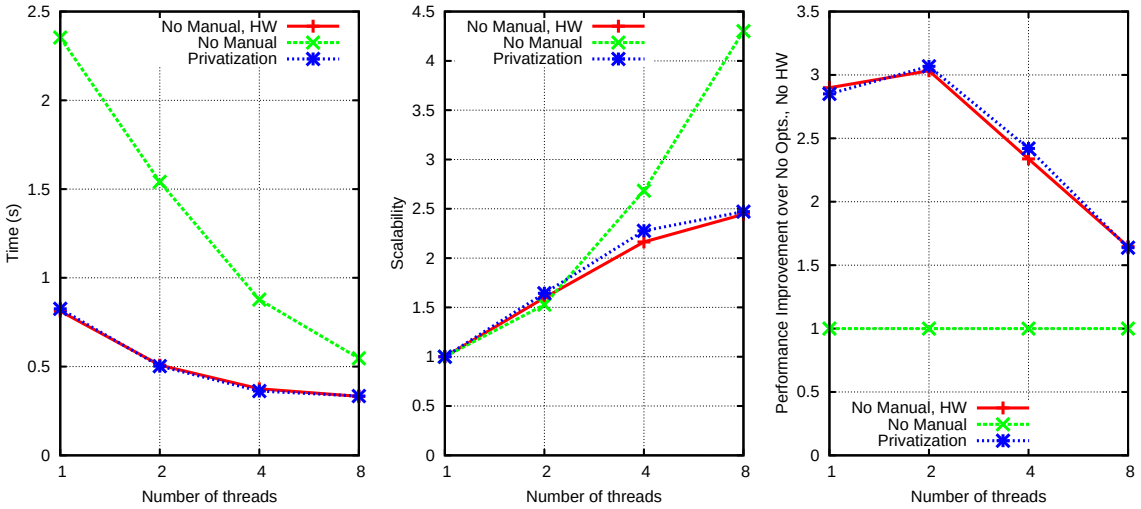
(a) Time **(b)** Scalability **(c)** Improvement

Figure 4.18: Gem5 Detailed Model: NAS Parallel Benchmark - FT class W



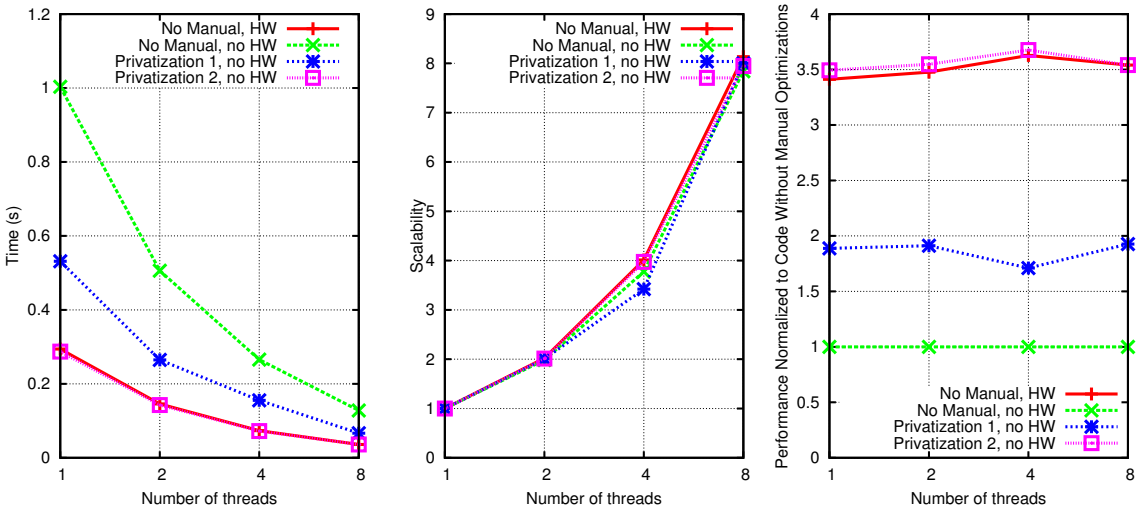
(a) Time **(b)** Scalability **(c)** Improvement

Figure 4.19: Gem5 Detailed Model: NAS Parallel Benchmark - IS class W



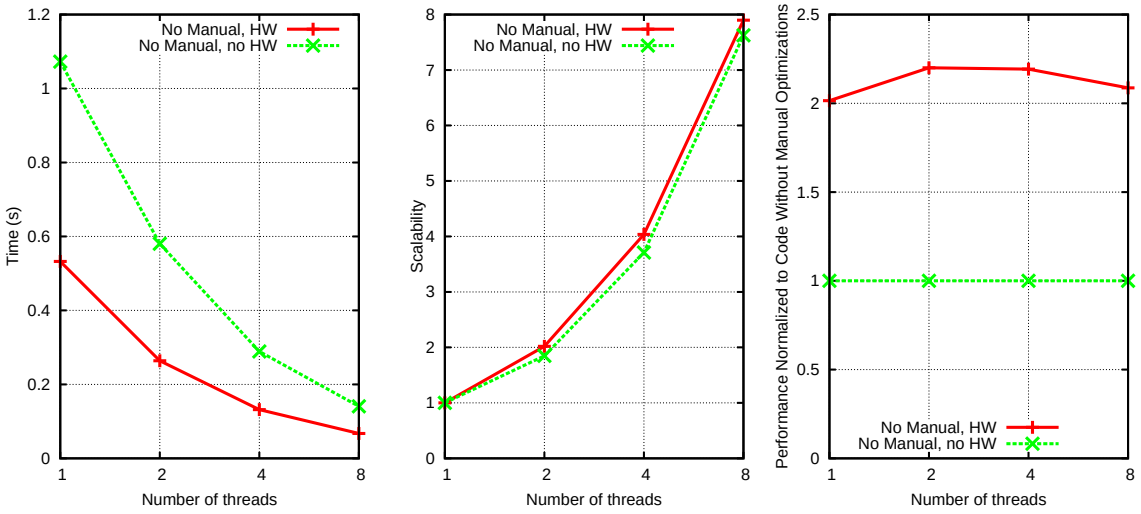
(a) Time **(b)** Scalability **(c)** Improvement

Figure 4.20: Gem5 Detailed Model: NAS Parallel Benchmark - MG class W



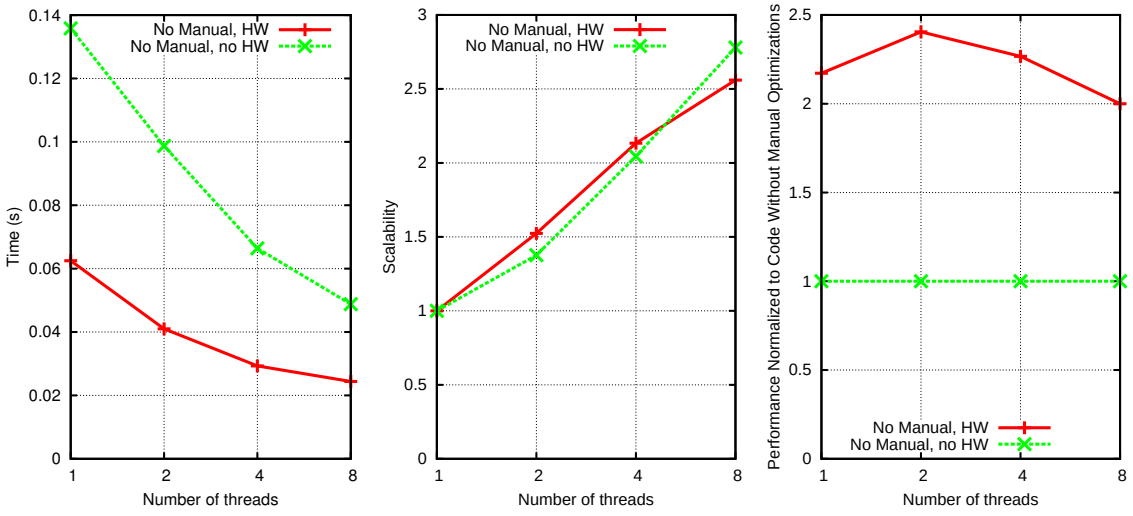
(a) Time **(b)** Scalability **(c)** Improvement

Figure 4.21: Gem5 Detailed Model: Matrix Multiplication 512x512



(a) Time **(b)** Scalability **(c)** Improvement

Figure 4.22: Gem5 Detailed Model: Sobel benchmark 4096x4096



(a) Time **(b)** Scalability **(c)** Improvement

Figure 4.23: Gem5 Detailed Model: Heat benchmark 64x64x64

the shared operations benefited from the hardware support. The *No Manual, HW* variant showed improvements up to 7x over the *No Manual* variant. The performance of the *No Manual, HW* variant approaches the fully optimized (privatization 2) level which provides an additional performance of 8%-14%.

For the Sobel and the Heat benchmarks (see Figure 4.8 and 4.9), the *Privatization* variant is not available, so only the 3x to 4.5x speedup obtained by the *No Manual, HW* variant is presented. All the shared pointers are compiled to use the hardware support.

4.4.2.2 Timing Model

Figures 4.10-4.16 present the results obtained using the Gem5 *timing* model. The *timing* model adds caches and memory timing to the simulation. The performance improvement of the hardware support is slightly reduced in proportion to the time spent accessing the memory. The single shared L2 cache also becomes a bottleneck for some benchmarks at 16 cores, as shown on the scalability graphs 4.10-4.16 (b). Nevertheless, the performance improvement of the hardware support is still substantial, showing up to a 4.7x improvement with the MG kernel. In this case, the *No Manual, HW* variant is comparable with the *Privatization* variant, which, in the worst case, is only 13% slower.

4.4.2.3 Detailed Model

Figures 4.17-4.23 present the results obtained using the *detailed* model. The *detailed* model supports a limited number of cores due to the relatively long simulation time (more than $35,000$ CPU hours were used to produce the results in this paper). Even though the performance gain of the hardware support becomes smaller, it is still substantial, ranging from 1.2x to 3.5x. The performance gain from the hardware support is reduced because the cores (with out-of-order capability) can improve performance through re-ordering the instructions and thus obviating part of the shared address manipulation activity.

4.4.3 Chapel

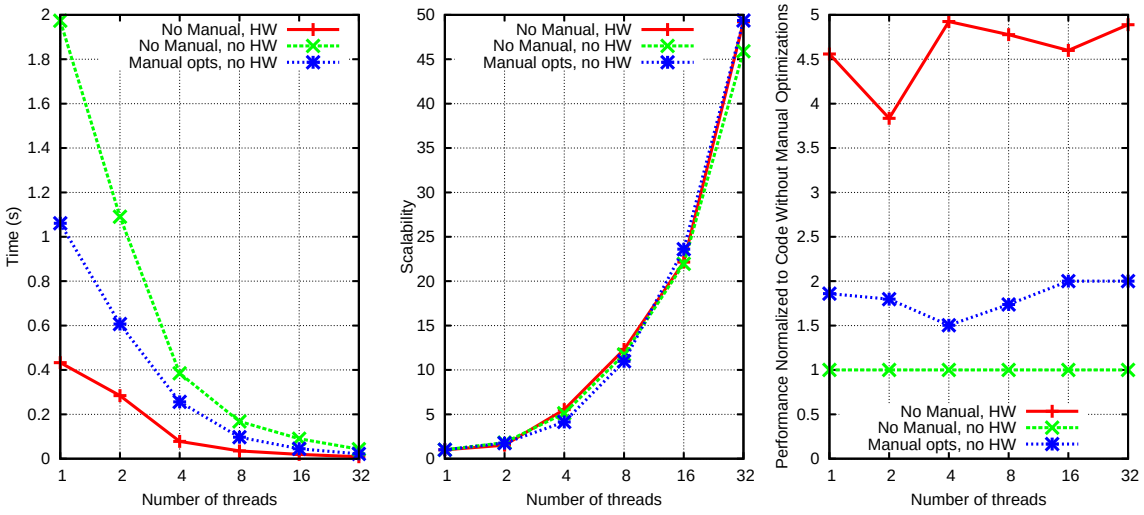
For Chapel four micro-benchmarks were used:

STREAM Scale Two vectors are added together, the second vector being scaled by a factor alpha. 4M elements double were used.

MM Matrix Multiplication 128x128 elements. (Similar to the UPC benchmark)

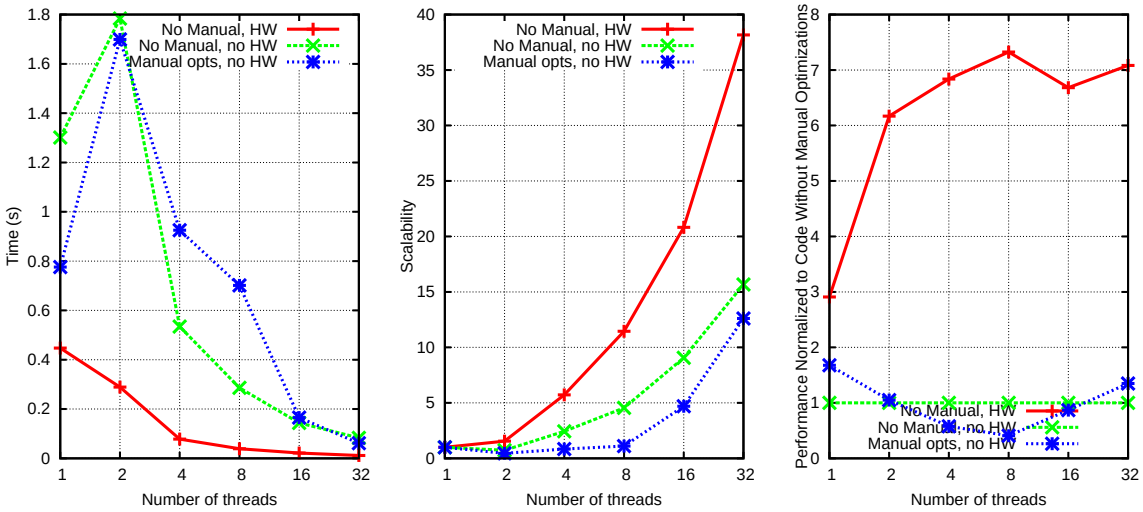
HEAT 3D Heat Diffusion (64x64x64) synthetic benchmark. (Similar to the UPC benchmark)

RA HPCC RandomAccess benchmark. The benchmark tries to update as many apparently random location per second as possible. It is designed to evaluate the system random memory access performance. The code is provided as an example with the Chapel compiler.



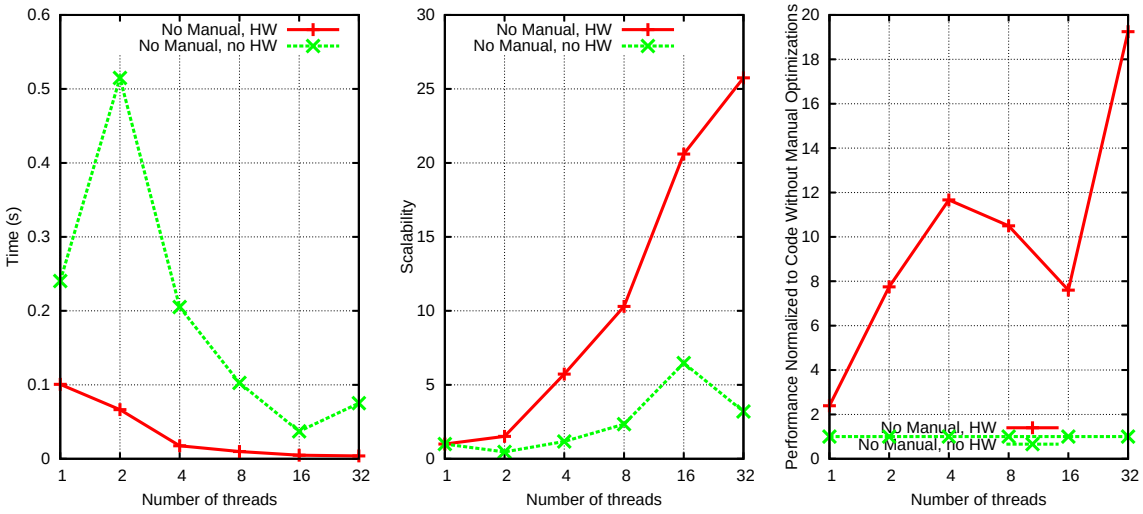
(a) Time **(b)** Scalability **(c)** Improvement

Figure 4.24: Gem5 Atomic Model: Chapel STREAM Scale - 4M



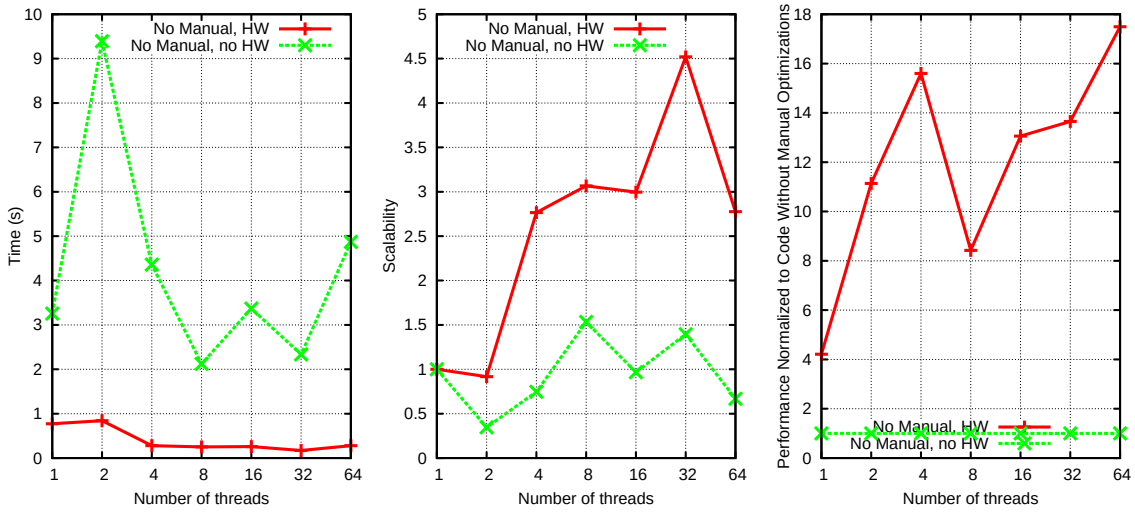
(a) Time **(b)** Scalability **(c)** Improvement

Figure 4.25: Gem5 Atomic Model: Chapel Matrix Multiplication 128x128



(a) Time **(b)** Scalability **(c)** Improvement

Figure 4.26: Gem5 Atomic Model: Chapel Random Access



(a) Time **(b)** Scalability **(c)** Improvement

Figure 4.27: Gem5 Atomic Model: Chapel 3D Heat Diffusion 64x64x64

It is important to be note that contrary to UPC, the Chapel compiler is still under heavily development and the performance and automatic compiler optimizations are expected to keep improving.

However, there is still also room for more improvement to the hardware support compilation. Also, we can see that the improvement obtained is already very substantial.

4.4.3.1 Atomic Model

Figures 4.24-4.27 present the results obtained with the atomic model. The 1 core results are using 1 locale and all the others are running with 2 locales (4 threads=2 locale with each running 2 threads).

For the STREAM scale benchmark (Figure 4.24, all the accesses are local, so the compiler would be expected to perform good optimizations. However, we can see that the hardware support provides a 2x speedup compared with the manually optimized version (in which all the accesses are specified as `local`).

For the matrix multiplication benchmark (Figure 4.25), the speedups are quite important

(from 4x to 8x with multiple locales).

The Random Access benchmark (Figure 4.26 is particularly interesting as compiler optimizations are very ineffective due to the random access pattern. The hardware support is able to provide a substantial performance gain.

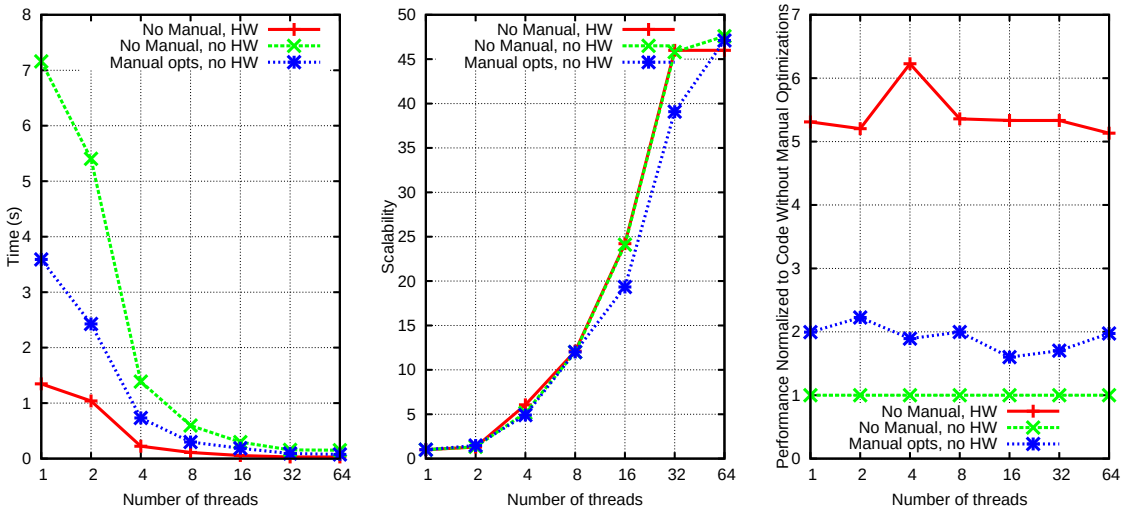
For the heat diffusion benchmark (Figure 4.27), the speedup for multiple locales varies from 8x to 13x.

The results are varying widely, especially at higher core counts, for a couple of reason:

1. the effective runtime of the benchmark inside the simulator is quite small due to the extremely long time necessary for the many-core simulation.

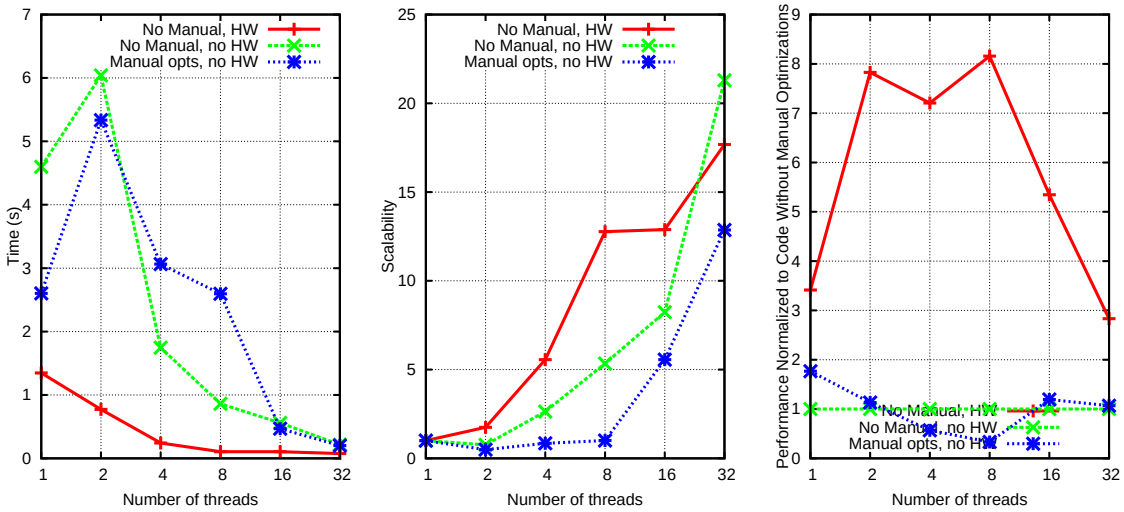
2. Compared to UPC, Chapel is a dynamic language that often creates new threads, and it takes some extra time for the Linux kernel to balance them efficiently across the available cores.

4.4.3.2 Timing Model



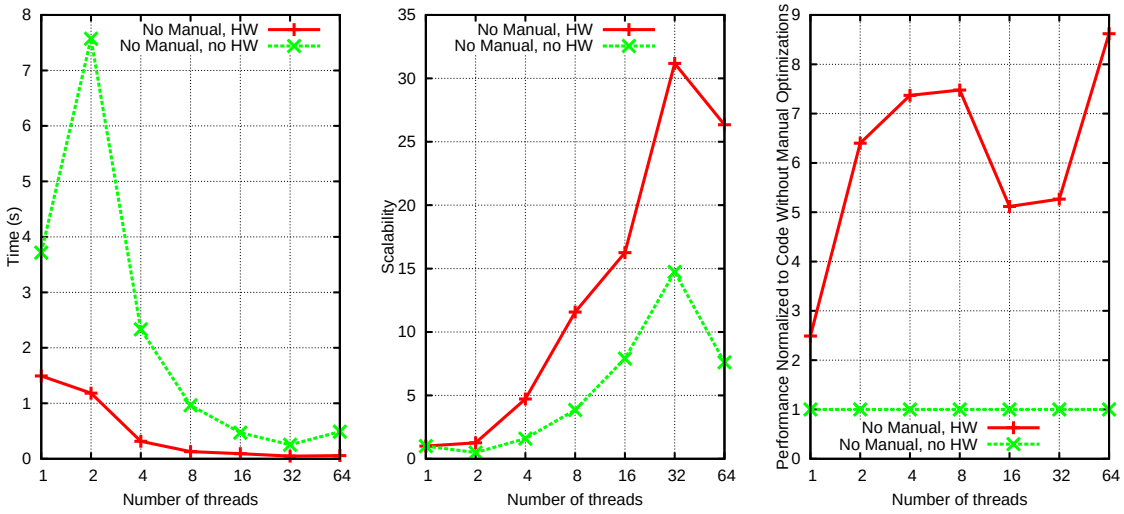
(a) Time **(b)** Scalability **(c)** Improvement

Figure 4.28: Gem5 Timing Model: Chapel STREAM Scale - 4M



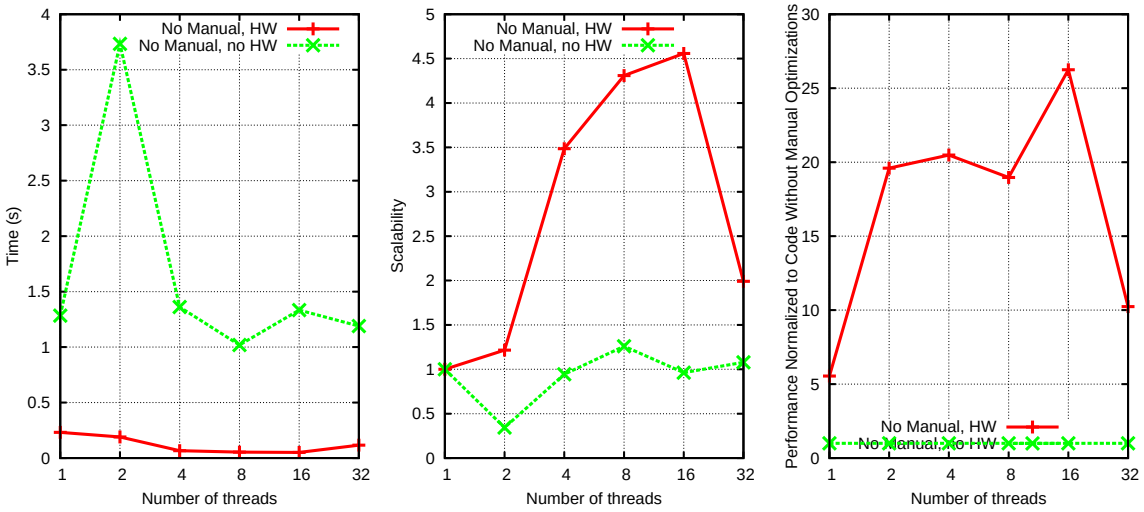
(a) Time **(b)** Scalability **(c)** Improvement

Figure 4.29: Gem5 Timing Model: Chapel Matrix Multiplication 128x128



(a) Time **(b)** Scalability **(c)** Improvement

Figure 4.30: Gem5 Timing Model: Chapel Random Access



(a) Time **(b)** Scalability **(c)** Improvement

Figure 4.31: Gem5 Timing Model: Chapel Heat Diffusion 64x64x64

With the timing model (Figures 4.28-4.31), speedup for multi-locales runs vary from 3x to 19x.

STREAM Scale and Matrix multiplication (Figure 4.28 and 4.29) provide on average, results similar to those obtained with the atomic model.

Random access speedups (Figure 4.30) are less important (but still significant) as proportionally more time is spent in the cache hierarchy. Heat Diffusion (Figure 4.31 provides an important speed up of 19x. Its regular accesses allow an efficient use of the cache which in turn make the extra overhead of address translation extremely costly.

4.4.4 Summary of Findings

Overall, our proposed hardware support for PGAS address mapping provides very significant speedup in all cases. Such speedup gains provide performance levels that are at least comparable to those of manually optimized codes. In some cases, it provides great performance improvement even compared to the best manual optimization.

Moreover, our approach does not require any modification or manual tuning of the

code to take advantage of the improved performance, thus it efficiently enables both the performance and productivity of PGAS languages.

Chapter 5 - Area and Timing Study

The full-simulation approach permits us to obtain accurate results in terms of the performance. However, it is not adequate in providing estimates of the hardware resource requirements, timing issues and effects on the processor clock frequency if any, as well as feasibility of the proposed hardware design.

To address this need, we implemented a full-system prototype on an FPGA. The following sections describe the details of the FPGA design and implementation, and summarize the results. In addition, sensitivity of the performance with respect to the latency of the custom instructions was also investigated.

5.1 Hardware Support Implementation on FPGAs

FPGAs are chips that can be programmed to implement different logic circuits, and can often be rapidly reconfigured from implementation to another. It is even possible to deploy a logic circuit that describes a software-programmable CPU on an FPGA. Leon3 is one such soft-core processor, i.e. it can be fully implemented on a FPGA using logic synthesis. As the HDL source of the processor is provided, it can be modified to incorporate our specific PGAS hardware support.

The FPGA prototype of our proposed hardware should allow us to evaluate the feasibility in terms of timing and area usage in a multi-core scenario. To enable this evaluation, an FPGA-based multi-core research platform was developed based on the Leon3 softcore processor. Then, the PGAS support was added to the experimental platform.

5.1.1 Platform for Multi-core Research

In order to be an accurate and realistic platform for multi-core processor evaluation, the FPGA-based multi-core platform has to meet the following requirements:

- Processor cores should be capable of running a full operating system in an SMP

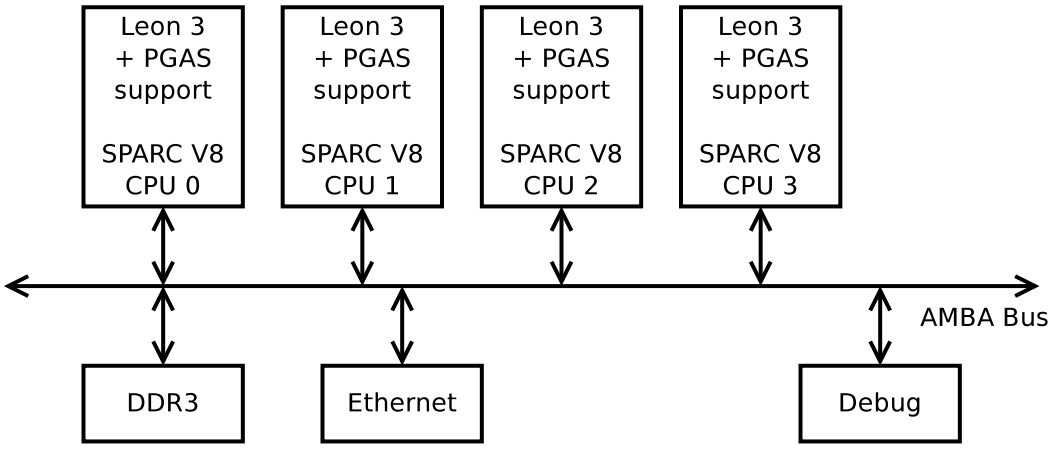


Figure 5.1: 4 Core Leon3 SMP with PGAS Support

configuration

- Virtual memory should be available in order to allow for process separation
- Physical shared memory must be supported with adequate cache coherency (to allow efficient data exchange and reuse between the processor cores)

Each processor core also should occupy only a small area on the chip in order to allow the placement of multiple cores on one FPGA.

We identified various mature soft-processor implementation and narrowed down our list to OpenSPARC, OpenRISC, Xilinx MicroBlaze, Altera NIOS, and Leon3.

The OpenSPARC open-source processors (1 and 2) were not selected due to their large area usage per core. The OpenRISC 1200 processor is smaller and does support SMP, but the Linux port does not support this feature, yet.

The Xilinx Microblaze and the Altera NIOS soft-core processors do not have support for cache coherency and modifications are prevented by their proprietary licenses.

Therefore, Leon3 was chosen as the base for the platform. It allows a cache coherent multi-core system to be implemented on an FPGA, runs GNU/Linux and allows extension of the instruction set through a coprocessor tightly coupled with the main pipeline.

There was no complete platform available for evaluating multi-core systems with hardware extensions to the processor. Therefore, we first modified Leon3 in order to support a generic, reconfigurable co-processor. This framework enabled us to make changes to and test the hardware extensions without rerunning the synthesis and place and route tools for the entire multi-core system, saving several hours on each design iteration. As a result, we were able to carry out rapid development and testing of the proposed hardware support. The results of this work was published in [52].

5.1.1.1 Related Work on Reconfigurable Processors

There has been a lot of reconfigurable processor research in the past, most notably, PRISC [53], OneChip [54], and Chimaera [55]. PRISC is a RISC based microprocessor that includes reprogrammable function units. OneChip is based on MIPS and integrates a tightly coupled reconfigurable core. Chimaera integrates a superscalar processor with a reconfigurable function unit; the Chimaera C compiler has the ability to automatically map instructions to the reconfigurable unit. All of these were simulation-based models without hardware prototypes.

Recently, multi-core reconfigurable base systems were studied, for example, Amalgam [56], which provides 4 cores and 4 reconfigurable cores without SMP, exchanging data through registers. In [57] a multi-core base system is simulated with Simics while the coprocessors run on a Virtex 5 FPGA. This is very realistic, however it is not suitable for hardware functions tightly integrated into the processor pipeline. In [58], a heterogeneous multi-core system on chip is presented, however the system is based on Microblaze which is not suitable for running a full SMP OS like Linux (Polyblaze[59] was later realized to add SMP support to Microblaze).

Depending on the granularity of the computations that are offloaded to a reconfigurable coprocessor, different degrees of coupling between the processor and reconfigurable logic

are beneficial. For instance, large, independent tasks can be deployed on a loosely coupled hardware unit, as the overheads of the initial data transfer can be amortized over its execution time. This also allows for a simpler system design. On the other hand, speeding up fine-grained computations such as ours demands a tightly integrated coprocessor [60], as the data transfer cost between the processor and its coprocessor is greatly reduced compared to a loosely coupled approach. Furthermore, both of them have access to the same memory hierarchy. A tightly coupled approach also simplifies programming as a single program controls both the processor core and its coprocessor.

5.1.1.2 Leon3

The Leon processors are developed by Aeroflex Gaisler, mainly for aerospace applications (they were originally developed by the European Space Agency).

The Leon3 32-bit processors in the form of synthesizable VHDL sources are publicly available, released under the GNU General Public License v2 (GPL), as part of the Gaisler IP library (GRLIB). The open-source version is fully featured, yet it does not include the radiation hardened Fault-Tolerance and the IEEE-754 Floating Point Unit (FPU) available in the licensed counterpart. The open-source nature of Leon3 makes it ideal for research [61]. It has the advantage of supporting cache coherent SMP systems allowing it to use the full GNU/Linux operating system. It also has already been used to study various specific hardware support possibilities [62, 63].

Leon3 is compliant with the SPARC V8 specifications, allowing it to run a great variety of software and operating systems, including GNU/Linux. Multi-processor operation is possible either in a symmetric or an asymmetric configuration (for example, see Figure 5.1, for a symmetric configuration). Leon3 is based on a 7 stage pipeline, see Figure 5.7.

Leon3 also supports a wide variety of development features: it can be configured with a Debug Support Unit allowing, in conjunction with the debug monitor (GRMON), to fully

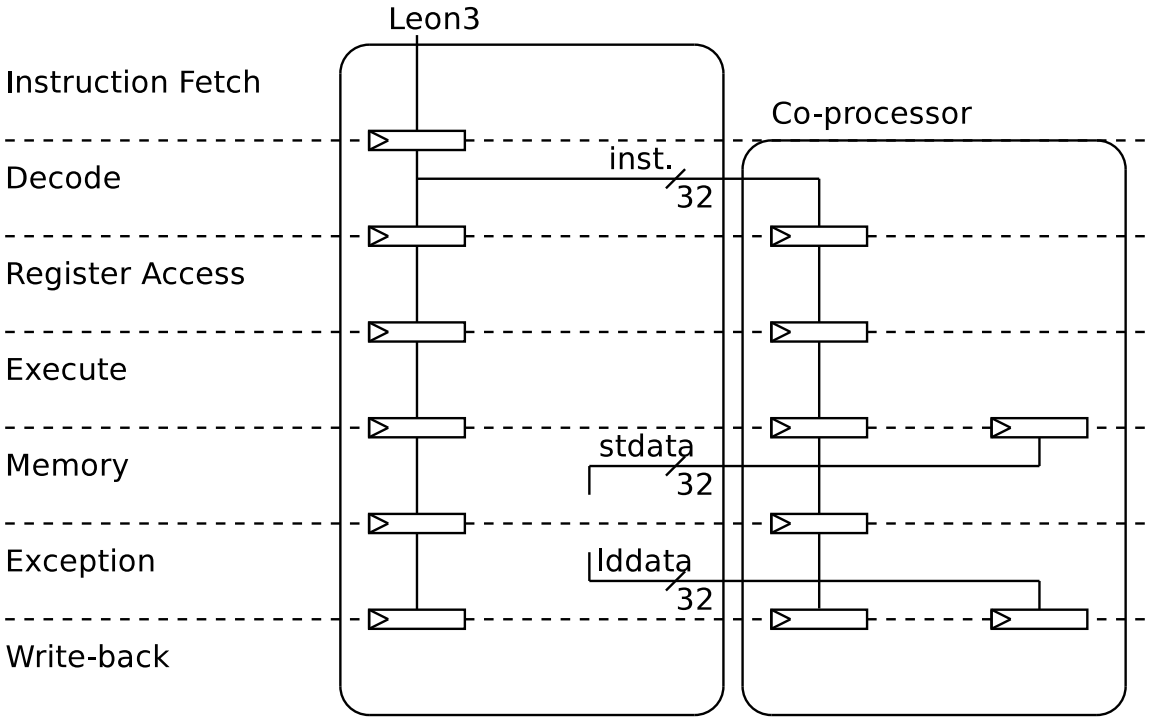


Figure 5.2: The Leon3 and the Proposed Coprocessor Pipelines. (Also represented, the pipeline timing for the load and store operations between the two units.)

debug the targeted system through operations such as reading/writing memory and registers, setting breakpoints and stepping through the code, attaching the GNU debugger (gdb), and monitoring the transactions on the AMBA bus. Simulators for Leon3 have also been developed by Aeroflex Gaisler, like TSIM2 and GRSIM, but they are closed-source and licensed products.

5.1.1.3 Coprocessor Interface

The Leon2 processor was widely used for reconfigurable processor research [64, 62] but unfortunately the simplified coprocessor interface which was provided with Leon2 was removed from the Leon3 source code release, making the development of a coprocessor much more difficult (to the best knowledge of the authors, no publication describes a Leon3 processor connected to a coprocessor).

The Leon3 processor still exposes an interface for coprocessors, which is in fact used for

connecting the GRLIB FPU unit. It uses the `rst`, `holdn`, `clk` signals plus two structures `fpc_in_type` and `fpc_out_type`. Those two structures basically expose all the signals from the main pipeline to the coprocessor. This makes the interface particularly complex to work with. Nevertheless, it enables a tighter integration into the pipeline as compared with its predecessor Leon2.

In total there are 497 signals going in and out from the coprocessor interface. A simplified interface for the coprocessor was designed by reducing the number of signals from 497 to 162. The VHDL records for the simplified interface are presented on Figure 5.3.

The Leon3 core was also adjusted to properly take into account the configuration settings and signals related to the coprocessor (e.g. `CPEN`, `holdn` and `dbg.data`). More importantly, the store datapath from the coprocessor to the memory was also missing and has been incorporated for this work.

The SPARC v8 architecture manual [65] also defines which instructions are available to control the coprocessor (see Table 5.1). The two first sets of instructions (LD and ST) are used to transfer data to and from the coprocessor. Depending on the addressing mode, the load/store instructions have the following format:

op1 (2)	rd (5)	op3 (6)	rs1 (5)	i (1)	zero (8)	rs2

Effective address = r[rs1]+r[rs2].

op1 (2)	rd (5)	op3 (6)	rs1 (5)	i (1)	simm (13)

Effective address = current address + sign ext(simm).

In both those instructions, `rd` is the destination register and its underlying semantics is dependent on the coprocessor implementation. The rest of the load/store instructions is interpreted by the host processor and cannot be changed by reconfiguring the coprocessor. For instance, a load operation would bring data into the coprocessor, but internally the

```
type fpc_pipeline_control_type is record
  pc     : std_logic_vector(31 downto 0);
  inst   : std_logic_vector(31 downto 0);
  cnt    : std_logic_vector(1 downto 0);
  trap   : std_ulogic; -- instr. trapped
  annul  : std_ulogic; -- cancel next inst.
  pv     : std_ulogic; -- valid
end record;

type cp_out_type is record
  stdata : std_logic_vector(31 downto 0);
      -- data to memory
  exc    : std_logic;  -- except
  cc     : std_logic_vector(1 downto 0);
  ccv    : std_ulogic; -- branch code valid
  ldlock : std_logic;  -- load lock
  holdn  : std_ulogic; -- hold
end record;

type cp_pipeline_control_type is record
  cnt    : std_logic_vector(1 downto 0);
  trap   : std_ulogic; -- trap
  annul  : std_ulogic; -- cancel next inst.
  pv     : std_ulogic; -- valid
end record;

type cp_in_type is record
  holdn  : std_ulogic; -- processor holdn
  flush  : std_ulogic; -- pipeline flush
  exack  : std_ulogic; -- exception acknowledge
  d      : fpc_pipeline_control_type;
  a      : cp_pipeline_control_type;
  e      : cp_pipeline_control_type;
  m      : cp_pipeline_control_type;
  x      : cp_pipeline_control_type;
  lddata : std_logic_vector(31 downto 0);
      -- data from memory
end record;
```

Figure 5.3: Reduced VHDL Interface for the Coprocessor

Table 5.1: The SPARC Instructions Reserved for the Coprocessor

Instruction	op1	op3	Description
LDC	11	110000	Load Coprocessor
LDDC	11	110011	Load Double Coprocessor
LDCSR	11	110001	Load Coprocessor State Register
STC	11	110100	Store Coprocessor
STDC	11	110111	Store Double Coprocessor
STCSR	11	110101	Store Coprocessor State Register
STDCQ	11	110110	Store Double Coprocessor deferred-trap Queue
CPOP1	10	110110	Coprocessor Operate 1
CPOP2	10	110111	Coprocessor Operate 2
CB...	00	N/A	Branch on Coprocessor Condition codes

coprocessor may direct it into a FIFO instead of a register. Timing information, in terms of the pipeline stages in which the coprocessor load/store operations are finalized, is shown in Figure 5.2.

The following instruction format is suggested by the SPARC architecture manual for the coprocessor function execution:

op1 (2)	rd (5)	op3 (6)	rs1 (5)	opc (9)	rs2 (5)

op1 and op3 fields are also decoded by the processor so they cannot be changed. The other fields are implementation specific and can be re-purposed. With the two CPOP instructions, there is actually 25 bits available to implement the coprocessor instruction set.

The CB (Coprocessor Branch) instructions are interpreted by the processor and are not customizable, they rely on the condition codes (signal `cc`) returned by the coprocessor.

More information about the multi-core research platform designed for this work can be found in our paper describing the prototype platform in detail [52].

5.1.1.4 Multi-core Self-Reconfigurable System

The GRLIB distribution supports Leon3 in a multi-core operation in a shared-bus configuration, as shown in Figure 5.4. The Advanced Microcontroller Bus Architecture (AMBA) is used as the system bus, with the Advanced High-performance Bus (AHB) component being used for all high-bandwidth components such as the CPUs, memory controller, and Ethernet and USB interfaces. It also contains an Advanced Peripheral Bus (not shown) that is used for connecting low-bandwidth devices such as the serial port. The CPU is a bus master on the AHB, and accesses all devices on the bus in a memory-mapped fashion by asserting data requests on the bus. The memory controller, which serves cache fill or write requests from the CPU, is an AHB slave and fulfills all CPU requests by reading and writing to DDR2 memory. By using the coprocessor interface, we have added a generic reconfigurable coprocessor to each processor core, as depicted in Figure 5.4 for 2 CPUs.

In any FPGA, the logic circuit functionality is defined by a configuration bitstream that is loaded onto the FPGA at startup, typically by an external host computer or an external controller. Furthermore, Xilinx FPGAs support *partial run-time reconfiguration*, which enables part of the FPGA to be reconfigured with the desired functionality while the rest of

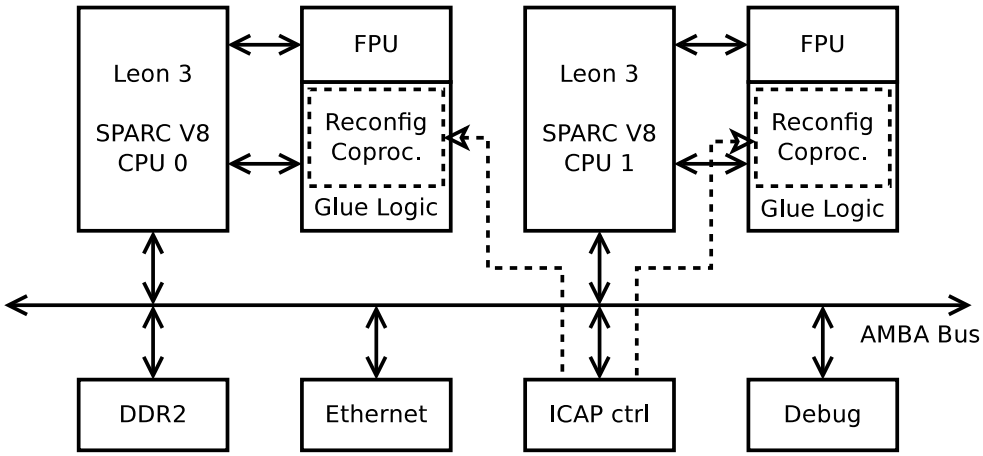


Figure 5.4: Leon3 SMP with Reconfigurable Coprocessors

the system is running. This is achieved by loading partial bitstreams into the chip. In order to allow our system to reconfigure the coprocessors without assist from an external controller, an Internal Configuration Access Port (ICAP) controller was written and connected to the AHB. This module enables self-reconfiguration by allowing the processors to load new partial bitstreams whenever needed. The controller behaves as an AHB slave and thus allows any processor core to reprogram any coprocessor.

5.1.1.5 Linux Kernel Driver

In order for the ICAP controller to be accessible by the user space programs, a kernel driver is needed and was thus written for our prototype. The kernel driver's first task is to register itself as a driver (via the `of_register_platform_driver` kernel function) for the ICAP controller. Once this is done, a new character device is created (`/dev/icap`) to allow the user space applications to communicate with the driver. When the Linux kernel probes the ICAP device, the driver maps the IO device memory to the kernel address space (with `of_ioremap`). The following 4 IOCTL operations are provided to the applications to configure and reprogram the coprocessor:

- IOCTL_ICAP_COPROCESSOR_ENABLE: Modify the Processor State Register (`psr`) of the CPU core running user process, to allow the use of the coprocessor instructions. This will also configure the glue logic to connect the signals of the coprocessor to the main pipeline.
- IOCTL_ICAP_COPROCESSOR_DISABLE: Disable the coprocessor.
- IOCTL_ICAP_COPROCESSOR_STATUS: Get the current status of the coprocessor.
- IOCTL_ICAP_PROGRAM: Program the coprocessor using the bitstream at the virtual address provided by the user. The programming itself is protected by a mutex, so that two different cores cannot access the ICAP at the same time.

Table 5.2: Area Cost for Different Configurations of Leon3 - Slice Resources

Configuration	Slice resources	
	Registers	LUTs
Leon3 minimal configuration	2,355 (3%)	3,615 (5%)
2x Leon3 cores (MMU, L1, HW mult 2c, FPU, debug, ICAP, DES)	16,886 (24%)	36,963 (53%)
4x Leon3 cores (MMU, L1, HW 2c, FPU, debug, ICAP, DES)	-	76,431 (110%)
4x Leon3 cores (MMU, L1, HW 2c, debug, ICAP, DES)	17,969 (25%)	32,258 (46%)
6x Leon3 cores (MMU, L1, HW 2c, debug, ICAP, DES)	25,493 (36%)	46,508 (67%)

5.1.1.6 Platform Evaluation

The Xilinx ML-509 board was used for the experiments. The board is powered by a Virtex-5 XC5VLX110T FPGA. It provides 17,280 slices, 148× 36Kb block RAMs (which can also be used as two independent 18Kb block RAMs) and 64 DSP48E slices (each providing a 25×18 multiplier). The board also includes a JTAG interface, a Gigabit Ethernet port and 256 MB of DDR2 RAM.

The following software tools were used: Synopsys Synplify Premier version E2011.03 to synthesize, Xilinx ISE version 13.1 to place and route the different designs, Leon3 from GRLIB version 1.1.0 B4100, LinuxBuild 1.0.0 to build the GNU/Linux environment with the Linux kernel version 2.6.36-4. All the designs target a clock speed of 70 MHz.

The area cost for the Leon3 processor is evaluated as follows. A minimal configuration of Leon3 is chosen and extra features are added progressively. The increased area usage in terms of LUTs and block RAMs is used to approximate the area cost of each features. Table 5.2 and Table 5.3 show the total cost of fully functional configurations. Table 5.4 details the cost of individual features. Both tables can be used to easily evaluate the area cost of a given configuration. Figure 5.5 illustrates the placement of a 4-core Leon3 SMP with the reconfigurable coprocessors laid out at fixed locations on the FPGA.

Table 5.3: Area Cost for Different Configurations of Leon3 - BRAM and DSPs

Configuration	BRAM		DSPs
	18kB	36KB	
Leon3 minimal configuration	6	2 (5%)	
2 Leon3 cores (MMU, L1, HW 2c, FPU, dbg, ICAP, DES)	50	20 (31%)	40
4 Leon3 cores (MMU, L1, HW 2c, FPU, dbg, ICAP, DES)	-	-	-
4 Leon3 cores (MMU, L1, HW 2c, dbg, ICAP, DES)	88	36 (56%)	16
6 Leon3 cores (MMU, L1, HW 2c, dbg, ICAP, DES)	126	52 (79%)	24

Table 5.4: Area Cost per Additional Feature to the Configuration of Leon3

Feature	Slice resources		BRAM		DSP	Per
	Registers	LUTs	18kB	36kB		core
MMU (8 entries, combined, 4kB page size)	857	746				yes
L1 I$ (8kB in 2 sets, 32 bytes/line, LRU)	77	250	4	2		yes
L1 D$ (16kB in 4 sets, 32 bytes/line, LRU)	700	2,028	12	4		yes
HW multiplier (5 cycles latency)	144	565			2	yes
HW multiplier (2 cycles latency)	93	362			4	yes
Floating Point Unit (GRFPU lite)	1,140	2,872				yes
Floating Point Unit (GRFPU full)	3339	9,423	22	8	16	yes
Coprocessor (DES encryption core)	523	525	1			yes
Debug support (JTAG, Ethernet GRMon, breakpoints)	1,813	3,366	4	4		no
AHB ICAP controller	10	132	4			no

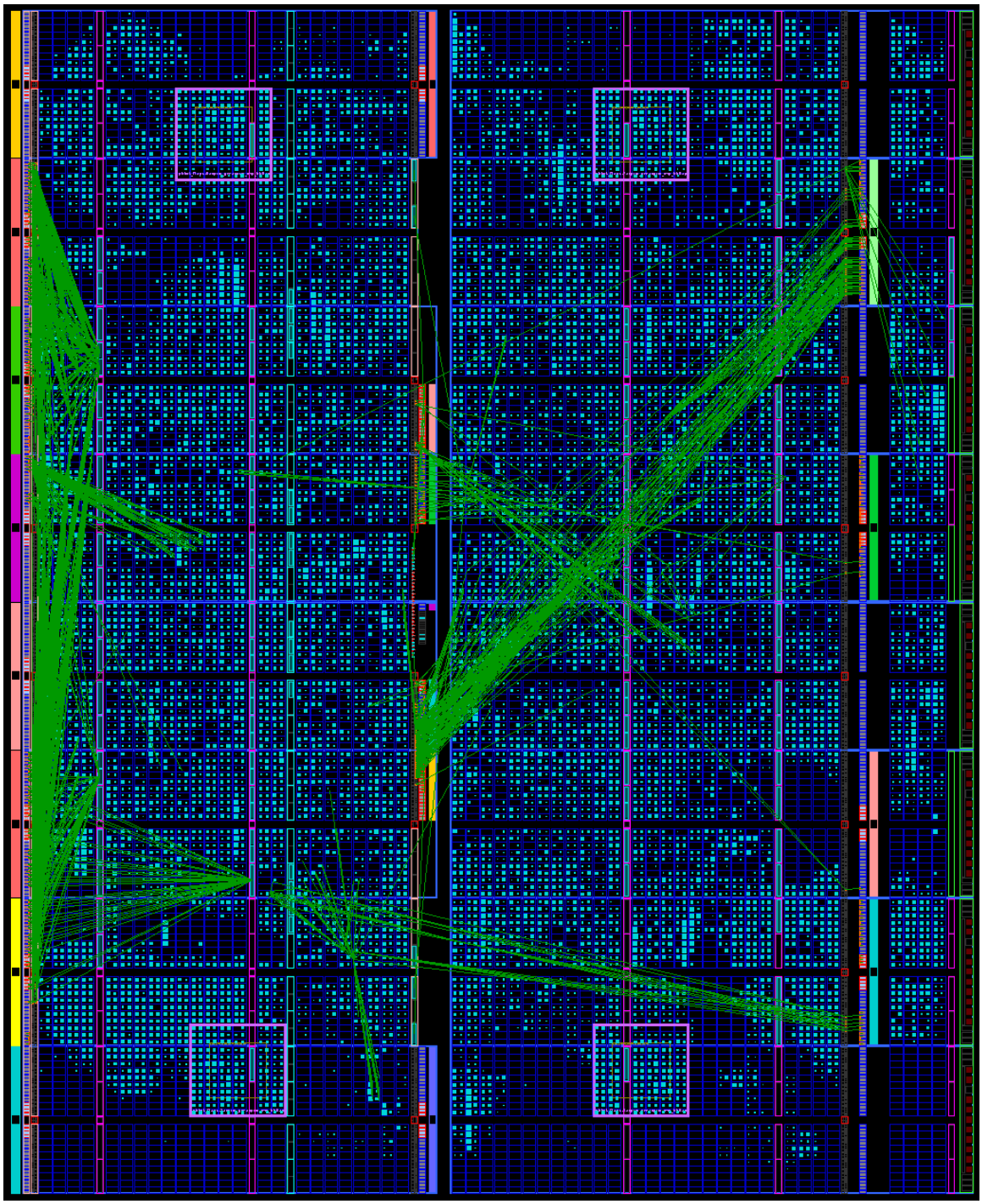


Figure 5.5: Placement of a 4-Core SMP Leon3 Design with Reconfigurable Coprocessors (the four squares represent the reconfigurable areas)

Table 5.5: Modified Stream Benchmark Performance Result (MB/s)

# of cores	Copy	Scale	Add	Triad
1	46.8	8.4	40.0	11.1
2	64.0	16.6	60.0	21.7
3	66.2	24.9	62.6	31.6
4	68.6	33.7	64.0	41.7

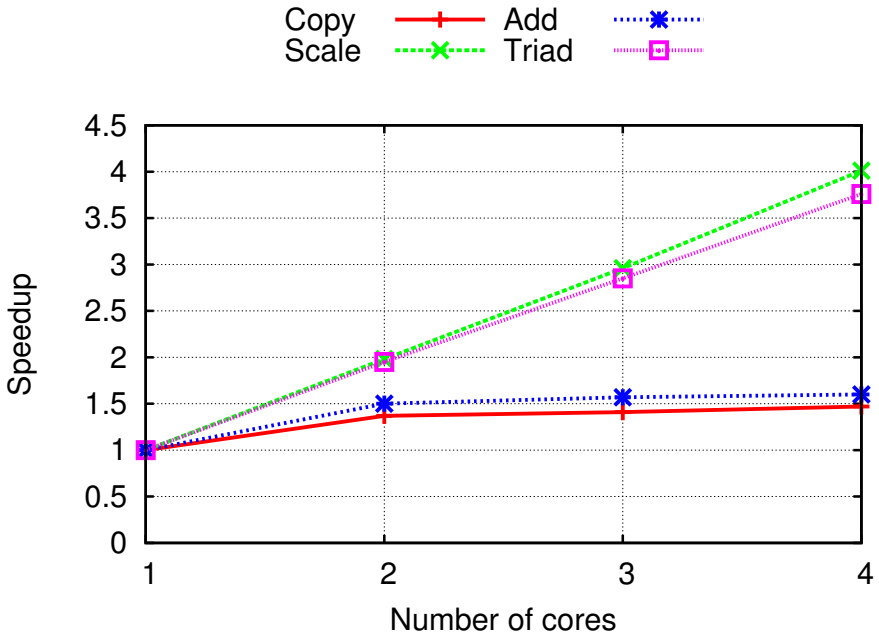


Figure 5.6: Scalability of the Modified Stream Test, (showing the contention on the AMBA bus)

Table 5.6: Implementation time for a 4 core SMP design

Implementation step	Full design	PR regions only
Synthesis	14m 55s	5s
Place and route	47m 24s	32m 28s
Total	62m 19s	32m 33s

In order to further evaluate our platform, we studied the memory access performance, particularly because all cores share the AHB in order to access memory. A modified version of the OpenMP Stream benchmark was used for this evaluation. On this modified version, the floating point operations were replaced by integer operations (double data types were replaced by long long), due to the lack of floating point unit on our test platform.

The results can be seen on Table 5.5. Figure 5.6 shows the scalability on our platform. We can see that the scaling is very good for the more compute intensive benchmarks, but scaling stops at 2 cores for memory intensive operations. Due to this limited scaling, it is important that the caches are dimensioned appropriately and used efficiently. Coprocessors can also use block RAM as buffer to reduce memory accesses.

The reconfigurable platform is also very helpful to reduce development times. A study presented in Table 5.6 shows the time needed to synthesis, place and route a design. The iteration time needed for a tightly coupled coprocessor is greatly reduced, as a only a small portion of the design is changed at a time. This greatly improve the development time and facilitate debugging.

The reduced development time was a great help for the design of the PGAS hardware support.

5.1.2 FPGA Implementation Details of the PGAS Hardware

The proposed hardware support for PGAS was designed and implemented on a Virtex-6 (XC6VLX240T) FPGA using the ML605 evaluation board. Xilinx ISE Release 13.4 was used to perform the logic synthesis, place, and route for the design. The final design runs

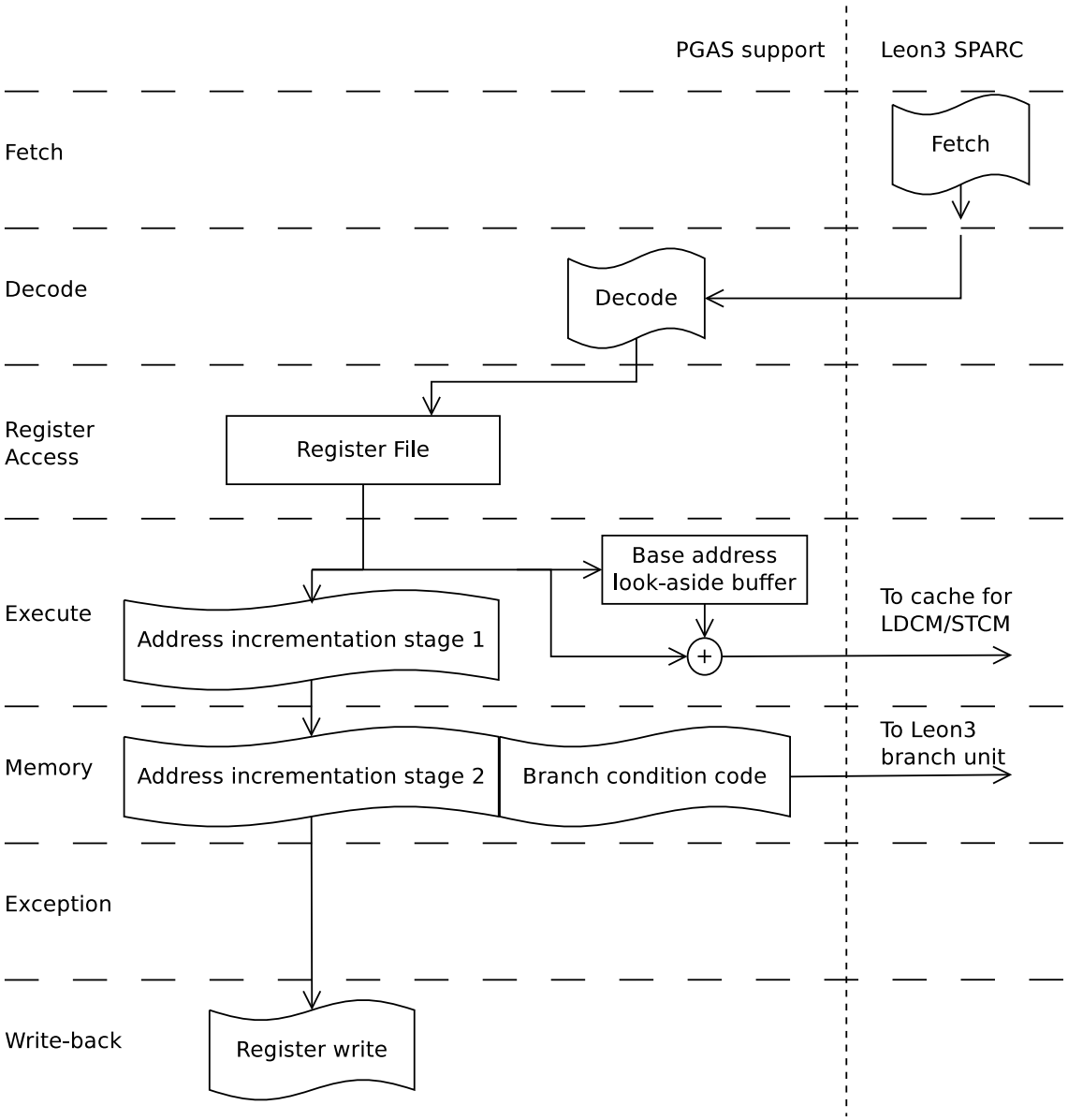


Figure 5.7: The 7-stage Leon3 Pipeline Extended with PGAS Support

at a frequency of 75 MHz. See Table 5.7 for the complete configuration. The processor core was extended with hardware support for PGAS shared addresses by using the reserved SPARC V8 instructions available for coprocessors.

The new instructions (see Table 5.8) are fully integrated with the main processor pipeline.

A new register file is introduced for storing the shared address pointers as those are 64

Table 5.7: Leon3 Configuration

	Configuration
Cores	4x SPARC cores (SMP)
Features	2-cycle multiplier, branch prediction
Cache	Cache coherent
L1 I	2 Sets, 8 kB/set, 32 bytes/line, LRU
L1 D	4 Sets, 4 kB/set, 16 bytes/line, LRU
FPU	Not implemented
BUS	AMBA AHB with fast snooping
Memory	Xilinx MIG-3.7 DDR3-800
Frequency	75 MHz
OS	GNU/Linux, Linux version 2.6.36

Table 5.8: PGAS Hardware Support SPARC V8 ISA Extension

Coprocessor Load/Store	
LDC	Load to Coproc. reg. (32 bits)
STC	Store from Coproc. reg. (32 bits)
Shared Address Load/Store	
LDCM	Load Long (32 bits)
STCM	Store Long (32 bits)
Branch	
CB123	Branch on locality
Shared Address Incrementing	
SH_ADD_INC	Immediate
SH_ADD_INC_REG	Register

bits, while Leon3 is a 32-bit processor. The register file is very similar to the register file used for floating point support in Leon3. It enables reading two 64-bit values and writing one 64-bit value per clock cycle. The extra register file would not be needed for 64-bit architectures, as was the case with our Gem5 Alpha implementation.

Figure 5.7 presents the Leon3 pipeline extended with hardware support for shared pointers. The address incrementation is fully pipelined over two stages, allowing a processor core to perform one address incrementation per clock cycle. It also generates a coprocessor condition code based on the locality of the incremented address. Four condition codes are possible:

- 0: Local (the pointed data is owned by the current thread),
- 1: Located on the same memory controller,
- 2: Accessible by load/store with the PGAS instructions,
- 3: Located on another node.

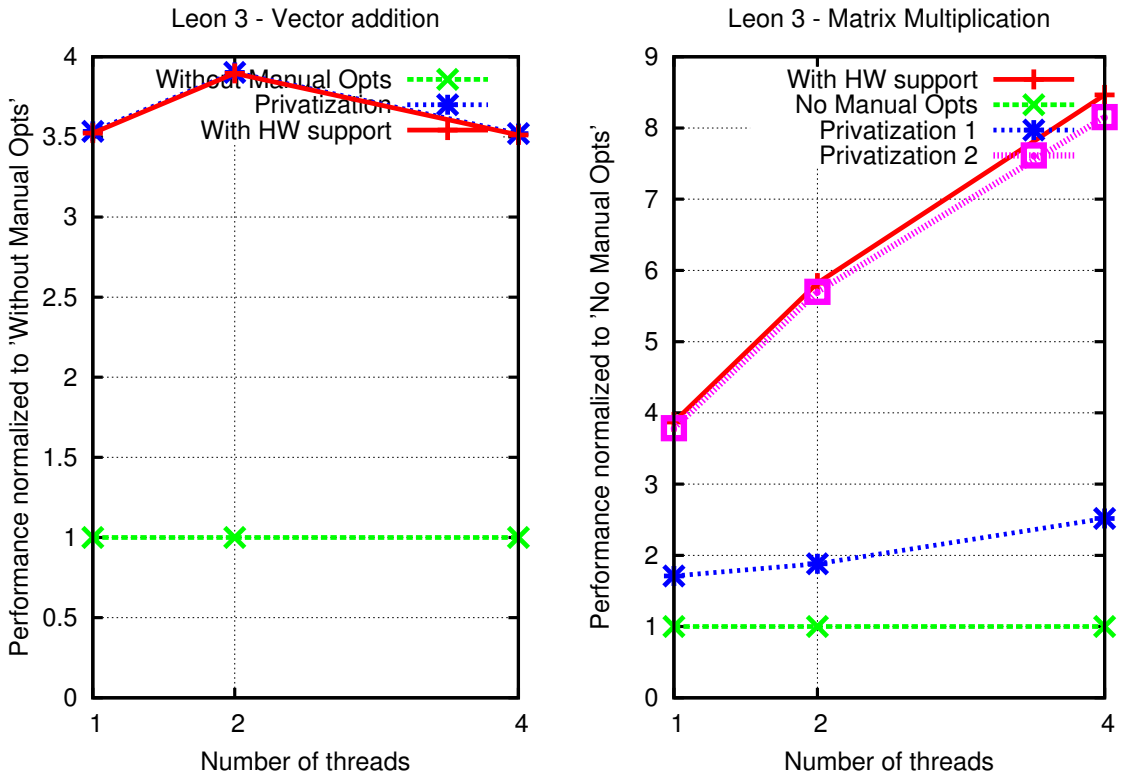
The Coprocessor Branch (CB) instruction allows branching based on a great variety of condition codes. Loads and stores from shared addresses (LDCM, STCM) are performed as fast as the normal SPARC load and store instructions (see Figure 5.7 for an overview of the pipeline timing).

In the SPARC architecture, the coprocessor instructions have to be enabled in the supervisor mode. A Linux kernel driver was written in that effect. It only allows one process to access the shared address register file, so that saving and restoring the register file on a context switch is not needed.

5.2 Performance Evaluation

Two micro-benchmarks (vector addition and matrix multiplication) were implemented to verify the functionality and performance of the hardware design implemented on the FPGA. They were compiled using Berkeley UPC 2.12.1 and GCC 4.4.2 for SPARC with all optimizations enabled (-O, -opt for BUPC, -O3 for GCC). As adding the support for the extra shared address register file in GCC would require considerable work, a full compilation support, as in our full system simulation work, was not possible. The hardware-accelerated code was partially written in assembly.

In the vector addition benchmark, two integer vectors are added together. Results are shown in Figure 5.8. Both the manually-optimized code (using only private pointers) and the non-optimized version using hardware support ran 3.5 times faster. The performance



(a) Vector Addition

(b) Matrix Multiplication

Figure 5.8: Leon3 with PGAS HW Support - Benchmarks

improvement shrinks as the number of threads increases, as the benchmark is quickly able to saturate the bandwidth-limited shared AMBA bus.

The matrix multiplication benchmark performs a 64x64 integer matrix multiplication. Two different levels of manual optimization were performed. The first one uses private pointers to access one of the matrices (privatization 1) and the second one uses a non-standard UPC extension to access all the matrices with private pointers. As seen in Figure 5.8, the code with hardware support matches the performance of the fully optimized version (8.4x faster than the non-manually-optimized code).

For both benchmarks, the hardware support performance matches or surpasses the performance of manually-optimized code.

Table 5.9: PGAS Instruction Stall Cycles Sensitivity Study - CG

	atomic			timing			detailed		
# Th	0/0	+1/0	+1/+1	0/0	+1/0	+1/+1	0/0	+1/0	+1/+1
1	1.20x	1.20x	1.20x	1.15x	1.15x	1.15x	1.09x	1.09x	1.08x
2	1.30x	1.30x	1.30x	1.22x	1.22x	1.22x	1.11x	1.10x	1.10x
4	1.41x	1.40x	1.40x	1.26x	1.26x	1.25x	1.04x	1.03x	1.03x
8	1.64x	1.62x	1.61x	1.34x	1.34x	1.34x	1.07x	1.07x	1.07x
16	1.89x	1.86x	1.85x	1.21x	1.20x	1.20x			
32	2.13x	2.08x	2.07x	1.11x	1.11x	1.11x			
64	2.62x	2.54x	2.53x	1.13x	1.13x	1.13x			

0/0 is the baseline, +1/0 adds 1 stall cycle for loads and +1/+1 adds 1 stall cycle for both loads and stores.

Table 5.10: PGAS Instruction Stall Cycles Sensitivity Study - FT

	atomic			timing			detailed		
# Th	0/0	+1/0	+1/+1	0/0	+1/0	+1/+1	0/0	+1/0	+1/+1
1	1.58x	1.55x	1.52x	1.44x	1.44x	1.43x	1.27x	1.24x	1.20x
2	2.27x	2.21x	2.15x	1.79x	1.78x	1.76x	1.45x	1.41x	1.37x
4	2.36x	2.30x	2.24x	1.75x	1.74x	1.73x	1.33x	1.30x	1.28x
8	2.34x	2.28x	2.22x	1.58x	1.57x	1.56x			
16	2.34x	2.28x	2.22x	1.21x	1.20x	1.20x			

5.3 Performance Sensitivity to Instruction Latencies

In the previous section, we presented a complete PGAS hardware support implementation with fully pipelined instructions that do not add additional instruction latency. However, there is a design trade-off between the instruction latency and the area of the chip used for the implementation. For example, a chip designer may prefer to reduce the area allocated for the new instructions by increasing the instruction latency (i.e. increasing the clock cycles/instructions to reduce the number of transistors used).

To explore this trade-off, we analyzed the sensitivity of the hardware support's performance to increased instruction latency in the following three cases: 0 extra cycles for loads

Table 5.11: PGAS Instruction Stall Cycles Sensitivity Study - IS

	atomic			timing			detailed		
# Th	0/0	+1/0	+1/+1	0/0	+1/0	+1/+1	0/0	+1/0	+1/+1
1	1.84x	1.77x	1.66x	1.75x	1.73x	1.69x	1.23x	1.21x	1.18x
2	3.07x	2.95x	2.77x	2.44x	2.41x	2.36x	1.30x	1.28x	1.25x
4	3.44x	3.30x	3.09x	2.50x	2.47x	2.43x	1.34x	1.31x	1.28x
8	3.88x	3.70x	3.47x	2.40x	2.37x	2.34x			
16	3.74x	3.57x	3.38x	1.94x	1.93x	1.91x			
32	3.64x	3.50x	3.33x	1.37x	1.37x	1.36x			
64	3.12x	3.04x	2.94x	1.07x	1.07x	1.07x			

Table 5.12: PGAS Instruction Stall Cycles Sensitivity Study - MG

	atomic			timing			detailed		
# Th	0/0	+1/0	+1/+1	0/0	+1/0	+1/+1	0/0	+1/0	+1/+1
1	4.25x	3.56x	3.50x	3.65x	3.45x	3.43x	2.93x	2.21x	2.16x
2	6.39x	5.37x	5.29x	4.76x	4.51x	4.48x	3.03x	2.42x	2.37x
4	6.71x	5.65x	5.56x	4.63x	4.41x	4.39x	2.34x	1.99x	1.96x
8	6.58x	5.56x	5.48x	4.25x	4.06x	4.04x	1.61x	1.47x	1.45x
16	6.54x	5.54x	5.45x	3.11x	3.02x	3.01x			
32	6.31x	5.38x	5.30x	1.83x	1.80x	1.80x			
64	5.73x	4.97x	4.90x	1.29x	1.28x	1.28x			

and stores, “0/0”; 1 extra cycle for loads alone,“+1/0” (we consider that the latency for the write is easier to mask out, as CPU write are generally posted writes); and 1 extra cycle for both loads and stores, “+1/+1”.

This analysis is performed with the modified Gem5 simulator running the UPC benchmarks as presented in Chapter 4.

We model the extra-cycles as a complete stall of the core which is the worst-case performance degradation. In a modern implementation, the stall would most likely only impact the PGAS functional unit and have a much moderate effect on the performance.

Tables 5.9—5.15 compare the speedup results of *No Manual Optimization with HW*

Table 5.13: PGAS Instruction Stall Cycles Sensitivity Study - MM

	atomic			timing			detailed		
# Th	0/0	+1/0	+1/+1	0/0	+1/0	+1/+1	0/0	+1/0	+1/+1
1	5.99x	5.00x	4.99x	5.14x	4.79x	4.79x	3.41x	2.34x	2.34x
2	7.10x	5.92x	5.92x	5.88x	5.48x	5.48x	3.45x	2.37x	2.36x
4	6.98x	5.82x	5.82x	5.88x	5.48x	5.48x	3.63x	2.49x	2.48x
8	7.05x	5.88x	5.88x	5.88x	5.48x	5.48x	3.45x	2.37x	2.37x
16	7.12x	5.93x	5.93x	5.83x	5.43x	5.43x			
32	6.95x	5.82x	5.82x	5.75x	5.37x	5.37x			
64	7.00x	5.86x	5.86x	5.61x	5.25x	5.25x			

Table 5.14: PGAS Instruction Stall Cycles Sensitivity Study - Sobel

	atomic			timing			detailed		
# Th	0/0	+1/0	+1/+1	0/0	+1/0	+1/+1	0/0	+1/0	+1/+1
1	2.70x	2.42x	2.38x	2.67x	2.55x	2.54x	2.01x	1.55x	1.50x
2	3.05x	2.73x	2.69x	2.97x	2.84x	2.82x	2.20x	1.69x	1.63x
4	3.06x	2.74x	2.69x	2.98x	2.85x	2.83x	2.20x	1.69x	1.63x
8	3.06x	2.74x	2.69x	2.96x	2.83x	2.81x	2.74x	2.11x	2.04x
16	4.61x	4.15x	4.08x	2.96x	2.84x	2.82x			
32	4.53x	4.08x	4.01x	2.95x	2.83x	2.81x			
64	3.00x	2.70x	2.66x	2.95x	2.82x	2.81x			

Support over the *No Manual Optimization* scenario.

The impact of the extra hardware cycles is quite small for CG, FT, IS and MG. At worst for MG, the speedup is reduced from 1.61x to 1.45x.

Matrix Multiplication and Sobel showed some additional reduction in performance improvement; for matrix multiplication from 3.45x to 2.37x and for Sobel a reduction in performance improvement from 2.74x to 2.04x using the *Gem5 detailed* model with 8 cores.

The performance benefit of the proposed hardware support remains significant across all the cases we studied. This gives a lot of freedom to the hardware designer to trade-off latency/area when designing the chip.

Table 5.15: PGAS Instruction Stall Cycles Sensitivity Study - Heat

	atomic			timing			detailed		
# Th	0/0	+1/0	+1/+1	0/0	+1/0	+1/+1	0/0	+1/0	+1/+1
1	2.45x	2.13x	2.09x	1.81x	1.75x	1.74x	2.17x	1.63x	1.57x
2	2.83x	2.58x	2.55x	2.21x	2.15x	2.14x	2.44x	1.96x	1.90x
4	2.73x	2.58x	2.56x	2.75x	2.69x	2.68x	2.27x	1.92x	1.88x
8	2.79x	2.69x	2.68x	2.86x	2.82x	2.81x	2.04x	1.85x	1.82x
16	2.70x	2.64x	2.64x	2.82x	2.79x	2.79x			
32	2.70x	2.67x	2.67x	2.85x	2.84x	2.83x			
64	2.65x	2.64x	2.64x	2.92x	2.91x	2.91x			

Table 5.16: Area Cost Evaluation for the PGAS Hardware Support

Configuration	Slice resources		BRAM		DSP48Es
	Registers	LUTs	18Kb	36Kb	
Leon3, 4 cores	46,718	59,235	106	34	16
Leon3, 4 cores + PGAS support	49,325	62,572	126	34	24
Virtex 6 - XC6VLX240T	301,440	150,720	832	416	768
Increase	2,607	3,337	20	0	8
Area increase, % of base	+5.6%	+5.6%	+18.9%		+50.0%
Area % of Virtex 6	+0.9%	+2.2%	+2.4%		+1.0%

5.4 Chip Area Dedicated to the Hardware Support

More importantly, the FPGA prototyping allowed us to evaluate the chip area needed for the PGAS hardware support.

The reconfigurable feature of our multi-core design platform was only used during the development and the final design does not use reconfigurability in order to avoid additional overheads.

Table 5.16 presents the FPGA resources used for a 4-core Leon3 SMP system with and without the hardware support. Results are both presented in terms of increase compared to the base Leon3 implementation, and in percentage of the FPGA chip used. The 5.6% increase in LUT resources may appear large at first but the area evaluation is very conser-

Table 5.17: Evaluation of the Transistor Count for the Base Leon3 Core

		Transistors
LUTs	$14809/4 \times 15 \times 5$	$1,776,960$
Block RAMs	$(106 \times 18 \times 1024 + 34 \times 36 \times 1024)/4 = 801,792$ bits	$4,810,752$
DSPs	$4 \times 21,000$	$84,000$
Total		$6,671,712$

Table 5.18: Evaluation of the Transistor Count for a Leon3 Core with the PGAS Hardware Support

		Transistors
LUTs	$15643/4 \times 15 \times 5 \times 6.4$	$1,876,800$
Block RAMs	$(126 \times 18 \times 1024 + 34 \times 36 \times 1024)/4 = 893,952$ bits	$5,363,712$
DSPs	$6 \times 21,000$	$126,000$
Total		$7,366,512$

vative as we are comparing against a very simple 32-bit processor core which does not even include floating point support.

Similarly, the 18.9% BRAMs and 50% DSP48Es increase may look significant at first, but they are mainly due to the extra 64-bit register file which would not be necessary on 64-bit architectures (as in our Alpha implementation on Gem5). Also, BRAMs and DSP48Es are scarcely used in the Leon3 design, and far from being fully used in the hardware support design, which makes the increase look larger than necessary. The proposed hardware support mechanism for 4 cores uses less than 2.4% of the overall FPGA chip.

In order to evaluate the area costs for an ASIC implementation, we can approximate the number of transistors needed for the Leon3 design and the hardware support. There are 4 LUTs per slice and each slice is equivalent to 6.4 logic cells [66]. Each logic cell in turn may be approximated to provide around 15 gates [67], and a logic gate on an average needs about 5 transistors [68]. A DSP block is approximated by 21,000 transistors using the data for a 32-bit multiplier [68]. We also use an approximation of 6 transistors per block RAM bit. Table 5.17 presents the estimation for the base Leon3 core. Table 5.18 presents the estimation for a Leon3 core with the additional PGAS hardware. We obtain an

estimated $694,800$ transistors for the hardware support which represents about 10% of the Leon3 core.

This can be compared with a 8-core Core i7 Haswell-E - $2,600,000,000$ transistors (325M transistors per core), or a 61-core Xeon Phi - $5,000,000,000$ transistors (82M transistors per core) [68]. The increase translates into a 0.2% increase on an Haswell and a 0.85% increase compared to a Phi core.

Chapter 6 - Conclusions and Future Directions

6.1 Summary and Conclusions

Ease of use and program efficiency are mutually opposing requirements in parallel programming models. The shared memory model is easy to use but unable to exploit locality. The message passing model, on the other hand, is difficult to use due to its fragmented address space and its need for explicit communications. The need for more productive, but also efficient, parallel programming models is quite evident.

The PGAS programming model provides a trade-off between those two models with a shared address space for productivity while allowing the exploitation of data locality. However, its performance can be hindered with the overhead associated with accessing and traversing its complex memory model.

Automatic compiler optimizations may help but they are not sufficient for achieving competitive performance. Hand tuning of PGAS code allows the user to achieve good performance levels, but it also strongly diminishes the productivity advantage.

In this work, we proposed the addition of a novel hardware support mechanism for addressing the inherent overheads in PGAS. It can be availed to and used easily by compilers though a simple extension of the instruction set. The proposed hardware thus affords minimal latency operation, enabling optimization of shared local accesses.

Testing and benchmarking were conducted using a modified version of the Gem5 full-system simulator, with many benchmarks including the well-accepted NAS Parallel Benchmark suite written in UPC. The results obtained with both the UPC and the Chapel languages show that the hardware support is very effective, and efficiently supported by compilers. The performance is, at a minimum, comparable to those obtained from hand-tuned code, which both demonstrates the performance and the productivity of this approach. For example, unmodified code with our proposed hardware support, achieved up to 18x speed-

up with Chapel and 5.5x speed-up with UPC as compared to the same code running without our hardware support but with full compiler optimizations. The work was also extended to include other modern PGAS paradigms, specifically Chapel, and has shown similar improvements when hardware support is used.

Moreover, the proposed hardware solution only requires a minimal increase in chip area as shown by our FPGA prototyping study. The PGAS support for a 4 core processor takes less than 2.4% of the FPGA chip used. A sensitivity study to the instruction latency was also performed and shows that a chip designer can easily chose to increase the instruction latency to further reduce the area overheads, enabling speed-area tradeoffs.

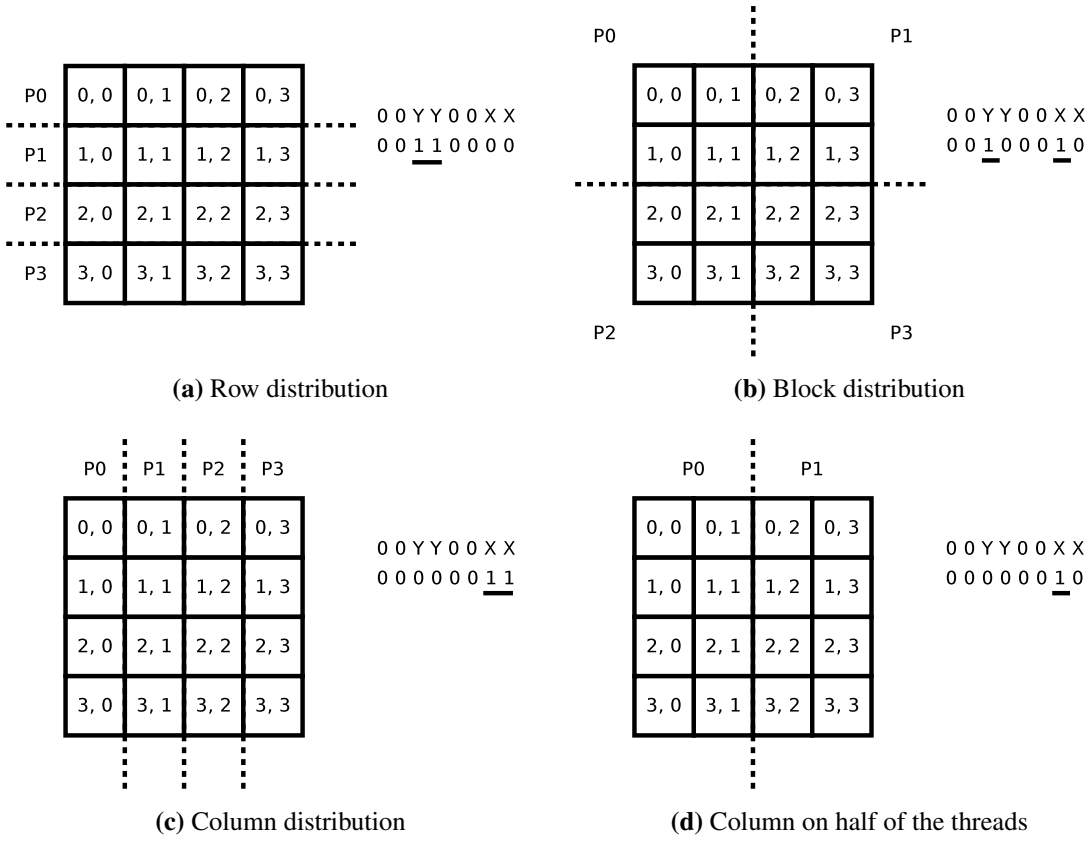
6.2 Future Directions

6.2.1 Other Data Distributions

For future work, we will extend the instruction set to support a greater variety of data distributions.

One possible way to support other distributions is to define bit-masks on array indices. The bit-mask is used to extract the thread number and the remaining bits are used to define an offset. Figure 6.1 illustrates some distributions possible with that scheme. For UPC, it would, for example, cover distributions with a multi-dimensional block factor as described in [69].

The hardware also has the advantage that it is fairly simple, see Figure 6.2. As part of the future work, it would be necessary to determine the practically achievable performance and the associated area cost. Furthermore, array allocation using this approach presents some difficulties, and would require some adapted algorithms.



(a) Row distribution

(b) Block distribution

(c) Column distribution

(d) Column on half of the threads

Figure 6.1: Distributions based on a Bit Select Function

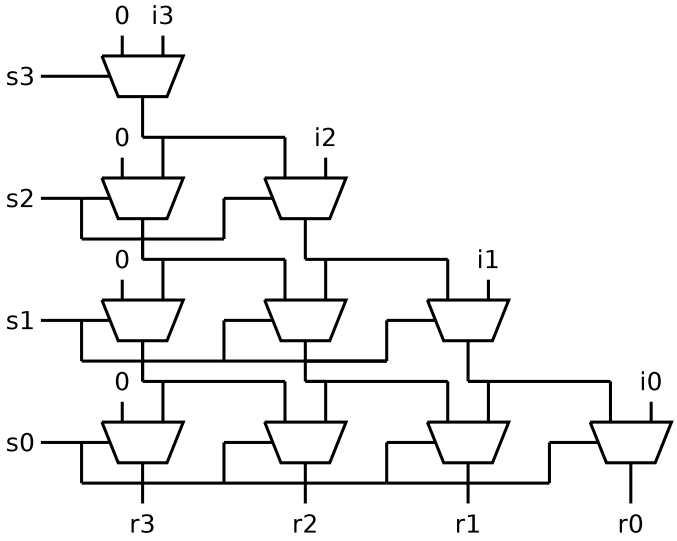


Figure 6.2: Bit Select Function Hardware

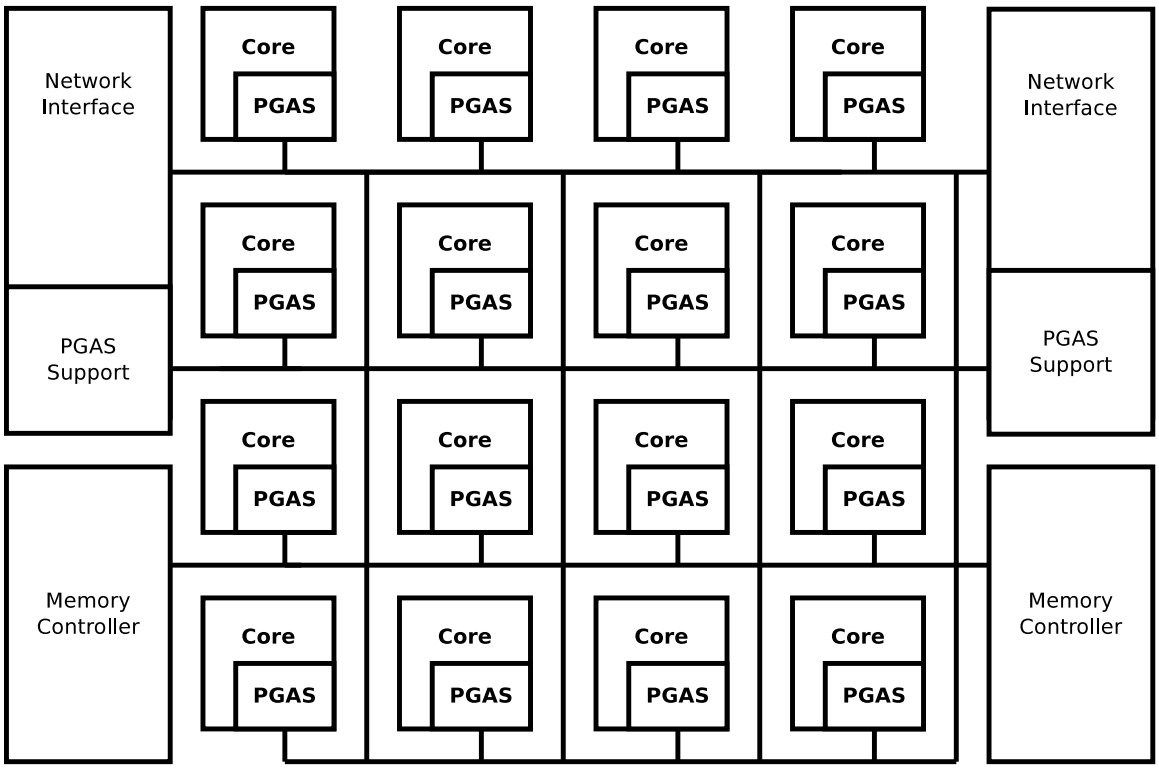


Figure 6.3: Many-core Hardware Support for PGAS

6.2.2 Extended Network Interface

The data mapping presented in this work is a first step and creates new opportunities for a better support for PGAS. The shared addresses allow a quick determination of the destination of a request across the system (through the `thread` field). This can be exploited for further improvements to remote data accesses across a full system of interconnected nodes.

This requires extending the PGAS hardware support to incorporate the network interface. Figure 6.3 presents such an extension for multi- and many-core chips.

We believe that the global solution will be hierarchical in order to limit the cost of additional hardware. In other words, individual cores would have PGAS support for accessing memory within the many-core node, and on top of that, the network interfaces would additionally incorporate PGAS support to serve a group of cores.

Such hardware will be able to provide features that are difficult or impossible to realize

with compiler support. Such as:

1. Aggregation: network interfaces are much more efficiently used with big packet sizes; the PGAS support hardware at the network interface will be able to group requests coming from different cores before being sent to the network;

2. Caching: a simple cache can be implemented at the network interface; this would be particularly efficient when multiple cores are trying to access the same data.

3. Prefetching: hardware prefetchers in modern processors are able to detect streams of sequential accesses to the memory, and automatically prefetch such future data blocks. A similar strategy driven by the program locality as exposed by PGAS can be employed at the network interface.

Bibliography

[1] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics Magazine Vol. 38, No. 8*, 1965.

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

[3] S. Borkar, "Thousand core chips: A technology perspective," in *Proceedings of the 44th Annual Design Automation Conference*, ser. DAC '07. New York, NY, USA: ACM, 2007, pp. 746–749. [Online]. Available: http://doi.acm.org/10.1145/1278480.1278667

[4] O. Serres, A. Anbar, S. G. Merchant, and T. El-Ghazawi, "Experiences with UPC on TILE-64 processor," in *proceeding of the IEEE Aerospace Conference (AeroConf)*, March 2011.

[5] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, M. Mattina, C.-C. Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook, "Tile64 - processor: A 64-core soc with mesh interconnect," in *IEEE International Solid-State Circuits Conference, ISSCC. Digest of Technical Papers.*, February 2008, pp. 88–598.

[6] T. El-Ghazawi, "Message of the PGAS 2015 General Chair," in *PGAS 2015: the 9th International Conference on Partitioned Global Address Space Programming Models*. IEEE, 2015.

[7] UPC Consortium, "UPC language specifications v1.2," May 2005.

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

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

[10] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," *Acm Sigplan Notices*, vol. 40, no. 10, pp. 519–538, 2005.

[11] F. Cantonnet, Y. Yao, S. Annareddy, A. Mohamed, and T. El-Ghazawi, "Performance monitoring and evaluation of a UPC implementation on a NUMA architecture," in *Proceedings of the International Conference on Parallel and Distributed Parallel Systems, IPDPS*, 2003.

[12] B. L. Chamberlain, "A brief overview of chapel," January 2013, pre-print of an upcoming book chapter.

[13] B. L. Chamberlain, S.-E. Choi, S. J. Deitz, and A. Navarro, "User-defined parallel zippered iterators in Chapel," 2011.

[14] O. Serres, A. Anbar, S. G. Merchant, A. Kayi, and T. El-Ghazawi, "Address translation optimization for Unified Parallel C multi-dimensional arrays," in *Proceedings of the 16th International Workshop on High-Level Parallel Programming Models and Supportive Environments, HIPS*. IEEE, 2011.

[15] F. Cantonnet, Y. Yao, M. Zahran, and T. El-Ghazawi, "Productivity analysis of the UPC language," in *Proceeding of the 18th International Parallel and Distributed Processing Symposium*, April 2004, pp. 254–.

[16] K. Ebcioglu, V. Sarkar, T. El-Ghazawi, and J. Urbanic, "An experiment in measuring the productivity of three parallel programming languages," in *Workshop on Productivity and Performance in High-End Computing (P-PHEC)*. IEEE, 2006.

[17] A. I. Stone, J. M. Dennis, and M. M. Strout, "Evaluating Co-array Fortran with the cgpop miniapp," in *Proceedings of the Fifth Conference on Partitioned Global Address Space Programming Models (PGAS)*, vol. 5, no. 8, 2011, p. 14.

[18] M. Weiland, "Chapel, Fortress and X10: novel languages for HPC," Tech. Rep., 2007.

[19] B. L. Chamberlain, S.-E. Choi, S. J. Deitz, and D. Iten, "HPC challenge benchmarks in Chapel," *HPC Challenge Awards Competition at SC09 (November 2009)*, 2009.

[20] B. L. Chamberlain, S. J. Deitz, S. A. Figueroa, D. M. Iten, and A. Stone, "Global HPC challenge benchmarks in Chapel," 2008.

[21] N. Dun and K. Taura, "An empirical performance study of Chapel programming language," in *26th IEEE International Parallel and Distributed Processing Symposium Workshops and PhD Forum (IPDPSW)*. IEEE, 2012, pp. 497–506.

[22] J. Milthorpe, V. Ganesh, A. P. Rendell, and D. Grove, "X10 as a parallel language for scientific computation: practice and experience," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2011, pp. 1080–1088.

[23] The Java Grande Forum benchmark suite, "http://www.epcc.ed.ac.uk/research/computing/performance-characterisation-and-benchmarking/java-grande-benchmark-suite."

[24] Y. Zheng, A. Kamil, M. B. Driscoll, H. Shan, and K. Yelick, "UPC++: a PGAS extension for C++," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. IEEE, 2014, pp. 1105–1114.

[25] Z. Zhang and S. R. Seidel, "Benchmark measurements of current UPC platforms," in *Proc. of IPDPS (PMEOPDS Workshop), 19th IEEE International Parallel and Distributed Processing Symposium*, 2005.

[26] H. Jin, R. Hood, and P. Mehrotra, "A practical study of UPC using the NAS parallel benchmarks," in *Proceedings of the Third Conference on Partitioned Global Address Space Programming Models*. ACM, 2009, p. 8.

[27] H. Shan, F. Blagojević, S.-J. Min, P. Hargrove, H. Jin, K. Fuerlinger, A. Koniges, and N. J. Wright, "A programming model performance study using the NAS parallel benchmarks," *Scientific Programming*, vol. 18, no. 3, pp. 153–167, 2010.

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

[29] C. Bell, D. Bonachea, R. Nishtala, and K. Yelick, "Optimizing bandwidth limited problems using one-sided communication and overlap," in *20th International Parallel and Distributed Processing Symposium, IPDPS*, April 2006, pp. 10 pp.–.

[30] R. Nishtala, P. H. Hargrove, D. O. Bonachea, and K. A. Yelick, "Scaling communication-intensive applications on BlueGene/P using one-sided communication and overlap," in *IEEE International Symposium on Parallel and Distributed Processing, IPDPS*. IEEE, 2009, pp. 1–12.

[31] E. Kayraklioglu and T. El-Ghazawi, "Assessing memory access performance of chapel through synthetic benchmarks," in *15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. IEEE, 2015, pp. 1147–1150.

[32] T. El-Ghazawi, F. Cantonnet, Y. Yao, S. Annareddy, and A. S. Mohamed, "Benchmarking parallel compilers: a UPC case study," *Future Gener. Comput. Syst.*, vol. 22, no. 7, pp. 764–775, 2006.

[33] F. Cantonnet, T. El-Ghazawi, P. Lorenz, and J. Gaber, "Fast address translation techniques for distributed shared memory compilers," in *Proceedings of the 19th International Parallel and Distributed Processing Symposium*, 2005.

[34] W.-Y. Chen, D. Bonachea, J. Duell, P. Husbands, C. Iancu, and K. Yelick, "A Performance Analysis of the Berkeley UPC Compiler," in *Proceedings of the 17th annual international conference on Supercomputing: June 23-26*, vol. 4, Association for Computing Machinery. ACM, 2003, pp. 63–73.

[35] W. Zhao and Z. Wang, "ScaleUPC: a UPC compiler for multi-core systems," in *PGAS '09: Proceedings of the Third Conference on Partitioned Global Address Space Programming Models*. New York, NY, USA: ACM, 2009, pp. 1–8.

[36] M. Alvanosl, J. N. Amaral, E. Tiotto, M. Farreras, and X. Martorell, "Reducing compiler-inserted instrumentation in Unified-Parallel-C code generation," in *2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE, 2014, pp. 270–277.

[37] M. Farreras, G. Almási, C. Cascaval, and T. Cortes, "Scalable RDMA performance in PGAS languages," in *IEEE International Symposium on Parallel and Distributed Processing, IPDPS*. IEEE, 2009, pp. 1–12.

[38] B. Dalton, G. Tanase, M. Alvanos, G. Almasi, and E. Tiotto, "Memory management techniques for exploiting rdma in pgas languages," in *Workshop on Languages and Compilers and Parallel Computing (LCPC)*, 2014.

[39] R. H. Arpaci, D. E. Culler, A. Krishnamurthy, S. G. Steinberg, and K. Yelick, "Empirical evaluation of the CRAY-T3D: A compiler perspective," in *ACM SIGARCH Computer Architecture News*, vol. 23, no. 2. ACM, 1995, pp. 320–331.

[40] H. Fröning and H. Litz, "Efficient hardware support for the Partitioned Global Address Space," in *IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pr 2010, pp. 1–6.

[41] S. L. Scott, "Synchronization and communication in the T3E multiprocessor," in *ACM SIGPLAN Notices*, vol. 31, no. 9. ACM, 1996, pp. 26–36.

[42] M. M. Mueller, "Efficient address translation," *Interner Bericht. Universität Karlsruhe, Fakultät für Informatik; 2000, 12*, 2000.

[43] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren, "Introduction to UPC and language specification," Center for Computing Sciences, Institute for Defense Analyses, Tech. Rep., 1999.

[44] O. Serres, A. Kayi, A. Anbar, and T. El-Ghazawi, "Enabling PGAS productivity with hardware support for shared address mapping: A UPC case study," *ACM Transactions on Architecture and Code Optimization*, vol. 0, no. 0, 0 2015.

[45] ——, "Enabling PGAS productivity with hardware support for shared address mapping; a UPC case study," in *HPCC 2014: the 16th IEEE International Conference on High Performance and Communications*, 2014.

[46] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, August 2011. [Online]. Available: http://doi.acm.org/10.1145/2024716.2024718

[47] A. Butko, R. Garibotti, L. Ost, and G. Sassatelli, "Accuracy evaluation of gem5 simulator system," in *7th International Workshop on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*. IEEE, 2012, pp. 1–7.

[48] D. H. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow, "The NAS parallel benchmarks 2.0", nas-95-020," Moffett Field, CA, USA, Tech. Rep., 1995.

[49] T. El-Ghazawi and F. Cantonnet, "UPC performance and potential: A NPB experimental study," in *Proceedings of the ACM/IEEE conference on Supercomputing*. IEEE Computer Society Press Los Alamitos, CA, USA, 2002, pp. 1–26.

[50] UPC NPB, "UPC NAS Parallel Benchmarks threads.seas.gwu.edu/sites/npb-upc," 2014.

[51] T. El-Ghazawi and S. Chauvin, "UPC benchmarking issues," in *International Conference on Parallel Processing (ICPP)*. IEEE, 2001, pp. 365–372.

[52] O. Serres, V. K. Narayana, and T. El-Ghazawi, "An architecture for reconfigurable multi-core explorations," in *proceedings of the International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, 2011.

[53] R. Razdan and M. Smith, "A high-performance microarchitecture with hardware-programmable functional units," in *Proceedings of the 27th Annual International Symposium on Microarchitecture, MICRO-27.*, November 1994, pp. 172–180.

[54] R. Wittig and P. Chow, "OneChip: an FPGA processor with reconfigurable logic," in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, April 1996, pp. 126–135.

[55] Z. Ye, A. Moshovos, S. Hauck, and P. Banerjee, "CHIMAERA: a high-performance architecture with a tightly-coupled reconfigurable functional unit," in *Proceedings of the 27th International Symposium on Computer Architecture*, June 2000, pp. 225–235.

[56] D. Gottlieb, J. Cook, J. Walstrom, S. Ferrera, C.-W. Wang, and N. Carter, "Clustered programmable-reconfigurable processors," in *Proceeding of the IEEE International Conference on Field-Programmable Technology (FPT)*, December 2002, pp. 134–141.

[57] L. Yan, B. Wu, Y. Wen, S. Zhang, and T. Chen, "A reconfigurable processor architecture combining multi-core and reconfigurable processing unit," in *Proceedings of the 10th IEEE International Conference on Computer and Information Technology (CIT)*, July 2010, pp. 2897–2902.

[58] T.-O. Kwok and Y.-K. Kwok, "On the design, control, and use of a reconfigurable heterogeneous multi-core system-on-a-chip," in *IEEE International Symposium on Parallel and Distributed Processing, IPDPS*, pr 2008, pp. 1–11.

[59] E. Matthews, L. Shannon, and A. Fedorova, "Polyblaze: From one to many bringing the Microblaze into the multicore era with Linux SMP support," in *22nd International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2012, pp. 224–230.

[60] F. Barat and R. Lauwereins, "Reconfigurable instruction set processors: a survey," in *Proceedings of the 11th International Workshop on Rapid System Prototyping, RSP*, 2000, pp. 168–173.

[61] Gaisler Research, "GRLIB IP library user's manual, version 1.1.0 b4100," 2010.

[62] P. Guironnet De Massas and P. Amblard, "Experiments around SPARC Leon-2 for MPEG encoding," in *Proceedings of the International Conference on Mixed Design of Integrated Circuits and System (MIXDES)*, June 2006, pp. 285–289.

[63] M. Danek, L. Kafka, L. Kohout, and J. Sykora, "Instruction set extensions for multi-threading in LEON3," in *2010 IEEE 13th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*. IEEE, 2010, pp. 237–242.

[64] G. Cardarilli, L. Di Nunzio, R. Fazzolari, and M. Re, "Algorithm acceleration on LEON-2 processor using a reconfigurable bit manipulation unit," in *8th Workshop on Intelligent Solutions in Embedded Systems (WISES)*, July 2010, pp. 6–11.

[65] R. Garner *et al.*, "The SPARC architecture manual: Version 8," 1992.

[66] Xilinx Virtex-6 Family Overview (DS150, v1.2), "http://www.xilinx.com/support/documentation/data_sheets/ds150.pdf."

[67] E. Stavinov, *100 Power Tips for FPGA Designers*. CreateSpace Independent Publishing Platform, 2011.

[68] Transistor count, Wikipedia, retrieved 2015-12-01, "https://en.wikipedia.org/wiki/Transistor_count."

[69] C. Barton, C. Caşcaval, G. Almási, R. Garg, J. N. Amaral, and M. Farreras, "Multi-dimensional blocking in UPC." Berlin, Heidelberg: Springer-Verlag, October 2007, pp. 47–62.