An Automated OpenCL FPGA Compilation Framework Targeting a Configurable, VLIW Chip Multiprocessor

by
Samuel Jon Parker

A Doctoral Thesis

Submitted in partial fulfilment of the requirements for the award of
Doctor of Philosophy of Loughborough University

© Samuel Jon Parker 2015

August 2015

Abstract

Modern system-on-chips augment their baseline CPU with coprocessors and accelerators to increase overall computational capacity and power efficiency, and thus have evolved into heterogeneous systems. Several languages have been developed to enable this paradigm shift, including CUDA and OpenCL. This thesis discusses a unified compilation environment to enable heterogeneous system design through the use of OpenCL and a customised VLIW chip multiprocessor (CMP) architecture, known as the LE1. An LLVM compilation framework was researched and a prototype developed to enable the execution of OpenCL applications on the LE1 CPU. The framework fully automates the compilation flow and supports work-item coalescing to better utilise the CPU cores and alleviate the effects of thread divergence. This thesis discusses in detail both the software stack and target hardware architecture and evaluates the scalability of the proposed framework on a highly precise cycle-accurate simulator. This is achieved through the execution of 12 benchmarks across 240 different machine configurations, as well as further results utilising an incomplete development branch of the compiler. It is shown that the problems generally scale well with the LE1 architecture, up to eight cores, when the memory system becomes a serious bottleneck. Results demonstrate superlinear performance on certain benchmarks (x9 for the bitonic sort benchmark with 8 dual-issue cores) with further improvements from compiler optimisations (x14 for bitonic with the same configuration).

Acknowledgements

Firstly I would like to thank my parents and grandparents for all their love and support (mentally, physically and financially) as I would have achieved nothing without you. You always nurtured my will to *'vestigate* and this has definitely been my biggest investigation to date!

My thanks also goes to my supervisors Dr. Vassilios Chouliaras and Dr. David Mulvaney for their guidance through my PhD and tuition during my undergraduate studies. Especially to Dr. Chouliaras whose undergradute project on his VLIW architecture gave me the opportunity to begin my adventures into compiler technology which continued into this research, and for his continued encouragement, guidance and patience. My gratitude also goes to Dr. Stevens who developed, and promptly provided bug fixes for, the simulator and assembler that I used to collect experimental data; and for his help and advice, particularly at the start of this research.

I would also like to thank the School of Electronic, Electrical and Systems Engineering at Loughborough for the opportunity and funding which made my PhD possible.

Finally I would like to thank the hundreds of developers that have worked on open source tools that I have used throughout this PhD, particularly to those who have contributed to the linux kernel, Debian, Ubuntu, Vim, Git, GDB and Clang/LLVM. Special thanks for anyone who has offered help and suggestions on the cfe-dev and llvm-dev mailing lists.

Contents

Abstract **i**

Acknowledgements **i**

List of Figures **vi**

List of Tables **xii**

Glossary **xiii**

1 Introduction **1**

1.1 Problem Formulation 1
1.2 Aims and Objectives 2
1.3 Outline of Areas of Research 3
1.3.1 Parallelism 3
1.3.2 Compilers 5
1.3.3 Heterogeneous Computing 5
1.4 Contributions of Thesis 6
1.5 Thesis Outline 7

2 Parallel Computer Architectures **8**

2.1 Chapter Objectives 8
2.2 Architecture Styles and Features 8
2.2.1 Memory Hierachy 9
2.2.2 Superscalar Execution 10
2.2.3 Vector Processing 11
2.2.4 Multiple Threads 13
2.3 Very Long Instruction Word Architecture 18
2.3.1 Philosophy 18

2.3.2 History and Evolution . . . 19
2.3.3 Modern Implementations and Applications . . . 23
2.4 Heterogeneous Computing Architectures . . . 27
2.4.1 Graphics Processing Units . . . 28
2.4.2 Many-core Accelerators . . . 30
2.4.3 Field Programmable Gate Arrays . . . 32
2.5 Hardware / Software Codesign . . . 35
2.5.1 ASICs and FPGAs . . . 35
2.5.2 Application-Specific Instruction-set Processors . . . 36
2.5.3 Codesign for Supercomputers . . . 37
2.6 Discussion . . . 38
2.7 Summary . . . 40

3 Parallel Programming **41**
3.1 Chapter Objectives . . . 41
3.2 Compiler Overview . . . 41
3.2.1 Code Representation . . . 42
3.2.2 Code Generation . . . 43
3.2.3 Types of Compilation . . . 45
3.3 Compiling for VLIWs . . . 47
3.3.1 Region Formation . . . 47
3.3.2 Scheduling . . . 52
3.4 Languages and Platforms . . . 56
3.4.1 Traditional Methods . . . 56
3.4.2 Programming for Heterogeneous Systems . . . 58
3.4.3 OpenCL Compilation . . . 63
3.5 Discussion . . . 68
3.6 Summary . . . 69

4 Identification of Research Area **70**
4.1 Chapter Objectives . . . 70
4.2 Compiling for VLIWs . . . 70
4.3 FPGA OpenCL Compilation . . . 71
4.4 Summary . . . 72

5 The LE1 Compiler **74**
5.1 Chapter Objectives . . . 74
5.2 Introduction to Clang and LLVM . . . 74

5.2.1 Program Representation . . . 75
5.2.2 TableGen . . . 77
5.3 Instruction Selection . . . 77
5.3.1 Instruction Legalisation and Lowering . . . 83
5.3.2 Instruction Description . . . 86
5.4 Instruction Scheduling . . . 91
5.4.1 Stable Branch . . . 91
5.4.2 Development Branch . . . 92
5.5 Register Allocation . . . 94
5.5.1 Register Files . . . 94
5.6 Instruction Packing . . . 95
5.6.1 Stable Branch . . . 96
5.6.2 Development Branch . . . 97
5.7 Code Emission . . . 99
5.8 Contributions . . . 100
5.9 Summary . . . 100

6 The LE1 OpenCL Driver 101

6.1 Chapter Objectives . . . 101
6.2 Client Driver . . . 101
6.2.1 API . . . 103
6.2.2 Core . . . 103
6.2.3 Device . . . 105
6.3 Source-to-Source Transformation . . . 106
6.4 Runtime Support . . . 111
6.4.1 Builtin Functions . . . 111
6.4.2 Softfloat . . . 111
6.4.3 Kernel Launching . . . 112
6.4.4 Data Transfer . . . 113
6.4.5 Simulation . . . 115
6.5 Contributions . . . 115
6.6 Summary . . . 116

7 Experiments 117

7.1 Chapter Objectives . . . 117
7.1.1 Validation . . . 117
7.1.2 ILP performance . . . 118
7.1.3 TLP performance . . . 118

7.2 Benchmarks . . . 118
7.3 Machine Configurations . . . 120
7.4 Results . . . 121
7.4.1 ILP Performance . . . 121
7.4.2 TLP Performance . . . 132
7.4.3 Development Branch . . . 141
7.5 Summary . . . 154

8 Conclusion 157
8.1 Chapter Objectives . . . 157
8.2 Summary of Thesis Objectives . . . 157
8.3 Contributions . . . 158
8.4 Findings . . . 160
8.4.1 Single Program, Multiple Data . . . 160
8.4.2 Very Long Instruction Word Processor . . . 160
8.4.3 Towards Heterogeneous Computing . . . 161
8.5 Limitations of Research . . . 161
8.6 Further Research . . . 162

References 164

Publications 181

A Source Code Repository 183

B Compiler and Simulator Target Generator Script 184

C Results 190

D Complete Results from Experiments 204

List of Figures

2.1 Memory hierachy in a modern smartphone. 9
2.2 Vector pipeline and register file with four lanes. 12
2.3 Power and performance ratio between doubling the cores and doubling the frequency. 13
2.4 Cache Hierachy in a chip-multiprocessor. 15
2.5 Processor utilisation with 4-way (a) fine-grained (*vertical*) multithreading and (b) simultaneous multithreading (*horizontal*), with the four threads in different colours. 16
2.6 Components and pipeline of a 4-wide, single-core LE1. 26
2.7 Quad-core LE1 with connecting memory system. 27
2.8 Internal components of an FPGA. 33

3.1 Example of *Single Static Assignment* (SSA) form. 43
3.2 Extended Basic Block Formation with the EBBs shown within the dashed lines. 49
3.3 Illustrated concept of software pipelining. 54

5.1 Three phase compilation with Clang and LLVM. 75
5.2 Graphical example of an AST produced by Clang. 76
5.3 Example of a SelectionDAG after target lowering and legalisation. 81
5.4 Code that informs the legalisation process which types are supported by each operation and in which registers the types can reside. 83
5.5 Calling convention definitions for the LE1. 84
5.6 SDNode definition for ADDCG and DIVS instructions. 85
5.7 Base class for instruction formats with arithmetic and logic operations definitions. 87
5.8 Instruction class and definition used for compare operations. 88
5.9 Pattern matching of conditional branch instructions. 88
5.10 Example of pattern matching for an unsigned value. 88

5.11 Multiplication instruction definitions and pattern matchers. 90
5.12 Processor itineraries which describe pipeline usage for each type of instruction. 92
5.13 Microarchitecture resource description and assignment to instruction types for use by the scheduler, in the updated compiler based on LLVM 3.4 93
5.14 Internal representation of LLVM MachineInstr with Bundles. 96
5.15 Class interactive for LE1 code emission. 99

6.1 Software system overview. 102
6.2 Driver target instantiation example. 103
6.3 Simplified relationship between the API and the objects in the ‘core’ and the ‘device’. 104
6.4 Simple kernel coarsening algorithm. 107
6.5 Algorithm outline for the second, and final, stage of kernel coarsening. 109
6.6 Permute source code after complete transformation. 110
6.7 Kernel launcher source code for Binary Search. 113
6.8 Workgroup execution model on a 4 core system, with the workgroups numbered and the different varying colours representing the separate cores. 114

7.1 Loop header for target machine generator script. 120
7.2 Total average stalls and NOP cycles for BFS_1 using 1 context across varying microarchitecture configurations. 122
7.3 Total average cycle count for BFS_2 using 1 context across varying microarchitecture configurations. 123
7.4 Total average stalls and NOP cycles for BFS_2 using 1 context across varying microarchitecture configurations. 123
7.5 Total cycle count for MatrixTranspose using 1 context across varying microarchitecture configurations. 126
7.6 Total stall and NOP cycle count for MatrixTranspose using 1 context across varying microarchitecture configurations. 126
7.7 Total average cycle count for nw_kernel1 from Needleman-Wunsch using 1 context across varying microarchitecture configurations. 128
7.8 Total average cycle count for nw_kernel2 from Needleman-Wunsch using 1 context across varying microarchitecture configurations. 128
7.9 Total average cycle count for FixOffset from Radix Sort using 1 context across varying microarchitecture configurations. 129
7.10 Total average stalls and NOP cycle count for FixOffset from Radix Sort using 1 context across varying microarchitecture configurations. 130

7.11 Total average cycle count for histogram from Radix Sort using 1 context across varying microarchitecture configurations. 130
7.12 Total cycles for Reduction using 1 context across varying microarchitecture configurations. 131
7.13 Total average stalls and NOP cycle count for Reduction using 1 context across varying microarchitecture configurations. 132
7.14 Speedup of Breadth-First Search, Binary Search and Bitonic Sort across a selection of multi-context, maximal microarchitecture, configurations. 133
7.15 Speedup of Fast Walsh Transform, Floyd Warshall and Gaussian Elimination across a selection of multi-context, maximal microarchitecture, configurations. 134
7.16 Speedup of Matrix Transpose, NBody Simulation, Nearest Neighbour and Needleman-Wunsch across a selection of multi-context, maximal microarchitecture, configurations. 135
7.17 Speedup of the kernels of Radix Sort and Reduction across a selection of multi-context, maximal microarchitecture, configurations. 137
7.18 Total stalls and NOPs of FixOffset, from Radix Sort, across maximal microarchitecture systems. 138
7.19 Speedup of a 2W-2A-1M-1L-4B LE1 system with two contexts using silicon data. 139
7.20 Speedup of a 2W-2A-1M-1L-4B LE1 system with four contexts using silicon data. 140
7.21 Speedup of a 2W-2A-1M-1L-4B LE1 system with six contexts using silicon data. 140
7.22 Speedup of a 2W-2A-1M-1L-4B LE1 system with eight contexts using silicon data. 141
7.23 Speedup of a 2W-2A-1M-1L-4B LE1 system with ten contexts using silicon data. 141
7.24 Total cycles of BinarySearch, on the single CU devices, using both compilers and optimisations. 142
7.25 Total IF stall cycles of BinarySearch, on the single CU devices, using both compilers and optimisations. 143
7.26 Total memory stall cycles of BinarySearch, on the single CU devices, using both compilers. 144
7.27 Total NOP cycles of BinarySearch, on the single CU devices, using both compilers. 144
7.28 Speedup of BinarySearch, relative to a single CU ,scalar device, and LLVM 3.2, across the maximal microarchitecture configurations. 145

7.29 Total average cycles of Bitonic Sort, on the single CU devices, using both compilers and varying optimisations . . . 146
7.30 IF stalls of Bitonic Sort. . . . 147
7.31 Memory stalls of Bitonic Sort across maximal microarchictectures with different compiler backends. . . . 147
7.32 Speedup of Bitonic Sort, relative to a scalar device with LLVM 3.2, across multi-context, maximal microarchitecture, devices, using both compilers and varying optimisations. . . . 148
7.33 Total average cycles for FloydWarshall using the different compilers and optimisations, for the single CU devices. . . . 149
7.34 Total average IF stalls of FloydWarshall for the single CU devices, using both the compiler backends. . . . 150
7.35 Total average memory stalls of FloydWarshall for the single CU devices, using both the compiler backends. . . . 150
7.36 Speedup, with respect to a scalar device and LLVM 3.2, for FloydWarshall using different compilers and optimisations across multi-CU systems of maximal microarchitecture configurations . . . 151
7.37 Total average cycles for Reduction using LLVM 3.2 and loop unrolling. . . . 152
7.38 Total average IF stall cycles for Reduction on the single CU devices using both compilers and optimisations. . . . 152
7.39 Total average memory stall cycles for Reduction on the single CU devices using both compilers and optimisations. . . . 153
7.40 Speedup, relative to a scalar device with LLVM 3.2, of Reduction on the single CU devices with maximal microarchitectures using both compilers and optimisations. . . . 153

C.1 Total cycles for BinarySearch using 1 context across varying microarchitecture configurations. . . . 190
C.2 Total IF stall cycles for BinarySearch using 1 context across varying microarchitecture configurations. . . . 191
C.3 Total average cycle count for BitonicSort using 1 context across varying microarchitecture configurations. . . . 191
C.4 Total average IF stall cycle count for BitonicSort using 1 context across varying microarchitecture configurations. . . . 192
C.5 Total average cycle count for BFS_1 using 1 context across varying microarchitecture configurations. . . . 192

C.6 Total cycles for FastWalshTransform using 1 context across varying microarchitecture configurations. 193
C.7 Total stalls and NOP cycles for FastWalshTransform using 1 context across varying microarchitecture configurations. 193
C.8 Total cycles for FloydWarshall using 1 context across varying microarchitecture configurations. 194
C.9 Total stall and NOP cycles for FloydWarshall using 1 context across varying microarchitecture configurations. 194
C.10 Total average cycles for Fan1 of Gaussian Elimination using 1 context across varying microarchitecture configurations. 195
C.11 Total average stalls and NOP cycles for Fan1 of Gaussian Elimination using 1 context across varying microarchitecture configurations. 195
C.12 Total average cycles for Fan2 of Gaussian Elimination using 1 context across varying microarchitecture configurations. 196
C.13 Total average stalls and NOP cycles for Fan2 of Gaussian Elimination using 1 context across varying microarchitecture configurations. 196
C.14 Total cycle count for NBody using 1 context across varying microarchitecture configurations. 197
C.15 Total stall and NOP count for NBody using 1 context across varying microarchitecture configurations. 197
C.16 Total cycle count for Nearest Neighbour using 1 context across varying microarchitecture configurations. 198
C.17 Total stall and NOP cycle count for Nearest Neighbour using 1 context across varying microarchitecture configurations. 198
C.18 Total average stall and NOP cycle count for nw_kernel1 from Needleman-Wunsch using 1 context across varying microarchitecture configurations. 199
C.19 Total average stall and NOP cycle count for nw_kernel2 from Needleman-Wunsch using 1 context across varying microarchitecture configurations. 199
C.20 Total average stalls and NOP cycle count for histogram from Radix Sort using 1 context across varying microarchitecture configurations. 200
C.21 Total average cycle count for permute from Radix Sort using 1 context across varying microarchitecture configurations. 200
C.22 Total average stalls and NOP cycle count for permute from Radix Sort using 1 context across varying microarchitecture configurations. 201
C.23 Total average cycle count for ScanArraysdim1 from Radix Sort using 1 context across varying microarchitecture configurations. 201

C.24 Total average stalls and NOP cycle count for ScanArraysdim1 from Radix Sort using 1 context across varying microarchitecture configurations. 202
C.25 Total average cycle count for ScanArraysdim2 from Radix Sort using 1 context across varying microarchitecture configurations. 202
C.26 Total average stalls and NOP cycle count for ScanArraysdim2 from Radix Sort using 1 context across varying microarchitecture configurations. 203

List of Tables

2.1 Architecture Comparison between NVIDIA Kepler and Maxwell. 29
2.2 Feature Set of Xilinx Virtex UltraScale Product Series 34

3.1 Modulo scheduling example 55
3.2 CUDA and OpenCL naming conventions 60

5.1 LE1 Arithmetic Operations. 79
5.2 LE1 Control Operations. 80
5.3 LE1 Memory Operations. 80
5.4 LE1 Multiplication Operations. 80
5.5 LE1 Logical Operations. 82
5.6 LE1 register usage. 95

7.1 Kernel work dimensions, sizes and execution iterations from AMD AMP. . 119
7.2 Kernel work dimensions, sizes and execution iterations from Rodinia. . . . 119
7.3 LE1 silicon data. 139
7.4 Average IRAM sizes across all configurations and compilers. 142

Chapter 1

Introduction

1.1 Problem Formulation

For several decades, since the inception of integrated circuits, transistors continued to shrink exponentially while maintaining the same power density [1][2]. This enabled reductions in power requirements, as the voltage and current demands both decreased with area, and also enabled higher clock frequencies. These two characteristics of shrinking transistors and maintained power density are respectively known as Moore's Law and Dennard scaling. Power in a digital circuit is composed of both its *static* and *dynamic* power. Static power is the energy that is wasted while trying to maintain state between switching states, and is caused by leakage current due to thermal excitations and quantum tunnelling. Where C is capacitance, V_{dd} is the the supply voltage and f being operating frequency, dynamic power (P_d) is given by [3]:

$$P_d = CV_{dd}^2 f \tag{1.1}$$

As transistors shrank, capacitance also diminished along with the supply voltage; allowing faster operating frequencies. With higher clock frequencies and more integrated functionality, processing capacity also increased exponentially with the reducing transistor size. For the software engineer, performance increase for their codes came for free, as microprocessors starting executing more simultaneous operations and at an increased rate. This type of parallelism is called *instruction-level parallelism* (ILP) and is utilised by having multiple functional units and usually some complex scheduling hardware. Segmenting the functional units into stages, called *pipelining*, allows for further ILP as multiple instructions can occupy different stages of the pipeline. This also allows for higher clock frequencies. However, shrinking transistor size also led to an increase in static power as more leakage

current was induced; as such Dennard scaling stopped around 2005 [4]. It has been suggested that high-performance scaling for double-gate FinFET designs would cease between 12 and 15nm, and both IBM and Intel are now releasing tri-gate FinFET microprocessors at 14nm [5].

With the possible physical properties of silicon being realised, computer architectures have had to scale up performance in different ways, rather than relying on faster operating frequencies and ILP exploitation of uniprocessors. The initial solution was to instantiate multiple processing cores on the silicon and/or have the cores running multiple threads of execution; today this is the ubiquitous paradigm of computing. The exploitation of this type of *thread-level parallelism* (TLP), does however require effort from the software developer and requires new languages, or extentions to traditional ones. As power densities have risen, new techniques have been required to keep microprocessors cool enough to operate. This power requirement can be maintained by using different power states, where the microprocessor operates at different frequencies and voltages as well as powering down parts of the device when unneeded. This allows a device to reduce operating power when less work is required, but also to use short bursts of higher power to meet performance requirements while staying within the power envelope, termed *thermal dynamic power* (TDP). Switching off parts of the device leads to '*dark silicon*' because though the device has the silicon capacity, it doesn't have the capacity to dissipate the heat, which utimately limits the capabilities [6].

The latest paradigm shift to emerge is the use of coprocessors to offload computation away from the host CPU. This type of execution has been developed because most common programs are still dependent on single-threaded performance, and so common CPUs are designed to support those applications. However, some computationally intensive codes are highly parallel and these types of applications are becoming more popular; as the world generates and processes more data. So modern computers need to accelerate in both single- and multi-threaded workloads. Coprocessors are generally comprised of many, simple, cores and are designed as throughput devices, but this has required algorithms to be re-implemented for the vastly different architectures. The algorithms generally use *divide-and-conquer* techniques to split the work across those cores and explicitly identify both thread- and data-level parallelism. A key problem with this paradigm is that data has to be transferred between the host and the accelerator, but this is being addressed through modifications to language standards and the inception of the Heterogeneous System Architecture (HSA) [7].

1.2 Aims and Objectives

The aims of the research conducted are:

- Enable OpenCL compilation for custom VLIW Chip-Multiprocessor (CMP).

- Enable execution of OpenCL kernels on a FPGA platform.
- Extensively benchmark the CPU as this has not been previously performed.

Objectives:

- Implement C compiler backend for unique VLIW CMP.
- Investigate parallel computing techniques.
- Develop an OpenCL library to support parallel computing on the VLIW CMP.
- Investigate compiler performance across varying microarchitecture configurations of the configurable VLIW CMP.
- Investigate whole system performance of the VLIW CMP.

The objectives are based around creating a toolchain to enable the execution of explicitly parallel programs upon a unique, and configurable, VLIW CMP. The majority of parallel languages can be accessed using the C programming language and they also generally require some form of runtime support. This requires that not only does the high-level language need to be compiled for the target, but an execution framework also needs to be developed to fully enable the programs to run.

1.3 Outline of Areas of Research

This section aims to give the reader an overview of the main areas of research where work was conducted through this thesis.

1.3.1 Parallelism

Exploiting parallelism in code is the fundamental technique for modern microprocessors to achieve their required performance. The three types of parallelism are instruction-level, data-level and thread-level and most modern microprocessors will address all these levels of parallelism. In the 1960s, Flynn coarsely categorised computer architectures, which still helps describe the type of parallelism that they exploit [8]. The three key categories are:

- SISD - *single instruction stream, single data stream* which are capable of extracting instruction-level parallelism from independent instructions within a single stream.
- SIMD - *single instruction stream, multiple data stream* where a single instruction will operate on multiple data elements simultaneously as the data elements are parallel.

- MIMD - *multiple instruction stream, multiple data stream* computers describe the execution of multiple programs, processes or threads that operate on independent data.

Another important execution model that has become particularly popular recently is single program, multiple data (SPMD) which utilises a single instruction stream but operates on multiple data elements from different threads. This type of execution model is commonly used in graphic processing units (GPUs) and will be discussed in Section 2.4.1. The use of GPUs for general purpose programming has also resulted in the introduction of new programming languages and application programmable interfaces (APIs); the next subsection will discuss both traditional and more recent methods of parallel programming.

Instruction-Level Parallelism (ILP) describes that some instructions will be independent of one another and so can execute in parallel. This can be achieved in hardware by including multiple functional units for the instructions to occupy concurrently and by also pipelining the microprocessor to have multiple instructions in-flight, but at different stages. This type of parallelism is implicit, the exploitation is the task of the compiler and/or the microprocessor, so no effort from the programmer is needed. Microprocessors that rely on the ILP discovery at compile-time are termed *Very Long Instruction Word* (VLIWs), and are discussed in Section 2.3. Dynamically scheduled ILP processors are called *superscalars*, which are discussed in Section 2.2.2.

Data-Level Parallelism (DLP) arises when multiple data elements can be operated independently on at the same time. Vector supercomputers exploit DLP, as do microprocessors that have SIMD instructions in their ISA and these will be discussed in Section 2.2.3. This type of parallelism often needs to be identified by the programmer and they also usually have to design their program for the underlying architecture. DLP can be found via the compiler, although auto-vectorisation techniques are an ongoing research topic [9].

Thread-Level Parallelism (TLP) has become particularly important in the last decade, as uniprocessor clock rates plateaued and the limits of ILP had also been discovered. TLP allows programs to be split into independent parts, to be operated on by multiple processors. TLP could also describe task-level parallelism in which multiple independent programs, or processes, can operate simultaneously on different data streams. This type of parallelism is also explictly identified by the programmer, and a key role of an operating system is to handle multiple processes. Multiprocessors and multithreading will be discussed in Section 2.2.4.

1.3.2 Compilers

Computer programs can be written in three forms: machine code (binary), the architecture's assembly language, or a higher level language such as C or Java. Writing machine code is only necessary in the absence of an assembler, which is very uncommon. High-performance, target specific codes and libraries, are often still written in assembly language [10][11]. However, most programs are written in higher level languages since they are more portable, quicker to write and easier to debug. It is the job of the compiler to translate higher level languages, via a series of stages or phases, into object code or machine language. Any basic compiler needs to have at least four phases [12]:

- lexical analysis which analyses the character strings presented to it and checks whether they are valid for the language,
- syntactic parsing which produces an intermediate-level representation and a symbol table of identifiers,
- static-semantic validity which checks that the intermediate code satisfies source language properties,
- the code generator which transforms the intermediate code into machine language.

Another phase that is essential to any modern compiler is the optimisation phase, and so a compiler can be logically split into three parts: the front-end, the high-level optimiser and the back-end. The front-end is language specific and organises the human-readable form into an intermediate representation (IR) and performs some language-specific optimisations. The high-level optimiser is responsible for machine-independent code transformations, including loop-level transformations, constant propagation and dead code elimination. The back-end is machine-specific and translates the IR into machine code and can perform machine-specific optimisations.

1.3.3 Heterogeneous Computing

Heterogeneous computing describes computer systems that comprise of two or more different computer architectures. These have been introduced as single-threaded performance improvements have slowed dramatically, with the failing of Dennard scaling. As well as the failing of Dennard scaling, another recent change in the microprocessor ecosystem is the kind of computation commonly being performed, as well as the amount of data being processed. In the consumer domain, the greatest growth is in smart mobile devices where graphics and camera performance are key. The business and server segments are heavily biased towards

Big Data [13] whereas supercomputers are used for evermore complex simulations [14][15]. These three market segments, consumer (embedded), server and supercomputing, target not only processing system raw performance but, equally importantly, power consumption. In that respect, HPC vendors had already moved away from bespoke vector computers to commodity x86 parts, as these were cheaper and more efficient [16]. With the introduction of General-Purpose Graphics Programming Units GPGPUs and the release of the proprietary API CUDA from NVIDIA, a trend towards the universal use of Graphic Processing Units (GPUs) in all these market segments is emerging.

The Open Compute Language (OpenCL [17]) was proposed as an open standard API for general-purpose computing across CPUs, GPUs and other accelerators in response to CUDA's performance advantage on NVIDIA-only hardware. This was standardized by the Khronos Group and nowadays, OpenCL drivers are offered by all the major graphic processor designers such as AMD, Intel, and Qualcomm [18]. Unlike CUDA, OpenCL is target agnostic which has enabled the emergence of an OpenCL implementation ecosystem around not only GPUs but also CPUs and Field-Programmable Gate Arrays (FPGAs). Heterogeneous computing is used to create some of the most powerful and efficient supercomputers [19][20], but is also utilised in low-power devices, such as smartphones, through the use of OpenCL, CUDA and RenderScript [21].

1.4 Contributions of Thesis

The work presented in this thesis focuses on OpenCL compilation for a configurable VLIW. This has been achieved through the use of fully programmable, FPGA-based, configurable VLIW CMP as the target hardware platform. The developed toolchain implements a runtime driver that enables the programmer to compile and execute OpenCL kernels using the VLIW CMP as an accelerator. The main contributions made within this thesis are as follows:

Automatic Compilation Methodologies: A compiler backend for our in-house VLIW CMP has been implemented and incorporated into an OpenCL driver which has also been developed. This enables the programmer to select a target, from a multitude of system configurations, to execute the OpenCL kernels upon.

SPMD Methodologies: The compiler has been developed to contain unique intrinsics, accessed via a custom runtime library, to enable the execution of OpenCL kernels on the target. A method of statically scheduling work across multiple cores has also been developed.

Source Transformer: To enable execution of OpenCL programs on the VLIW CMP, it has been necessary to transform the kernel code. The source-to-source transformer has been implemented to perform this automatically and produces code that is generic, so could be used on other shared memory multiprocessors.

1.5 Thesis Outline

The remainder of this thesis is organised as follows:

- Chapter 2 is a detailed survey into how computer architectures exploit various forms of parallelism to increase single threaded execution speed, throughput and efficiency.
- Chapter 3 reviews the programming languages and software required to develop parallel applications on modern CPUs, accelerators and FPGAs.
- Chapter 4 summarises the background review and identifies the area of research that this thesis focuses on.
- Chapter 5 provides an indepth description of the compiler for the LE1.
- Chapter 6 describes the OpenCL driver for the LE1, for which the compiler is an essential component.
- Chapter 7 presents results and analysis from executing industry standard benchmarks, using the researched driver, on a cycle accurate simulator.
- Chapter 8 concludes this research and proposes possible extensions to the work.

Chapter 2

Parallel Computer Architectures

2.1 Chapter Objectives

The objective of this chapter is to give an overview of existing computer architectures and styles and how they exploit different types of parallelism to improve throughput and efficiency. The chapter also contains an in-depth history of the development of the VLIW architecture and how they are used today, with some modern architecture examples including the configurable VLIW CMP that the research in this thesis is based upon. The focus of modern architectures is still to continue improving throughput and execution, but also with a keen focus on power consumption and heat dissipation. Heterogenous computer architectures and hardware/software codesign are also introduced as these modern approaches aim to target both criteria.

2.2 Architecture Styles and Features

Modern CPUs still predominantly focus on improving single threaded performance and this is achieved through concurrently executing multiple instructions and reducing the number of stalls when accessing memory. Originally microprocessor designers aimed to achieve higher performance through the use of multiple, pipelined, functional units to better utilise the hardware to have multiple instructions in-flight. This section will introduce numerous methods, and implementations, for exploiting the various levels of parallelism identified in Section 1.3.1: with dynamic scheduling of multiple instructions, extensions to instruction sets to exploit DLP and the execution of multiple concurrent tasks.

2.2.1 Memory Hierachy

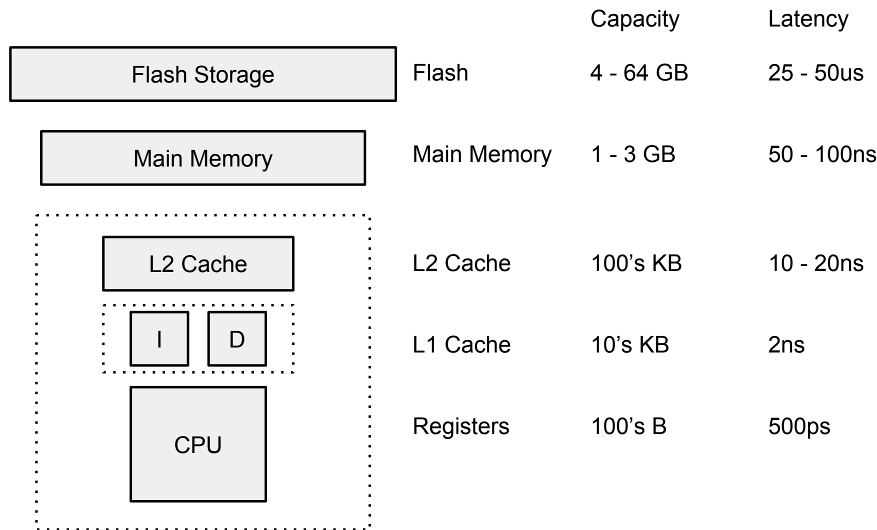

Figure 2.1: Memory hierachy in a modern smartphone.

As shrinking transistor sizes improved CPU performance exponentially, through faster clocks and more functional units, memory capacity also increased, however memory latency did not improve at the same rate as CPU frequency. This led to a significant performance bottleneck and so memory hierarchies were introduced into computer systems to help minimise the effect of slower memories. Figure 2.1 shows a typical, simplified, memory hierarchy of a modern smartphone, the added complexities of caches in multi-core systems will be further discussed in Section 2.2.4.1. The diagram shows that there are several levels, each getting larger, slower and cheaper the further away it is from the processor. CPU caches are implemented in SRAM as these are faster and require less power while DRAM is used as main memory because it is much more dense and cheap, but at the cost of speed and power requirements [22]. These characteristics lead to the tiered heirachy.

The purpose of the caches are to exploit spatial and temporal locality so that data that are likely to be needed by the processor, are closer to it. The latencies of the higher levels of memory can be hidden by keeping the required data in the nearest cache, and by keeping the lower caches small, the faster they can operate.

2.2.2 Superscalar Execution

In Sections 2.3 and 3.3, an in-depth introduction will be given into the methods used by VLIW architectures, and their compilers, to statically exploit ILP at compile time. ILP can also also be identified at runtime by the hardware and this type of processor is termed a dynamically scheduled superscalar. They too have multiple functional units, but also include additional hardware to resolve dependencies at runtime. Though this makes the hardware more complex, superscalars have the advantage of being able to react, and predict, runtime hazards such as branches and cache misses. There are two execution models for superscalars: in-order, where instructions are issued and executed in program order, and out-of-order in which instructions are fetched in-order but can execute and finish out of program order.

The original superscalar computer, in the 1960's, was the in-order CDC-6600 [23]. Superscalars remained exclusive to the supercomputer market until they were introduced to workstations in the early and mid-1990's with the IBM RS/6000 (POWER1)[24] and the MIPS R10000 [25], which were capable of dispatching four instructions per cycle. Both architectures supported superscalar execution by using separate functional units and associated register files, speculative instruction dispatch and *register renaming*. Register renaming requires that there are more physical registers than architected so as instructions are reordered, their values can be stored in other registers to maintain the correct read/write ordering. Renaming also requires a table to map architected registers to physical ones and a list or queue of free registers. Executing instructions out-of-order requires that all the statuses of the in-flight instructions are maintained during their lifetime and may require additional queue structures. Out-of-order Superscalars have remained the most popular architecture style, with Intel's current Haswell microarchitecture having a four wide in-order instructon fetch engine that can dispatch up to eight instructions per clock [26]. To utilise this width, complex branch predictors and loop detectors are required so that many instructions are available to the hardware scheduler.

However, superscalar architectures are now also implemented in low power and embedded systems too. The ARM Cortex-A15 is a high-performance, low-power, 32-bit microprocessor designed for smartphones. It has an out-of-order superscalar engine with pipelines that vary between 15-24 stages [27][28]. The instruction fetch unit can feed three instructions to the decode unit and also contains branch prediction hardware. The decode unit performs register renaming and also contains a loop buffer to reduce power consumption in small instruction loops. Three instructions are then dispatched to the *execution cluster* queues on each cycle. The clusters are split by functionality: simple integer ALU, floating-point / vector, branch, multiply / divide and load/store. The load/store cluster enables out-of-order loads, but they cannot bypass stores, while stores issue in order but only require the address source

to issue. The Cortex-A7 is functionally compatiable with A15, however it is designed to be more power efficient than the A15; it is an 8-stage, partial dual-issue, in order superscalar [29]. This difference results in the A15 operating at ∼6x higher dynamic power than the A7 and occupying 5x the area [30].

2.2.3 Vector Processing

Vector processors exploit data-level parallelism to speed up execution, by issuing single instructions that operate across multiple data elements. They are characterised by high bandwidth requirements, wide registers and a high-throughput but at the cost of a startup delay. Multiple pipelines can be incorporated to allow multiple instructions to be issued, and those pipelines can be split into lanes to operate on multiple data elements simultaneously to further increase IPC. Their well defined partitioning of the register file and pipelines, shown in Figure 2.2, reduces complexity and allows vector architectures to easily increase throughput.

Vector processors include registers that enable clean computation of varied length arrays and sophisticated memory accesses to increase the amount of code that can be vectorised [31]. Designated registers are used to:

- set the length of the vector to perform the calculation on,
- perform conditional operations across elements,
- set the stride length of a memory access to aid the vectorisation of multidimensional arrays,
- set an index vector for gather-scatter memory accesses.

Vector processors, especially from Cray and later NEC, represented the most powerful supercomputers from the 1970s through to the 1990s [32][33]. The size and cost of vector processors confined them to supercomputers, but as transistor sizes shrank and became cheaper, vector processing become viable for the wider market. Multimedia applications drove the inclusion of instruction set extensions for x86, from MMX [34] up to the current day AVX [35], and has seen adoption for all popular general-purpose architectures too. These instructions are generally termed as SIMD extensions.

The most basic of SIMD extensions, such as those included in the ARMv6 ISA [36] and Intel MMX [37], simply partitioned existing register and pipelines to operate on multiple, smaller data elements, as many graphic and audio applications operate on 8- or 16-bit data instead of the full 32- or 64-bit native width. Later extensions, such as PowerPC Altivec [38], Intel SSE [39] and ARMv7 NEON [40], added separate, larger, registers to support

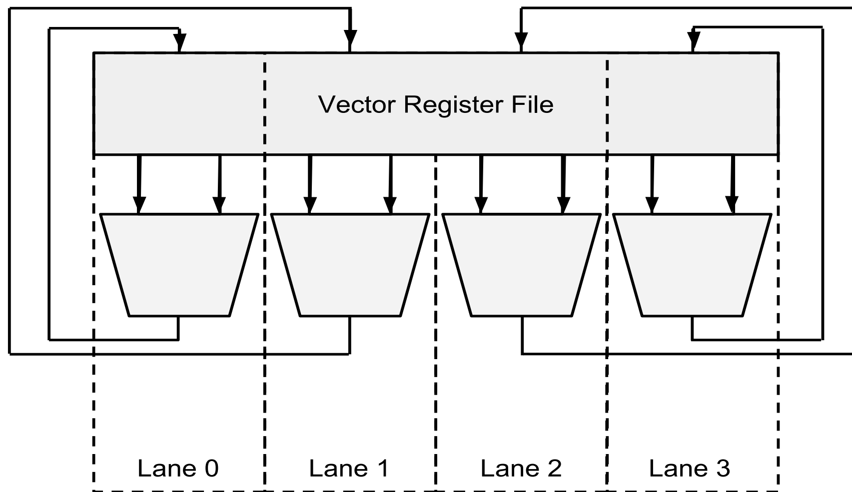

Figure 2.2: Vector pipeline and register file with four lanes.

SIMD instructions on larger data types, as well as increasing the number of operands for smaller element operations. Though all these extensions allow computation across multiple data elements, they do not support the memory accesses and conditional execution of vector processors, nor are they as flexible as the vector length usually encoded within the instruction. The latest AVX-512 from Intel does however support conditional execution across the elements using eight opmask registers.

The Advantage of vector (SIMD) processing comes from the ability to perform many calculations from a single instruction, which can help reduce power and hardware complexity while improving execution speed. The reduction in the number of instructions can reduce dynamic power as far less instructions need to be fetched and decoded, and can help improve instruction cache misses too. Power can be further reduced while maintaining the same throughput because the clock rate can be lowered as the number of lanes increases, this does however increase the die size. Their pipelines can be easily expanded by introducing new lanes and the register file can be kept simple, even with the increase in ports, by partitioning it for the dedicated lanes. These advantages make SIMD extensions a popular choice, particularly in contemporary architectures which target power restricted systems where the power requirement and silicon expense of wide, out-of-order processors is not viable. The downside of such extensions is that to use them, generally requires the programmer to use architecture specific libraries and/or compiler intrinsics which makes the code less portable.

2.2.4 Multiple Threads

Single-threaded performance has been viewed as a performance limiter since the 1960s, with machines such as the IBM System/360 which could come in a dual-socket configuration [41]. These computers were very large, and their speed was often constrained by their physical size due to the propagation delay along the connecting wires. Introducing another CPU into the computer saved space and power, due to most of the resources being shared, and it was also easier to double the performance this way than to design a new CPU to be twice as fast as the previous one. However, single threaded performance continued to increase greatly with the introduction of microprocessors while multi-socket computers were only used in servers and supercomputers, where many independent transactions would need to occur concurrently and algorithms could be written specifically for the underlying hardware. In the consumer and embedded markets, applications were generally written to use a single thread and the OS was used to time-multiplex tasks and processes.

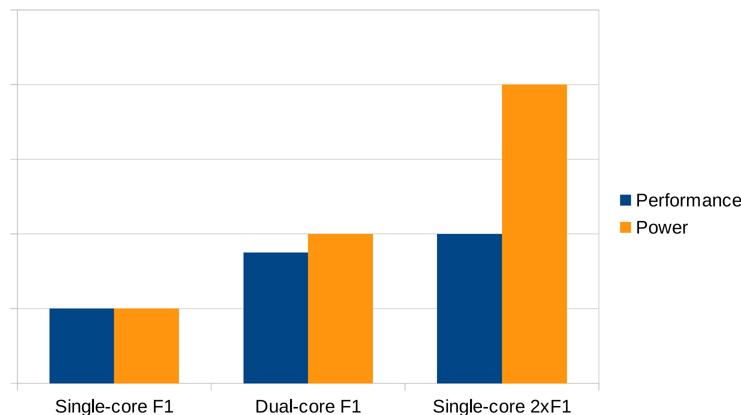

Figure 2.3: Power and performance ratio between doubling the cores and doubling the frequency.

Up to the early 2000's, the shrinking process sizes had enabled increased clock frequencies and the inclusion of extra hardware, such as on-chip L2 caches, without affecting the power density. This improvement, in both architecture and technology processes, allowed CPU performance to improve rapidly. But by the mid 2000's, the physical limitations were being discovered once again. This time it was the amount of heat being dissipated as Dennard scaling had stopped and the leakage current increased static power. The power dissipation problem was compounded by the increase in the dynamic power required by aggressive superscalars that continued to pursue increased clock frequencies. The diminishing returns of single-threaded ILP processors and their increased power requirements made them par-

ticularly unsuitable for embedded systems, which are typically limited by passive cooling. Figure 2.3 depicts the power and performance ratio of designs that double single core frequency and ones that double the number of cores and maintain the same frequency [42]. This is based on the assumption that doubling the frequency will half the execution time while paralellising the problem will incur some overhead. The large increase in the power required as the frequency is increased stems from the power scaling linearily with frequency. But this increase in power would also be worse as increasing clock frequencies also generally requires increasing the voltage, which in turns expontentially increase power dissipation.

2.2.4.1 Cache Coherence

As discussed in Section 2.2.1, modern computer architectures employ cache memories to counteract the speed differential between a processor and main memory. A typical CMP cache hierachy is shown in Figure 2.4, which leads to the issue of cache coherence in multiprocessor systems. This is because each processor has its own private cache, and the values stored within it may not match what is visible to the other cores. To solve this problem, the memory system needs to be aware of what memory locations are shared between caches and which contain the most recent value. Two popular implementations to this solution are: directory based and snooping [43][44][45]. With a directory protocol, the sharing status of a block is stored in one location, often centralised with the main memory or highest level cache. On the other hand, snooping caches each track the status of the shared memory blocks. A snooping protocol can be implemented as write invalid or write broadcast. With write invalid, an extra bit is used to identify whether the shared block is valid or not. On a cache write, the cache will invalidate the other shared locations which will in-turn cause a cache miss on those other caches if/when they try to access the data. A write broadcast protocol writes the data to the cache and also broadcasts the new value to the other caches, reducing cache misses but increasing bandwidth requirements. Write-through cache simplify the situation since the latest value is always stored in memory, but again, this increases bandwidth requirements. To reduce bandwidth, another bit can be used on each cache line to identify whether a cache is the owner of that line. This changes the status of the block, from being shared, to being exclusive; allowing the owner to modify the value without having to broadcast invalidations to the other caches.

2.2.4.2 Symmetric Multiprocessors

Symmetric multiprocessors comprise multiple homogeneous processing cores that share the same centralised main memory; this includes chip multiprocessors (CMP) with a single chip containing a number of cores, as well as multi-socket workstations and servers. Larger scale

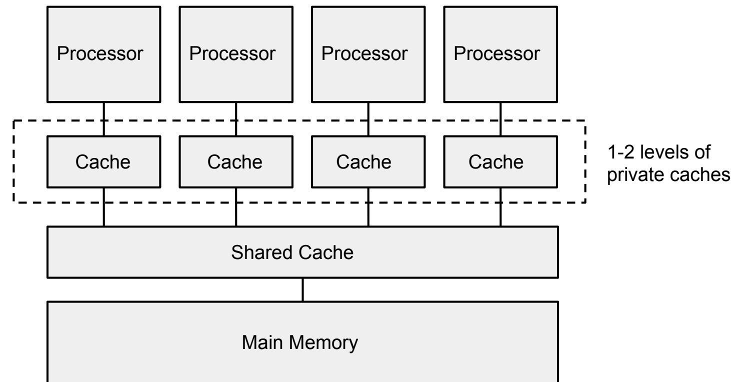

Figure 2.4: Cache Hierachy in a chip-multiprocessor.

computers may have separate memory for individual processors, yet share the same address space, and these are called distributed shared memory (DSM) systems and are not discussed here.

A novel SMP implementation is that of ARM's big.LITTLE architecture. ARMs microprocessor architectures focus on the low power markets and, as consumers have wanted more performance from such devices, have engineered higher performance designs too. This began with the release of the Cortex-A9 which was a dual-issue, out-of-order superscalar, and could be configured into a quad-core SoC [46]. The requirement of higher performance within the same, small, power envelope has driven ARM to offer high-performance cores alongside more power efficient cores [47]. The system architecture is designed to contain two clusters of ISA compatible cores, each with a shared L2 cache, one cluster containing the high performance cores and the other for power efficiency. The clusters, containing 1-4 cores supporting both the ARMv7 (A7, A15) and ARMv8 (A53, A57) ISAs are connected via a cache-coherent interconnect.

Threads are dispatched to cores dependent upon their performance requirements and this can be handled by several methods: two are migration modes that use modifications to the dynamic voltage and frequency scaling (DVFS) mechanisms to move work between cores or clusters, the other is called Global Task Scheduling which is performed in software at the kernel level. In migration modes, each big core is paired with a LITTLE core and only one core is active at any one time; this makes the OS only see half of the total number of cores. The Global Task Scheduler however is aware of the compute capacity differences between the cores, and they are treated as independent so clusters can contain non-equal number of cores. The scheduler uses statistical data from previous runs along with current load and CPU frequency data to select the core on which to run the individual thread.

2.2.4.3 Multithreading

Multithreading enables more efficient use of hardware since it allows resources to be shared when resources may be wasted while executing a single thread, due to cache misses or low amounts of exploitable ILP. There are two key paradigms to multithreading: *fine-grained* (also known as *interleaved*, or *vertical*) where resources are shared through time-multiplexing and the other is simultaneous (or *horizontal*) where pipeline resources are shared simultaneously amongst the threads.

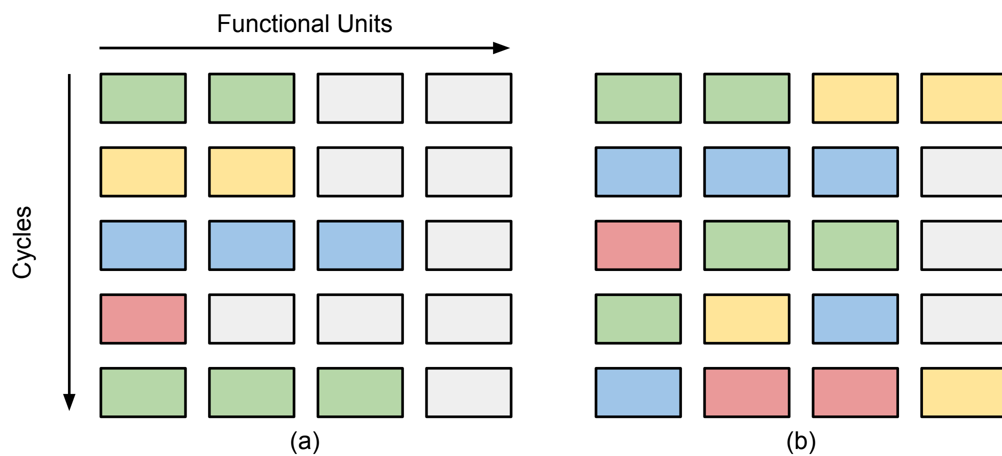

Figure 2.5: Processor utilisation with 4-way (a) fine-grained (*vertical*) multithreading and (b) simultaneous multithreading (*horizontal*), with the four threads in different colours.

2.2.4.3.1 Fine-Grained Mulithreading was implemented in Sun's Niagara architecture which was designed for increased throughout at a lower power envelope for data centres. This was in response to aggressive ILP processors of the early 2000's that focused on single-threaded performance and consumed much more power [48]. The architecture was designed solely to exploit thread-level parallelism (TLP) with a single single-issue, six-stage pipeline that was fine grained multithreaded between four threads. Each chip contained eight cores for a total of 32 threads. The thread select logic would switch between threads on each cycle, with a 0-cycle penalty, from a list of available threads. Priority was given to the most least recently used thread and threads could become unavailable if they were performing a long latency operation such a multiply/divide or a branch.

The commercial server applications that the architecture targetted exhibited poor locality of reference and were difficult to predict and so the memory system was designed accordingly. Each chip contained a large, 640 64-bit, register file supported by a private

16KB L1 cache and a shared 3 MB 4-way banked L2 cache. The load/store unit of each core also contained four 8-entry store buffers that could also bypass load data. The processors and L2 cache were connected via a crossbar that supported 200 GB/s bandwidth with four independent on-chip memory controllers providing in excess of 20 GB/s main memory bandwidth. Sun implemented a simple cache, due to the relatively high number of cores and threads, to save area and it was designed so that the four threads could hide the latencies of L1 and L2 misses. The L1 cache was 4-way set associative and used a random replacement algorithm with a write-through policy. The L2 was 4-way banked between the cores and used a write-back policy. It also held a directory that maintained a sharers list at the L1-line granularity. The crossbar maintained memory transaction ordering between the same and different L2 banks.

2.2.4.3.2 Simultaneous Multithreading (SMT) is an effective method for improving throughput of out-of-order superscalars which can make them more efficient [49]. The IBM POWER7 architecture is a wide out-of-order superscalar: with two load/store pipelines, two fixed-point units, four double-precision floating-point pipelines, one vector and a pipe for branch logic [50]. The core is capable of fetching up to eight instructions, decoding and dispatching six and issuing and executing eight per cycle. It is also capable of 4-way SMT, but threads can be disabled to allow a single thread to utilise all of the execution units. A single chip contains eight cores and a system can comprise of 1 - 32 sockets, for a total supported 1024 concurrent threads.

Again, like Niagara, the memory system had to be carefully designed to meet the compute capabilities of so many threads. Each chip has two memory controllers, each supporting four channels for a sustained bandwidth of 100 GB/s. Each core has access to both a private L1 and L2 and shared a large, 32MB, on-chip eDRAM L3 cache. The L2 cache was designed to be fast, using 8-way association and a 8-cycle latency, to compliment the large L3 cache. Each core is assigned a local 4MB region of the L3 which is used as a victim cache for the private L2, but can also be used as a target for evicted data from other local sections. The cache coherence protocol and interconnect is designed to support more than 20,000 concurrent operations with the off-chip SMP interfaces providing 60 GB/s coherence bandwidth per core. Coherency operations are snooped by the L2 caches, L3 cache regions, the memory controllers and the I/O controllers providing an on-chip coherence bandwidth of 450 GB/s. IBM also implemented a speculative localised-scope coherence broadcast protocol using additional cache states and a distributed directory in memory, with prediction heuristics, to further multiply potential bandwidth by the number of localised scopes.

2.2.4.3.3 Both SMT and vertical TLP are implemented in AMD's Bulldozer microarchitecture which combines two integer cores, a single floating-point unit (FPU) and a shared L2 cache into a module; which is capable of executing two threads [51]. The front-end is shared by the threads via vertical multithreading and is capable of fetching and decoding four instructions per clock cycle to either one of the integer cores as well as the FPU. The integer cores are 4-wide out-of-order superscalars and operate as single threaded pipelines. The FPU, however, was designed to be shared due to its large size and is a 4-wide out-of-order pipeline that supports 2-way SMT. In the latest microarchitecture iteration, Steamroller, the instruction decoder is no longer shared between threads [52].

2.3 Very Long Instruction Word Architecture

This section introduces the *Very Long Instruction Word* (VLIW) processor philosophy, and its hardware design history, before surveying modern usages and describing the configurable VLIW microprocessor that this research has been conducted upon.

2.3.1 Philosophy

The most basic of forms of processor will finish one operation before initiating the next, executing each instruction in the order which the compiler or programmer wrote the code. In typical RISC (Reduced Instruction Set Computing) processors the goal was to achieve one instruction per cycle (IPC), with the most basic model this would be impossible since some operations will have greater latencies than one cycle and there will be branches in the code which will also take up extra time. A technique used in almost every modern microprocessor, to increase the IPC, is to have a pipelined organisation to allow the processor to carry out more than one operation at a time. By pipelining the processor, multiple instructions can occupy different pipeline stages and thus enabling multiple instructions to be in-flight. However, it would be impossible to achieve more than one IPC if no more than one instruction is issued each clock cycle.

VLIW processors use their long instructions and extra functional units to issue multiple instructions simultaneously, exploiting ILP to improve execution speed. The other form of ILP processors, the *superscalar*, is discussed in Section 2.2.2; suffice to say that VLIWs identify and statically schedule for ILP at compile-time where as superscalars use complex hardware to perform the task at runtime. This allows VLIWs to require less hardware than superscalars, enabling more compact and power efficient designs, making them appropriate architectures for embedded applications [53]. They can have lower design time and cost and are easier to simulate since the hardware does not generally perform operations that have

not been explicitly programmed by the software.

2.3.2 History and Evolution

2.3.2.1 Eli-512 and Multiflow

The term VLIW was coined by Joseph A. Fisher with the idea being developed through the early 80's with the Enormously Long Instructions (ELI) project [54]. Fisher proposed a method to increase the ILP that can be found within a program by enlarging the scope of the scheduler beyond basic blocks. He termed this method Trace Scheduling and this is described in Section 3.3.1.3. With a belief in the possibilities of trace scheduling and ILP, Fisher's team at Yale started work on the ELI-512. It was never fully completed but was designed to be a clustered architecture with eight clusters, each containing two register banks, two integer ALUs, a floating adder, a floating multiplier and a memory port. It was designed to be capable of issuing a total of 32 operations each cycle, and of performing multi-way branches. The clusters were organised in a circle, each cluster connected to its neighbours as well as to two others. The system was also designed without a data cache and had a slow central memory controller. The target speed was 5MHz.

The compiler was called Bulldog, and the focus of John R. Ellis's PhD thesis [55]. As the focus of the VLIW philosophy was to reduce the hardware complexity, the complexity had to be incorporated into the compiler. All of the execution details were explicitly communicated via the compiler, including: instruction ordering, functional unit selection, cluster selection, what registers to operate on and which register bank. The problems induced from these tasks were closely interdependent:

- To minimise data transfer between the clusters, operations needed to be designated to the clusters which had direct access to the register banks which contained the operands. At the same time, data needed to be spread to reduce pressure on the banks and the functional units.

- When transfers were required, the compiler needed to calculate the most effective datapath through the system. Memory accesses via the memory controller were slow (the memory controller only handled one access at a time), so the compiler needed to try to access memory directly (each direct access to a different bank could be performed in parallel). To perform direct access the compiler needed to perform memory bank disambiguation to specify which bank to access.

The Eli-512 was designed for scientific computing in which performing calculations on large data sets is very typical. In this case, performance can be particularly hindered if

memory operations cannot be performed in parallel. The purpose of *memory disambiguation* is to resolve whether memory operations are independent. Bulldog expressed the array indices in terms of loop invariants, loop induction variables and other variable definitions using use-def chains [56]. The derived symbolic equations could then be compared to the memory locations that the two indices represented to identify independent memory operations. A similar approach was used to determine the specific memory bank of a value. To make it possible for the compiler to vectorise more code, Ellis implemented assertions that the programmer could use to guarantee that the memory locations were separate.

After the ELI project was disbanded, many of the members went onto start Multiflow in 1984 (closed in 1990). During that time they had six production model machines over two series, both series had three widths: a 7-wide, a 14-wide and a 28-wide. The wider issue machines were constructed from the 7-wide functional units, which they called a cluster, with each containing two integer ALUs, two floating-point units and a branch target unit [57]. Instructions were issued every 130ns with two 65ns beats per instruction, integers could issue in both beats whereas the floating-point and branches could only issue in the early beat, giving a maximum issue width of 7.

2.3.2.2 Cyrdra 5

Around the same time as Multiflow, Cydrome were also designing a minicomputer; the heterogeneous system was designed as two tightly integrated subsystems, one with a single numerical processor and the other was a general purpose system with several processors handling data I/O and the user interface [58]. The two systems shared the same memory and were controlled by the same OS, a Unix implementation named Cydrix. The architecture of the numeric processor was called 'Directed-Dataflow' and based upon the philosophy of dataflow architectures [59]. At the time of the Cydra 5, no dataflow architecture had been a commercial success, due to the extremely high runtime overhead, so Cydrome decided to move most the complexity out of the hardware and run-time and into the compiler; and so it followed the same ideals of VLIWs. The numerical processor had seven functional units: a floating-point adder/integer ALU, a floating-point multiplier/divider, two memory reference ports, two address arithmetic units and a branch unit. Each FU had an associated register file but could also acquire operands from the register files of the other FUs, as well as read and write to a general-purpose register file.

The compiler of the Cydra 5, the Cydrix F77, used loop scheduling which is described in detail in Section 3.3.2.2. To increase support for scheduling loops, the hardware included important additions to provide mechanisms to deal with conditional statements and register allocation. To solve the register allocation problem, the Cydra 5 included a hardware register called the Iteration Control Pointer (ICP) from which the real register address could be

calculated from the sum of the register specified in the instruction and the ICP. Conditional operations and the code reduction was solved using several other registers. The loop counter (LC), epilogue stage counter (ESC) and iteration control register (ICR). Almost all of the instructions could be performed conditionally by querying the 1-bit wide ICR register file. The compiler would control prologue and epilogue by making the instructions in the different stages conditional on different bits in the ICR and the ICR would be changed accordingly as iterations completed. Sections of conditional code were converted into straight line code using if-conversion [60]; the operations on different paths were executed conditionally depending on different bits in the ICR and a select instruction was used to merge alternate values of a variable from different control paths.

2.3.2.3 HP and Intel

PlayDoh was a research project by HP to explore hardware and software methods to extract ILP, with reduced hardware complexity, aiming to address the shortcomings of VLIWs for general-purpose computing [61]. These shortcomings were: the rate of branch instructions in non-scientific codes and the inability to effectively execute a binary from one microarchitecture on another. To address the issue of running code compiled for one implementation on another, the HP PlayDoh (HPL-PD) used a non-unit assigned latency (NUAL) model. This allowed the code to describe the expected latency of an operation, which could enable the issue of an instruction before its operands were available, while very simple hardware was used to insert latency stalls to maintain semantics if the physical latencies varied from the virtual ones. Predicated execution was used to reduce the negative effect of branch instructions; by removing the branch instructions, the delay that is caused by them is removed and this also allows instructions to be freely moved across branch boundaries. This was an enhanced version of what was used in the Cydra 5. Compare instructions were introduced that wrote to predicate registers, these registers could then be referenced for any other instruction to be conditionally executed. If-conversion was handled using the IMPACT algorithm to construct hyperblocks, which is a region formation technique described later in Section 3.3.1.3. They also used predicate promotion to move instructions out of the inner loops and speculatively execute them. HPL-PD also inherited the architectural features of the Cydra 5, with rotating registers and special loop control registers for modulo scheduling.

The hardware supported runtime memory disambiguation through three related families of operations: data speculative load (LDS), data verify load (LDV) and data verify branch (BRDV) [62]. When used, LDS operations logged their operands into a table which were then checked against when a store was executed; the log checked whether the store potentially wrote to the same physical address that a speculative load accessed. If this was true, the entry in the log was invalidated by the processor. LDV operations were then used,

which again, queried the log to see if there were any valid matching entries and if so the processor wouldn't need to do anything else and the data would need to be reloaded. BRDV instructions provided a method to be able to issue instructions dependent on the speculative loads. BRDV detected whether a store aliased a previous LDS operation and if so, branched to a section of compensation code, otherwise the program flow was able to continue.

As the HPL-PD could execute most instructions speculatively, HP devised a system for handling exceptions from speculative instructions. This was necessary since exceptions that arose should not have reported until the instruction was proven to have been executed correctly. Instruction tag bits were used to denote whether an instruction was being executed eagerly. If an instruction caused an exception, it set the speculative tag of the destination register instead of throwing an error. The register tag bits were read by instructions; if a non-speculative operation read one or more source registers with their bits set, it then triggered the exception.

The memory hierarchy had an extra level than usual called the data pre-fetch cache which was generally designed to hold the elements of data arrays that weren't going to be reused, freeing up the cache for items that needed to be cached. The compiler controlled the management of the memory hierarchy, almost all loads had two modifiers to indicate its expected latency and also which cache the data item was likely to be in. The store instructions also had a modifier to specify a cache target.

HP teamed with Intel to graduate the HPL-PD from just a research project into a commercial architecture. The design philosophy, closely related to VLIWs, was termed *Explicitly Parallel Instruction Computing* (EPIC) [63]. The only commercial EPIC architecture has been the product line of an Intel and HP collaboration; the Itanium series. The first Itanium was released in 2001 and quickly followed up by the Itanium 2 [64][65]. All integer and ALU operations completed in one cycle with each of the units fully bypassed, the floating-point units were also fully bypassed and had a fixed latency of 4 cycles. Most the silicon of the processor was used by the 3MB of L3 cache, and many of the microarchitecture features were designed to minimise cache delays and misses. There was an instruction-streaming buffer to hold cache lines from the L2 and L3 caches and was used for instruction pre-fetching and branch prediction; software engaged the instruction pre-fetching by issuing hint instructions about future branches. The branch prediction was performed using two levels of branch history, L1 holding the taken/not taken history while the L2 holds the history of branches evicted from L1. It also contained an advanced-load address table (ALAT) which provided hardware support for greater speculative loads, through dynamic memory disambiguation.

2.3.3 Modern Implementations and Applications

The only significant mainstream, general purpose, CPU to be based upon the VLIW philosophy was the Itanium series, but this was not capable of displacing Intel's other popular architecture. VLIWs have however found their niche in embedded systems, primarily as digital signal processors (DSPs). The reduced complexity of VLIW leads to smaller die sizes and lower power consumption which are the most important design concerns of embedded systems. DSP codes also generally contain both ILP and DLP, with less control-flow than general-purpose programs, making VLIW designs ideal.

2.3.3.1 Qualcomm Hexagon

Qualcomm design high performance mobile SoCs and offer the full stack of hardware components, including their Hexagon DSPs [66]. The original versions were integer-based and designed for voice and audio processing, such as echo cancellation, vocoding and music playback. The latest version includes support for floating-point types and extends the application target to image processing for cameras, facial recognition and sensor input. The core is a statically scheduled, multithreaded, four-way VLIW with 32 32-bit general purpose registers. The Hexagon V5 in the Snapdragon 800 is three-way multithreaded and runs at speeds up to 800MHz with a 16KB I-cache , 32KB D-cache and unified 256KB of L2. It is connected to main memory via a 64-bit bus which runs at 240MHz. Both the CPU and DSP share main memory and coherency must be maintained through software.

The architecture has been designed for high performance with low power and the instruction set has been selected to effectively support VLIW execution of DSP applications [67]. The ISA supports a multitude of data types: 8- 16- 32- or 64-bit fixed-point, 32-bit IEEE floating-point, 32- or 64-bit complex as well as 64-bit vector data. There are four functional units within the core, two dedicated to arithmetic operations (including SIMD) and two dedicated to memory operations that can also support simple ALU operations. The instructions are grouped into packets of up to four instructions and are dependent on the available resources. To reduce branching overhead, many instructions can be executed conditionally using one of three predicate registers, and 'dot-new' instructions can be used define and use the same predicate register within the same packet. Two hardware loop instructions are also implemented for performing nested loop branching with zero overhead. Another an instruction extends this functionality to reduce prologue code of software pipelined loops.

The original versions of Hexagon (V1-V4) implemented round-robin interleaved multithreading (IMT) with three threads executing in a time sliced manner with a three-deep pipeline. This allows a packet in the thread to complete execution before the next packet begins and the simple implementation reduces power, but does reduce efficiency and through-

put when threads are stalled or idle. The latest iteration introduces a *dynamic multithreading* (DMT) which executes some packets faster if threads are idle or stalled. Threads have their own separate register files and communicate via the shared memory; the instruction set includes atomic memory operations to implement semaphores and mutexes.

2.3.3.2 Texas Instruments C6424

The Texas Instrument C6424 is a fixed-point DSP in the TMS320C64xx series designed for telecom, audio and industrial applications; it is based on the third generation VelociTi VLIW architecture [68][69]. The core is capable of dispatching eight instructions per clock at rates of 400-700MHz. The instructions are issued to the two clustered data paths; each cluster has access to 32 32-bit registers and contains a memory unit, a multiplier, an ALU and a unit that calculates shifts, compares and branches. The functional units can execute SIMD instructions with both the arithmetic and multiply instructions capable of operating on quad 8x8-bit or dual 16x16-bit. The data paths are connected by two cross paths which allow each path to read a source operand from the other on each cycle, however this does induce a stall cycle. The ISA is fully predicated, has zero-overhead branching, and also a loop buffer to increase the efficiency of pipelined loops.

2.3.3.3 Tilera Tile-Gx

The Tile-Gx series of processors are based on the Tilera Tile architecture and aimed at embedded networking, multimedia and cloud datacentre markets [70]. The processors comprise of a 2D array of identical tiles, connected via their proprietary network-on-chip called iMesh, and come in configurations of 9, 16, 36 and 72 tiles. Each of the tiles are designed to operate at 1.2GHz and only consume 500mW. The tiles each contain a processor core with TLBs, L1 and L2 cache and interfaces to the iMesh network, which connects them to their four nearest neighbours and the L2 cache. The core has access to two 2-way set associative 32KB L1 caches, one for instruction and one for data. The unified 256KB L2 cache is 8-way and supports Error Correcting Code (ECC). The L2 cache controllers, along with the iMesh network, implement a global cache coherence protocol. The processing cores are 3-wide VLIWs, with short pipelines to reduce complexity and branch penalties, and support SIMD and MAC operations. As well as the programmable VLIW cores, the processor also implements a number of on-chip, but off-core, hardware accelerators for packet processing, cryptographic and compression tasks. The architecture is designed to be programmable with ANSI C/C++ and runs SMP Linux.

2.3.3.4 Kalray MPPA-256

The Kalray MPPA-256 is a single-chip, many-core processor designed for accelerating a variety of algorithms, from video encoding to HPC [71]. Kalray report that the processor is 20x more energy efficient in H264 video encoding than a quad-core, multithread, i7-3820 with all multimedia extensions enabled. The device is designed so that it can operate in stand-alone mode or as an accelerator connected to a host via PCIe. The MPPA-256 contains 16 compute clusters and 4 quad-core I/O subsystem clusters. Each compute cluster holds 16 processing elements (PEs) and a single resource manager (RS) along with shared memory and a DMA engine. The compute clusters have their own private address space and contain a 2MB ECC shared memory, organised into 16 banks, with an aggregate bandwidth of 38.4 GB/s. Communication between clusters is via the network-on-chip.

The compute core is a 5-way VLIW, clocked at 400MHz, containing 2 ALUs, 1 MAC / FP unit, 1 load/store unit and a branch unit. The functional units have access to 64 32-bit registers and each core has two 8KB, 2-way set associative, caches; one for instruction and one for data. The development tools allow programmers to use one of three programming models:

- cyclostatic dataflow (CSDF) language [72], named $\sum C$, which alleviates the task of synchronisation and difficulties of shared memory for the programmer through the use of dataflow computation.
- traditional POSIX processes are spawned to execute on compute clusters and then POSIX threads and OpenMP can be utilised within the clusters.
- OpenCL via their (still in development) compiler.

2.3.3.5 The Configurable VLIW Chip Multiprocessor, The LE1

The LE1 VLIW CMP is a configurable system-on-chip multiprocessor system, designed to accelerate signal and image processing algorithms on field-programmable silicon. It is an in-house design from Loughborough University by Dr Chouliaras and the target hardware platform for this research. The LE1 implements a partially predicated ISA, based on the Lx architecture [73]; which itself is based upon Multiflow. So its genesis is deeply routed in the original VLIW architectures as well as their modern evolution into DSPs. The LE1 builds upon the capabilites of the Lx architecture through the capability of configuring a many-core system and the use of multithreaded cores, as well as incorporating pipeline interlocks. The configurability of the LE1 creates a large architecture exploration space for research and its design as an accelerator makes it an ideal candidate for OpenCL research.

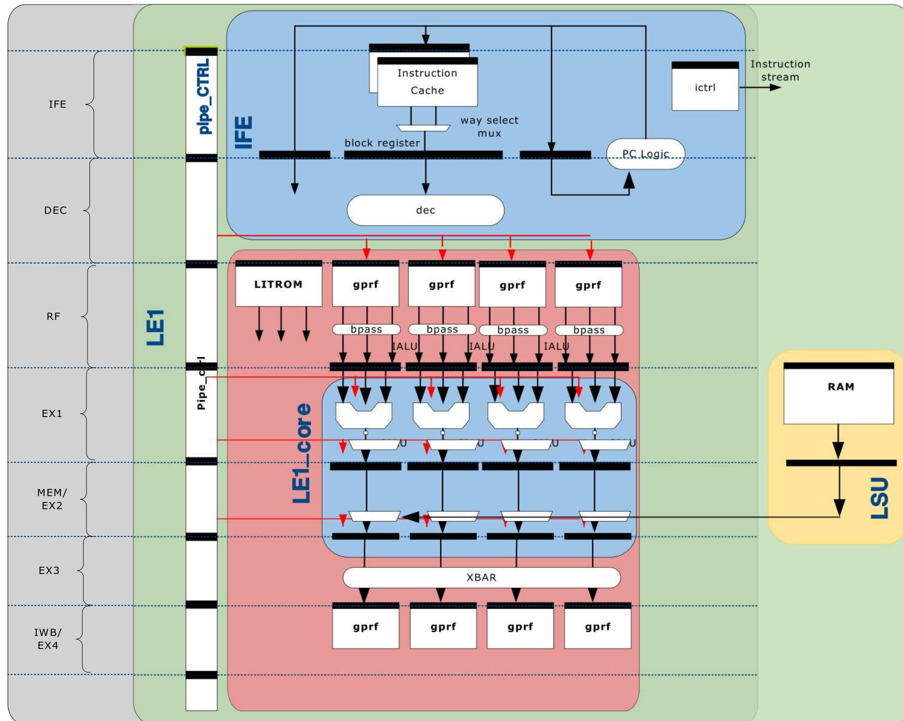

Figure 2.6: Components and pipeline of a 4-wide, single-core LE1.

An LE1 system can incorporate up to 256 cores, or *contexts*, in a shared memory organisation. Each context contains one or more architected states, or *hypercontexts*, which can execute an independent thread of execution. For this research, each context only has a single hypercontext. The processing core has an 8-stage integer pipeline and can be partitioned into a configurable number of clusters, up to a maximum of 16. The ISA, named VT32PP, specifies a configurable number of clusters, each cluster consisting of up to 64 static general purpose registers, 8 single-bit predicate registers (used for computing branch conditions and conditional selection), a PC and a Link register (LR). Implementations can define the number of contexts, hypercontexts, clusters, integer ALUs, multipliers and load/store units. A view of a 4-wide LE1 microarchitecture configuration is depicted in Figure 2.6[74].

The CPU consists of the Instruction Fetch Engine (IFE), the execution core, the pipeline controller (PIPE_CTRL) and the Load/Store Unit (LSU). The IFE can be configured with an instruction cache or alternatively, a closely-coupled instruction RAM (IRAM). These are accessed every cycle and return a long instruction word (LIW) consisting of multiple RISCops for decode and dispatch. The IFE controller handles interfacing to the external memory for ICache refills and provides debug capability into the ICache/IRAM. The PIPE_CTRL, the

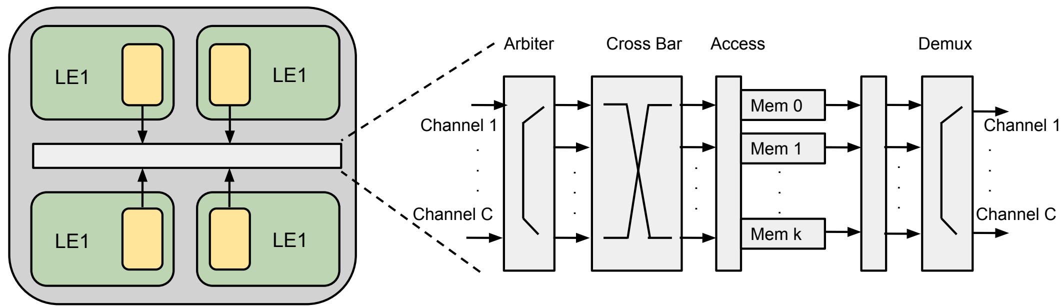

Figure 2.7: Quad-core LE1 with connecting memory system.

primary control logic, is collection of interlocked, pipelined state machines, which schedule the execution datapaths and monitor the overall instruction flow down the processing and memory pipelines and also maintains the decoding logic and control registers of the CPU. The LSU is the primary path of the core to the system memory and allows for up to the architected width of memory operations per cycle and directly communicates with the shared data memory. The memory is a multi-banked, 2 or 3-stage pipelined cross-bar architecture and the number of channels and banks do not have to be equal. A quad-core LE1 system with the connecting memory system is shown in Figure 2.7. Further details of the ISA will be discussed later in Chapter 5 in the context of compiler development. The LE1 is designed to run as an accelerator attached to a host CPU, which runs the OS and coordinates communication between the two systems. The host loads both the instruction (IRAM) and data RAM (DRAM) on the LE1 system and then initiates the execution.

Long instruction words (LIWs) consist of a variable number (up to the architectural width of the processor) of RISC-type operations known as syllables or RISCops. Instruction accesses are byte- aligned (to allow for future implementations where variable instruction lengths are supported) and control transfer instructions target the first syllable of the target LIW. The architectural width of the processor is communicated to both the compiler and the processor RTL and is used in the software and hardware compilation processes respectively. There can be only one control transfer instruction (branch/jump/call/ret) at the end of a LIW.

2.4 Heterogeneous Computing Architectures

Most computer systems need to perform a variety of tasks, and it is becoming increasingly difficult for uniprocessors, or homogeneous multiprocessors to accelerate computation effectively across the spectrum of tasks. Heterogeneous systems include a variety of architectures that are more specialised to different tasks, which can make the system more efficient and

increase both single- and multi-threaded performance. This section identifies the types of architectures that are used for accelerating software, and the programming languages are discussed in Section 3.4.2

2.4.1 Graphics Processing Units

Modern GPUs have been evolved from their original fixed-function pipelines into fully programmable processors. GPUs are throughput devices as their architectures are designed to primarily calculate millions of pixels near a fixed frame rate. Because of this, GPUs have high bandwidth requirements and concurrently execute hundreds, or thousands, of threads to mask memory latencies. GPU shader programs often have to perform the same calculation on a large set of vertices and *fragments* and so high-performance devices utilise a *Single Instruction, Multiple Thread* (SIMT) style of execution. This is similar to SIMD, but the different data elements are taken from different threads, which this means that performance is dependent on how each thread executes. GPUs achieve maximum throughput when the executing threads maintain the same program counter (PC), allowing the single issued instruction to execute with different data from the various threads. These constraints have little effect on highly-regular graphic shader programs, but throughput can dramatically decrease in the presence of control-flow (thread-divergence) with multiple bespoke solutions proposed to alleviate thread divergence [75] [76] [77]. Divergent threads execute serially as only a single PC can be used resulting in substantially reduced utilization of the chip datapaths (Processing Elements).

2.4.1.1 NVIDIA Kepler and Maxwell

NVIDIA's GPUs consist of 'Graphical Processing Clusters'(GPCs) which are comprised of many 'streaming multiprocessors' (SMs), which contain many CUDA cores, load/store units, special function units, a large register file and a scheduler. A CUDA core is a very simple with just an ALU and a FPU. Each SM schedules threads in groups of 32 parallel threads called warps. Each SM can issue an instruction from two warps per clock. The SM features of NVIDIAs latest architecture, Maxwell, are not too different from their previous design, Kepler, as shown in Table 2.1. The Kepler architecture is implemented in the NVIDIA K1 mobile SoC [78], but Maxwell is designed to be even more efficient: Kepler is capable of 12.7 GFLOPs/W where Maxwell can now achieve 21.76 GFLOPs/W; with both implementations using a 28nm process [79]. The key difference is that the SMs in Maxwell are separated into 4 smaller processing blocks, distributing the functional elements and register file evenly, and it's the partitioning that simplifies the design scheduling logic which reduces area and power.

Table 2.1: Architecture Comparison between NVIDIA Kepler and Maxwell.

Feature	**Kepler**	**Maxwell**
SMs per GPC	2	5
CUDA cores per SM	192	128
SFU per SM	32	32
LD/ST per SM	32	32
Warp schedulers per SM	4	4

2.4.1.2 Graphics Core Next

Graphics Core Next (GCN) is AMD's latest GPU architecture that is also designed for GPGPU programming [80]. The architecture is broken down into compute units (CUs), each capable of executing 2560 work-items in SIMT style. Each CU contains four SIMD units and although they are called SIMD, they are capable of issuing multiple instructions. The vector ALU within each SIMD unit has 16 lanes, taking 4 cycles to operate on a wavefront of 64 work-items, and each unit has an instruction buffer for at least 10 wavefronts. Both the CU front-end and SIMD unit are capable of issuing five instructions per cycle with the CU selecting the SIMD unit in a round-robin fashion. The CU contains six execution pipelines which handle both scalar and vector data and there are also special instructions which are consumed within the instruction buffers in a single cycle. To maintain program order, only a single instruction from each wavefront can be issued per cycle and only a single instruction of each type may be issued also. The seven types of instruction choices are: vector arithmetic logic unit, scalar ALU or scalar memory read, vector memory access, branch / message, local data share (LDS), export or global data share (GDS), internal within the instruction buffer.

AMD's latest R-Series Accelerated Processing Units (APUs) are designed for embedded applications and combine a multi-core x86 CPU with a GCN graphics core [81]. In this APU, both the CPU and the GPU share address space which enables more fluid GPGPU programming as less data needs to be copied and transferred. AMD has designed the device around the Heterogeneous System Architecture (HSA), which is introducted in Section 3.4.2.4.

2.4.1.3 Broadcom's VideoCore IV

The VideoCore IV is the mobile orientated GPU found within the Raspberry Pi [82] and is designed for some GPGPU programming as well as graphics [83]. The scalable compute capacity is provided via multiple instances of a shader processor termed a Quad Processor (QPU) which can be grouped into up to four slices. To the programmer, the QPU is a 16-

way SIMD processor however it is physically 4-way and multiplexed over four cycles. The QPU core is dual-issue and contains two floating-point ALUs, one for multiply instructions and the other for add / logic type instructions. Both ALUs support 8-bit vectors for some operations. To keep the pipeline simple, results written to the register file are available two cycles later and no forwarding paths are provided. The QPU core contains two single-ported register files which are used along with six accumulators registers, which do not suffer the two cycle write-back penalty to provide the extra argument.

2.4.2 Many-core Accelerators

GPUs have proved as a disruptive technology, requiring a shift in programming paradigms, that have enabled much more efficient, yet powerful, systems. But, by their own nature, they are still designed to accelerate graphics and still consume a relatively large amount of power. This subsection looks at the alternative many-core accelerators that are not based around GPU architectures. Many are designed to be capable of operating as standalone CPUs but are mainly used as accelerators alongside a host CPU. Most of them target low power, with high performance per watt, and can be programmed with traditional languages and APIs as well as more modern approaches.

2.4.2.1 Cell Broadband Engine

The IBM Cell Broadband Engine (CBE) was a project set up between Sony, IBM and Toshiba to develop a new architecture to meet the demands of next generation game and multimedia applications [84], specifically the Sony Playstation 3. The console was designed to enable internet gaming and media streaming and so the architecture was designed for those target applications. IBM identified these key barriers that were preventing existing architectures from being suitable:

- Memory latency and bandwidth, as both media and streaming require high memory bandwidth, and low memory latency is key in maintaining high performance within the CPU.
- Power and power density, as the consumer product would have to be suitably sized and not eject too much heat.
- Responsiveness, as the system would have to react to inputs from different players, locally and on the network, while keeping audio and video synchronised.

IBM viewed the existing, deeply pipeplined, out-of-order, superscalars to be too demanding on the power budget of an embedded system. The deep pipelines would also create

longer instruction latencies and greater mispredict penalites, both of which would decrease the responsiveness of the system. So IBM designed the CBE processor which consisted of a central control processor with eight, *synergistic processors*, coprocessors. The central Power Processor Element (PPE) used direct memory access and synchronisation mechanisms to communicate with the Synergistic Processor Elements (SPEs). The PPE was a 64-bit Power architecture with a two-way SMT, dual-issue, in-order microarchitecture. It had a 23 stage pipeline with 32 64-bit fixed-point registers and 32 64-bit floating-point / vector registers, thus exploiting many different forms of parallelism.

Each SPE contained a processing unit (SPU), a channel unit and a DMA engine and was connected to the rest of the system via a high speed interconnect bus capable of 96B/cycle [85]. The SPU was capable of issuing two instructions per clock and executed 128-bit SIMD operations. It contained a large 128 entry register file as well as a 256KB local store to reduce the memory bandwidth requirements and to enable the compiler to exploit more ILP. Memory accesses by the SPU accessed the local store with the system memory being accessed via the DMA engine. The DMA engine was capable of processing 16 commands simultaneously, each fetching up to 16KB of data, and commands included scatter-gather operations from the system memory. To reduce the complexity of the hardware, as well as improving bandwidth utilisation, the software was responsible for data prefetching and branch prediction. The channel unit was a message passing interface between the SPU core and the rest of the system. The CBE supported various programming models including: offloading library functions from the PPE to a SPE, partitioning the work amongst SPEs for them to execute in parallel, and streaming data through the SPEs with each applying their own computational kernel.

2.4.2.2 Many Integrated Cores

Intel's Many Integrated Cores (MIC) architecture is mainly a response to the introduction of GPU accelerators into the HPC market, with Xeon Phi coprocessors being implementations of the architecture [86][87]. The x86 cores, based on the old Pentium 5, are dual-issue, in-order, 4-way (fine-grained) multithreaded with 64-bit support. The instruction set includes FMA, 512-bit SIMD instruction (including scatter/gather vector memory operations), and hardware support for single-precision transcendental instructions. The cores are arranged in a bidirectional ring network with fully coherent 512KB L2 caches, and have a 32KB, 8-way set-associative, L1 cache. The coprocessors come in various configurations, but all have more than 50 cores clocked at 1GHz or more, with access up to 16GB RAM. The highest performing Phi coprocessor, the 7120x, is a 300W device with 61 cores with a peak performance of 1.2 TFLOPS, thus operating at 4 GFLOPS/W. Software development with the MIC architecture is designed to be very similar to that of programming for a

modern x86 CPU with a linux-based operating system running on the coprocessor. The Intel software development kit (SDK) supports thread-based languages, including: Intel Threading Building Blocks (TBB), Intel Cilk Plus and OpenMP [88].

2.4.2.3 Epiphany

Adapteva's Ephiphany is a many-core 32-bit architecture which is designed to be scalable up to 4095 processors, sharing the same address space [89]. The Epiphany III is found in the 16-core Parallela development board [90] and Epiphany IV (E16G401) will be its successor, with 64 cores [91]. The chip consists of many RISC cores (eCores) connected in a 2D mesh network with each node having connections to only its nearest neighbours. The nodes in the mesh comprise of an eCore, a DMA engine, local memory and the network interface. The eCore is a 2-wide, in-order, superscalar with both integer and floating point ALUs and a 64-entry 32-bit register file. The E64G401 eCores can be clocked up to 800MHz and provide a theoretical peak performance of 102 GFLOPS, when utilising the fused multiply-add and multiply-subtract instructions, consuming 2W. The eCores share a 2MB on-chip distributed memory with an aggregated local memory bandwidth of 1.6 TB/s, 102 GB/s NoC bisection bandwidth and 6.4 GB/s off-chip bandwidth. The architecture is designed to excel in image and signal processing, encryption and compression tasks, and is programmable in C and C++ with some support for OpenCL.

2.4.3 Field Programmable Gate Arrays

Field Programmable Gate Arrays (FPGAs) are programmable silicon devices, programmable by the user, to create digital circuits [92]. Traditionally, FPGAs filled the use case of high-performance custom designs, where low volume production could not justify an application specific integrated circuit (ASIC) implementation. FPGAs make this possible due to their low entry costs, though costs per device are higher. The key market area for FPGAs has traditionally been in real-time systems and as digital signal processors (DSPs) where high speed, custom logic blocks and custom data widths are necessary. They also have the key advantage of being reconfigurable, allowing bug fixes and modification to be made in the field or allowing the device to be configured for a completely different task.

Modern FPGA fabrics consist of many thousands of *configurable logic blocks* (CLBs) which are interconnected, arranged in an array, and surrounded by configurable I/O blocks (IOBs) [93]. The key components of the CLBs are: lookup tables (LUTs), DSP *slices* and block RAMs, which are connected via multiplexers that are controlled by the configuration program. The configuration program is stored in on-chip SRAM, which means that an FPGA has to be reconfigured every time it powers up. On initilisation, the bitstream is

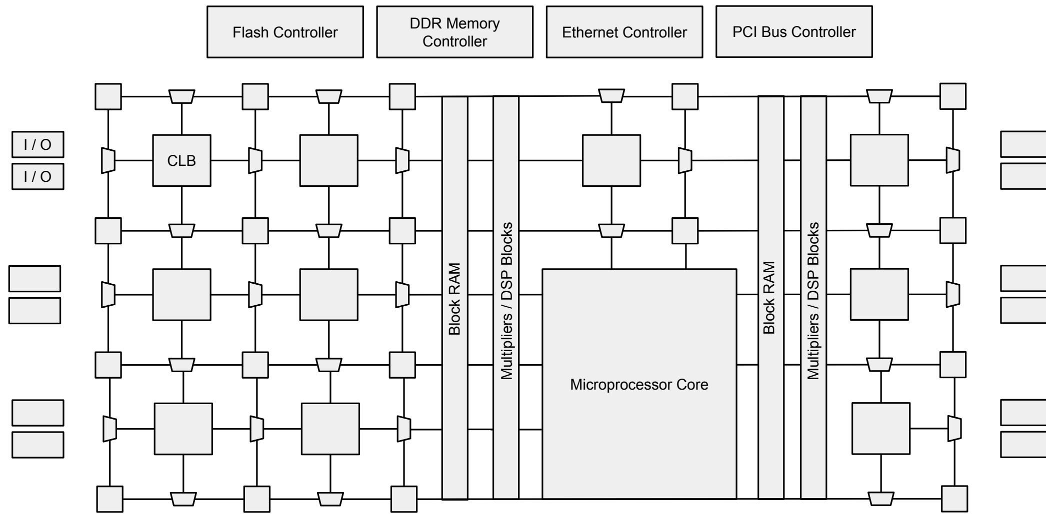

Figure 2.8: Internal components of an FPGA.

shifted into the SRAM programming cells from off-chip memory. The CLBs are surrounded by wiring channels, multiple bits wide, that are separated into segments by switchboxes at the intersection of the channels; this is depicted in Figure 2.8. Switchboxes are matrices of programmable pass transistors that are each controlled by a memory cell; the transistor can be switched on and off by the value in the cell to either make or break the connection. Connection boxes are implemented to select inputs to components within the CLB from the horizontal and vertical segments.

Key metrics in FPGA fabric design are size, speed and cell utilisation. These attributes are interlinked and so architecture parameters are determined in a trade-off. Cell utilisation is directly affected by the connectivity between the CLBs and this is determined by how flexible the interconnect is [94]. Flexibility is a function of the channel width and the number of connections in the switch- and connection boxes, with flexibility rising with an increase in connections. But the number of transistors used for the connections also affects area and line delay as they take up physical space and induce capacitance, which contributes to determining achievable clock frequency. Area on the FPGA is taken up by the CLB and the interconnect. The area required by a CLB is dependent upon the amount of fixed hardware within the block and the number of it's inputs and as block complexity increases, less blocks are required to implement a circuit [95]. As the number of inputs increase, however, so does the area required for routing which can consume the majority of the available area. Therefore, the increase in CLB complexity needs to reduce the number of required blocks and also compensate for the increase in routing area.

Table 2.2: Feature Set of Xilinx Virtex UltraScale Product Series

Feature	**UltraScale Series**
Logic Cells	626,640 - 4,407,480
CLB Flip Flops	716,160 - 5,037,120
CLB LUTs	358,080 - 2,518,560
Maximum Distributed RAM	3.9 - 28.7 Mb
Total Block RAM	44.3 - 132.9 Mb
DSP Slices	600 - 2,880
DSP Performance	4,268 GMAC/s
Transceivers	36 - 120
Peak Transceiver Speed	33 Gb/s
Peak Serial Bandwidth (full duplex)	5,886 Gb/s
PCIe Interface	2 - 6
DDR3/4 Memory Interface Performance	2,400 Mb/s
I/O Pins	365 - 1,456

The feature set of Xilinx's high-performance UltraScale FPGAs are listed in Table 2.2 [96][97]. Each CLB contains one slice and is comprised of 8 LUTs and 16 flip-flops, as well as arithmetic carry logic and multiplexers. The LUTs can be configured as 64-bit RAMs or as shift registers. The DSP slices contain 27x 18-bit multipliers and a 48-bit accumulator which provide both multiply add and multiply accumulate functionality. The multipliers can be bypassed to feed a SIMD ALU for 2-way, 24-bit, add/sub/accumulate or 4-way 12-bit operations. The DSP slice is also pipelined and contains a programmable width 96-bit-wide XOR function and a 48-bit-wide pattern detector.

As well as more traditional FPGAs, such as the UltraScale architecture, vendors are now offering SoCs that combine fully programmable general purpose CPUs with an on-chip programmable FPGA. Xilinx offers the Zynq-7000 platform which combines a dual-core ARM Cortex-A9 coupled with either a Artix-7 or Kintex-7 FPGA [98][99]. The Altera Stratix 10 is the company's upcoming, high-performance SoC which integrates a 64-bit, quad-core, ARM Cortex-A53 in a FPGA fabric built on Intel's new 14nm proces s[100]. The Stratix 10 further differentiates itself from the competition by being the first FPGA to contain dedicated hardware for floating-point calculations, and doubles the previous generation's clock rate up to over 1GHz. Altera claim that the device will be capable of 10 TFLOPs and is capable of supporting a total off-chip bandwidth of 1.382 Tbps.

2.5 Hardware / Software Codesign

As microprocessors use power gating to stay within the confines of their TDP limits, it results in percentages of the silicon being unusable and thus limiting their performance. To be capable of effectively using more dense logic, the silicon has to be more efficiently utilised. The most efficient way to utilise silicon for a task is to create an *application specific integrated circuit* (ASIC). In devices where efficiency is particularly important, such as smart mobile devices, ASICs are included to support common tasks that would otherwise be very CPU intensive, such as multimedia encoding. These ASICs are fabricated on the same die as the CPU and so the resulting chip is called a system on-chip (SoC); as the multi-core paradigm has grown, these are now called multiprocessor SoCs (MPSoCs). The decision of this mix between hardware specific circuits and software implementations is called hardware / software (HW/SW) co-design and the focus is usually a balance of performance, power and area.

2.5.1 ASICs and FPGAs

It has been suggested that cores within an MPSoC should operate at reduced voltages and frequencies to increase throughput instead of ramping voltage and frequency for some cores while switching off others [101]. Researchers have also suggested using 20-30% of the silicon area for reconfigurable logic, or ASICs, to support lower power cores. The choice between FPGAs and ASICs can be decided upon how much commonality the kernels have and how much coverage the ASICs can provide. The low power dissipation of the reconfigurable logic and low power cores, allows for greater utilisation of dense transistor logic, whilst also accelerating execution [102]. It is noted however, that as processing cores and the use of FPGAs and ASICs increases, bandwidth requirements of heterogeneous MPSoCs will increase significantly.

The design of the hardware and modifications to the software are usually performed by a *design space exploration* (DSE) tool that often operates on graph, or tree, representations of the task [103][104]. The DSE tool is fed the source code, which is sometimes annotated to aim hardware synthesis, as well as design constraints. These constraints are based upon area, power and performance requirements which are tightly interconnected. The tool will explore the nodes of the graph and use cost functions to calculate whether the operation, or group of operations, would be better performed by hardware or software. For special hardware to be instantiated, the ASIC needs to be capable of performing the operation(s) faster than the host processor, and this includes the communication overhead of using an accelerator. Cost estimation software are usually based upon the latency of operations whereas hardware costs are based upon the necessary area and power requirements.

2.5.2 Application-Specific Instruction-set Processors

In cases where multiple target applications share similar types of computation, incorporating instruction extensions (IEs), instead of an ASIC for each application, can be a better use of area and enable greater flexibility [105][106]. Calculating effective IEs is similar to designing an ASIC, however the custom hardware block will be included within the microprocessor pipeline and instructions need to be included in the ISA. IEs have the advantage of being more readily reusable and do not suffer the communication overhead incurred when using ASICs. However, they are further constrained by the, possibly fixed, microarchitecture; such as the number of ports to the register file and clock frequency. Like ASIC integration, the IEs need to have good coverage and resusability for the application. It is the task of the compiler or intrinsics to utilise the additional instructions and it needs to be able to match the complex instruction patterns that arise from merging of multiple instructions into one.

The ASAM project, of the ARTEMIS program, aims to develop a heterogeneous MPSoC platform, designed through HW/SW codesign, that is capable of tera-flop performance within a mobile SoCs power budget [107]. The platform is based around the Intel SiliconHive application specific instruction-set processor (ASIP) which is a configurable and extensible VLIW core with both SIMD and MIMD support. The MPSoC is comprised of several ASIPs, customised for particular parts of the application, and distributed shared memories, connected via a bus or NoC. The DSE tool is split into four, inter-communicating, stages: system DSE, ASIP DSE, global communication and memory DSE (GC&M) and HW/SW synthesis and prototyping. System DSE feed the application C code and is responsible for the entire design and defines the network of the several ASIPs and the distributed memories. ASIP DSE consists of a simultaneous co-tuning of the ASIP architectures and their embedded software, including parallelising techniques for software optimisations and high-level synthesis. The GC&M DSE targets the exploration and optimisation of the global communication and shared memory of the system. The system DSE creates a communication graph of the application tasks and feeds this to the GC&M stage which can use simulation and prototyping to validate designs. The HW/SW synthesis takes an abstract architecture description and produces the RTL description and compiles the software tasks for each of the ASIPs.

In the early 1990's, the MOVE framework was conceived to utilise a *Transport-Triggered Architecture* (TTA) to aid in the creation of ASIPs [108][109][110]. The TTA architecture was designed to exploit ILP and use a compiler to find that ILP, much in the same way as VLIWs. However, the architecture was designed to fundmentally differ from VLIWs to make them more suitable to be used as template architectures. Multiple functional units are connected by a number of buses, which together define the number of operations that

can be issued per cycle. Operands are transported to the functional units and execution is triggered the by the arrival of the complete number of operands. Register files (RFs) are also treated as functional units and are not necessarily available to all of the FUs, this can be implementation defined. All reads and writes between the RFs and FUs are programmer-visible, including the registers of the bypass logic, which makes the bypass registers the first level of memory available to the program. These modifications allow for smaller and less complex register files as they are partitioned and both read and writes can be reduced by the explicit use of software controlled bypass logic.

The TTA-based Codesign Environment (TCE) continues in the same manner by providing a toolchain to automatically design a processor, from C code, using a DSE tool, or manually using a graphical tool [111]. TCE provides a template architecture that can have its number and mix of FUs, the pipelines they contain and the register files and their connectivity all uniquely specified. The tool is split into three main phases: (1) processor design space exploration, (2) code generation and analysis, and (3) program image and proccesor dseign generation. During the processor design phase, the tool estimates the die area, consumed energy and runtimes and stores the information into a database. The code generation phase uses a compiler together with an architecture simulator to produce code and estimate its energy consumption and cycle time. The tool could then select the most appropriate design based upon some specified criteria. The final stage of the tool generates an HDL file which is verified by the simulator.

2.5.3 Codesign for Supercomputers

Codesign is of interest in the supercomputing space too, even where power dissipation capability is high. DARPAs Ubiquitous High-Performance Computing (UHPC) program has challenged researchers to achieve 50 GOPs/W with a 20MW power envelope to be capable of reaching exascale performance. The US Department of Energy's exascale computing initiative has identified that hardware-software codesign could be essential to achieving exascale performance [112]. Many researchers are drawing upon embedded systems and toolchains, which already have to operate within strict and low power requirements, and are investigating the potential of using semi-custom hardware using hardware-software codesign processes.

2.5.3.0.1 Runnemede was one such project where all layers of the computing stack were codesigned to minimise the amount of required energy, targeting low clock frequencies at near-threshold voltages [113]. The runnemede architecture was designed to be modular, with the CPU separated into blocks with both L3 and L4 memory; each block contained a single scalar control engine, multiple execution engines and some L2 memory. Their initial design used custom, single-issue, in-order RISC cores for the execution engines while

the control engines were based on the Siskiyou Peak [114] synthesisable core. A dataflow-inspired execution model was used to break computation into 'codelets' which could be executed by the execution engines; this model allowed for software-managed caches and reduced synchronisation overhead. Codesign was used to find suitable extensions to the ISA to reduce the energy used on key functions as well as finding suitable memory optimisations; in their case study their codesign optimisations reduced total energy consumption by 75%.

2.5.3.0.2 The Green Flash many-core processor was designed to accelerate climate simulations at an necessary estimated rate of 70 PFLOPs [115]. The design team calculated that the problem could be split into 20,971,520 subdomains, and so with each core being assigned a subdomain, a single core would need a computational rate of 3.5 GFLOPS. The design team, from Lawrence Berkeley National Laboratory, decided to look towards embedded systems design tools for their system to minimise power requirements. A tool from Tensilica (now Cadence) [116] was used which allows a hardware designer to start with a base architecture which can then have features added or removed, the instruction set is expandable and the toolchain also creates a compiler, testbench and simulator for the newly designed architecture. Software autotuners were used to select compiler optimisations using domain-specific knowledge of the algorithm to produce an optimal code. The tuned software was then used, in an iterative co-design process, as a reference to tailor the hardware to achieve greater efficiency and find a suitable configuration to meet their power and performance requirements.

2.6 Discussion

This chapter has the shown the breadth in variety of architecture styles and the types of computation that they excel in. With the fact that many styles are in existance, plus resurgence of accelerators and co-processors, underlines that no magic bullet has yet been conceived for hardware. However, there are four key reoccurring themes and features that are part of most of the implementations described: the rise of SoCs, the exploitation of multiple forms of parallelism, higher memory bandwidth and lower power envelopes.

Almost all modern designs are SoCs: they contain ASIC blocks for media codecs, programmable VLIWs for DSP applications, integrated GPUs for graphics, and, with the very recent acquisition of Altera by Intel, we may soon see FPGAs being embedded too. To utilise the components of the SoC, the vendors have to develop kernel code, drivers and SDKs for third party developers but it enables smaller chips with a greater performance/watt ratio. The transition to SoCs will continue as more functionality is brought off the motherboards and onto the chip as this reduces the power requirements of external buses, reduces latency

and enables the vendor to known exactly what components the system contains, allowing for system optimisations in both hardware and software. The inclusion of more ASICs and application-specific instructions will also most likely continue as more demanding codecs are developed, encryption becomes more important and as target power consumption continues to shrink. This will be aided by the continuing development of FinFET technology which is greatly reducing static power, which will allow the inclusion of more specialised hardware without the detrimental affect of power drain and subsequent heat dissipation, even when it is not being utilised.

Most of the processors described also exploit multiple forms of parallelism, with SIMD extensions and threading the most popular evolutionary steps. SIMD instructions are currently the main way that architects can improve the single threaded performance of CPUs. Intel's AVX is evolving to be able to operate more like a vector machine by being more flexible and so it would be reasonable to expect that this is the way that other vendors will also investigate, until finally a full vector processor is a common functional unit of a CPU. Data-level parallelism is quite abundant in many popular applications but it is certainly not always exploited, this is often because of how code is written and/or the autovectorisation capabilities of the compiler. Vendors currently need to ship specific libraries to fully utilise the functionality of their designs which makes portability difficult and performance less optimal; which is bad for both the vendor and the user. More collaboration between hardware and software teams will be needed so that hardware features can be developed to ease the task of autovectorisation. Multithreading enables more effective use of superscalar pipelines to make up for the lack of ILP but also to hide memory latencies, but it can also harm single-threaded performance. So it can be imagined that different cores within a processor will be instantiated with varying multithreading capabilites so to hide more memory latency as well as maintaining the execution speed of key threads.

The growth rate of the compute capabilites of microprocessors has not be matched by the development of DDR RAM technology. This has made it necessary for more SoC silicon to be used on memory with larger register files and deeper an larger cache hierarchies, such as on-chip EDRAM caches. These cache memories are mainly useful for applications with large data sets and these types of application are growing. A focus on the memory system is also evident in GPUs where threads are used to mask the latencies caused by accessing memory. In the desktop market, GDDR5 has evolved to be faster and wider at the expense of larger cards and high power requirements. The power required by GDDR5 and ever growing bandwidth requirements has pushed AMD and NVIDIA to develop 3D stacked memories which are now coming to market [117]. Innovation is also happening in system interconnects to provide higher bandwidths, so to enable more cores and the greater use of GPUs and accelerators: NVIDIA has developed NVLink [118], for which IBM is a customer,

and Intel continue to develop their on-chip ring interconnect for their Xeon processors. As big-data and graphics applications and many-core systems demand higher bandwidth and lower latency, it only is logical that development in memory and interconnects will be absolutely vital in improving system performance - more so than the interals of individual cores.

2.7 Summary

This section has reviewed the current state of computing architectures. It has been shown that recent trends and paradigms have been changing as silicon-based CPUs have hit the *power wall*, where they can no longer clock faster without dissipating too much heat. This has lead computer architects to utilise data- and thread-level parallelism to increase performance past what is possible with ILP based uniprocessors. System designers have also looked towards accelerators to increase through-put and efficiency. Power consumption is one of the most important factors in modern computing, through rising running costs, the want for longer battery lives, and the physical limitations of silicon systems. Codesign systems have been researched to target greater throughput while also reducing the power requirements. Accelerators have also been introduced to increase through-put because of the huge increase in the amount of data that is commonly produced. The LE1 VLIW CMP has been introduced which aims to exploit ILP and TLP, and its configurable architecture lends itself to codesign systems. It was used as the hardware platform for the ENOSYS European FP7 project, which used high-level UML and a DSE tool to produce both a program binary and hardware configuration [119]. Current accelerators are based on many simple cores to mainly exploit TLP, and this is further augmented with multithreading capabilities to mask memory latencies; the ubiquitous accelerator is a GPU. The next chapter reviews the new languages and platforms that have been developed for these arising computing paradigms.

Chapter 3

Parallel Programming

3.1 Chapter Objectives

The aim of this chapter is to introduce both the traditional and more recently developed programming languages and frameworks that have been designed to exploit the architectures that were described in the previous chapter. The chapter will cover both old and new languages and focus upon targetting OpenCL to multi-core CPUs, DSPs and FPGAs. The chapter begins with a review of compiler technology, focusing on VLIWs, as they are the key software tool to enable the adoption of high-level languages.

3.2 Compiler Overview

High-level languages have been introduced to make software more portable and more easily read, which speeds development and debugging. Compilers are a key tool for software developers as they convert, and optimise, the high-level languages to the assembly language of the target architecture. The importance of the compiler varies between architecture styles. VLIW and statically scheduled superscalars rely heavily on the decisions made by the compiler, whereas dynamically scheduled superscalars can re-order instructions and rename registers, making them less reliant. Compilers can be designed to accept multiple languages and support multiple target architectures, but their compatibility and feature support can vary. Compiler writers and hardware vendors can supply libraries which include intrinsic functions that target specific hardware features of certain architectures. Likewise, languages such as OpenMP extend C with pragmas which can be either accepted by the compiler or ignored.

3.2.1 Code Representation

As a program passes through the many stages of a compiler, it is represented in different formats to be more suitable to the current task. The program is stored in various data structures and forms, depending on the task and how far the process is into compilation. Some structures are better for representing the details of source languages while others resemble machine language. The program can often be represented in more than one format at a time to combine the benefits of the different intermediate representations (IRs). IRs generally take two forms: graphical ones, such as trees and graphs, and linear sequences of operations [120].

Syntax Trees are often used by the frontend of a compiler as the original source code is scanned and parsed. A parse tree is a graphical representation that corresponds to the input program, containing a node for each grammar symbol in the derivation and communicates both the precedence and meaning of the expressions. Some of these nodes in a parse tree are unnecessary for the rest of the compiler, so it can be simplified into an Abstract Syntax Tree (AST). When used at a high level, ASTs often contain source-level abstractions which make them particularly useful for source-to-source transformations.

Directed Acyclic Graphs (DAGs) are graphs which contain no back edges. When used in the frontend, these further contract the AST by allowing nodes to have multiple parents and sharing identical subtrees. This can help remove redundancies so it is more efficient to store in memory than an AST. The removal of redundancies also makes DAGs usual throughout the compilation process, aiding in effective instruction selection and removing duplicate code. Control-Flow Graphs (CFGs) model the flow of control between the basic blocks of a program. The nodes of the graph are the basic blocks, connected by edges that represent the possible runtime flow of the program execution. Data-dependence Graphs (DDGs) encode the flow of values from their definitions to their uses. The nodes in the DDG represent operations and the connecting edges between nodes indicate a value that is defined in one node, being used by another node. This creates a partial ordering to nodes, as it does not encode important control-flow information.

Single Static Assignment (SSA) form is a linear IR in which variables are uniquely named for each assignment to them. This form explicitly represents *du-chains*, where a du-chain represents all the possible uses of a definition. This representation is important for several optimisations such as constant propagation, invariant code motion and partial redundancy elimination. Figure 3.1 gives a code fragment with its accompanying CFG which has been modified into SSA form. Two changes have been made to the CFG: (1) a unique name has been given for each location where a variable has been assigned a value and (2) ϕ nodes have been inserted to select between values from converging control-flow paths.

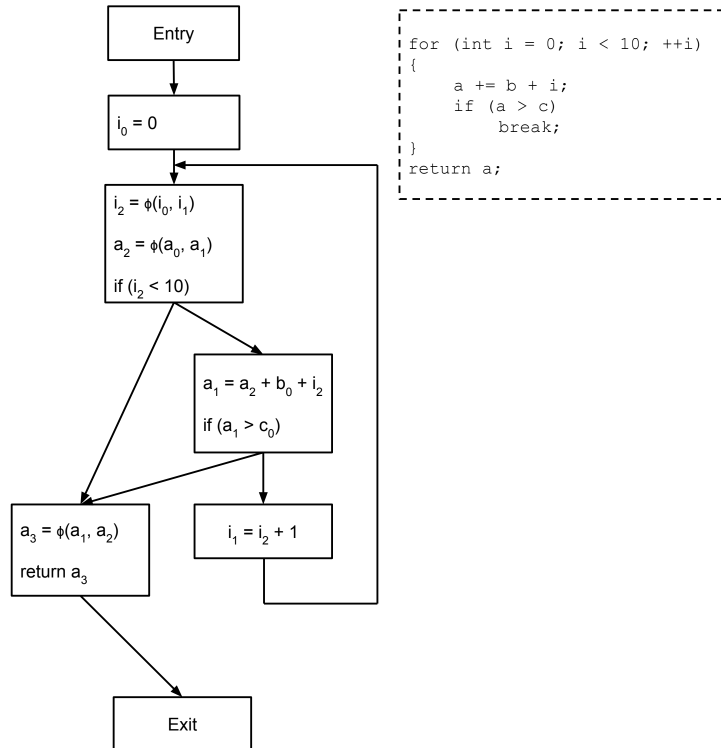

Figure 3.1: Example of *Single Static Assignment* (SSA) form.

3.2.2 Code Generation

Code generation is the final part of compilation where the program is transformed from an internal representation into the assembly format of the target architecture.

3.2.2.1 Instruction Selection

Instruction selection is the process of converting a program from the IR into instructions that the target architecture can execute. It is not only important for the execution speed of the program, but also the code size which is often more important than speed in embedded systems. Instruction selectors are very often produced automatically from a machine description which can make the retargeting of a compiler a more simple task. The old approach was to perform syntax directed techniques by parsing tree grammars and matching these to patterns defined in pre-computed tables, such as the Granham Glanville technique [121]. This involves assigning target machine instruction patterns to the IR and storing the information within a table. Depending on the abstraction level and the target architecture, many machine instructions may be mapped to a single instruction in the IR, or it can be

the other way. As most architectures provide many ways to perform the same operation, the table is ordered in a 'best instruction first' manner and the table is iterated from the top during the search. Selection can also be performed using DAGs, which is more complex than tree pattern matching as nodes can be shared in the graph; some of this complexity can be reduced by decomposing the DAG into trees to work upon but can produce suboptimal results. DAGs can also be parsed [122] and the same dynamic programming technique as tree patterns can be applied where nodes/non-terminal combinations are labelled with their costs. Shared nodes are then considered during a reduction phase, and so the subgraphs are reused.

3.2.2.2 Instruction Scheduling

Instruction scheduling is the process of trying to select an optimal order for all the instructions in the program to issue. List scheduling has been the dominant technique for scheduling for several decades, because it is easily adaptable and produces good results [123]. It is a technique rather than a set algorithm and so can be modified to fit different circumstances. An implementation of a list scheduling algorithm will usually operate on renamed code to avoid anti-dependences, it then builds a dependence graph and assigns latencies on the edges from source values to their sinks. Each operation is then set a priority utilising a scheme; this is where the technique can be highly adaptable and define how effective the scheduler is [124]. The classic scheme is to base priority of the nodes on the length of the longest latency path from a node to the root. The scheduler then uses two lists, one for instructions that are ready to be issued and one for currently active operations. A cycle count is kept so when enough cycles have passed, it means that an operation in the active list would have completed and so it is removed. The scheduler can then check the removed operations successors to determine whether they are ready to be issued, if so there are added to the ready queue. Operations in the ready queue are issued depending on the scheme used.

3.2.2.3 Register Allocation

Register allocation is the stage where the code is mapped from a set of infinite virtual registers into the limited register set of the target architecture. This can happen before or after instruction scheduling. The allocator needs to try to reduce the effect of generated spill code (the code to load and store a value to and from memory), this includes the time for address computation with the execution frequency of the operation as well as the code space these instructions occupy. Allocation can occur locally over individual basic blocks. However this requires load and store operations to be used to safely pass values between basic blocks, which can dramatically reduce the performance. Therefore most allocators work globally

using live variable analysis to understand what variables are maintained throughout the program; live ranges bound the definitions and uses of a value. These live ranges can be used as nodes on a graph, with edges indicating which live ranges interact with one another. This is called an interference graph and is used for allocation and spilling [125]. The task is to colour the interference graph so that no adjacent nodes are coloured the same, with k number of colours to choose from, k being the number of registers of the architecture. If the graph cannot be k-coloured then spill code has to be inserted and a re-attempt made.

3.2.3 Types of Compilation

Compilers are not just tools used by software developers to create distributable binaries and libraries. Some programs are distributed in portable formats which require compilation when the application is installed, while some languages are dynamic and self-modifying which can be sped up by compilation and optimisation during runtime.

3.2.3.1 Static Compilation

Static compilation is the traditional method of creating an executable program from source code. The application developer feeds the source of the program into the compiler which then outputs the machine code or object file, the format of which will be specific to the target platform. Static compilation allows the compiler more time to perform optimisations which can result in a faster executable. Sometimes, however, the binary can be significantly larger than the source file; especially if it is large *statically linked* program. Linking is the process of including library function and data within an executable, this can either happen statically at compile time, or dynamically at runtime. Statically linking ensures that the executable has (otherwise external) data contained within it which helps make the program more portable. Dynamic linking reduces the program size by allowing programs to share libraries, however source programs need to be compiled to target the right version of a library; otherwise binary incompatiabilities can arise.

3.2.3.2 Just-in-Time

Just-In-Time (JIT) compilation encompasses creating machine code at program runtime and then executing that code, also at program runtime. The primary advantages to this model is that portable code can be distributed and it enables dynamic code creation. Some programming languages are *interpreted*, where the interpreter translates the source code at runtime and skips the output of machine code, performing the execution immediately. This negates the, sometimes lengthy, compilation times and makes the code portable since it can run where ever the interpreter is installed. This also keeps the code size small. However,

interpreted languages are significantly slower because less optimisations are performed [126]. JIT compilers aim to take the advantages from both statically compiled and interpreted languages by distributing small, portable, programs that can be modified and compiled at runtime. JIT compilers can be implemented within interpreters to selectively compile and optimise certain, or all, parts of the program.

The importance of JIT compilation can be highlighted in the fact that it enables the internet experience of today. Nowadays, web pages and client side web apps are accessed from smart mobile devices and PCs running Windows, BSD (iOS, OSX) and Linux (Android) on both x86 and ARM architectures, all of which requires that code is portable. The language that powers most client side applications is Javascript and so all major browsers contain a Javascript engine: Apple's Webkit, Mozilla's SpiderMonkey and Google's V8. All the engines have the ability of performing JIT compilation, as well as interpretation, of Javascript to increase execution speed across multiple platforms [127][128][129]. Server side applications have also historically run on different operating systems and different architectures, such as POWER, SPARC and x86. Both Java and C# programs are compiled and distributed in a bytecode format which is JIT compiled within the VM, as is the case with the .NET Common Language Runtime [130], C# Mono runtime [131] and Java's HotSpot [132]. The portability of the application only requires that the VM has to be ported to different platforms and architectures.

High-level, dynamic, languages have become popular in the scientific community for easily expressing algorithms. Two examples are: (1) SciPy [133] a collection of Python tools and packages while Python is generally interpreted and compiled into a bytecode to run within a VM [134] and, (2) Julia, a high-performance, concurrent, language which also uses JIT compilation [135].

3.2.3.3 Ahead-of-Time

Ahead-of-Time(AOT) compilation aims to increase execution speed of portable applications by compiling a target independent representation of a program to machine code at install time. This compilation generally improves upon JIT compilers, as it has more time to perform optimisations, but can also increases the size of the executable, compared to the source code or bytecode, and can increase initial startup time. Most of the runtime engines mentioned in the previous sections also have AOT compilers embedded within them and the latest version of Android uses one when installing applications [136].

3.3 Compiling for VLIWs

This section covers the various compiler techniques that have been developed, alongside hardware features, to enable statically scheduled computers to exploit as much ILP as possible. The key to these analysis passes and optimisations is that they aim to provide the instruction scheduler with as many options for instruction issue as possible; the actual scheduling algorithm does not necessarily have to be any more complicated than that of a superscalar. Thus, the most important part of compilation is the code formation for the scheduler.

3.3.1 Region Formation

Programs are generally broken down into basic blocks, which are maximal straight-line sections of code without any control-flow within them; they are bounded by control-flow statements. For ILP processors it is advantageous to group basic blocks that are executed most frequently together so that more instructions can be analysed. Code placement and structure is also very important when using caches, as minimising cache misses is of the utmost importance due to the huge difference in access speeds to memory.

3.3.1.1 Profiling

Code formation techniques for VLIWs often include enlarging the scheduling regions, which could lead to substantial code size increases if the whole program was optimised in the same way. Code size can affect the performance of the instruction cache as a larger instruction stream will require a larger cache to avoid performance degradation. Complex optimisations and transformations also lead to longer compile times, which is a particularly important metric in JIT systems. Profiling can be used to find the suitable sections of code to optimise to help improve compile time and to produce higher performing code with less code growth [137]. Profiling counts the occurrences of events during the execution of the program; this can be individual instructions, but is more often performed on basic blocks. Dynamic profiling gathers information during the runtime of a program and is more accurate than static profiling, which is performed by the compiler, and can be particularly useful in JIT runtime systems [138]. Profile information is vital for many optimisations and transformations as it enables a compiler to focus on optimising areas which are executed heavily. There are different types of profile information that can be collected, such as basic block execution counts and execution time [139].

3.3.1.2 Speculation and Predication

The compiler can utilise speculation and/or predication to overcome control dependencies by removing them or converting them to data dependencies. This allows larger basic blocks to be constructed which gives way to greater optimisations as well as better use of the pipeline. Software speculation is where the compiler speculatively moves code above branches that are highly weighted in one direction, and so breaks data and control-flow dependencies [140]. This can reduce the critical path of computation, increases ILP, can help tolerate latencies and helps issue long-latency instructions earlier. It does have some drawbacks too though; it can increase register pressure, can increase the critical path of other parts of code and adds complexity to the exception-handling mechanism. Control speculation is executing an instruction before knowing that its execution is required or needed; it's very helpful when branches are predicted correctly, potentially increasing ILP, but can potentially waste resources when speculated incorrectly. Data speculation is executing an instruction before knowing it can be executed correctly.

Predication offers a more effective technique for ILP processors as it is used to collapse short sequences of alternative operations after a branch that is nearly equally likely in each direction. This has the effect of transforming control dependencies into data dependencies and is called *if-conversion* [60]. Predicated or guarded execution refers to the conditional execution based upon the value of a operand called a predicate. If-conversion is used to convert condition statements to predicate defining operations and the following statements into predicated instructions. As well as reducing the number of cycles taken from branching, eliminating branches means there are less branches that can be predicted incorrectly, and so less time is wasted recovering from those mistakes. There are two models; partial and full predication [141]. If the architecture supports full predication, each instruction is provided with an additional source operand to hold a predicate specifier, and so all instructions can be predicated. Partial predication support requires a small number of instructions that are conditionally executed, this helps keep the ISA simple but reduces the useful scope.

3.3.1.3 Region Enlargement

Finding parallelism in most programs requires the compiler to operate on larger areas than basic blocks, two of the more basic techniques are by using extended basic blocks and loop unrolling. An extended basic block (EBB) is a group of basic blocks where the only beginning block can have multiple predecessors but each proceeding block has one predecessor, this is shown in Figure 3.2. Global code motion can be used to then produce code that the scheduling can work with to find more parallelism. Loop unrolling is a classic technique which unrolls the body of a loop so that more work is done for the overhead of the loop

branching back in on itself, as well as revealing more instructions to the scheduler to take advantage of parallelism. Obviously this creates larger code so often there is a limit to unrolling and/or it is reserved for particularly hot parts of code; this information can be gained from profiling.

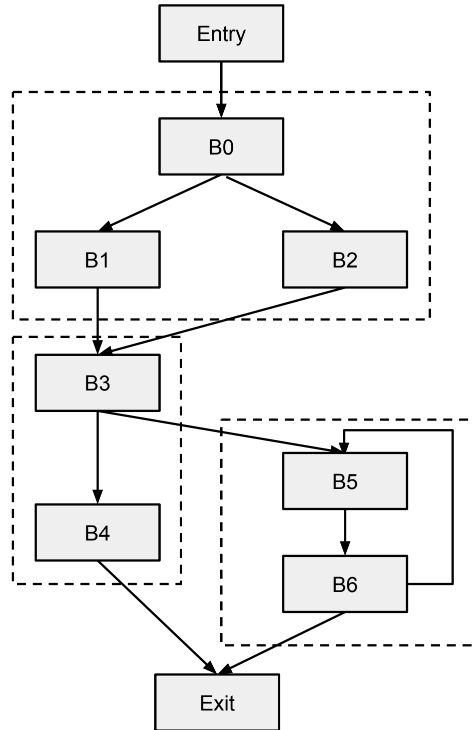

Figure 3.2: Extended Basic Block Formation with the EBBs shown within the dashed lines.

3.3.1.3.1 Trace scheduling provides a way to group basic blocks into much larger streams of instructions and the decision of what blocks get grouped together is performed using profile information or heuristics [142]. The selection algorithm works from the most frequently executed blocks to the least until all blocks have been scheduled. The group of basic blocks to be scheduled together is called a trace, it is a linear path through code which can have multiple entrances and exits. The Bulldog compiler used Traces, and creating them requires more than just selecting the basic blocks. Firstly, schedulers were designed to operate on basic blocks where control entered at the top and exited at the bottom, however traces allowed conditional statements within traces (splits) and allowed jumps in too (joins). So at the trace boundaries (entry, splits, joins and exit), the generator had to report the locations of all the live variables. Conditional jumps also meant that extra edges had to be added

between certain nodes to maintain correct live ranges and ensure *write-after-read*(WAR) dependences were maintained. By grouping and scheduling the most frequently executed blocks first, it reduced the amount of compensation code added into the critical path, which otherwise could degrade performance.

3.3.1.3.2 The Superblock is similar to a trace, but formation and scheduling is designed to reduce the complexity of the compiler [143]. A superblock is a trace without side entrances; it has one entry but can have many exits. Superblocks are formed in two steps: First the traces are identified using profile information, as with trace scheduling, and the second uses tail duplication [144] to eliminate any side entrances to the trace. The process of tail duplication involves replicating basic blocks, which can be entered from outside the trace, so that side entrances do not occur. This maintains program semantics but changes the CFG. Superblocks are similar to extended basic blocks, their difference is mainly in their formation because profile information is used to construct superblocks. Once a superblock is formed, it can then be enlarged to increase ILP though it is important to note that enlarging optimisations are only performed on the most frequently executed parts of a program. Superblocks are enlarged using three techniques:

- Branch target expansion expands a superblock when a likely control transfer ends with another superblock. The target superblock is copied and appended onto the original.
- Loop peeling modifies a superblock loop (a superblock which ends in a likely transfer to itself) but peeling the loop into straight line code. This is performed when the likely number of iterations is low, this expected number can be found through profiling. The original loop body is copied to the bottom of the code to handle extra iterations.
- Loop unrolling copies the body of a superblock which tends to iterate many times.

To increase ILP further, a number of optimisations are performed to remove data dependencies, again these are performed in a controlled way as they can increase the number of executed instructions. Five optimisations are used:

- Register renaming removes anti and output dependencies by assigning unique registers to different definitions of the same register.
- Operation migration moves an instruction from the superblock, who's result is not used in the superblock, to a less frequently executed superblock.
- Induction variable expansion creates new names for variables in unrolled loops. Induction variables are used within loops to index data structures, anti, output and flow dependencies can then limit ILP when the loop is unrolled.

- Accumulator variable expansion is similar to induction variable expansion in that variables are renamed in unrolled loops. Accumulator variables accumulate a sum or product in each iteration of the loop.

- Operation combining eliminates flow dependences between pairs of instructions with compile-time constant source operands, this often arises in address calculation and memory access operations. The dependence is removed by substituting the expression of the first instruction into the second.

3.3.1.3.3 Hyperblocks use predication to form a region that includes many control flow paths, whereas superblocks utilise speculation [145]. As with superblocks, they have a single point of entry but can have many exits and the basic blocks are also selected using profile information. Hyperblocks are usually formed from basic blocks that typically make up the body of an innermost loop. The basic blocks are selected carefully so that hyperblock formation doesn't have a detrimental effect of the performance. By removing infrequently executed paths, the constraints imposed on the frequently executed paths are relaxed. Larger blocks are given a lower priority than smaller blocks since less ILP is likely to be discovered in the smaller blocks by themselves compared to the larger. Also, blocks with hazardous operations such as procedure calls and memory accesses have a lower priority in selection as these types of instructions reduce the effectiveness of the optimisations.

Blocks are selected if they have no incoming control paths other than to the entry block and if they contain no nested inner loops. Tail duplication and *loop peeling*, where loop iterations are extracted out of the loop, are used to transform the blocks to meet these requirements [146]. Node splitting may also be performed which can completely remove merge points by duplicating sufficient amounts of code, obviously this can dramatically increase relative code size but it's particularly effective on wide issue machines and control-intensive programs. The selected blocks are then if-converted and formed in a hyperblock. If-conversion first calculates all the control dependences between all the basic blocks in the hyberblock, then one predicate register is assigned to the basic blocks with the same control dependence. The predicate defining instructions are then added into the basic blocks which contain the source of the dependences. Finally the conditional branches between blocks are removed leaving a single hyperblock of predicated code.

3.3.1.3.4 Treegions A treegion is a region containing a tree of basic blocks which is a subgraph of the CFG, they have a single entry and multiple exits and are formed without the aid of profile information [147]. They can contain multiple, independent, control paths and so they are classed as non-linear, whereas superblocks and hyperblocks are linear regions. Treegions are grown by traversing the CFG absorbing basic blocks if they are not merge

points. Once all edges have been explored, the basic blocks at the treegions leaves become saplings which become the new roots of another region. This process continues until the entire CFG has been consumed.

3.3.2 Scheduling

3.3.2.1 Acyclic Scheduling

List scheduling can be used for VLIWs and requires larger scheduling regions to utilise the available hardware resources efficiently, such as the ones described in the previous subsections. To be most effective for ILP processors, the scheduler needs to operate on a representation that doesn't induce false dependencies; so ideally before register allocation. A data-dependence graph can be modified so that each node (instruction) contains the functional unit requirements and the latency of the operation. The operations without any predecessors can be added to a queue of ready nodes which the scheduler then has access too. The scheduler can then use an internal clock together with some form of resource hazard table that tracks the usage of resources by each of the instructions in flight. On each cycle of the clock, the scheduler can select instructions from the ready queue and assign them to available resources and increment the clock once the queue is empty or no resources are available.

The quality of the produced code is dependent on how the scheduler selects instructions from the queue, and a key metric is usually the *height* as the higher value represents a node on a more critical path. The scheduler will also need to employ some form of decision making function when nodes in the available queue have the same height. A useful metric is using the latency of the operations so that longer latency operations are scheduled earlier or by how many descendants the node has so that more nodes may be added to the available queue. Another issue is that the scheduler needs be aware of how greedy it is being and this is for two reasons: (1) scheduling too many instructions to be inflight may increase register pressure and lead to spill code being inserted which will almost definitely reduce performance of the generated code, and (2) some instructions could occupy a resource for multiple cycles which could block more critical instructions later in the schedule.

This method of scheduling is not much different for that of a pipelined scalar architecture, but scheduling VLIWs can be significantly different if the architecture is clustered, as such was the case for the Eli-500 and the TI C6000 DSP; described in Sections 2.3.2 and 2.3.3. Clustering the datapaths can greatly reduce hardware complexity and allow for wider and faster machines, but it does increase the complexity of the compiler as it needs to decide where the instruction executes as well as when. There are two key approaches to this: either select the cluster before scheduling or perform cluster assignment simultaneously

with instruction scheduling.

The bottom-up greedy (BUG) algorithm was implemented in the Bulldog compiler and used two passes over the DAG to assign clusters before instruction scheduling. The algorithm traversed the DAG from the exit nodes up to the entry nodes, estimating the best FU for each node. It then worked back down to the roots making the final assignments on the way, estimating the cycle in which the FU can compute the operation. BUG kept track of FU usage for each cycle in a table and updated the table after each node was assigned a location. The unified assign and schedule (UAS) algorithm was developed to address the issue that BUG could not eliminate the oversaturation of the interconnect buses since it did not have a complete knowledge of the interconnect availability and ignored copy instruction scheduling delays [148]. Early clustering was also viewed as constraining for the scheduler. The UAS algorithm integrated cluster selection into the list scheduler of the instructions by treating the bus interconnect as a machine resource and by accounting for the inter-cluster copy latencies of the instructions. For each instruction selected by the scheduler, a cluster was chosen from a priority queue for it to be assigned to. The researchers found that the best priority function was one where priority was given to clusters that would produce the source operands late in the schedule; they also found UAS to outperform BUG.

3.3.2.2 Cyclic Scheduling

Many programs spend most of their time in loops, however acyclic schedulers do not account for this. Acyclic schedulers handle loops by unrolling the loop body (the acyclic region) to expose more instructions to the scheduler. However, this is more difficult when the loop count is unknown at compile time, it also enlarges the code size and does not account for the loop back edge. The region formation techniques of the previous can also grow the code significantly which is particularly undesirable in embedded systems. *Software pipelining* is a class of global cyclic scheduling algorithms that exploit inter-iteration ILP and so are useful for VLIWs [149] and generally do not induce code growth of unrolling. They do this by pipelining instructions from the body of the loop across multiple iterations in order to take advantage of the available resources within one iteration. This trades a longer single iteration latency for greater throughput. Successive iterations are initiated at regular intervals, called the *initiation interval* (II), and the goal of the scheduler is to minimise this value [150]. Resource constraints and interiteration data dependences place a lower bound of the II, called the *resource-constraint minimum* and *recurrence-constraint minimum* respectively. As the scheduled loop body contains instructions from multiple iterations, some code is required to setup the loop before the first iteration begins and this is called the *prologue*. Similarily, code is also required to finish the instructions of the final iteration and these are handled in the *epilogue*. Software pipelining schedulers still operate on a per cycle basis, but

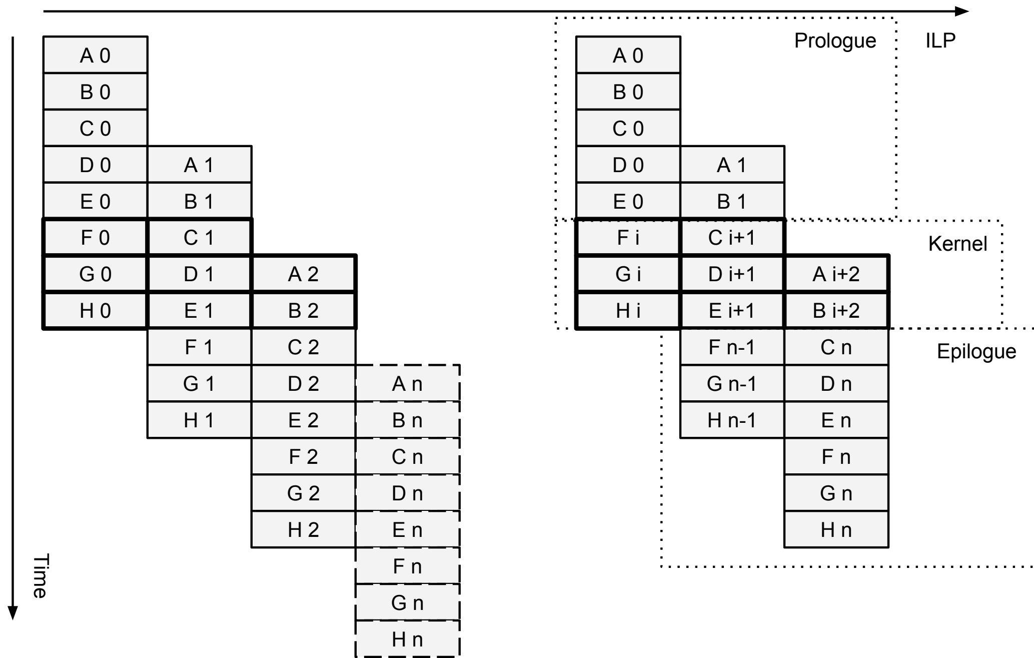

Figure 3.3: Illustrated concept of software pipelining.

more complex than acylic as they need to schedule for intra- and interiteration dependences and resource usage.

The developers of F77 compiler, for the Cydra 5, created a software pipelining algorithm called *modulo scheduling* which conceptually unrolls the loop to find a recurring set of instructions which can be scheduled together in a single loop body [151]. This body is called the *kernel* and is constructed so that neither data dependence nor resource conflicts arise; Figure 3.3 depicts this concept. To track resources across multiple concurrent iterations, the modulo scheduler uses a *modulo reservation table* (MRT) which ensures that within the single iteration the same resource is never used more than once at the same time, modulo the II. Once a target II has been selected, the scheduler can work can select instructions from a ready queue, using the current cycle and the II to check for resources in the current cycle and future iterations. Table 3.1 shows a worked example of how a loop, performing `c[i] = a[i] * b[i]` on the LE1, would be modulo scheduled with II = 5; the upper part of the table shows the original schedule, followed by the new prologue (p0-p9), kernel (k0-k3) and epiplogue (e0-e6).

Though the MRT guarantees that functional resources are not oversubscribed, it does not guarantee that there are enough registers available for allocation. Register allocation becomes an issue when the life time of a value is longer than the II, since the following

Table 3.1: Modulo scheduling example

Cycle	**ALU 0**	**ALU 1**	**MUL**	**LSU**	**BR**
0	sh2add r6, r2, r4	sh2add r7, r2, r3			
1	sh2add r9, r2, r5	add r2, r2, 1			
2				ldw r16, r6[0]	
3				ldw r17, r7[0]	
4	nop				
5			mulhs r8, r1, r16		
6			mullu r10, r17, r16		
7	nop				
8	add r11, r10, r8	cmplt b0, r2, 100			
9	nop				
10				stw r9[0], r11	br b0, loop
p0	sh2add r6, r2, r4	sh2add r7, r2, r3			
p1	sh2add r9, r2, r5	add r2, r2, 1			
p2				ldw r16, r6[0]	
p3	sh2add r6, r2, r4	sh2add r7, r2, r3		ldw r17, r7[0]	
p4	add r2, r2, 1				
p5			mulhs r8, r17, r16		
p6		cmplt b0, r2, 98	mullu r10, r17, r16	ldw r16, r6[0]	
p7				ldw r17, r7[0]	
p8	add r11, r10, r8				brf b0, e6
p9	nop				
k0	sh2add r6, r2, r4	sh2add r7, r2, r3	mulhs r8, r17, r16	stw r9[0], r11	
k1	sh2add r9, r2, r5	add r2, r2, 1	mullu r10, r17, r16		
k2				ldw r16, r6[0]	
k3	add r11, r10, r8	cmplt b0, r2, 98		ldw r17, r7[0]	br b0, loop
e0		add r2, r2, 1			
e1			mulhs r8, r17, r16		
e2	sh2add r9, r2, r5		mullu r10, r17, r16	stw r9[0], r11	
e3	nop				
e4		add r11, r10, r8			
e5					
e6				stw r9[0], r11	

definition will overwrite the previous value. The Cydra 5 introduced hardware support in the form of rotating registers to address this issue, but it can also be handled by the compiler. The scheduled loop body can be unrolled by a number of times so that the new length covers the lifetime of the value, this technique is called *modulo variable expansion*. Another issue arises in the presence of control-flow within the loop body; to handle this effectively the architecture should support a predicated ISA so that the control dependences can be converted into data dependences.

3.4 Languages and Platforms

This section gives an overview of the currently available languages that have been designed to enable different forms of parallel programming. The traditional methods of threading and message passing are described, as well as the more recent developments into languages and platforms designed for heterogeneous computing. OpenMP, described in Section 3.4.2.1, is both old and new as it is an API that has been used for shared memory multiprocessing for over 15 years and has also recently been updated for heterogeneous computing. The section finishes with a survey of the different methods, reported in literature, that have been developed to compile OpenCL programs for a variety of differing platforms and architectures.

3.4.1 Traditional Methods

Traditional implementations towards parallel programming primarily focus on utilising multiple separate processors, either in a shared or distributed memory system.

3.4.1.1 POSIX Threads

POSIX threads (Pthreads) defines an API for creating and manipulating threads by defining a set of C programming languages types, functions and constants in the `pthread.h` header file of POSIX conformant operating systems [152][153]. Pthreads exist within UNIX processes, sharing the memory, and are much more ‘lightweight’ than processes as they only need to maintain their own program counter, stack pointer and registers. The creation of the threads can be 10-100x faster than creating a new process using `fork()` as the kernel does not need to make a new independent copy of the process. Operating in a shared memory environment also greatly reduces the communication overhead between threads as all global variables are visible to each thread. This does however require that the programmer explicitly controls access to shared data.

`pthread_t` type defines a thread handle, which is used by the `pthread_create()` function to create a new thread along with the name of the function that will execute as the

thread. The new thread then terminates when the function returns. The `pthread_join()` function is passed a thread Id and waits for that thread to end, suspending the current thread while it waits. This functions joins two threads back into one and is necessary for the proper cleanup after a thread terminates. The `pthread_mutex_t` type defines a *mutex* which enables a mutually exclusive lock, accessed using the `pthread_mutex_lock()` and `pthread_mutex_unlock()` functions. If a thread tries to lock access to a mutex, but finds that it is already locked, the thread goes to sleep and is effectively placed in a queue waiting for access to the data. By using mutexes, it is possible to ensure that only a single thread operates on the data at one time.

3.4.1.2 Message Passing Interface

The Message Passing Interface (MPI) is an open specification for a library API that enables communication between processes in a distributed memory environment [154], currently at version 3.0. The API is designed to be cross-platform and language independent The interface addresses the message-passing parallel programming model which involves data being moved between different processes / cores / computers. MPI is designed to allow efficient communication by avoiding memory-to-memory copying, by allowing the overlap of computation and communication and to enable the use of co-processors. The standard is designed to enable both point-to-point and collective communication.

Point-to-point communication is primarily enabled by using the `MPI_SEND` and `MPI_RECV` functions to send a buffer of data and to signal that the data has been received and saved. The `MPI_SEND` encodes the location, size and type of the send buffer as well as an `envelope`, which specifies the message destination and contains distinguishing information that can be used by the receive operation. The `envelope` contains a `communicator` which is used to specify a context for communication; messages from different contexts do not interfere with one another. The communicator specifies the set of processes that share the communication context with the `MPI_COMM_WORLD` communicator provided by MPI to enable communication with all the processes. Both blocking and non-blocking communication is supported.

Collective communication involves a group(s) of processes, with the communicator defining them and providing the communication context. There are two types of communicators: *intra-communicators* and *inter-communicators* which respectively identify a single group of processes and two distict groups. MPI defines collective communication functions including:

- `MPI_BARRIER` to provide synchronisation across all members of a group.
- `MPI_BCAST` to broadcast a message from one member to all members of a group.
- `MPI_GATHER` to gather data from all members of the group to one member.

- `MPI_SCATTER` to scatter data from one member to all the members of a group.

3.4.2 Programming for Heterogeneous Systems

3.4.2.1 OpenMP

OpenMP is an API for C/C++ and Fortran programs that defines code constructs and a runtime library to support parallel execution. Originally, OpenMP only supported parallel processing in shared memory systems [155] but the latest specification (4.0) also supports offloading computation to other devices [156]. The execution model is thread based, with a *host device* and zero or more *target devices*, but data-level directives are also available; thus supporting both explicit TLP and DLP. OpenMP has a relaxed consistency, shared-memory model where the programmer is responsible for data consistency and avoiding race conditions. Each thread is allowed to have it's own *temporary* view of main memory, whether this be in registers, cache or some local store. Each thread can also have *threadprivate* memory. Several directives are provided to enable the programmer to maintain consistency between memory views and to avoid race conditions:

- The `critical` directive restricts execution to the associated block to a single thread.
- The `ordered` directive sequentialises loop iterations, which allows outer regions to still operate in parallel.
- The `barrier` directive must be executed by all threads, executing within a parallel region, before any can continue beyond the barrier.
- The `flush` directive forces the thread to write it's temporary value to main memory.
- And `atomic` is used to specify that a storage location is accessed atomically.

Constructs are supplied with various clauses which inform the runtime on parameters such as thread count, the target device, data share (*shared* and *private*) and can also make their execution conditional. The `parallel` construct identifies a parallel region of code. When a thread encounters a `parallel` construct, it creates a *team* of threads, of which it becomes the master, to execute the region. The number of threads remains constant throughout the execution of the region and so if one thread terminates, all threads terminate. Within parallel regions, other constructs can be used to further distribute the execution amongst the team members. Loop constructs (either `omp for` or `omp do`) enables the loop to be split into *chunks* to each be executed by the threads within the team. Loops can also be vectorised using the `simd` construct. The `sections` construct encloses multiple `section` constructs which each get executed by only one of the threads. A `single` construct

specifies a region that is only executed once, by a single thread. All these constructs have an implicit barrier proceeding them which blocks execution of the master thread until all threads executing the construct have completed. This can however be overridden using a `nowait` construct.

The `target data` and `target` are used to setup data for a *target device* and execute a region on a specified, or not specified, device. The following constructs are designed for sharing a large number of threads, mainly on a target device which is likely to be a highly multithreaded GPU. The `teams` construct creates a *league* of thread teams to execute the region. The `distribute` construct can then be used within the region to distribute iterations of a loop across the teams. Parallelism can be further enhanced by using the `distribute simd`, which distributes the iterations in the same way, but also executes using SIMD instructions. The `distribute parallel` loop construct distributes the iterations of the outer loop across teams, with the resulting loop iterations being further distributed across the threads within the team; and this too has a SIMD counterpart.

3.4.2.2 CUDA

The Compute Unified Device Architecture (CUDA) is a parallel computing platform and programming model, developed by NVIDIA, to enable general purpose computing on their GPUs [157]. A CUDA program is split into two components: the *host* and the *device* code. CUDA C extends C by allowing the programmer to define functions, called *kernels*, to represent a CUDA thread. The programming model assumes that the CUDA threads execute on a physically separate device, such as a NVIDIA GPU, while the rest of the program runs on the host CPU. CUDA threads are executed in parallel across the many parallel ALUs (CUDA cores) of NVIDIA GPUs. The threads are organised in thread blocks, which are parallel groups of work, with each block containing up to 1024 threads. The blocks are organised into a *grid* and the device can choose any order in which to execute the blocks as they are completely parallel to one another. Built-in variables can be used within the kernel to identify the thread within the execution space; which can be split into three dimensions. The `threadIdx` variable is used to read the thread index, the `blockIdx` for the block index and the dimension of the thread block is accessed using the `blockDim` variable.

The CUDA memory model is partitioned so that each thread has private local memory, each thread block has shared memory that is visible to all the threads within that block, and global memory which is accessible to all threads. There are also two additional global memory spaces that are read-only: constant and texture memory. The programming model also assumes that both the host device maintain separate memory spaces in DRAM. As some memory is shared and threads within a block can execute in any order, a method of thread synchronisation is provided. The intrinsic `__syncthreads()` function is used within

the kernel to synchronise the threads as it acts as a barrier at which all threads in the block must execute before any is allowed to proceed.

As well as CUDA C, kernels can be written in NVIDIA's assembly form called *PTX*. NVIDIA's CUDA compiler, `nvcc`, is used to compile CUDA C or PTX kernels into a binary format and can do this offline or at runtime. In offline mode, nvcc first separates the host and device code and compiles the device code into PTX form. The host code is modified by inserting the necessary calls to the runtime library to load and launch the compiled kernel. The compiler can output the modified C code or continue and produce an object file. The runtime library is also used by the programmer to transfer memory between the host and the device and can provide low level access to the driver if the programmer requires. JIT compilaton loads the PTX code and compiles it for the target and automatically caches a copy of the generated binary to avoid repeating compilation in subsequent invocations of the application.

3.4.2.3 OpenCL

Table 3.2: CUDA and OpenCL naming conventions

Programming Term	**CUDA Name**	**OpenCL Name**
Vectorisable Loop	Grid	NDRange
Body of vectorisable loop	Thread Block	Workgroup
Sequence of SIMD lane instructions	CUDA Thread	Work item
Thread of SIMD instructions	Warp	Wavefront

OpenCL, like CUDA, is a programming language and execution framework designed to allow programmers to offload compute intensive codes (*kernels*) to accelerators [17][158]. There is close correspondence between OpenCL and its CUDA counterpart as shown in Table 3.2. OpenCL programs are split into two parts: host and device code. The host code can be written in a variety of languages (C, C++, with bindings for Python [159], Java [160] amongst others) and runs on the host CPU; the purpose is to discover accelerator devices, submit work and manage data transfers to and from them. The host program is also free to make use of all the native parallelism, in the form of the OpenMP and MPI APIs, which are supported in the host environment. The OpenCL programming model views the accelerator device as consisting of compute units (CUs), each containing multiple processing elements (PEs). OpenCL explicitly views the application parallelism in the form of an N-dimensional index space where each point (work-item) within that space is an instance of a small function called a kernel. Kernels represent the smallest unit of work and will be executed thousands of times across the device with a work-item assigned to a PE. Work-items are grouped together

in a second-tier hierarchy to form workgroups and these are assigned to CUs. Kernels can be compiled offline and loaded onto the accelerator as a machine object, or at runtime where better optimisations can be made with knowledge of the specific architecture. In the latter case, a runtime compilation framework is assumed to be in place. There is also a specification for a device independent intermediate representation for kernel code distribution called SPIR [161].

The language, an extension of C99, explicitly supports both DLP and TLP. This is achieved by exposing DLP through the inclusion of vector data types and by expressing the code as a thread. This is further supported through built in functions for thread synchronisation and key mathematical operations. The `barrier` function enables thread synchronisation between work-items within a workgroup, however synchronisation between workgroups is not permitted. The memory model is split into four address spaces:

- Constant: A section of the global memory that stays constant during kernel execution.
- Private: PE Scope: A section of memory allocated per PE and inaccessible by any other PE
- Local: CU Scope: A section of memory accessible by all PEs in the current CU but inaccessible by other CUs
- Global: Device scope: A section of memory visible to all PEs, across all CUs. Potentially cached.

The OpenCL 2.0 specification introduced the shared virtual memory (SVM) model which extends the global memory region into the host memory region. This allows pointers to be passed between the host and device which can remove the requirement of data transfers. There are three possible types of SVM in OpenCL: coarse-grained buffer, fine-grained buffer and fine-grained system. Coarse-grained buffers enable kernel instances to share pointer based structures with the host program with consistency being enforced using map/unmap commands. Fine-grained buffers offer sharing at the load/store level within buffer objects with memory consistency being guaranteed at synchronisation points. Finally, the fine-grained system model enables individual load/stores to access anywhere within the host memory, and again, consistency is guaranteed at synchronisation points.

3.4.2.4 HSAIL

The Heterogeneous System Architecture (HSA) [162] is designed to support a wide variety of data- and thread-parallel programming models, and is particularly amenable to OpenCL. HSAIL is the intermediate language (IL) used by HSA. The purpose of HSA is to provide

programmers with a virtual machine (VM) to make heterogeneous code more portable and also to make it possible to tightly integrate compute elements in a system. Compute elements that participate within the HSA memory model are called *agents*. An HSAIL VM consists of multiple agents, including at least one host CPU and one *HSA component* which communicate via *Architected Queuing Language* (AQL) queues. The host CPU runs the operating system as well as the HSA runtime. An HSA component is an agent which supports the HSAIL instruction set and the AQL packet format and contains multiple compute units. The host CPU can send commands to components using the AQL queues, and components can also send commands to each other, and themselves, via the same method. As the format is portable and targets a VM, the IL has to get compiled, or *finalised*, to execute on a target architecture. The finaliser can be invoked at various times: statically at build time, when the application is installed, when it is loaded or during execution.

Like OpenCL, HSAIL programs are written in kernels which each represent a *work-item* and represent a point of execution within a larger, multidimensional *grid*. The grid is split into smaller parts called *workgroups* with work-items within a workgroup being gang-scheduled in wavefronts, in SIMT style, so the programming model and naming conventions are the same as OpenCL. The AQL queues also offer the same functionality as command queues do in OpenCL. The memory model is also similar to OpenCL 2.0, with a unified flat model where all agents can use the same pointers and a pointer can address any segment of memory. The seven segments are:

- *Global* that is visible to all work-groups and to all agents.
- *Group* that is visible to a single work-group.
- *Private* that is visible to a single work-item only.
- *Kernarg* which is read-only memory used to pass arguments into a kernel.
- *Readonly* remains constant during the execution of a kernel and can be treated as part of global memory.
- *Spill* is used to load or store register spills and is visible to a single work-item only.

Unlike OpenCL, HSAIL defines a language that is at a much lower level and resembles RISC assembly; more like OpenCL's SPIR language format. The three-address format operates on registers and the specification defines a virtual register set with four types:

- 8 1-bit control registers for comparison results and branches.
- 128 32-bit registers for signed/unsigned integers and floating-point values.

- 64 64-bit registers for signed/unsigned long integers and double float values.
- 32 128-bit registers for holding packed data.

3.4.2.5 Renderscript

Renderscript is a framework developed by Google for running computationally intensive tasks on Android [163]. The runtime parallelises work across all the processors available in a device including the GPU and DSPs. It is primarily designed for data-parallel applications such as image processing and computer vision. Like CUDA and OpenCL, the offloaded parts of algorithms are written in an extension of C, specifically C99, and are written as *kernels*. The kernels reside within a script file which can also contain global values, static globals, invokable functions and static functions. The invokable functions are single-threaded Renderscript functions that are callable from the Java program and are useful for inital setup; though a dedicated `init` function is also provided which is invoked automatically when a script is first instantiated. Global values are also accessible by the Java program, however static globals and functions are not.

A kernel is a parallel function that executes across every *Element* with an *Allocation*. Allocations are objects that hold a fixed amount of data and are used to pass arguments to the kernel to receive the result. The result of the computation is returned by the kernel function. Google provide an API within Android for setting up and running Renderscript kernels. A runtime library is also supplied with contains a large collection of math functions and data type definitions for vectors.

3.4.3 OpenCL Compilation

OpenCL has become widely adopted by industry, and though the specification is target agnostic, GPUs are still the primary target for application developers. This is because most compute systems with a user interface will contain a GPU. However, much research has been conducted into enabling OpenCL execution on different platforms and architectures and this section will summarise them.

3.4.3.1 OpenCL on Multi-core CPUs and DSPs

The MCUDA framework was developed to enable CUDA programs on multi-core CPUs[164]. As the overhead of managing and executing thousands of threads on a CPU can have a detrimental effect on performance, the unit of work was increased to the thread block. Loops were introduced to run the CUDA threads serially; *deep fission* being used to maintain synchronisation statement semantics within control structures. Such synchronisation points are control statements such as `gotos` and `labels` and conditional regions that contained

the `__syncthreads` command were also partitioned. For variables live past synchronisation points, the authors selectively replicated the necessary thread dependent variables by expanding them into arrays and simply removing the shared keyword from thread block variables; since these are not private to each thread. MCUDA used POSIX threads to issue thread blocks across the cores of the CPU.

Twin Peaks was a software system designed to better utilise the system resources by executing kernels on CPUs as well as GPUs, taking into consideration the memory hierarchy of the CPU [165]. The aim was to use the CPU for smaller kernels as the communication overhead between the CPU and GPU address spaces is significant. The framework assigned a single CPU thread to a workgroup, utilising all the cores until all the workgroups had completed. In the absence of any barriers each work-item was completed and then another work-item scheduled. Whereas in the presence of barriers; the `setjmp` function, from the C standard library [166], was used to save the program state before executing the next work item. Once all the work-items had reached the synchronisation point, the `longjmp` function was used to restore the context of the work-item to carry execution of the kernel.

A framework was devised to enable OpenCL capability on heterogeneous multi-core system with local memory, such as the IBM Cell BE processors [167]. The CBE contains one general-purpose processor core (GPC) and multiple accelerator processor cores (APCs), with the GPC generally performing system management tasks while the APCs are dedicated to compute-intensive workloads. The APCs are connected via a bi-directional interconnect (ring each way), each having DMA-driven local memory with software managing data coherency amongst the APCs. A similar technique to MCUDA was used to transform the source code to embody the kernel body within a triple nested loop, a technique known as *work-item coalescing*. Private variables live beyond the scope of the nested regions were expanded into arrays, but the authors also used a *web* of variable values to reduce memory usage. This web comprised of all the du-chains of a variable across all the threads, and so illustrated where threads shared common values for that variable. For where they shared the same value, it was unnecessary to create a thread private value.

The PACDSP is a five-way, dual-clustered VLIW DSP core with SIMD instructions and a distributed register file; each cluster contains a load/store unit and an ALU, with the fifth execution slot utilised by a shared scalar unit [168]. The PACDUO is a platform with a dual-core PACDSP coupled to an ARM core [169]. OpenCL is enabled on this device through three source transformations:

- kernel serialisation - create a multi-dimensional loop around the body of the kernel.
- vector translation - the compiler is for C/C++ and not OpenCL, therefore the researchers had to translate OpenCL vector types to C++ classes with accompanying

overloaded functions which used target intrinsics to operate upon them.

- kernel vectorisation - use of software thread integration [170] to merge conditional statements of concurrent threads and use intrinsics to assign work-items to explicit clusters.

3.4.3.2 OpenCL on FPGAs

FCUDA was built upon the work of MCUDA, but instead of using CUDA to target CPUs, it used HLS targeting FPGA silicon [171]. The framework uses the same methods of serialising kernels as MCUDA but also makes use of annotations (synthesis directives) on the kernel source to drive HLS. AutoPilot [172] was used as the HLS tool and the flow includes transformation of kernels into AutoPilot C; the latter is a subset of C designed for hardware synthesis. Such annotations are `FCUDA COMPUTE` and `FCUDA TRANSFER`, with the annotated sections extracted into their separate functions; known as task functions by AutoPilot. `FCUDA COMPUTE` functions perform computation proper while `FUDA TRANSFER` functions contain data communication tasks; which usually inferred DMA burst transfers between off-chip memory and on-chip BRAM arrays. These directives were also used as synchronisation points along with the existing statements from MCUDA. The task functions were then called from the constructed workgroup loop with the custom pragmas `AUTOPILOT REGION` and `AUTOPILOT PARALLEL`, which specified the multiplicity of parallel processing cores.

MARC was a many-core architecture developed by researchers at Berkeley, comprising of a single control processor and a variable number of algorithmic processing cores [173][174]. The control processor was a simple RISC CPU while the algorithmic cores were simplified MIPS cores with fine-grained multithreading and an extensible ISA. A barrier-like instruction was added to the ISA alongside an atomic swap instruction for inter-kernel communication via shared memory. The private memory was implemented in distributed LUT RAMs with local and global memories residing in block RAMs. The global memory size could be extended by using external memory and the compilation architecture was based on LLVM. The researchers designed application-specific processing cores by transforming the LLVM IR instructions to an optimised, predicated SSA form, directly mapping to pre-determined hardware primitives.

POCL is a portable OpenCL implementation used within the TTA-based Codesign Environment (TCE) which targets a configurable TTA processor in which both the host and device codes are merged into a single program [175]. The target architecture is configurable in the number and mix of functional units as well as having the capability of custom instructions to help accelerate the given algorithm. After the user has specified any custom operations, the system iteratively adds in functional units and register files, to satisfy the

computational requirements (and ILP) of the kernel. A set of closely related low-level LLVM passes, that operate on the IR level, are used to modify the kernel to chain several instances together, analogous to loop unrolling. These passes also maintain the parallel semantics that OpenCL kernels explicitly provide, while code size is kept under control by setting an upper limit on the number of chained instances; any instances above the limit were rolled into a loop.

SOpenCL is an architectural synthesis tool that also maps OpenCL kernels to FPGA fabrics [176][177]. The tool uses an architectural (hardware) template that can be instantiated to match the target application dataflow using a network of FUs, stream units and distributed control logic to reconfigure the datapaths between producer and consumer units. The compilation front-end uses source-to-source transformations to convert the OpenCL kernel into a C function, while also coarsening it to represent a workgroup instead of a work item. The kernels are limited to containing one nested loop only and are coarsened by encapsulating the kernel within the body of a triple-nested loop to represent the three possible grid dimensions. Loop fission is used at barrier call locations and the system conducts live variable analysis to identify live variables beyond synchronisation points; for these, variable privatisation is used to create copies for each thread. The coarsened kernel is optimised and converted into a single basic block through if-conversion [60]. The code is then split into three slices, to generate three parts of the system:

- the input stream, which loads data and calculates the required addresses,
- the output stream, which stores output data and calculates the required addresses,
- the computation kernel which is the accelerator core performing, consuming input data and producing the output stream.

A similar approach was taken to create an OpenCL compiler for a coarse grained reconfigurable architecture (CGRA) [178]; specifically the SRP from Samsung, which is a VLIW architecture coupled with a CGRA. SRP has a simple memory hierarchy, using a scratchpad memory instead of a data cache. The CGRA is used to accelerate the kernel and consists of an array of PEs such as FUs and register files connected by dedicated connection buses. The compilation framework serialises the work-items (Kernel code) into loops via source-to-source transformations and in the process, it re-writes it in standard C. The loops of that C application are then unrolled and modulo-scheduled to fully utilise the available functional units of the VLIW engine and the CGRA.

Altera has been the first adopter of OpenCL for their FPGA silicon [179] with the aims of reducing the very steep learning curve of high-throughput FPGA design while ensuring that algorithms are portable across different FPGA families and computation silicon. Altera's

OpenCL compiler (ACL) transforms OpenCL kernels into deeply pipelined circuits to be mapped onto the FPGA fabric. The pipelined design allows for thread data to be clocked in sequentially so that each stage of the pipeline can be used by different instances of the work-item. Multiple pipelines can also be instantiated in parallel to further increase throughput. As well as the kernel computation circuit, the compiler also creates memory interfaces: global loads and stores are performed using LSUs connected via a global interconnect to off-chip DDR DIMMS, whereas local accesses target on-chip static RAMs. The vendor has shown that, in some cases, FPGA implementations can be significantly faster and more efficient than CPUs and GPUs [180]. For fractal video compression the FPGA design was 3x faster than the GPU and 114x faster than the CPU, while consuming 12% of the GPUs power and 19% of the CPUs.

Prior to the very recent announcement for OpenCL support from Xilinx in the Vivado design suite 2014.2 [181] research was undertaken to convert kernels into AutoESL C code and subsequently, synthesise them to silicon [182]. The Clang AST libraries were used alongside Graphtool to transform these kernels for processing by the synthesis tool; this involved converting barrier calls to barrier hit and barrier done signals as well as creating interfaces to the block RAMs for the kernel arguments. The researchers used a Convey HC-1 hybrid system consisting of an Intel Xeon CPU and four Virtex-5 (XC5VLX330) FPGAs, the CPU acting as the host while each FPGA performed as a compute unit in the compute device. The global memory was implemented in DDR2 modules, on-chip block RAMs were used for the local memory and finally, registers were used for private memory. High-level and logic synthesis were performed by Xilinx AutoESL and ISE respectively with compilation performed offline due to excessive runtimes; in this system, work size and dimensions are fixed at compile time.

Very recent research has been conducted on improving the speed of HLS from OpenCL programs by using virtual coarse-grained reconfigurable contexts [183]. The authors use *intermediate fabrics* (IFs [184]) which provide the virtual coarse-grained resources atop a physical FPGA. The IFs map behavior onto application-specialised resource, such as floating-point units, instead of thousands of LUTs. To create an IF, the OpenCL kernels are compiled into LLVM IR and custom intrinsics are used for OpenCL builtin functions. The IR is then used to create a control dataflow graph, mapping LLVM instructions to compatible cores provided by a user-specified library. The framework analyses the requirements of kernels and clusters the kernels into reconfiguration contexts, based on their functional similarity. Each context can implement one kernel at a time, time-multiplexing instances of it, with the work-items being carefully pipelined to exploit data reuse. The clustering enables order-of-magnitude faster compilation and reconfiguration between kernels executions. For when the current context does not support a kernel, an alternate context is loaded via an

existing bitfile. The aim of the clustering algorithm is to maximise the number of resources reused across the kernels in a context while minimising the area of individual contexts. The authors report a compilation speedup of 4,211x while incurring a total of 1.8x additional area requirement to implement a system of 20 kernels, compared to traditional synthesis techniques.

3.5 Discussion

Currently, the primary development in programming languages is to improve built-in parallel semantics as well as enabling the use of multiple different architectures as accelerators. The languages covered in this chapter have mainly focused on ones that extend or support C/C++ development as they remain the incumbent languages for the majority of tasks. C and C++ both now have thread support built into their standard library and the specification, so maybe these support will phase out the use of Pthreads in C code. If the C/C++ continue to evolve fast enough, they may be able to continue being the most popular system languages. As for the languages and platforms developed for heterogeneous, I doubt many of them will enjoy the longevity of C or C++. As CUDA is specifically for NVIDIA it naturally has limited reach and though it is popular in the scientific computing and NVIDIA have a strong presence in the desktop markets, key companies such as Apple and Adobe ship AMD hardware and support OpenCL. Renderscript lacks any formal specification and as such mobile GPU vendors mask their OpenCL drivers with a renderscript frontend. HSAIL seems too low level, having a restricted register file seems odd and too targeted at AMD's GPUs. To me, it would make more sense for HSAIL and SPIR to converge to have a low-level representation of host/device program without the unnecessary detail that HSAIL specifies. Low-level APIs are becoming popular in graphics so I expect this to continue to the compute APIs too. As an extension to C, I believe OpenMP will still be widely used, having an already well established extension for parallel computing, as the current revision now supports overloading to accelerators.

A growing trend across the industry is the use of runtime systems and compilers to support high-level and portable programming. This can be seen with the distribution of OpenCL kernels, as source code or as the SPIR bytecode, and with Javascript. Ease of use and portability are, in many cases, the most important aspects of language selection. Patterns are emerging to suggest that more programs will be distributed in a pre-compiled bytecode format that enables portability but also takes advantage of the performance gains from statically compiled languages. This will require continued compiler development that will focus on fast transforms for runtime compilation and increasingly complex static analysis, whole program analysis and vectorisation optimisations.

3.6 Summary

New programming languages and APIs have been developed to support the accelerator paradigm, and most of them focus on threading with explicit constructs, and data types, for DLP as well. The burden of increased performance has been put on the programmer, with parallel semantics, data transfer and synchronisation to consider. The explicit parallel constructs makes the task easier to compiler engineers however, but the languages do need complicated runtime libraries to support them. The most popular open API for accelerators, OpenCL, is target agnostic. OpenCL has been shown to run across a variety of architectures: from GPUs, multi-core CPUs, DSPs and FPGAs.

Chapter 4

Identification of Research Area

4.1 Chapter Objectives

The purpose of this chapter is to discuss, relative to Chapter 2 and 3, the area of research this thesis focuses upon.

4.2 Compiling for VLIWs

Compilation techniques that target ILP exploitation, for both VLIW and superscalars, have been out of focus of research since the release of the EPIC architecture. Aggressive pursuit of ILP processors has given way to thread-level parallelism, and VLIW architectures have found their niche as DSPs where there is large amounts of parallelism within the codes. Mainstream adoption of VLIWs, for general purpose computing, was not successful for two reasons: (1) maintaining binary compatiability across microarchitectures is difficult and originally required porting, and (2) the performance was sub-par. Aggressive, dynamically scheduled, ILP processors also began to dissipate too much heat to be suitable for embedded applications. Therefore, as the hardware platform is capable of exploiting TLP, through multiple processors in a shared memory configuration, the research undertaken has focused on this instead of aiming to develop more exotic compilation methods to exploit ILP.

The SPMD programming model has been selected because it explicitly expresses TLP, which can be used to exploit DLP. The compiler will need to be capable of supporting SPMD programs, and this will be provided by implementing unique intrinsics that will be available through the custom runtime library for the LE1. As the multiple data elements in SPMD programs are spread across threads, some DLP can be converted into ILP in the compiler, by serialising the threads into loops and then using loop unrolling to enlarge the

scheduling region. Thus the compiler must still be capable of issuing multiple instructions simultaneously. It also needs to contain multiple microarchitecture targets to explore how it effects the system performance so that the configurability of the LE1 architecture can be utilised successfully. The compiler also needs to utilise the instruction set properly to speedup address computation and remove control-flow statements. Many programs will also require floating-point calculations which are not supported in hardware by the LE1, so a method will need to be devised to provide emulated floating-point functions at runtime.

OpenCL has been selected as the language and platform for the following reasons:

- It is an open specification;
- The language is supported by Clang and LLVM;
- The programming model and language explicitly represents parallelism; and
- The capability of JIT compilation alleviates the barrier of binary incompatibility across different microarchitectures.

To facilitate OpenCL execution using the LE1, a software platform will need to be developed to execute the host code and enable the execution of device code, with source compilation along with the necessary data transfers. A cycle accurate simulator has already been developed for the LE1 [185], along with a provided API to allow control and access to the memory, which will be used by the developed platform (driver). The LE1 compiler will need to be integrated within the driver to enable automatic compilation of OpenCL kernels, but the software developer will need to have control over which system- and microarchitecture to use. Therefore, the driver will need to contain many (hundreds) of configurations available to aid in an investigation into how microarchitecture and system architecture decisions affect the performance of specific algorithms.

4.3 FPGA OpenCL Compilation

Though the conducted research will use a simulated LE1 system, the configurability of the LE1, along with the JIT compilation model of OpenCL, lend themselves well to execution on FPGAs. The API for the simulator has been developed so that it can easily be extended to control a physical FPGA target, so the driver would also be capable of using an FPGA target. The XML file that will be generated alongside the compiler target is also sufficient to generate VHDL for a physical platform. In general, researchers have followed two routes to mapping OpenCL applications to FPGAs: a) ESL-based methodologies in which OpenCL is the input language to high-level synthesis tools, and b) Using template architectures. The

key difference is that the LE1 target architecture is not a template but a fully programmable and highly configurable/extensible embedded VLIW CMP.

SOpenCL used a template architecture for synthesis and used a source transformation that is similar to the one developed during this research, except this work is not limited to kernels with a maximum of single nested loops. The source transformation of this thesis is also similar to the one developed for the Samsung SRP, except that their transformation output the kernel in C, while the developed transformation will output valid OpenCL C. POCL uses the TTA templated architectures and uses multiple, tightly coupled passes on the LLVM IR to serialise the work-items and transform it for the ILP processor. The TCE platform, which encompassed POCL, also transformed the code so that both the host code and device code ran on the TTA target. This will not be the case in this research as the developed software is intended to enable heterogeneous computing using the LE1. This thesis will also focus on a two-phase source-to-source transformation that is also target agnostic.

The MARC processor was not a template architecture, but a configurable, many-core architecture. The host was a simple RISC CPU which was coupled with custom, multithreaded cores as the accelerators which utilised a custom instruction to handle the OpenCL barrier functions. The two differences between this approach and this research are that (1) the LE1 has not been configured to use multithreading and it is a VLIW microarchitecture and (2) the barrier functions will be removed during the source transformation as the work-items are serialised and partitioned.

4.4 Summary

This chapter presented the areas of research which will be the focus of this thesis. Initially work will be carried out to create a compiler target for the configurable LE1 VLIW CMP, followed by a driver to enable automatic runtime compilation. A source-to-source transformer will be developed to transform the OpenCL kernels into a form that is suitable for effective execution on the VLIW CMP. A script will also be implemented to generate both the compiler target and the hardware description for the simulator, which could further be used to create VHDL for a real FPGA target. Finally, a test suite will be used to extensively analyse the performance of the compiler, across many microarchitectures, as well as the driver's ability to schedule work across multiple cores. The key areas of work are:

- Develop an LLVM compiler backend for the LE1, including compiler intrinsic functions to support OpenCL kernels.
- Develop a userland driver to encompass the compiler to enable automatic compilation of kernels, as well as controlling data transfers.

- Transform OpenCL kernels into another SPMD form, more suitable for the LE1 VLIW CMP.
- Implement a runtime library to support execution of the kernels on the LE1.
- Develop a method to schedule OpenCL workgroups across the multiple cores of the CMP.
- Investigate how the choices in microarchitecture and system architecture effect the performance of the system.

Chapter 5

The LE1 Compiler

5.1 Chapter Objectives

This chapter describes the compiler for the LE1, which is based on LLVM, and used within the driver to compile OpenCL kernels to the assembly language of the LE1. An overview of Clang and LLVM will be given before the details of code generation for the LE1. The LE1 code generator is the backend part of the LLVM framework that handles the target-specific passes of a compiler. The scheduling and register allocation is performed using the default passes that are part of the LLVM code generator that uses target-specific information but a target-independent algorithm. The results presented in Chapter 6 use both the 'stable' code generator, which is based on LLVM 3.2, and a development branch that is based on LLVM 3.4. The development branch was begun to help improve the performance of the compiler but it is not as complete as the stable version. Where the methods differ within the two code generators, both methods will be discussed.

5.2 Introduction to Clang and LLVM

LLVM is the umbrella project that encompasses a modular, library-based, framework for program analysis, optimisation and compilation [186]. LLVM has a well defined IR, enabling the three main stages of compilation ('front', 'middle' and 'back') to be effectively decoupled from one another [187]. This allows the compiler optimisations to be used by multiple frontends, supporting different languages, and code generators for different target architectures. It has also been designed from the bottom-up to be a flexible set of libraries, so that parts of the framework can easily be incorporated into other projects as required. The optimisation and analysis passes are compiled into archive libraries and are individually

accessible to a program which wishes to use them. This is to say that the LLVM libraries don't *do* anything by themselves, some form of driver is required to utilise the libraries.

Clang is the official frontend for LLVM, it too is a set of libraries but also encompasses a compiler driver to utilise LLVM [188]. Clang supports C-based languages such as C++, Objective C and OpenCL. Clang performs the task of parsing and lexical analysis of the source code, representing the code as an AST. Storing the code in an AST enables source-to-source transformations to be performed and this functionality is used by the driver, which is discussed later in Section 6.3. On the input of legal code, Clang tranforms the AST into the LLVM IR which is then passed into the target independent optimisation phase. The optimised IR is then used by the selected backend to perform target dependent optimisations and code generation, as shown in Figure 5.1.

Modifications to Clang to support the LE1 have been minimal, their purpose has been to inform the compiler driver that the LE1 exists with simple information; such as it's endianness, supported data types and the intrinsic functions used by the OpenCL runtime library. All the real work is contained within the LLVM backend which is responsible for generating assembly code from the LLVM IR.

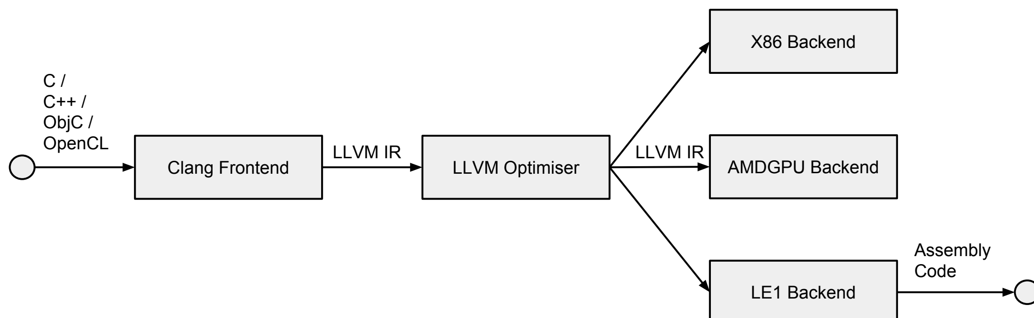

Figure 5.1: Three phase compilation with Clang and LLVM.

5.2.1 Program Representation

The LLVM code representation is available in three different forms: as a human-readable assembly language, an in-memory compiler IR and as an on-disk bitcode file. LLVM bytecode represents the program at a relatively low level, which is where the original name, *low level virtual machine*, comes from. Though the IR is at a low level, it has to be generic enough to be useful in a retargetable compiler. This is enabled by encoding target and platform dependent type information into the module in a `target datalayout` object. It also has a typed representation that can contain arbitary length integer types, general floating-point

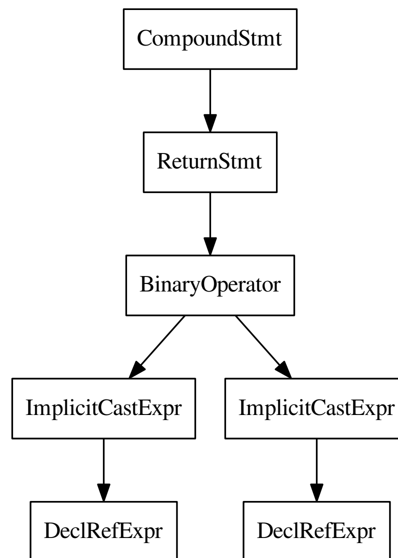

Figure 5.2: Graphical example of an AST produced by Clang.

types (half, float, double, fp128), pointers, vectors, arrays and structures. As the compilation process advances, the program is represented in different forms which allow more specific optimisations to be performed.

The human readable form looks similar to the assembly language of a RISC machine as it is in three address form and uses load and store operations to access memory. The key differences are that the IR has an infinite number of registers and uses SSA form. The contents of both the assembly and bitcode file define a *module*, which is the top level data structure. A module then contains a sequence of functions as well as global variables, function prototype declarations and the target data layout. The functions within the module contain a sequence of basic blocks, which contain sequences of instructions. In-memory representation models the assembly and bitcode format closely. It uses a `Module` object to aggregate all of the data from the translation unit. Modules contain instances of the `Function` class, which contain instances of the `BasicBlock` class, which contain instances of the `Instruction` class.

The LLVM IR is created from an AST when using Clang as the frontend. Figure 5.2 is an example of Clang's AST for a function which accepts two integer values and returns the result of their addition. Clang AST represent declarations (`Decl`), statements (`Stmt`)

and types (`Type`) and all nodes in the tree inherit from one of these classes. `CompoundStmt` objects contain other statements and expressions; in this case a `ReturnStmt` which uses a `BinaryOperator` expression, while the `ImplicitCastExpr` nodes are used for every use of a variable to cast it to its required type. As the AST represents the program close to the source code form, the nodes contain `SourceLocation` objects which are used to provide detailed debugging information back to the user and allows the source code to be rewritten. The AST is not designed to be modified, but designed to enable clients to traverse the original source code and rewrite it in a buffer.

5.2.2 TableGen

Much of the target architecture information is contained with '.td' files, which are parsed by a program called 'TableGen'. TableGen is a program developed for LLVM to help develop and maintain records of domain-specific information, including the register files, the instruction set and the instruction selector, and other architecture features. Information described in TableGen files are considered as 'records' and are split into two types: 'classes', which are abstract records that are used to build and describe other records, and 'definitions', which are a concrete form of a record. Classes can be used to build definitions by leaving some of their fields as variables to be completed when a definition uses the class. Multiclasses are groups of abstract classes that can result in multiple definitions. As the name of the program suggests, the resulting data structures are tables that are accessed by specific TableGen backends with instruction selection being one of the most complicated.

5.3 Instruction Selection

Instruction selection in LLVM is performed using DAGs, specifically a `SelectionDAG`, where each node is a `SDNode` with the key attribute being the operation code (opcode). The backends perform the task over several phases: target legalisation, target lowering and machine instruction selection. The selection is an iterative process, consisting of several steps:

- Build initial SelectionDAG from the LLVM input code.
- Optimise SelectionDAG.
- Legalise SelectionDAG types by transforming the nodes to eliminate any types that are unsupported by the target.
- Optimise the SelectionDAG by removing redundancies induced through the legalisation step.

- Legalise Selection operations by transforming nodes to remove any operations that are unsupported by the target.
- Optimise the DAG to remove inefficiencies introduced by the operation legalisation step.
- Perform instruction selection to convert the DAG into one of target-specific instructions.

The legalisation and preliminary lowering occurs within `LE1ISelLowering.cpp`. The legalisation converts the original DAG to use only legal types for the target architecture, while target lowering enables the insertion of custom nodes as well as handling target specific situtations such as calls and returns. Figure 5.3 shows the graphical representation of a `SelectionDAG` after target lowering and legalisation, which is the form that instruction selection is performed upon. The DAG represents a small function that multiplies two of the integer arguments and places the result at the address of the third argument. Opcodes, operand locations and result types are all depicted within the nodes. The blue dashed lines represent a partial ordering to the nodes, and are called 'chain' values, they are used to maintain non-data dependencies such as control-flow and an order between memory operations.

The instructions, listed in Tables 5.1, 5.2, 5.3, 5.4 and 5.5 are described within the TableGen file, `LE1InstrInfo.td`, in which most instructions are matched automatically by a TableGen backend. The LLVM system has many in-built opcodes which match almost 1-1 with the instructions in the IR. Many of LE1's instructions have used the standard nodes, such as arithmetic and logic operations and most of the other instructions can be described using a combination of the LLVM opcodes. This allows patterns to be described in the form of DAGs, effectively a tile to be matched in the `SelectionDAG`. For instructions which need custom handling, custom SDNodes need to be created and the instruction selection generally performed in `LE1ISelDAGToDAG.cpp`. Instruction selection between the stable and development branch by two factors:

- conditional branches are handled by TableGen pattern matching in the stable branch, whereas they are handled in the target lowering phase in the development branch.
- the development branch is aware that some immediates are encoded within the instruction word and some require an additional 32-bit immediate.

Table 5.1: LE1 Arithmetic Operations.

Operation	**Description**	**C Macro**
ADD	Add	(s1) + (s2)
ADDCG	Add with carry and generate carry	t = (s1) + (s2) + ((cin) & 0x1);
		cout = ((cin) & 0x1) ?
		(UINT32(t) <= UINT32(s1))
		: (UINT32(t) <UINT32(s1));
AND	Bitwise AND	(s1) & (s2)
ANDC	Bitwise complement and AND	∼(s1) & (s2)
DIVS	Division step with carry generate	unsigned tmp = ((s1) <<1) | (cin);
		cout = UINT32(s1) >>31;
		t = cout ? tmp + (s2) : tmp - (s2);
MAX	Maximum signed	(INT32(s1) >= INT32(s2)) ? (s1) : (s2))
MAXU	Maximum unsigned	(UINT32(s1) >= UINT32(s2)) ? (s1) : (s2)
MIN	Minimum signed	(INT32(s1) <= INT32(s2)) ? (s1) : (s2)
MINU	Minimum unsigned	(UINT32(s1) <= UINT32(s2)) ? (s1) : (s2)
OR	Bitwise OR	(s1) | (s2)
ORC	Bitwise complement and OR	(∼(s1)) | (s2)
SH1ADD	Shift left 1 and add	((s1) <<1) + (s2)
SH2ADD	Shift left 2 and add	((s1) <<1) + (s2)
SH3ADD	Shift left 3 and add	((s1) <<1) + (s2)
SH4ADD	Shift left 4 and add	((s1) <<1) + (s2)
SHL	Shift left	(INT32(s1)) <<(s2)
SHR	Shift right signed	(INT32(s1)) >>(s2)
SHRU	Shift right unsigned	(UINT32(s1)) >>(s2)
SUB	Subtract	(s1) - (s2)
SXTB	Sign extend byte	UINT32((INT32((s1) <<24)) >>24)
SXTH	Sign extend half	UINT32((INT32((s1) <<16)) >>16)
ZXTB	Zero extend byte	((s1) & 0xff)
ZXTH	Zero extend half	((s1) & 0xffff)
XOR	Bitwise exclusive OR	(s1) ˆ (s2)

Table 5.2: LE1 Control Operations.

Operation	Description
GOTO	Unconditional relative jump (byte-aligned, 20-bit offset from PC)
CALL	Unconditional relative call (byte-aligned, 20-bit offset from PC)
BR	Conditional relative branch on true condition (byte-aligned, 16-bit offset from PC)
BRF	Conditional relative branch on false condition (byte aligned, 16-bit offset from PC)
RETURN	Pop stack frame and goto link register

Table 5.3: LE1 Memory Operations.

Operation	Description
LDW	Load word (word aligned)
LDH	Load halfword signed (half-word aligned)
LDHU	Load halfword unsigned (half-word aligned)
LDB	Load byte signed (byte aligned)
LDBU	Load byte unsigned (byte aligned)
STW	Store word (word aligned)
STH	Store halfword (half-word aligned)
STB	Store byte (byte aligned)

Table 5.4: LE1 Multiplication Operations.

Operation	C Macro
MULLL	(s1) * INT16(s2)
MULH	(s1) * INT16((s2) >>16)
MULHS	((s1) * INT16((s2) >>16)) <<16
MULLU	(s1) * UINT16(s2)
MULHU	(s1) * UINT16((s2) >>16)
MULLLL	INT16(s1) * INT16(s2)
MULLH	INT16(s1) * INT16((s2) >>16)
MULHH	INT16((s1) >>16) * INT16((s2) >>16)
MULLLU	UINT16(s1) * UINT16(s2)
MULLHU	UINT16(s1) * UINT16((s2) >>16)
MULHHU	UINT16((s1) >>16) * UINT16((s2) >>16)

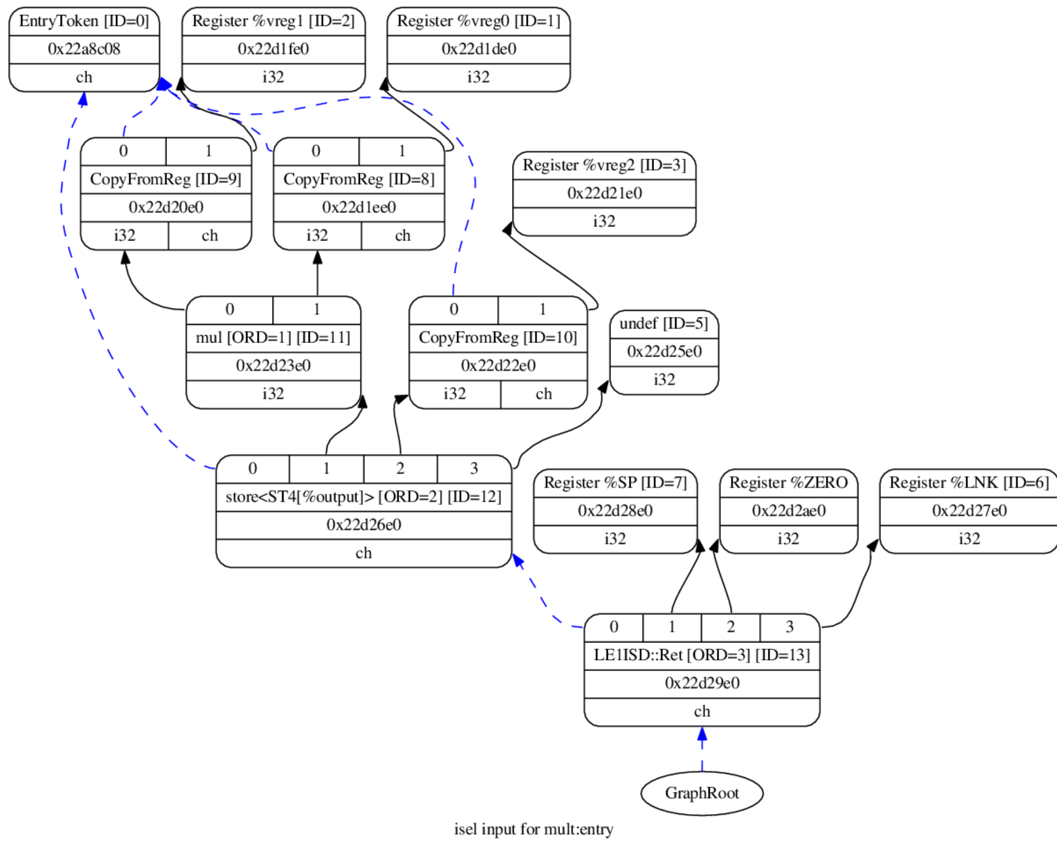

Figure 5.3: Example of a SelectionDAG after target lowering and legalisation.

Table 5.5: LE1 Logical Operations.

Operation	**Description**	**C Macro**
CMPEQ	Compare (equal)	((s1) == (s2))
CMPGE	Compare (greater equal - signed)	(INT32(s1) >= INT32(s2))
CMPGEU	Compare (greater equal - unsigned)	(UINT32(s1) >= UINT32(s2))
CMPGT	Compare (greater - signed)	(INT32(s1) >INT32(s2))
CMPGTU	Compare (greater - unsigned)	(UINT32(s1) >UINT32(s2))
CMPLE	Compare (less than equal - signed)	(INT32(s1) <= INT32(s2))
CMPLEU	Compare (less than equal - unsigned)	(UINT32(s1) <= UINT32(s2))
CMPLT	Compare (less than - signed)	(INT32(s1) <INT32(s2))
CMPLTU	Compare (less than - unsigned)	(UINT32(s1) <UINT32(s2))
CMPNE	Compare (not equal)	((s1) != (s2))
ANDL	Logical AND	((((s1) == 0) | ((s2) == 0)) ? 0 : 1)
NANDL	Logical NAND	(((s1) == 0) | ((s2) == 0)) ? 1 : 0
NORL	Logical NOR	(((s1) == 0) & ((s2) == 0)) ? 1 : 0
ORL	Logical OR	(((s1) == 0) & ((s2) == 0)) ? 0 : 1
SLCT	Select on true condition	UINT32(((s1) == 1) ? (s2) : (s3))
SLCTF	Select on false condition	UINT32(((s1) == 0) ? (s2) : (s3))

5.3.1 Instruction Legalisation and Lowering

The task of the instruction lowering and legalisation phases is to transform a DAG of LLVM SDNodes into a form that may contain target-specific SDNodes and only contain legal types for the target. Three actions can be performed for the legalisation of instructions and data types: 'promote', 'expand' and 'custom'. The promote action enlarges the data type to the next largest legal data type for the set operation, whereas expansion does the opposite and breaks a data type into smaller types until a legal type is found (such as breaking vectors into smaller vectors or scalars). For more complex handling, custom functions can be defined to lower the IR into a form suitable for the target. This phase also handles the calling conventions and the insertion of possible calls to runtime libraries. For the LE1 backend, this is the phase that comprises:

- All integer types are converted into the two legal types: the boolean 'i1' type that is stored within predicate registers, and 'i32' which are stored in the general purpose registers.
- Divide operations are custom lowered to use LE1 specific SDNodes.
- Floating point operations are converted into calls which emulate the functionality with integer operations.
- Call and return ABIs are implemented.
- Intrinsic functions are also lowered.

```
// 32-bit integers are stored in the GPRs, boolean values are stored in the branch registers
addRegisterClass(MVT::i32, &LE1::CPURegsRegClass);
addRegisterClass(MVT::i1, &LE1::BRegsRegClass);
// ROTL is not part of the LE1 instruction set so expand into a combination of other operations.
setOperationAction(ISD::ROTL, MVT::i32, Expand);
// Integer divison is custom lowered using LE1 specific SDNodes.
setOperationAction(ISD::SDIV, MVT::i32, Custom);
// Replace 32-bit FMUL operations with calls to float32_mul.
setLibcallName(RTLIB::MUL_F32, "float32_mul");
setOperationAction(ISD::FMUL, MVT::f32, Expand);
```

Figure 5.4: Code that informs the legalisation process which types are supported by each operation and in which registers the types can reside.

The legalisation of types begins with informing the process about which types are supported by the register set of the architecture, as well as which operations support those

types. Examples of this are given in Figure 5.4. The legalisation phase defines that only 'i32' and 'i1' are natively supported and there are separate register classes to handle those types. As no register classes are defined as supporting floating-point (FP) types, LLVM defaults to replacing nodes that operate on those types with function calls to the compiler-rt library which contains functions to emulate FP functionality. For most of the FP operations, the compiler has been left to make the default calls to compiler-rt; however, for multiplication the function name has been replaced with the function from the SoftFloat library. Although a type may be supported by the target's register set, it does not necessarily mean that instruction set will support all those types for all the operations, or support all of the instructions of the LLVM IR. This is the case with the LE1, for example, with the ROTL LLVM instruction which needs to be expanded into a combination of other operations that are supported.

```
def RetCC_LE1 : CallingConv<[
   CCIfType<[i1, i8, i16], CCPromoteToType <i32 >>,
   CCIfType<[i32], CCAssignToReg <[AR0, AR1, AR2, AR3, AR4, AR5, AR6, AR7] >>,
   CCIfType<[i32], CCAssignToStack<4, 32>>
]>;
def CC_LE1 : CallingConv <[
   CCIfType <[i1, i8, i16], CCPromoteToType <i32>>,
   CCIfType <[i32], CCAssignToReg <[AR0, AR1, AR2, AR3, AR4, AR5, AR6, AR7]>>,
   CCIfType <[i32], CCAssignToStack <4, 32 >>
]>;
```

Figure 5.5: Calling convention definitions for the LE1.

The calling conventions and return ABI are defined by the VEX system, the register use is described in detail in Section 5.5. Eight of the GPRs are allocated for arguments and return values and values are promoted to 32-bits in size. The calling conventions are defined with `LE1CallingConv.td`, shown in Figure 5.5, which TableGen uses to generate the `LE1GenCallingConv.inc` header file. Functions are defined with the header file which specify which registers are used and the order in which they are allocated, as well as defining stack usage. To lower a call, a `CCState` object is used to analyse the operands, by using the Tablegen'd header file, and sets the destination location and handling of the call operand. The call sequence begins by using the LLVM SDNode `CALLSEQ_START` and `CopyToReg` are used to copy values from virtual registers to the designated physical registers, with any others values being stored onto the stack. The `LE1ISD::Call` has been created to represent the call which takes the link register, the callee address and a variable number of register arguments. The call sequence DAG is finalised by using the in-built `CALLSEQ_END` SDNode.

```
def SDT_LE1CarryUseGen : SDTypeProfile<2, 3, [ SDTCisSameAs<0, 2>,
                                               SDTCisSameAs<0, 3>,
                                               SDTCisInt<0 >, SDTCisVT<0, i32>,
                                               SDTCisSameAs<1,4>,
                                               SDTCisInt<1>, SDTCisVT<1, i1>]>;
def le1_addcg : SDNode<"LE1ISD::Addcg", SDT_LE1CarryUseGen, []>;
def le1_divs : SDNode<"LE1ISD::Divs", SDT_LE1CarryUseGen, []>;
```

Figure 5.6: SDNode definition for ADDCG and DIVS instructions.

For function returns, `CopyToReg` nodes are used once again to copy values in virtual registers into the designated physical ones. The `LE1ISD::Ret` has been created to represent the return; it takes the stack pointer register, an immediate and the link register as arguments. The `return` instruction of the LE1 is used to unwind the stack and the link register is used as it holds the address to which the program counter needs to return to. As the final size of the stack is unknown until after register allocation, the `Ret` node is modified in the `LE1FrameLowering` class which knows the final size of the stack. On the return from the call, the results are copied out of the physical registers and back into virtual ones by using the `CopyFromReg` nodes.

Some nodes of the other DAG are custom lowered where promotion or expansion do not apply, such as the division and remainder operations. The LE1 does not have a single instruction to perform these operations but does have specific instructions designed for the task: `ADDCG` and `DIVS` are used together over numerous iterations, the order of which was obtained from the output of the VEX compiler. For signed division, the polarity of two operands are first checked to obtain the absolute value of each and then the division step process is begun. The `select` node visible in the diagram is responsible for selecting the absolute value of one of the division operands. The result is finally corrected for any polarity changes after the stepped division calculation. Custom SDNodes had to be created for these two instructions as they are not similar to any of the default SDNodes within LLVM, the code is shown in Figure 5.6. The `SDTypeProfile` defines the number, and the restrictions on, the inputs and outputs of a node. The `ADDCG` and `DIVS` both require three inputs, the first two being 'i32' and the final as an 'i1', and they produce two results; an 'i32' and a 'i1'.

5.3.1.1 Intrinsic Functions

Several intrinsics operations, unique to the LE1, have been implemented to enable the LE1 to execute OpenCL kernels and these are not accessed in user generated code but in the runtime

library. An intrinsic has been defined to differentiate the cores within the execution space by reading the value of the CPU Id. This value actually relates to the hypercontext that is currently executing and the architecture supports up to 256 hypercontexts per context. For this research however, only a single hypercontext was used per context (core) and so the value of the hypercontext Id was divided by 256. The other two types of intrinsics are to write and read a counter value that allows the contexts to iterate through the number of workgroups assigned to it, with the group counters stored in a reserved area of memory. On the stable branch a byte is reserved for each core, and so each core is capable of executing 256 workgroups in each dimension. However this unnecessarily low limit was increased to a word for each dimension of each core in the development branch.

5.3.2 Instruction Description

The instructions are described with TableGen as classes and definitions: base classes are first used to describe the binary encoding format, then classes extend these to group types of instructions, and finally, instructions are defined by specifing the details of the instruction type classes. Instructions can have zero or more inputs, from registers and immediate and can produce zero or more results. Automatic pattern matching is only possible for instructions that produce a maximum of one computed value.

5.3.2.1 Stable Branch

Figure 5.7 shows both the class and instruction definitions for three arithmetic and logic operations that use two general purpose registers as their operands, as well as the base format class. The `ArithLogicR` class defines that the instruction has two inputs, each in a register, and has a single output which is also contained within a general purpose register (GPR). The assembly code string template is also defined and specified by the instruction definition through the passing of the instruction string, such as `add.0`. This implementation of instruction descriptions also enable automatic pattern matching, which is achieved by defining the DAG pattern matcher within the square brackets in the class definition. The differentiator of the instructions is their opcode, and these are passed from the instruction definitions: `add`, `sub` and `and` are all in-built SDNodes.

Figure 5.8 shows example definitions of the compare instructions that take one input from a GPR and the other from an immediate, but the result can either be placed in a GPR or a branch register. The class for the condition set operations is different from the arithmetic class because a SDNode is not passed from the definition; instead a `PatFrag` is used. `PatFrag`s are used to define reusable patterns, in these cases they define a `setcc` (set condition code) node which take two values and the condition to be tested.

```
class LE1Inst<dag outs, dag ins, string asmstr, list<dag> pattern, InstrItinClass itin>:  Instruction
{
      field bits<32> Inst;
      let Namespace = "LE1";
      bits<6> Opcode = 0;
      let Inst31-26 = Opcode;
      let OutOperandList = outs;
      let InOperandList = ins;
      let AsmString = asmstr;
      let Pattern = pattern;
      let Itinerary = itin;
}
class ArithLogicR<string instr_asm, SDNode OpNode, bit isComm = 0>:
      LE1Inst<(outs CPURegs:$dst), (ins CPURegs:$src1, CPURegs:$src2),
               !strconcat(instr_asm,   "$dst, $src1, $src2"),
               [(set CPURegs:$dst, (OpNode CPURegs:$src1, CPURegs:$src2))], IIAlu> {
                     let isCommutable = isComm;
}
def ADD      :  ArithLogicR<"add.0", add, 1>;
def SUB      :  ArithLogicR<"sub.0", sub>;
def AND      :  ArithLogicR<"and.0", and, 1>;
...
```

Figure 5.7: Base class for instruction formats with arithmetic and logic operations definitions.

For instructions that do not match 1-to-1 within pre-existing SDNodes, pattern matchers can be defined as DAGs in the TableGen language too. Just as with instruction descriptions, classes can also be used to describe multiple similar patterns. The code that describes the pattern matching for the branch instructions is shown in Figure 5.9; using a class enables the pattern matching of 60 instructions in less than 30 lines of code. These patterns are required because the LE1 has no single instruction that performs a compare and branch, and so those nodes are matched to two instructions: one performing the comparison and then the second performing the branch based on the result of the comparison. The `PatFrags` provided for condition testing include the comparison of both signed and unsigned types, yet the LLVM type system does not differentiate between the two; instead operations are used to handle those types. Figure 5.10 shows that a shift left by 16 bits followed by a logical right shift by 16 bits provides a 16-bit unsigned value.

```
class SetCCI<string instr_asm, PatFrag cond_op, RegisterClass DestRegs>:
      LE1Inst<(outs DestRegs:$res), (ins CPURegs:$lhs, i32imm:$rhs),
                !strconcat(instr_asm, " $res, $lhs, $rhs"),
                [(set DestRegs:$res, (cond_op CPURegs:$lhs, imm:$rhs))], IIAlu>;

let isCompare = 1 in {
   def CMPEQri : SetCCI<"cmpeq.0", seteq, CPURegs>;
   def CMPGEri : SetCCI<"cmpge.0", setge, CPURegs>;
   ...
   def CMPEQi : SetCCI<"cmpeq.0", seteq, BRegs>;
   def CMPGEi : SetCCI<"cmpge.0", setge, BRegs>;
   ...
```

Figure 5.8: Instruction class and definition used for compare operations.

```
multiclass BranchPats<Instruction BrOp, Instruction CmpOpR, Instruction CmpOpI, PatFrag cond> {
   def : Pat<(brcond (i1 (cond CPURegs:$lhs, CPURegs:$rhs)), bb:$dst),
            (BrOp (CmpOpR CPURegs:$lhs, CPURegs:$rhs), bb:$dst)>;
   def : Pat<(brcond (i1 (cond CPURegs:$lhs, imm:$rhs)), bb:$dst),
            (BrOp (CmpOpI CPURegs:$lhs, imm:$rhs), bb:$dst)>;
   def : Pat<(brcond (i1 (cond CPURegs:$lhs, 0)), bb:$dst),
            (BrOp (CmpOpR CPURegs:$lhs, ZERO), bb:$dst)>;
}
defm : BranchPats<BR, CMPEQ, CMPEQi, seteq>;
...
```

Figure 5.9: Pattern matching of conditional branch instructions.

```
   // MULLHU = ui16(s1) * ui16(s2 >> 16)
   def : Pat<(mul (srl (shl CPURegs:$lhs, (i32 16)), (i32 16)), (srl CPURegs:$rhs, (i32 16))),
             (MULLHU CPURegs:$lhs, CPURegs:$rhs)>;
```

Figure 5.10: Example of pattern matching for an unsigned value.

For instructions that cannot be matched using TableGen, C++ code can also be used to define `PatFrag`s and `ComplexPattern`s; the latter is used for address selection in the LE1 backend. Hard-coded instruction selection and `ComplexPattern`s are defined within `LE1ISelDAGToDAG.cpp`. The only LE1 specific nodes that need to be selected with C++ is the `ADDCG` and `DIVS` instructions as these produce two numeric values. Address selection is performed in C++ as there are multiple legal forms for the address to be encoded. The LE1 is capable of accessing memory using a register, with a possible immediate offset, or by using the name of the global variable; with or without an offset. For global variable accesses, `r0.0` is used as the base address as it constantly provides a zero value. Hard-coded matchers for these type of matchers take an SDValue (the potential match) and return a specified number of values that are used by the instruction. In the case of address calculations, the function assigns values from the input SDValue to the 'base' and 'offset' values used by the load or store.

5.3.2.2 Development Branch

A key difference between the stable and development branch is that the development branch is aware that different length immediates result in different instruction encodings. The instruction descriptions have also been setup to provide the binary encodings if such functionality is required. These two factors made the instruction description more complex, especially for pattern matching as each opcode can generally accept two types of immediates. To alleviate some of the TableGen pattern matching, some operations were lowered to target specific nodes before the final instruction selection, primarily the compare and branch instructions. This happened in two stages: the initial lowering and then during the DAG combining phase, which is part of the optimising phase that is run after each legalisation step. During these steps, `SETCC` nodes are lowered to LE1 specific nodes that have been used to describe the comparison instructions. The results of the comparison can be either 'i32' or 'i1' type, but the code generator defaults to a boolean value; the DAG combiner phases check whether the result of the comparison is extended to 32-bit, and if so, the extending node and comparison is combined into a single compare node which produces a 32-bit result. The `BRCOND` and `BR_CC` SDNodes are then lowered to the LE1 specific `BR` node. The results presented in Section 7.4.3 suggest that this approach is not as effective as using the in-built nodes until actual instruction selection. In the benchmarks in which control-flow operations comprise of a significant number of execution cycles, the development branch compiler performs worse than the stable branch. This can be attributed to the LLVM optimisation passes that do not understand the semantics of the target specific nodes.

A different approach was taken for the updated description of the multiplication instructions. The LE1 SDNodes for the various multiplication operations are equilavent to between

```
def simm9 : ImmLeaf<i32, [{return (isInt<9 >(Imm)); }]>;
def maskU16 : PatFrag<(ops node:$in),
                      (srl (shl node:$in, (i32 16)), (i32 16))>;
def srl16  : PatFrag<(ops node:$in),
                      (srl node:$in, (i32 16))>;
def mullh  : PatFrag<(ops node:$lhs, node:$rhs),
                      (mul (maskU16 node:$lhs), (srl16 node:$rhs))>;
...
multiclass MULT<bits<5>opcode, string instr_asm, PatFrag OpNode, bit isComm = 0>{
   def NAME#r  :  Mult_R_RR<opcode, instr_asm, OpNode, isComm>;
   def NAME#i32 :  Mult_R_I32R<opcode, instr_asm, OpNode>;
   def NAME#i9  :  Mult_R_I9R<opcode, instr_asm, OpNode>;
}
defm MULLH : MULT<0x14, "mullh.0", mullh>;
...
```

Figure 5.11: Multiplication instruction definitions and pattern matchers.

two and five in-built SDNodes, and there are eleven different multiplication operations, all of which can operate on either two registers, a register and a 9-bit immediate or a register and a 32-bit immediate. Three classes have been created for the multiplication instructions, to handle the three different combinations of operand, and then pattern fragments combined with a multiclass have been used to create the definitions of the instructions, as shown in Figure 5.11. As well as the `PatFrag` and `multiclass`, the code also shows an `ImmLeaf` that is defined to pattern match the size of an immediate. This pattern matcher has C++ embedded within it to perform the match. This approach of breaking down the pattern matchers into several reusable parts enables the simultaneous description and pattern matching of the 33 multiplication instructions, equating to over 70 in-built nodes, in less than 60 lines of code. Delaying the matching until the last phase of selection also enables the selector to make better informed choices, especially useful for shift nodes as there are also dedicated 'shift and add' instructions for address computations.

Address pattern matching is more complicated in the development branch as the memory operations can encode an 8-bit immediate or 12-bit immediate within the instruction word as well as encodings that use an additional 32-bit immediate. Three functions have been created, accessed through `ComplexPatterns`, to match the different forms of the addresses based on the length of the offset. Memory accesses that directly use the address of the global value require that the address is stored within a additional 32-bit immediate. The address matching functions check the size of the offsets, much like the `ImmLeaf` pattern, attempting

to encode the offset within the instruction where possible.

5.4 Instruction Scheduling

LLVM instruction schedulers operate upon basic blocks and representes the code as a DAG, specifically a `ScheduleDAG`. The nodes in the DAG (`SUnits`) are connected via their dependences (`SDeps`) upon one another. Both compilers employ the default scheduler that operates on machine instructions before register allocation, as this helps improve ILP as the register allocator would introduce false dependencies. The scheduling framework implements a list scheduler and is extendible by allowing schedulers to use their own heuristics to select from the available nodes. The default machine scheduler tries to choose the best candidate from the ready queue by analysing register pressure and target resource consumption; it also avoids serialising long latency dependence chains. Post register allocation schedulers are also available. The scheduling objects for each target are generated using a script that can be found in Appendix B, which also generates an accompanying XML model for the simulator and can be further used to generate VHDL.

5.4.1 Stable Branch

Scheduling in the stable branch is based upon instruction and processor itineraries. `InstrItinClass` classes are named to represent different types of instructions for the architecture, and then these classes are specified for each subtarget as subtargets may have different pipelines or mix of functional units. `InstrItinClass` are specified with subtarget-specific data using `InstrItinData` objects, which contain detailed information about resource usage, including the number of micro ops, latency, pipeline stages and any bypass pipes that may be available.

For the LE1, five `InstrItinClass` objects are defined to describe the five different scheduling types of instructions of the LE1: arithmetic, multiplication, memory, branch, and pseudo operations. Each instruction is assigned one of these classes for the scheduler to understand the resource requirements and latencies of it. The pseudo class is used for operations that are defined in the backend, yet are not real instructions such as operations to adjust the stack frame. An example of these classes being defined is presented in Figure 5.12; the microarchitecture used is a 2-wide device with two ALUs and a single multiplier and load/store unit.

An object, one for each subtarget (microarchitecture), is created that inherits from `ProcessorItineraries` in which the resource information is defined for each instruction type. The `ProcessorItineraries` are given a list of available `FuncUnits` that can be

```
// FuncUnits are individually named across subtargets so the bundling framework can differentiate between
them.
def ALU_0_2w2a1m1ls1b    :  FuncUnit;
def ALU_1_2w2a1m1ls1b    :  FuncUnit;
def MUL_0_2w2a1m1ls1b    :  FuncUnit;
def LSU_0_2w2a1m1ls1b    :  FuncUnit;
def BRU_0_2w2a1m1ls1b    :  FuncUnit;

def LE12w2a1m1ls1bItineraries :  ProcessorItineraries< [ ALU_0_2w2a1m1ls1b,
                                                          ALU_1_2w2a1m1ls1b,
                                                          MUL_0_2w2a1m1ls1b,
                                                          LSU_0_2w2a1m1ls1b,
                                                          BRU_0_2w2a1m1ls1b ], [/*ByPass*/], [
   // ALU operations have a 2 cycle latency, with 2 FUs to choose from
   InstrItinData<IIAlu, [InstrStage<2, [ ALU_0_2w2a1m1ls1b, ALU_1_2w2a1m1ls1b ]>], [3, 1]>,
   InstrItinData<IIMul, [InstrStage<2, [ MUL_0_2w2a1m1ls1b ]>], [3, 1]>,
   InstrItinData<IILoadStore, [InstrStage<2, [ LSU_0_2w2a1m1ls1b ]>], [3, 1]>,
   InstrItinData<IIBranch, [InstrStage<5, [ BRU_0_2w2a1m1ls1b ]>], [6, 1]> ]>;
```

Figure 5.12: Processor itineraries which describe pipeline usage for each type of instruction.

utilised by the instructions, and each functional unit is given a unique name for the microarchitecture so to be compatiable with the pass that groups instructions into bundles. Each instruction itinerary is broken down into pipeline stages (`InstrStage`) which is given a list of FUs that can be used as well as the latency information. Each instruction class is defined as only having a single pipeline stage because no forwarding paths are used. As such, each stage is defined as having a two cycle latency, with the instruction issued in cycle 1 and the result produced in cycle 3. This method enables a detailed description of the pipeline activity when an instruction is issued, and is more detail than what is actually required for scheduling for the LE1, and so a simpler method is used on the development branch.

5.4.2 Development Branch

For the development branch, based on LLVM 3.4, the scheduling framework was ported to the updated machine scheduling techniques. `ProcResources`, `SchedWrites` and `WriteRes` were used in the place of `FuncUnits`, `ProcessorItineraries` and `InstrStages`. The

ProcResource objects replaced the old FuncUnits by enabling the naming of a resource as well as the number of said resource. As there are multiple target microarchitectures in the backend, resources were named to reflect the number of units in the system; for instance ALU4 would define four ALUs. This method enables multiple targets with varying number of functional units to exist the backend. SchedWrite objects are then used to associate types of instructions with the available resources, the definition of these and the resources are shown Figure 5.13. The SchedWrite objects are synonymous to the InstrItineraries of the original compiler as it is these objects that are associated to the instructions to define the resources taken to produce a result. To do this, a WriteRes object is used that is supplied the SchedWrite and a list of resources that can be used.

```
// Define all the resources across all the subtargets
def ALU1  :  ProcResource<1>;   def ALU2  :  ProcResource<2>;
def ALU3  :  ProcResource<3>;   def ALU4  :  ProcResource<4>;
def MULT1 :  ProcResource<1>;   def MULT2 :  ProcResource<2>;
def LSU1  :  ProcResource<1>;   def LSU2  :  ProcResource<2>;
def BRU   :  ProcResource<1>;
def PRED1 :  ProcResource<1>;   def PRED2 :  ProcResource<2>;
def PRED3 :  ProcResource<3>;   def PRED4 :  ProcResource<4>;

// Declare the objects that will use the resources in producing a result
def WriteA   :  SchedWrite;     def WriteAI   :  SchedWrite;
def WriteM   :  SchedWrite;     def WriteMI   :  SchedWrite;
def WriteLS  :  SchedWrite;     def WriteLSI  :  SchedWrite;
def WriteB   :  SchedWrite;     def WriteBI   :  SchedWrite
def WriteP   :  SchedWrite;

let SchedModel = LE1Model2w2a1m1ls in {
   def :  WriteRes<WriteAI, [ALU2]> {
      let Latency = 2;
      let NumMicroOps = 2;
   }
   ...
}
```

Figure 5.13: Microarchitecture resource description and assignment to instruction types for use by the scheduler, in the updated compiler based on LLVM 3.4

Each of the different microarchitectures defines these objects separately as there are different numbers of functional units across the devices, this is achieved by creating a `SchedMachineModel` for each of the microarchitectures. This object defines the issue width, the load latency as well as tying resources to `SchedWrite` objects. An example of this is given in Figure 5.13, which defines instructions that use `WriteAI` as being capable of issuing on either two of the resources defined by `ALU2`. The definition sets the pipeline latency and increases the default number of micro ops from one to two. This number is used by the scheduler and the packer to understand if instructions take up more resources once executing in the target architecture. This is designed for architectures, such as x86, where instructions get decoded and broken down into smaller operations by the hardware; but it is used for the LE1 to indicate whether an additional word is used in the IRAM for a 32-bit immediate.

5.5 Register Allocation

The task of the register allocator is to replace the unlimited virtual registers in the list of machine instructions with physical registers from a limited set. Register allocation is performed after scheduling to avoid the induction of false dependencies, as the task of the register allocator is to generally reduce live ranges and thus the number of registers in use. As VLIW architectures are designed to exploit ILP, they are designed to have many variables live at one time and so have a typically larger register file. The default allocator, called 'greedy', is used which prioritises values with longer live ranges and is capable of splitting live ranges to reduce the amount of spill code. The allocator is also designed to work well with the specified register usage of target ABIs and can also perform dead code elimination.

5.5.1 Register Files

The LE1 has several types of register that are available to the compiler, or programmer, to use. There are 64 32-bit general purpose registers (GPRs), 8 1-bit predicate registers and a 32-bit link register for calls and returns. The register use and application binary interface (ABI) has been taken from VEX, but with one slight modification in that one register is reserved for copying branch registers. This is because there is no instruction defined to load and store branch registers, and so a general purpose register is required as an intermediate storage container for the memory operation. A VEX system can partition a processing core into clusters, and so register numbers are prefixed with the cluster number, but this compiler only assumes a single cluster. Thus in this case, all register numbers are prefixed with '0' for cluster 0. The registers and their usages are described in Table 5.6.

In describing register files in an LLVM backend, registers are grouped into classes: for

Table 5.6: LE1 register usage.

Register	Class	Usage
r0.0	Constant	always zero
r0.1	Special	stack pointer
r0.2	Scratch	struct return pointer
r0.3 - r0.10	Scratch	argument / return values
r0.11 - r0.56	Scratch	temporaries
r0.56	Special	branch copy / load / store
r0.57 - r0.63	Preserved	temporaries (callee saved)
l0.0	Special	link register
b0.0 - b0.7	Scratch	temporaries

the LE1 there is a `CPURegs` class that holds the GPRs, a `BRegs` class for the branch / predicate registers and a `LReg` class that holds the link register. The registers are named and added to these classes along with the value types that they can contain, which is `i32` for the GPRs and the link register and `i1` for the branch registers. These registers are alll defined in `LE1RegisterInfo.td` using TableGen.

Tablegen is used to create the `LE1GenRegisterInfo` struct which is inherited by `LE1RegisterInfo` and used to implement target-specific functions that provide information about the registers, such as which ones are reserved, and the lowering of `FrameIndex` nodes to use the stack pointer register and an offset. Function call and return ABIs are implemented in the `TargetLowering` class where `CopyToReg` nodes are used to inform the register allocator which registers have to be used as function arguments and return values.

5.6 Instruction Packing

As the LE1 is a VLIW architecture, it is capable of executing multiple instructions simultaneously. The instruction words of the LE1 are variable length groupings of the RISC operations described in the previous sections. Instructions within the group are executed atomically as a single unit, so no RAW or WAW dependencies are allowed within the packet but WAR are as all operands are read before any results are written. The maximum size of the packet is determined by the issue width, and the mix of operations is determined by the available parallelism within the basic block and the number and mix of functional units of the microarchitecture.

The two tasks of the instruction packer is to (1) group instructions into packets which can be executed simultaneously and (2) inserts no-operation instructions (NOPs) between

the packets to adjust for the two cycle pipeline delay. The instruction packer is implemented as a `MachineFunctionPass` which, as the name suggests, is a global pass that operates on the code at the function level once it is represented in machine instructions. The packing is performed after scheduling and register allocation, on lists of `MachineBasicBlock`, so the pass has to iterate through the instructions and not change the order, or change the semantics of the code. In LLVM, groups of instructions are referred to as *bundles*, so 'bundle' and 'packet' will be used interchangeably throughout the description. The internal representation of bundled `MachineInstrs` is depicted in Figure 5.14.

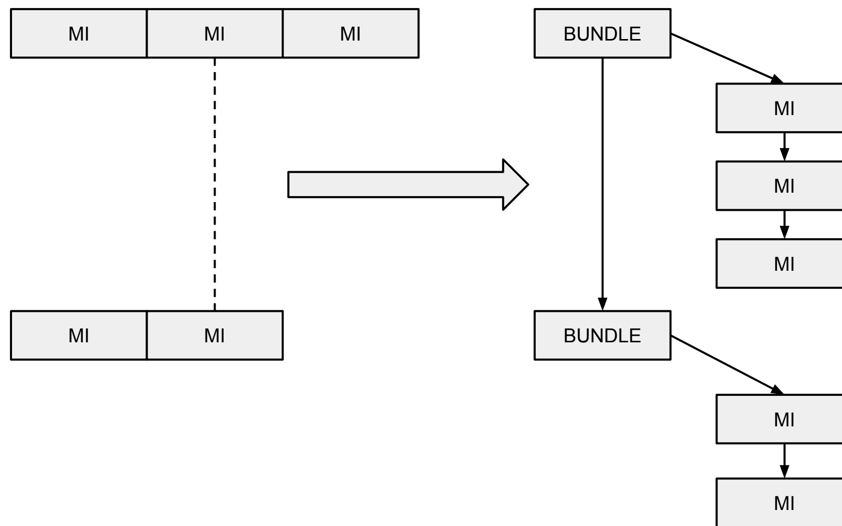

Figure 5.14: Internal representation of LLVM MachineInstr with Bundles.

5.6.1 Stable Branch

Instruction packing on the stable branch uses the techinque developed by Qualcomm for their Hexagon VLIW DSP. Their solution was to create a compiler compile-time table to implement a deterministic finite automaton (DFA) to represent the instruction packet. The DFA has three main elements: inputs, states and transistions. The input being the instruction currently trying to be added to the packet, the state being the current possible consumption of hardware resources, and, the transistions occur with the addition of another instruction to the packet. The absense of a transistion from the current state with a certain state indicates that there is no legal mapping for that instruction within the current packet. The DFA is constructed at compile time, into `LE1GenDFAPacketizer.inc`, and required that all the functional units had different names and also does not support understanding

instructions with multiple `InstrStages`.

The `LE1Packetizer` is a `MachineFunctionPass` that contains a `LE1PacketizerList` object which has access to the DFA. The pass, executed just before code emission, first iterates through each of the basic blocks of the function and removes the pseduo instructions. The basic blocks are then revisited and each instruction is evaluated in turn against the state of the current packet. The `LE1PacketizerList` objects checks whether there are another dependencies between the instructions and will not allow any data dependencies within the packet. It also checks that the size of the packet is not larger than the issue width and that the packet ends when a control-flow instruction is added. Once the instruction in a basic block have been bundled, the latencies between the bundles are checked so that NOPs can be inserted to maintain the two cycle pipeline delay. The instructions within adjacent bundles are checked for data dependences and the process stops as soon as one is found, with a NOP being inserted between the two bundles.

5.6.2 Development Branch

A different implementation was used for the development branch because the DFA method was not necessary as the execution ports of the LE1 do not handle various types of operation. Instructions can be dispatched as long as the required functional units are available and so this is does not effect the issuing of different typed operations. This means that all that is required for packing for a resource table of functional units, created at runtime, instead of a statically compiled automaton. The packer still checks for, and prevents, data dependences within a packet and then checks for hardware resource availability. The small resource table is queried for a free FU depending on the type of the instruction and also the number of micro ops (instruction word + possible long immediate) that the instruction uses compared to the size of the current packet. The pass still inherits from `MachineFunctionPass` but also from `MachineSchedContext` so that a data dependence graph can be built from the incoming list of machine instructions. The insertion of NOPs is also handled differently in the development branch because of the IRAM alignment optimisation that has been introduced.

5.6.2.1 IRAM Alignment

As discussed by Stevens in his thesis [185] and confirmed in the testing of the compiler, the LE1 can suffer from decreased performance due to instruction fetch (IF) stalls in the front-end. This is due to instruction packets being split across lines in memory that cannot be fetched in a single cycle. This happens because of the variable length encoding of the instruction words and through the use of long immediates (32-bit) that can be encoded

in a proceeding instruction packet. A key motivation for the development branch of the compiler was to remove these decode stalls to improve the performance of the more narrow configuration, which would be more suitable while using basic blocks for the scheduling regions.

The first step to removing IF stalls was to enable the compiler to understand the requirements of instructions that used additional words in the IRAM for immediates. This was performed by encoding more information into the instruction selection and scheduling phases described in Section 5.4.2. Any instruction that uses an extra word is assigned an extra micro-op in its description, and during the packing pass the number of micro ops of the instructions is queried against the current size of the packet so that the immediates are always packed within their parent operation.

The second step was to effectively disable the use of variable length instruction words. This was performed by padding out the instruction packets to the issue width of the machine as the instruction fetch width is the same as the issue width. Once the packer decides that the current packet needs to be closed and when the size of the packet does not match the issue width, arithmetic operations are inserted to fill the packet to the issue width. These operations have no effect on the system as they only read and write to and from R0, which is constantly wired to 0. For most packets, the instructions are inserted after the final 'real' instruction of the packet. But for instructions that are scheduling boundaries, such as branches and calls, the instructions are inserted at the start of the packet due the way LLVM represents terminators of basic blocks. The specific instruction that is chosen to be added is dependent on the current state of the current and the available pipeline resources. If an ALU is available, an AND operation is used, otherwise a multiply instruction is inserted. If two words padding is required, then the inserted operation uses a 32-bit immediate. The method requires that the total number of ALUs and MULs at least match the issue width.

The third step is to remove NOPs, because these are single operations that do not equal the issue width, unless in a scalar configuration. Instead of NOPs, bundles were inserted that consisted purely of the operations that are used to pad out instruction bundles. Like the padding, these packets are created to match the issue width and have no semantic effect on the code. The final modification was to the pre-existing assembler, that almost contradicts the previous modification. For a program to execute on the LE1, it needs a `main` function that starts at address 0x0 and so this means that all the other functions in the program are placed in memory after this function ends. On the exit of the main function, the execution of the program needs to end and this means that the pipeline has to be emptied. To enable the LE1 to empty its pipeline and end the program execution, there are 11 NOPs inserted after the call to exit which separate the call from the beginning of the next function stored in the IRAM. However, the number of NOPs used does not align with any issue widths used,

except for a scalar device, and so an extra NOP was inserted. This means that the rest of the functions within the program are offset by 12 operations, of which all the tested issue widths are factors.

5.7 Code Emission

The output of the compiler is an assembly file which is passed to the pre-exisiting assembler to create both an instruction- and data RAM file for the simulator. Within LLVM, the 'MC Layer' is used to represent and process code at the raw machine level. This layer is used to either produce assembly- or machine code, and can also be used to produce standalone assemblers and disassemblers. The `MCStreamer` class is used as an API, with a method per directive, that is implemented differently depending on the type of file output. The LE1 uses the `MCAsmStreamer` class along with its target specific classes to write the assembly file. Much of the textual information of the instruction is described in the TableGen file for the instructions, but three classes are implemented to fill in the details (such as operand formats) : `LE1AsmPrinter`, `LE1MCInstLower` and `LE1InstPrinter`.

Code emission begins with the program represented by `MachineFunction`, with each `MachineInstr` passed to the `LE1AsmPrinter`. The instructions are then lowered to `LE1MCInst` objects, using the `LE1InstLower`, and emitted through the `MCAsmStreamer` using a `LE1InstPrinter` instance. TableGen automatically builds the `LE1GenAsmWrite.inc` file, in which it defines the `PrintInstruction` function of `LE1InstPrinter`. This relationship is depicted in Figure 5.15.

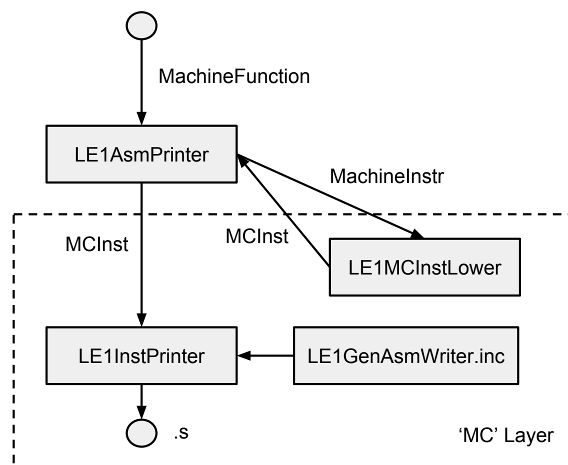

Figure 5.15: Class interactive for LE1 code emission.

5.8 Contributions

The compiler described in this section targets our in-house microprocessor, the LE1. Before this compiler was constructed, the VEX compiler was used to produce machine code for the LE1. The VEX compiler implements a TRACE scheduling compiler and focuses strongly on ILP exploitation but is limited to C89 code, with no C++ support, and is not designed for a multi-core ISA. The compiler presented in this thesis enables the execution of C99, C++ and OpenCL applications on the LE1. This is achieved using the Clang frontend, which supports these languages, coupled with a unique backend which has access to the CPU Id instruction of the LE1 as well as target specific intrinsic instructions. This compiler is also capable of removing the instruction fetch stalls that were induced with the VEX compiler.

5.9 Summary

This chapter has described the LLVM backend for the LE1 for both the stable and development branches of the compiler. The register files, the instruction set and the machine resources of the LE1 have all been described using LLVM's Tablegen language which results in a concise and accurate machine description. The full selection of the LE1's instruction set are available for the compiler to use, and target specific intrinsic operations have been added to enable the execution of OpenCL kernels. The code generator uses the default pre-RA list scheduler as well as the 'greedy' register allocator that is the default for optimised builds. Clang, the frontend, is able to create LLVM bitcode for the LE1 as it has been modified to accept the custom intrinsic functions and obtain data layout information of the target. The code generator is capable of packing independent instructions together in a VLIW format and produces assembly code ready for the pre-existing assembler. The development branch has been optimised for the LE1 by encoding more information about the instructions to enable better instruction selection and scheduling. There are also modifications to the instruction packer to work around an issue with the instruction fetch hardware, the results which are presented later in Section 7.4.3.

Chapter 6

The LE1 OpenCL Driver

6.1 Chapter Objectives

This chapter describes the function and implementation of the OpenCL driver for the LE1, which contains the compiler that was described in the previous chapter. The driver connects the user to the compiler and enables the execution of OpenCL kernels on a cycle accurate simulator for the LE1. This chapter describes the system architecture which enables that to happen and explains what is necessary to enable OpenCL kernels on the configurable VLIW CMP. This includes:

- the user-facing client driver,
- the source-to-source transformer,
- the method of statically scheduling OpenCL workgroups,
- the runtime support; and
- the execution using the simulator

6.2 Client Driver

The software driver provides a layer of abstraction from the LE1 hardware for the application developer, so its task is to allow communication between them; allowing the programmer to control the execution of the OpenCL program but also hiding the device specific details. For this, the OpenCL 1.1 standard API has been implemented and the fact that the kernel is coarsened to the workgroup, statically linked and run on the cycle accurate simulator, is hidden from the application developer. The proposed software is in the form of the

GNU/Linux shared object, named libOpenCL.so, that can be used just as any other OpenCL driver. The driver is built around Clang / LLVM libraries, which are statically linked into the driver; they allow the transformation and compilation of OpenCL kernels at runtime. The driver is composed of three main parts, depicted in Figure 6.1:

- the front-end client driver, which implements the OpenCL 1.1 API calls, allowing the user to control the rest of the driver;
- the source transformer, which converts (coarsens) the kernel source from work-item -based to workgroup-based, taking into account barrier synchronisation, variable life-time etc;
- the backend compiler, which links in the developed runtime library and produces the assembly code which executes on the LE1 CA simulator.

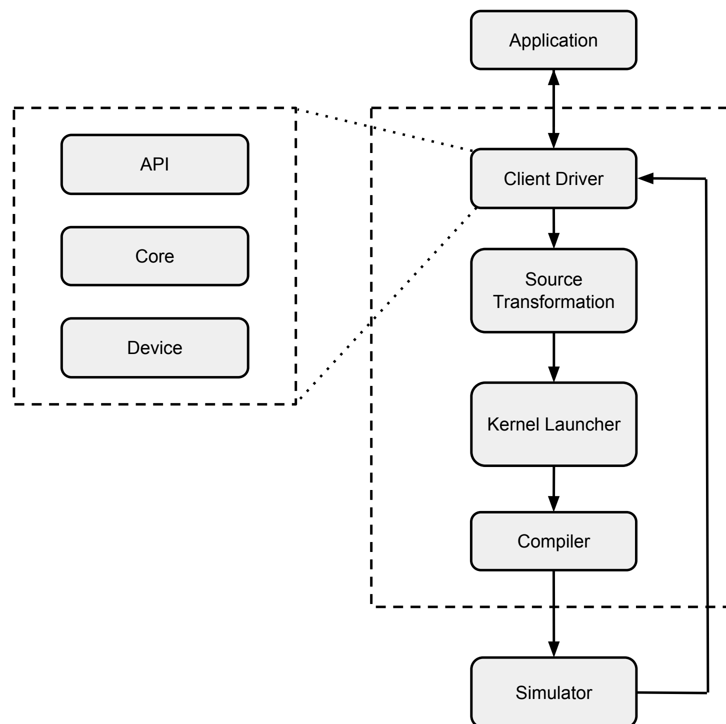

Figure 6.1: Software system overview.

```
static Coal::LE1Device LE1Devices[240] = {
//              cores, width, alus, muls, lsus, banks
   Coal::LE1Device(    1,    1,    1,    1,    1,    1),    // 1
```

Figure 6.2: Driver target instantiation example.

6.2.1 API

The frontend is based upon Clover [189], a project which was merged into the Gallium3D [190] graphics layer. It implements the OpenCL 1.1 API and supports the OpenCL embedded profile (no 64-bit arithmetic or 16-bit floating-point). The API layer implements the OpenCL API and so enables the programmer to create OpenCL objects, such as contexts, programs and kernels, as well as enabling the execution of kernels. The original Clover implementation supports `Image2D` and `Image3D` creation, but these have not been tested for the LE1. The API layer holds the 240 statically instantiated devices that represent the LE1 targets, and the user can query and select this using the specified API functions. These targets are automatically generated via a script, described in Section 7.3, and an example of a target instantiation is shown in Figure 6.2. Once the user has created a context and and a command queue, the API layer uses these to interface with the core.

6.2.2 Core

The core contains the classes for OpenCL objects such as buffers, kernels and programs which are used as containers to carry information from the API layer to the target devices. These objects map to their respective OpenCL types. The key component to the core is the command queue, which is used to transport commands, as well as their respective data structures, between the three layers of the driver. A `Context` object populates a list of possible target devices (`DeviceInterface` objects), which the user then selects and assigns a `CommandQueue` to it. The `CommandQueue` holds both the `Context` and the `DeviceInterface` and also maintains a list of `Events` intended for the target. Events are used for the (possibly asynchronous) tasks of memory transfer and kernel execution. `BufferEvent`s, containing `Buffer` objects, are used to perform memory transfers between the host and the device while `KernelEvent`s, containing `Kernel` objects, are used to initialise kernel execution. The relationship between the the API, core and device layer is depicted in Figure 6.3.

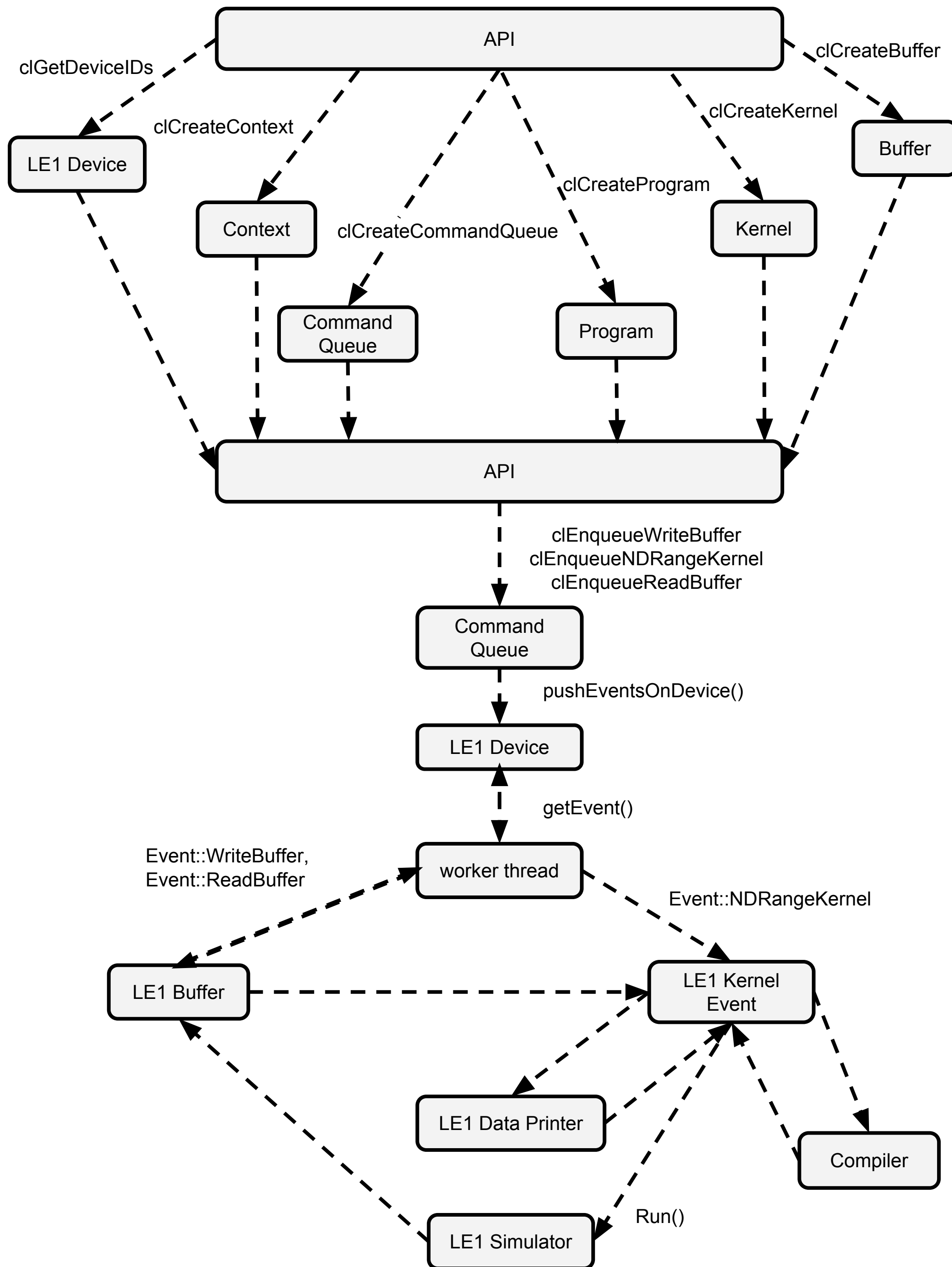

Figure 6.3: Simplified relationship between the API and the objects in the 'core' and the 'device'.

An OpenCL `Program` object is capable of loading the program from either source code or as a binary. The OpenCL `Program` assigns target devices to itself, creating a target-dependent version of itself and constructing a target-dependent `Compiler` too. When source is used as the input, the `Compiler` object is first used to check for errors and compiles the kernel source code into a LLVM `Module`. With no errors in the input file, any macros in the code are expanded, and functions are inlined in preparation for the source transformation. The `Module` is then used by the `Program` to find the names of the kernels by using metadata that the Clang front-end applies to OpenCL kernels. When the user instructs the driver to create any `Kernel` objects, the `Program` then uses this metadata to verify the existence of such a function; which is then assigned to the `Kernel`. A `Kernel` object creates an argument list for the given function by scanning the LLVM `Module`, recording the address space and its type. The argument data are stored in their own `Arg` objects which each contain hooks and helper functions for access to their underlying data, such as buffers. Buffers are represented as `MemObject` which use their parent `Context` object to know how many target buffers to create. This base class is inherited by `Buffer`, `SubBuffer` and both `Image2D` and `Image3D`.

6.2.3 Device

The device layer takes the generic program objects, from the command queue, and specifies them for itself. Each device is instantiated in the API layer, and `DeviceInterface` objects represent all the LE1 architecture and microarchitecture variations that have been investigated. As well as the compiler target and simulator model, each `LE1Device`, has a worker thread which queries its associated queue for events to act upon, such as buffer operations and running of kernels. Once the user calls `clEnqueueNDRangeKernel` all the necessary data is available for the kernel to be transformed. The `LE1Buffer` objects creates a copy of the buffer created in the core layer, but does not currently check against the total memory allocated on the device; this is delayed until the last phase of data transfer. This is so that the memory required by stack allocations can then be taken into consideration once the program has been compiled. This check is yet to be implemented though. The `LE1Kernel` also contains little more than a copy of the generic object in the 'core', but sets the device-specific maximum workgroup size of 256. The `Program` object is specialised into a `LE1Program` object but it's only function is to transfer the modified kernel source code to a `LE1KernelEvent`.

The `LE1KernelEvent` class is the final one to control the execution as it holds all the information required to finalise the kernel; it is this class that utilises the `Compiler` the most as it drives the source transformation and compilation. The transformed source code is again compiled to a LLVM module, optimised, and scanned for any floating-point operations that will require the support of the runtime library, which is then linked to the kernel. Global

data within the module is then extracted so that it can be written to the device in the same manner as the user defined buffers. A runtime kernel-launching function is also created and linked to the kernel code, and, finally the whole module is compiled into assembly language. An `LE1Simulator` object is also created and controlled from the event, including writing the data and reading the results back after the execution to update the host buffers. The `LEDataPrinter` class is used to write the data and is also responsible for calculating the addresses to be assigned to the device buffers. To write the data, it is extracted from the host buffers and written into the assembly code file.

6.3 Source-to-Source Transformation

The unit of work is enlarged from the work-item level to the workgroup level through AST source-to- source transformations, using Clangs libraries. Performing the transformation at a high-level allows the coarsening to only happen once, even in the presence of different multi-core accelerators in the heterogeneous system. The transformation takes place in three phases: a) code expansion and function inlining, b) basic workgroup coarsening and c) barrier call and control-flow handling. Function inlining happens at the source level as barriers, can contained within any function, and so several passes from C-Reduce [191] have been integrated for this. Macros also need to be expanded to be able to successfully rewrite the coarsened source since the Clang libraries are unable to handle macros in the rewriter. In the absence of any barrier calls, the kernel body only needs to be enclosed in one or more for loops; one for each required dimension; any return statements are replaced with a `goto`, effectively skipping the current work-item. The transformation takes place once the programmer has requested the kernel execution, so the local size can be hard coded into the loop declaration in the hope of aiding loop transformations. The kernel initialiser algorithm in shown in Figure 6.4 and is an implementation of the `RecursiveASTVisitor` class, that is part of Clang.

In the presence of barriers, the regions in the source in which the work-items would execute completely independently need to be found so that the kernel can be divided between those sections; for this loop fission is used [192]. Wherever there is a barrier, the workgroup loop is closed before the barrier and re-opened after the barrier with the barrier call finally removed. This guarantees that all the work-items have completed before continuing past the original barrier call as the OpenCL specification requires. If there are barriers located within nested regions, such as a for-loop (but not the outermost workgroup loops inserted by the transformation engine), those regions boundaries are also used as fission points. This is necessary since a barrier within a loop would define that all work-items have to complete up to the barrier for the same iteration before any can pass it. Other statements, such as break

```
∀ Function f ∈ Module
  do if isKernel(f)
    then EncloseBodyWithNestedLoop(f)
      InsertExitLabel()
      ∀ DeclStmt ds ∈ f
        do if NonSingleDeclStmt(ds)
          then split(ds)
      ∀ CallExpr ce ∈ f
        do if isOpenCLBuiltin(ce)
          do if isIdCall(ce)
            then replace ce
          do if isLocalSize(ce)
            then replace ce with immediate
          do if isBarrier(ce)
            then barrierList.add(ce)
      ∀ ReturnStmt rs ∈ f
        returnList.add(rs)
      do if barrierList = ∅
        then ∀ ReturnStmt rs ∈ returnList
          replace rs with a goto
```

Figure 6.4: Simple kernel coarsening algorithm.

or continue, complicate this situation further since they could skip work-items or the whole workgroup. If these statements exist within a cyclic region that also contains a barrier, the specification mandates that if one work-item executes the statement, all of them will for that same iteration - otherwise some work-items would execute the barrier while others would not, causing a livelock. As a result, `continue` and `break` statements are also used as fission points.

Local variables are created for variables that are live past the chosen fission points, and dependency analysis is applied to determine whether the variable is thread-dependent to decide whether the variable needs to be expanded or not. As the kernel is explored depth-first from the outer thread loop, statements are checked to find whether their definition is ever dependent upon the work-item ID - if not, the variable does not ever need to be expanded since the same value is computed for each work-item. Thread dependent variables are ones that have a data dependency on either `get_local_id` or `get_global_id`, so any variables that are defined using those calls are added to a list of dependent variables. The `get_workgroup_id` is not used to identify thread dependent variables as workgroups are

executed as a single unit upon a single core of the device. This list is then used to find further thread-dependent variables by examining whether any variables refer to any members of the list. As the code is explored, and the algorithm enters a region that is executed conditionally upon a thread-dependent variable, any variables defined within that region are also added to the list. For variables that are thread-independent, it would be possible to move them outside of the workgroup loop but for now it is left to the LLVM loop optimisations to do this. For scalar expanded values, all references of the original variable are visited and rewritten as array accesses using indices of the workgroup loop(s). The algorithm is described in Figure 6.5 with an example transformation shown in Figure 6.6.

```
∀ Function f ∈ Module
  do if isKernel(f)
    then ∀ DeclRefExpr dre ∈ f
      declRefExprList.add(dre)
    ∀ ForStmt outer ∈ f
      do if outer = outerLoop
        then TraverseRegion(outer) {
            regionMap.add(outer)
            ∀ Stmt s ∈ outer
              MapStmt(s, outer)
              FindThreadDeps(s)
            ∀ Stmt inner ∈ outer
              TraverseRegion(inner)
          }
    SearchThroughRegions() {
      ∀ Stmt region ∈ regionMap
        do if isNotParallel(region)
          then HandleNonParallelRegion(region)
    }
    FindReferencesToExpand() {
      ∀ Stmt region ∈ regionMap
        ∀ DeclStmt ds ∈ region
          ∀ DeclRefExpr dre ∈ declStmtMap(ds)
            do if SeparatedByFissionPoint(ds, dre)
              then ScalarExpand(ds)
    }
```

Figure 6.5: Algorithm outline for the second, and final, stage of kernel coarsening.

```
__kernel void permute(__global const uint* unsortedData,
                      __global const uint* scanedBuckets,
                      uint shiftCount,
                      __local ushort* sharedBuckets,
                      __global uint* sortedData) {
  size_t localId[64], globalId[64];
  unsigned __esdg_idx = 0;
  for (__esdg_idx = 0; __esdg_idx < 64; ++__esdg_idx) {
    size_t groupId = get_group_id(0);
    localId[__esdg_idx] = __esdg_idx;
    globalId[__esdg_idx] = get_group_id(0) * 64 + __esdg_idx;
    size_t groupSize = 64;
    for(int i = 0; i < ( 1 << 8 ); ++i) {
      uint bucketPos = groupId * ( 1 << 8 ) * groupSize + localId[__esdg_idx] * ( 1 << 8 ) + i;
      sharedBuckets[localId[__esdg_idx] * ( 1 << 8 ) + i] = scanedBuckets[bucketPos];
    }
    //barrier(1);
  }
  for (int i = 0; i < ( 1 << 8 ); ++i) {
    for (__esdg_idx = 0; __esdg_idx < 64; ++__esdg_idx) {
      uint value = unsortedData[globalId[__esdg_idx] * ( 1 << 8 ) + i];
      value = (value >> shiftCount) & 0xFFU;
      uint index = sharedBuckets[localId[__esdg_idx] * ( 1 << 8 ) + value];
      sortedData[index] = unsortedData[globalId[__esdg_idx] * ( 1 << 8 ) + i];
      sharedBuckets[localId[__esdg_idx] * ( 1 << 8 ) + value] = index + 1;
      //barrier(1);
    }
  }
  for (__esdg_idx = 0; __esdg_idx < 64; ++__esdg_idx) {
__ESDG_END: ;
  }
}
```

Figure 6.6: Permute source code after complete transformation.

6.4 Runtime Support

Once the kernel has been coarsened, it is then compiled into an LLVM module. Before the module is compiled into machine assembly, it has to be linked with the runtime library which supports the builtin OpenCL functions as well as the emulated floating point operations. Any embedded data from the module also needs to be extracted for data transfer to the device and a control function has to be created to execute the workgroups at runtime.

6.4.1 Builtin Functions

The OpenCL standard defines many library functions that need to be included in a conforming implementation. Libclc [193], a subproject of LLVM, is an existing library that has been incorporated into the driver for this purpose. This library has been modified to enable an LE1 system to operate throughout the execution space that is defined by workgroup dimensions. This is achieved by using the intrinsics defined in the LE1 compiler backend for accessing the CPU Id and designated areas of memory for kernel specific information. These key functions are:

- `get_global_size` accesses `__builtin_le1_read_global_size`
- `get_num_groups` accesses `__builtin_le1_read_num_groups`
- `get_group_id` accesses `__builtin_le1_read_group_id`

Neither `get_local_id` or `get_local_size` need to be implemented as the local size becomes hard coded into the source file, and the local Id becomes the loop controlling induction variable. The `get_global_id` function is also rewritten to multiply group Id by the local size and add the result to the local Id. The rest of the functions have also been left untouched. As the library is a subproject of LLVM, it uses LLVM's runtime library 'compiler-rt [194]' for its optimised functions and LLVM intrinsics for key mathematical operations. This reliance on compiler-rt requires that these functions are also included in the OpenCL runtime library. The library has not, however, been ported to the LE1 as there has not been sufficient time; it has merely been compiled into an LLVM module with the target data layout information of the architecture. This is currently the key flaw of the driver, which not only means that the library has not been optimised for the LE1, but it may well also contain code that produces erroneous on the device.

6.4.2 Softfloat

As well as the functionality defined by the OpenCL specification, the runtime library for the LE1 also needs to support emulating floating-point calculations. For this, the softfloat library

[195] and functions defined in compiler-rt have been compiled into the runtime module. But unlike the functions of the OpenCL standard, calls to the floating-point functions are not resolved until the code reaches the backend of the LE1 compiler. And as compilation is happening just-in-time, but without LLVM's JIT framework, the required softfloat functions need to be included within the final module before the compilation to assembly begins. To enable this, once the kernel source has been compiled into an LLVM module, all the IR operations are iterated through to see whether they operate on floating-point data types. This is possible by querying the opcode of the operation, or the Id of the intrinsic call. When such an operation is found, the supporting function, or functions, is declared within the module which identifies to the linker which functions are required in the module. The compiler backend will then lower the operations to calls to those functions which are finally resolved by the assembler. The support for floating-point is not exhaustive, the functionality has been added ad-hoc as the tested kernels have required, but simple arithmetic (add, sub, mul, div) and compare operations are currently supported.

6.4.3 Kernel Launching

The coarsened kernel source represents the work done by a workgroup instead of a work-item, typically however, many workgroups will execute that code. The instantiation of the workgroups is performed in software purely running on the LE1 system. The kernel is linked with a small function which calls instances of the newly created workgroup. This launcher function, as shown in Figure 6.7, uses the CPU Id and work dimension counters to calculate which workgroup the unit should be computing. The LE1 has an instruction which allows the user to query the CPU ID (SYSTEM:CONTEXT:HC tuple), which returns the value of the currently executing hypercontext. This value is used at the context level to determine the execution space; an intrinsic is used to read the CPU ID, which is then used to offset the buffer address for each of the cores. Intrinsics are also used to keep count of the number of workgroups completed. The launcher also checks whether the core is even supposed to operate as some data sets will not split over the whole system evenly, meaning that sometimes cores need to exit early and not perform the kernel operation.

Figure 6.8 depicts how workgroups are viewed, the number in the centre is the Id given to the workgroup and the different colours represent the different cores. For a 4-core device, the cores would execute these groups:

- Core 0 (green) executes groups 0, 4, 8, 12, 16, 20, 24 and 28,
- Core 1 (blue) executes groups 1, 5, 9, 13, 17, 21, 25 and 29,
- Core 2 (yellow) executes groups 2, 6, 10, 14, 18, 22, 26, 30,

```
extern int BufferArg_0;
extern int BufferArg_1;
int main(void) {
  int id = 0;
  int num_cores = 1;
  int total_workgroups = 8;
  int workgroupX = 8;
  int workgroupY = 0;
  int x = 0;
  int y = 0;
  id = __builtin_le1_read_cpuid();
  while (id < total_workgroups) {
    x = id;
    if (x >= workgroupX) {
      y = x / workgroupX;
      x = x % workgroupX;
    }
    if (y > workgroupY)
      return 0;
    __builtin_le1_set_group_id_1(y);
    __builtin_le1_set_group_id_0(x);
    binarySearch(&BufferArg_0, &BufferArg_1, 20000);
    id += num_cores;
  }
  return id;
}
```

Figure 6.7: Kernel launcher source code for Binary Search.

- Core 3 (pink) executes groups 3, 7, 11, 15, 19, 23, 27 and 31.

6.4.4 Data Transfer

Once the kernel source has been transformed, compiled into a module, scanned for floating-point operations and finally linked with the runtime library, it is then scanned for any embedded data. This is global data that has been included from either the soft-float library or compiler-rt and is required by the functions that have been linked into the module. Any constant data found within the module is saved into an `EmbeddedData` object which

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31

Figure 6.8: Workgroup execution model on a 4 core system, with the workgroups numbered and the different varying colours representing the separate cores.

currently supports scalars, arrays, vectors and structs containing both floating-point and integer data. This data needs to be sent to the device along with the buffers that the application programmer has sent up using the API. The final code is compiled into an assembly file and all the necessary data is appended onto the end in a format ready for the assembler to create the corresponding DRAM file. The stack on the LE1 begins at the highest address and is decremented towards zero, so all the kernel attributes and global data are stored beginning at address 0x0 followed by the buffer data and memory reserved for local buffers. All kernels begin with set attributes:

- 0x0 - number of work dimensions,
- 0x4 - global work size for all three possible dimensions,
- 0x10 - local work size for all the three possible dimensions,
- 0x1c - the number of work groups for all the dimensions,
- 0x28 - global Id offset,
- 0x34 - number of cores in the system,
- 0x38 - current group id for each core.

As the contents of the buffers are printed out into the existing assembly file, the data is converted from the little endian format of the x86 host to big endian that is used by the LE1.

Local buffers are reserved on a per-core basis as a core only executes a single workgroup at any one time and all these buffers are initialised to zero. All buffers are also word aligned. The data printer supports chars, shorts, ints, 32-bit float and also vectors of these types. It also supports buffers of user defined `struct` types though this has only been tested with one benchmark which used naturally packed elements. The `LE1DataPrinter` is also responsible for assigning addresses to the OpenCL buffers, this is so that their contents can be updated after the execution of a kernel. When this happens, data can be read out of the simulator by requesting a read from the beginning of an address for a set number of bytes.

6.4.5 Simulation

The execution of the OpenCL kernels is performed using a cycle accurate simulator, named Insizzle. An API is provided to gain low-level access to the simulator, which is encapsulated within the `LE1Simulator` object that each `LE1KernelEvent` creates. The simulator requires three files to operate: an XML file that describes the system architecture and both an IRAM and DRAM file of the kernel. Statistics are collected from each context after the program has finished executing, comprising of: the total number of cycles, number of NOPs, instruction fetch stalls, branches taken, branches not taken, control-flow changes and memory access stalls. The statically instantiated targets are destroyed as the host program ends, and in the process they print the data to a CSV file, taking the mean average if the kernel was been run for multiple iterations.

6.5 Contributions

The driver produced as part of this research enables the SPMD programming model on the LE1. This is achieved by enabling OpenCL kernels to execute upon the, currently simulated, LE1. Previous parallel applications had to be written specifically for the LE1 using a pthread derived functions. The JIT compilation of the kernels, which the driver enables, also allows portable programs to be distributed and run upon various configurations of the LE1 without having to worry about the underlying microarchitecture. This alleviates the key issue of compiling and distributing programs for VLIWs, such as the LE1. The configuration files, that are used by the driver to target the various micro- and system architectures, can be passed via the same API as the simulator so that physical FPGA designs can be instantiated. With this method, the driver has been designed to offload the computation of OpenCL kernels onto an FPGA platform using a fully programmable CPU architecture. I am unaware of any available OpenCL drivers that support the execution of OpenCL kernels on a configurable VLIW CMP architecture.

6.6 Summary

This chapter has described the OpenCL driver that enables the execution of OpenCL kernels using a cycle accurate simulator for the LE1. This is achieved by embedding Clang and LLVM libraries, including the LE1 code generator, into the driver. The workgroups are statically issued across the multiple contexts, in SPMD fashion, by using a small control function to launch the kernel functions and to set which workgroup should be executed. The kernel source files have been transformed so that they represent a workgroup and not a work-item. This has been achieved by introducing loops within the kernel to iterate through all the work-items and ensuring to maintain the semantics by splitting the loops around barrier functions and control-flow statements. The runtime library is provided through an amalgamation of three open-source libraries which provide functions defined by the OpenCL specification and emulated floating point operations.

Chapter 7

Experiments

7.1 Chapter Objectives

This chapter presents the investigations into the validation and performance analysis of the compilation framework and the LE1. This was performed by running 12 OpenCL benchmarks across 240 different machine configurations. The performance of the compiler was evaluated using 20 different microarchitectures, significantly less than the number of machine configurations due to the compiler not knowing about the number of LE1 contexts or memory banks in the system. The purpose of using such a large number of machine configurations was so that the collected data could also also be used in the future as a training set for designing a hardware / software codesign system. Appendix C contain the rest of graphs not shown is this chapter. The graphical results do not present all that was collected during the experiments so the full tabulated sets are available online, listed in Appendix D

7.1.1 Validation

The key aspect of the experiments was to validate that the compiler was producing correct code and that the driver was transforming code legally and correctly executing the programs. The benchmarks chosen contain correctness testing code within them, and the results presented here are from successful completion of those benchmarks. The kernels chosen cover a wide range of applications and requirements for handling, such as: vector data, floating point operations, barriers as well as inter- and intra-kernel communication.

7.1.2 ILP performance

As well as validation, the experiments needed to show the compiler's ability to effectively use the instruction set of the LE1. The key abilities are being capable of removing control-flow using `select` instructions, using suitable registers for comparison operations and taking advantage of the shift-and-add instructions for simplifying address computations. Region formation techniques have not been used, so performance will be based upon the compiler's ability to find enough parallelism within basic blocks and performing effective scheduling for the multiple functional units of multi-issue machines. The final section of this chapter evaluates the effect of using loop unrolling to enlarge the scheduling regions as well as comparing those results with the unstable branch of the compiler.

7.1.3 TLP performance

ILP performance is the responsibilty of the compiler and will be dependent upon the characteristics of the kernels and the microarchitecture configurations, though the static scheduling of the workgroups will also effect performance. TLP performance is primarily dependent of the system architecture of the LE1, and the experiments needed to explore the capabilities of it. This is performed by using silicon data alongside the results obtained from the simulator.

7.2 Benchmarks

The OpenCL benchmarks were taken from the AMD AMP SDK [196] as well as the Rodinia benchmark suite [197]. It was necessary to change the host programs in some cases where the OpenCL context is created by type, as the driver defines the LE1 as a `CL_DEVICE_TYPE_ACCELERATOR` whereas most benchmarks query for a GPU device and in its absence, default back to the x86 host CPU. The benchmarks also contained an AMD specific memory optimisation that was disabled during the testing. The benchmarks represent a mix of real-world applications that should test the capabilities of both hardware and software. The selection of kernels include complex control-flow, barriers, vector data types and are both integer- and floating-point based. Several benchmarks are comprised of multiple kernels which are also run for a number of iterations, requiring intra- and inter-kernel data transfer. The amount of work performed by each kernel is described in Tables 7.1 and 7.2.

Table 7.1: Kernel work dimensions, sizes and execution iterations from AMD AMP.

Kernel Name	**Global sizes**	**Local sizes**	**Workgroups**	**Iterations**
BinarySearch	131072, -	128, -	1024	1
BitonicSort	16384, 1	256, -	64	120
FastWalshTransform	2048, -	256, -	8	12
FloydWarshall	128, 128	16, 16	64	128
MatrixTranspose	32, 32	16, 16	4	1
NBody	1024, -	256, -	4	1
Reduction	16384, -	256, -	64	7
Radix Sort				
Histogram	16384, -	256, -	256	4
ScanArraydims2	64, 256	64, 1	256	4
ScanArraydims1	256, -	256, -	1	4
Permute	64, -	64, -	1	4
FixOffset	64, 256	variable, variable	variable	4

Table 7.2: Kernel work dimensions, sizes and execution iterations from Rodinia.

Kernel Name	**Global sizes**	**Local sizes**	**Workgroups**	**Iterations**
Breadth-First Search				
BFS_1	4096, 1	256, 1	16	8
BFS_2	4096, 1	256, 1	16	8
Gaussian Elimination				
Fan1	16, 16	variable	variable	15
Fan2	16, -	variable	variable	15
Needleman Wunsch				
nw_kernel1	variable (16-256), -	16, -	variable (1-16)	16
nw_kernel2	variable (16-256), -	16, -	variable (1-16)	15
NN	42816, -	variable, -	variable	1

7.3 Machine Configurations

The machine configurations were automatically generated via a python script, shown in Figure 7.1, which created both the LLVM backend and the input file for the simulator. The task was to create a homogeneous system for the simulator in the form of a XML file, which included the number of contexts and the DRAM configuration, and TableGen files for the compiler target; which did not need to know the number of contexts, nor the DRAM configuration. The XML file can also be used as the configuration file for generating VHDL to synthesis the LE1 system. The script was limited to instantiating a maximum of 2 load/store units (LSUs) and 2 multipliers (MULs) per context. Each context also needed to have a number of arithmetic logic units (ALUs) that was equal to at least half the issue width, and the issue width had be a factor of four. As the script progressed, the number of contexts were doubled and the number of DRAM banks was never more than the multiple of contexts and the number of LSUs within them and was capped at eight. This resulted in a total of 240 configurations. The LE1 CMP device is treated as an OpenCL compute device, with each context equating to an OpenCL compute unit (CU); context and CU will be used interchangeably throughout the analysis of the results. The full script can be found in Appendix B.

```
for context in [1, 2, 4, 8] :
   for width in [1, 2, 4]:
      for alus in range(1, width+1):
         if ((width != 1) & (alus < (width / 2))):
            continue
         for muls in [1, 2] :
            if (muls > width) :
               continue
            for lsus in [1, 2] :
               if (lsus > width) :
                  continue
               for banks in [1, 2, 4, 8]:
                  if (banks > (lsus * context)):
                     continue
```

Figure 7.1: Loop header for target machine generator script.

7.4 Results

7.4.1 ILP Performance

This section evaluates the single context performance of all the benchmarks to ascertain the potential performance increases from exploiting just ILP. Thus, the results in this section represent the capabilites of the compiler. All the kernels were compiled with Clang's default aggressive optimsiations which are equivalent to using '-O3' on the command line.

7.4.1.1 Binary Search

The performance response for Binary Search, in Figure C.1, clearly shows that no ILP is usefully exploited. The increase in NOPs, from 3484, with the scalar device to ∼5500 in the 2-wide configurations shows that ILP is discovered but that it is not enough to fill the pipeline continously. With a lack of available ILP, the increase in IF stalls, shown in Figure C.2, results in all the 2-wide devices performing slower than the scalar configuration by ∼7%. Increasing the issue width to four yields improvements because less IF stalls are encountered, and the 4-wide devices only perform 0.6-0.7% better than a scalar configuration. The scalar configuration is the most efficient as it has the lowest number of wasted cycles (due to NOPs, IF stalls and mispredict penalties) with 71.4% compared to the other configuration which all spend ∼86-87% of the cycles not performing any calculations.

7.4.1.2 Bitonic Sort

The performance response to the microarchitecture for Bitonic Sort is shown in Figure C.3 with the IF stall cycles in Figure C.4. A small amount of ILP is exploited in the 2-wide devices with singular FUs, with ∼2% performance increase. This is a modest increase due to the increase in NOPs, showing that there is not enough ILP to mask the pipeline latencies. An increase in IF stalls in the 2-wide devices with two LSUs cause these devices to perform little over 1% faster than the scalar devices, while the two machines with two ALUs but singular LSUs execute 10% faster. In the 4-wide configurations, the IF stalls decrease again enabling these configurations to perform better than the 2-wide machines. The increase in ALUs and MULs improves performance by 2-3% for each added unit but performance is capped at three ALUs and two MULs; these devices perform 17.81% faster than the scalar configuration. The total wasted cycles for this kernel remains high though: the scalar machine wastes 37%, the 2-wide waste 47-62% while the 4-wide waste 58-65%.

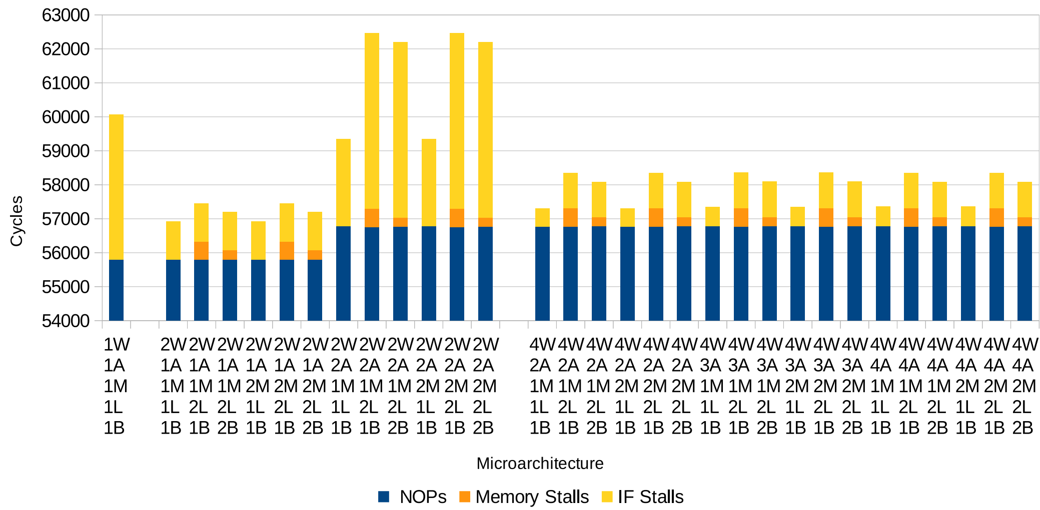

Figure 7.2: Total average stalls and NOP cycles for BFS_1 using 1 context across varying microarchitecture configurations.

7.4.1.3 Breadth-First Search

Both kernels from the breadth-first search implementation (BFS_1, BFS_2), shown in Figures C.5 and 7.3, exhibit similar behaviour to one another. The compiler is not able to exploit much ILP for either of the kernels: the largest configurations only achieve 4.46% speedup for BFS_1 and 7.95% for BFS_2 and this is only ∼1% more than the 2-wide configurations with single FUs. Both kernels also suffer from increased IF stalls in some of the 2-wide machines, for BFS_2 this occurs in the two devices with two ALUs and a single LSU whereas all the 2-wide devices incur the stalls in some degree for BFS_1. The variation in the memory also contribute to the variation in total cycles, as shown in Figures 7.2 and 7.4, but overall the response to the microarchitecture remains relatively constant.

Figure 7.3: Total average cycle count for BFS_2 using 1 context across varying microarchitecture configurations.

Figure 7.4: Total average stalls and NOP cycles for BFS_2 using 1 context across varying microarchitecture configurations.

7.4.1.4 Fast Walsh Transform

The single CU performance for the FastWalshTransform kernel is depicted in Figure C.6, it shows that this kernel responds positively to increased issue widths and ALUs but is largely unaffected by the other variables in the configuration. The largest configuration achieves a 19.14% reduction in cycles. But the more simple 4-wide devices, each with one MUL, LSU and bank but with two and three ALUs, achieve 18.33% and 17.83% improvements respectively. The 2-wide devices, with two ALUs, all achieve ∼9% improvement over the scalar, but these devices also incur a significant increase in IF stall; as shown in Figure C.7. The performance gain for the 4-wide devices largely comes from the decrease in these stalls, which suggests that any significant ILP is discovered in the 2-wide machines but their performance is purely hindered by the IF stalls. However, IF stalls do increase again for the larger 4-wide devices and yet performance still improves slightly.

7.4.1.5 Floyd Warshall

Figures C.8 and C.9 shows that the performance of the FloydWarshall benchmark is closely linked to both the IF and memory stalls of the system, and that the rest of the microarchitecture details have very little effect. The increase in IF stalls for the 2-wide configurations, with a single ALU, are reflected in the total cycles which means that any ILP exploited is not enough to counter the stalls. However the next increase, when using two ALUs, is not reflected in the total output which means that enough ILP is discovered to counter the detrimental affect, but still all the 2-wide devices perform worse than the scalar device by an average of 6.1%. The decreased IF stalls throughout the 4-wide configurations enable them to execute faster than the scalar, but only by ∼6-11% for the devices with just one LSU.

7.4.1.6 Gaussian Elimination

The single CU results for Fan1 are shown in Figure C.10. The microarchitecture only seems to effect the performance of Fan1 in two changes to the configuration: (1) increasing the number of ALUs to two in the 2-wide configuration and (2) where the issue width increases from two to four; with both enchancements to the architecture yielding the same performance increase of ∼8%. Again, as with other kernels, the 2-wide devices with two ALUs suffer an increase in IF stalls, shown in Figure C.11; the reduction in the stalls leads to the improvement seen in the 4-wide devices. The best configuration is the 4-wide with two ALUs and one MUL and LSU as it performs as well as the largest configuration.

The response of Fan2, shown in Figure C.12, is not so flat, but the same improvement is observed when doubling the issue width from two to four and the small spikes are where

there is a mismatch in the number of LSUs and DRAM banks. Figure C.13 shows an increase in IF stalls for the 2-wide devices, with two ALUs, which limits the effects of any ILP discovered in these devices. The 4-wide devices achieve ∼16% for Fan1 whereas the same configurations vary between ∼12-16% for Fan2 as performance improves up to three ALUs and IF stalls are more varied. The most effective configuration for this kernel is the 4-wide with three ALUs, two MULs and one LSU.

7.4.1.7 Matrix Transpose

The response of matrix transpose is very volatile and highly dependent of the memory configuration, the greatest dependence of all the benchmarks used, as presented in Figure 7.6 and 7.6. Cycle times improve for each model where the number of LSUs are increased, along with the number of DRAM banks to support them; each on the prodominant peaks represents where the number of banks does not match the number of LSUs and these configurations perform significantly worse than the scalar device by ∼9-16%. The IF stalls are also volatile: the reduction in IF stalls for the 4-wide machines occurs in this benchmark, but there are also general increases for the larger configurations which peak when two LSUs are combined with two DRAM banks. The NOPs are also affected by the number of LSUs as these only vary in the 4-wide devices as the LSUs varies between one and two. The graphs show that there is very little to gain from increasing the complexity of microarchitecture beyond a 4-wide device with two of each FU and two banks, this device executes 21.38% faster than the scalar while the largest configuration achieves 22.01%.

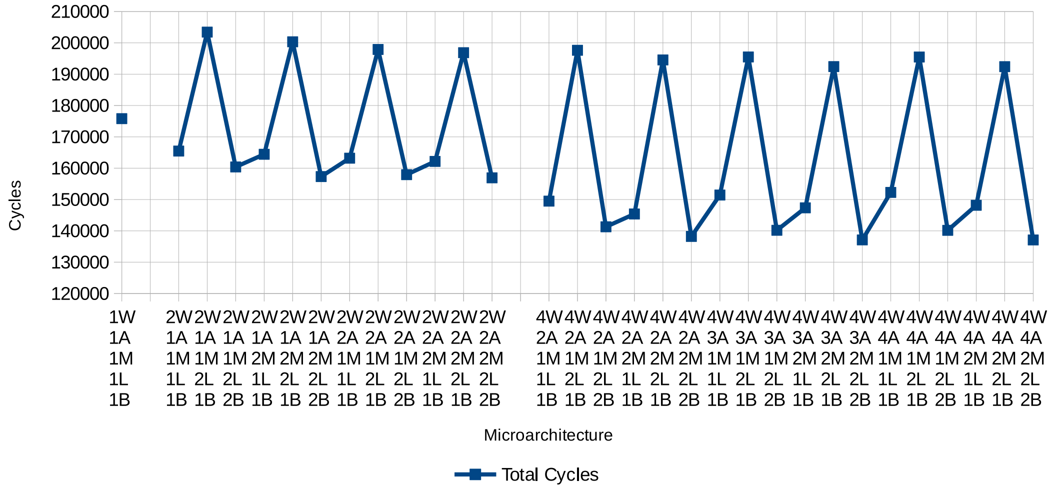

Figure 7.5: Total cycle count for MatrixTranspose using 1 context across varying microarchitecture configurations.

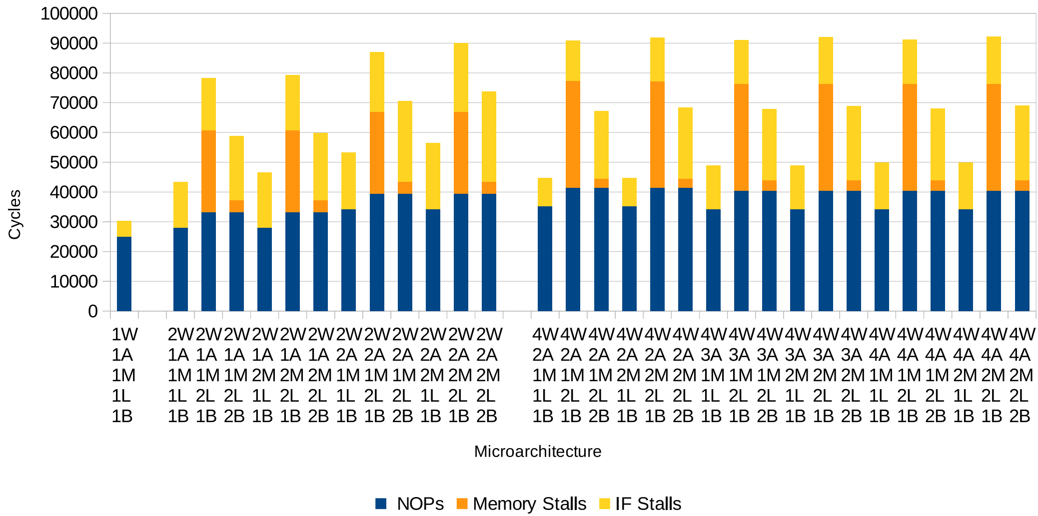

Figure 7.6: Total stall and NOP cycle count for MatrixTranspose using 1 context across varying microarchitecture configurations.

7.4.1.8 NBody Simulation

Single CU performance for NBody is shown in Figure C.14 with NOPs and IF stalls in Figure C.15. The 2-wide devices gain meager improvements over the scalar machine with execution cycles decreased by ∼3-5%. The 2-wide devices with two ALUs incur increases in both NOPs and IF stalls compared to the other 2-wide configurations, yet performance remains about the same; this suggests that enough ILP is discovered to counter both the NOPs and IF stalls. The number of NOPs remains relatively constant for the 4-wide machines, but the decrease in IF stalls enable these configurations to perform ∼10% better than their 2-wide counterparts. Increasing the number of ALUs to three, while maintaining one of each other FU is the most effective configuration as this performs 15.42% faster than the scalar device.

7.4.1.9 Nearest Neighbour

The single CU performance for nearest neighbour is shown in Figure C.17, the result is very similar to NBody simulation which is unsurprising since they are both complex kernels for the LE1 since they based upon floating-point calculations. The 4-wide devices do not suffer the IF stalls induced in the 2-wide devices and so execute the fastest. Doubling the issue width, while maintaining the minimal mix of FUs, results in performance gains of 5.33% and 10.34% with the largest configuration achieving a 13.72% reduction in cycles. The most effective configuration is the 4-wide device with three ALUs, one MUL and two LSUs with matching DRAM banks; this executes 14.33% faster than the scalar.

7.4.1.10 Needleman-Wunsch

The single CU performance of both kernels for the Needleman-Wunsch benchmark are presented in Figures 7.7 and 7.8. The response of nw_kernel1 is quite flat in the 2-wide devices, but the performance of the 4-wide machines varies with differences in the IF and memory access stalls; as presented in Figure C.18. The effect of the large increase in IF stalls in the 2-wide, two ALU, devices is noticeable in the total cycles; though the performance degradation is less than expected which suggests ILP exploited to help negate the issues. Differences in the performance can also be seen in nw_kernel2 and these too correlate to the differences the in the number of IF stalls observed and the memory access stalls to a lesser extent, as shown in Figure C.19. As the variations in IF stalls appear to negate most of the potential performance increases from ILP, only 4.64% and 8.37% improvements are achieved in nw_kernel1 and nw_kernel2 respectively.

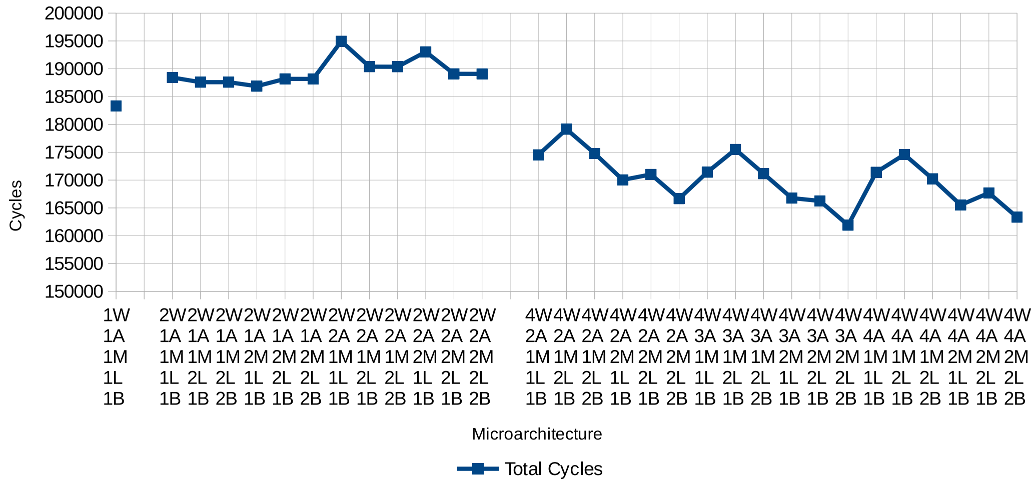

Figure 7.7: Total average cycle count for nw_kernel1 from Needleman-Wunsch using 1 context across varying microarchitecture configurations.

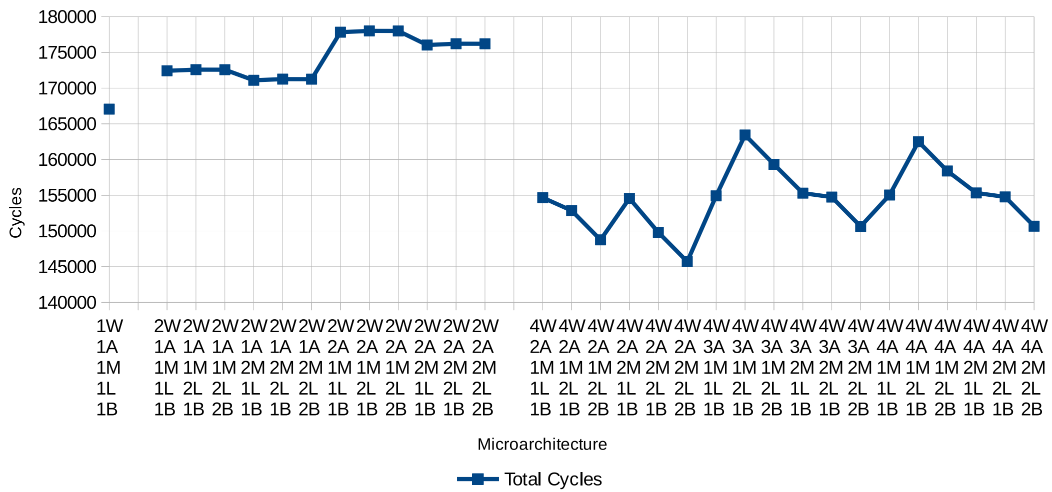

Figure 7.8: Total average cycle count for nw_kernel2 from Needleman-Wunsch using 1 context across varying microarchitecture configurations.

7.4.1.11 Radix Sort

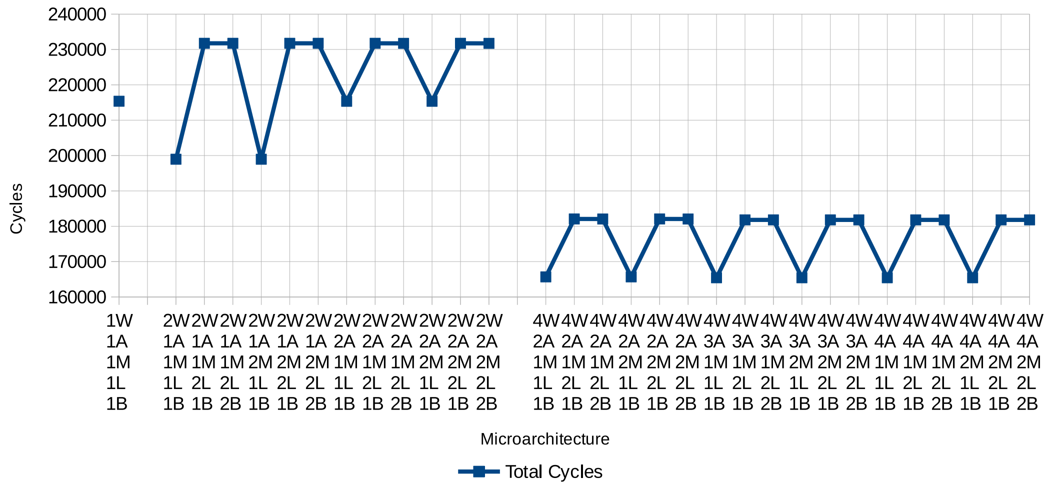

Figure 7.9: Total average cycle count for FixOffset from Radix Sort using 1 context across varying microarchitecture configurations.

The single context performance for the FixOffset kernel, from Radix Sort, is depicted in Figures 7.9 and 7.10. The response to the microarchitecture is mainly dominated by the effect of the memory stalls that occur whenever two LSUs are used. For the 2-wide devices with a one ALU, these memory stalls also coincide with an increase in IF stalls resulting in only two of the 2-wide machines to perform better than the scalar device. These devices achieve a 7% speedup while the others achieve no speedup for perform 7% worse. The 4-wide machines gain a significant decrease in the IF stalls resulting in them performing 15-23% faster than the scalar configuration. There is no gain from increasing the complexity of the microarchitecture past a 4-wide with two ALUs as this configuration performs as well as the largest single context device.

The single context performance for the histogram kernel, from Radix Sort, is shown in Figure 7.11 while the memory and IF stalls are in Figure C.20. Little ILP is found in the kernel and it is largely invariant to the microarchitecture except for when the number of LSUs and banks are mismatched and the large decrease in IF stalls in the 4-wide machines with three ALUs. The fastest configuration is 4-wide with three ALUs, one MUL and one LSU as the memory stalls from using two LSUs, as well as the increase in IF stalls when using four ALUs, causes the largest configurations to execute slightly slower.

For the permute kernel, presented in Figures C.21 and C.22, little ILP is found once again. However, the ~50,000 IF stall cycle increase in the 2-wide devices is not reflected in

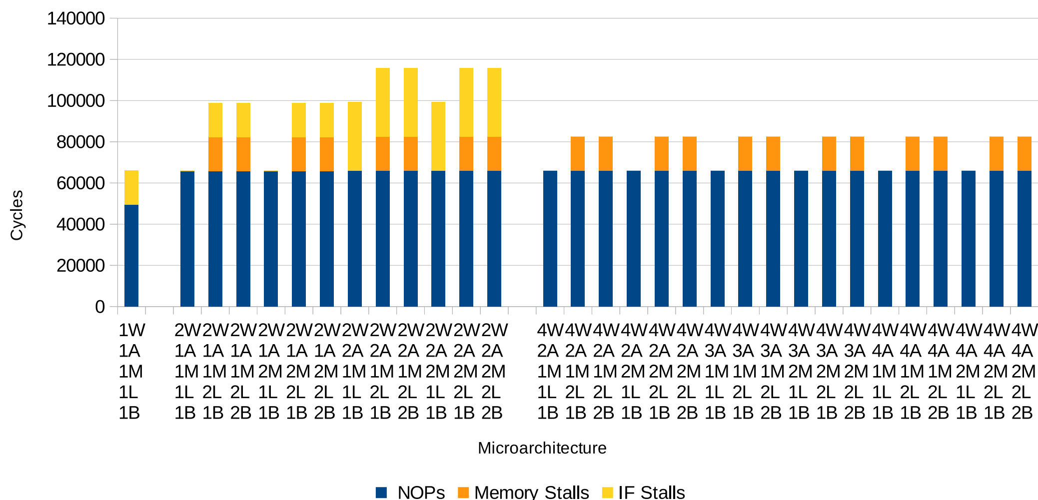

Figure 7.10: Total average stalls and NOP cycle count for FixOffset from Radix Sort using 1 context across varying microarchitecture configurations.

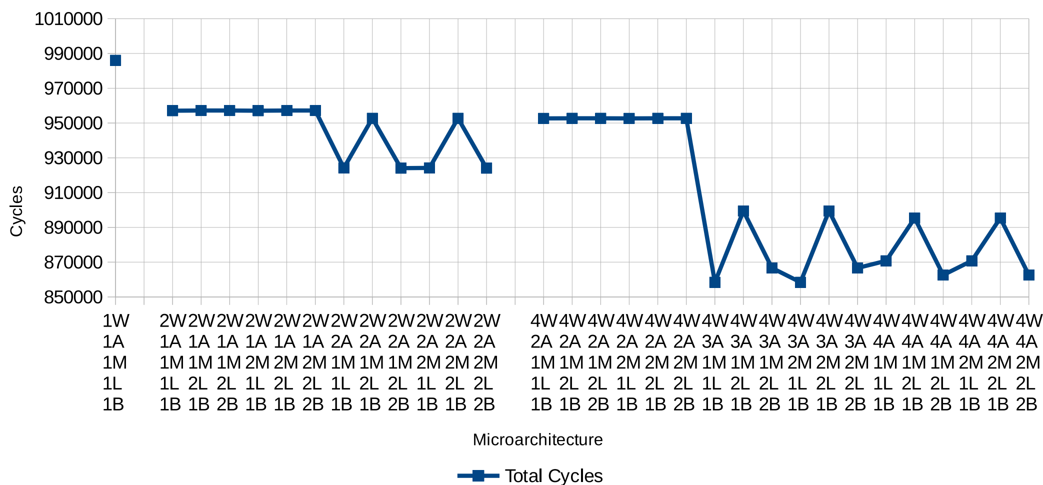

Figure 7.11: Total average cycle count for histogram from Radix Sort using 1 context across varying microarchitecture configurations.

the total cycle count. For the devices with one ALU this suggests that the ILP discovered is just enough to counter the stalls, and with two ALUs the increase is overcome and the devices perform 6% faster. The decrease in IF stalls for the 4-wide configurations enables them to perform 12-13% faster than the scalar machine. The response of ScanArraysdim1,

in Figure C.23, is very similar to that of histogram but more ILP is discovered; with the 2-wide devices achieving 7-9% and the 4-wide devices achieving an 17.75% reduction in cycles. The results for ScanArraydims2 are somewhat different as the 4-wide machines with three and four ALUs all suffer another increase in IF stalls. This increase results in those devices achieving ∼10% improvement over the scalar, compared to ∼13% for the devices with two ALUs. Memory stalls, from mismatched LSUs and banks, also contribute to the smaller performance gain in those configurations.

7.4.1.12 Reduction

The results from this benchmark show that it is largely variant to the microarchitecture configuration; Figures 7.12 and 7.13 show that this is due to memory and IF stalls. All the configurations that contain two LSUs suffer from significant memory stalls, even when there is a bank to support each LSU, and the stalls are higher in the 4-wide devices than the 2-wide. These stalls lead the performance of 4-wide devices to vary by ∼5-6%. In the 2-wide machines, the sharp reduction in IF stalls when using two ALUs leads to a performance increase. The best performing architecture each have two ALUs, one MUL and a LSU: the 2-wide achieves 10.19%, while the 4-wide achieves 13.58% reduction in cycles.

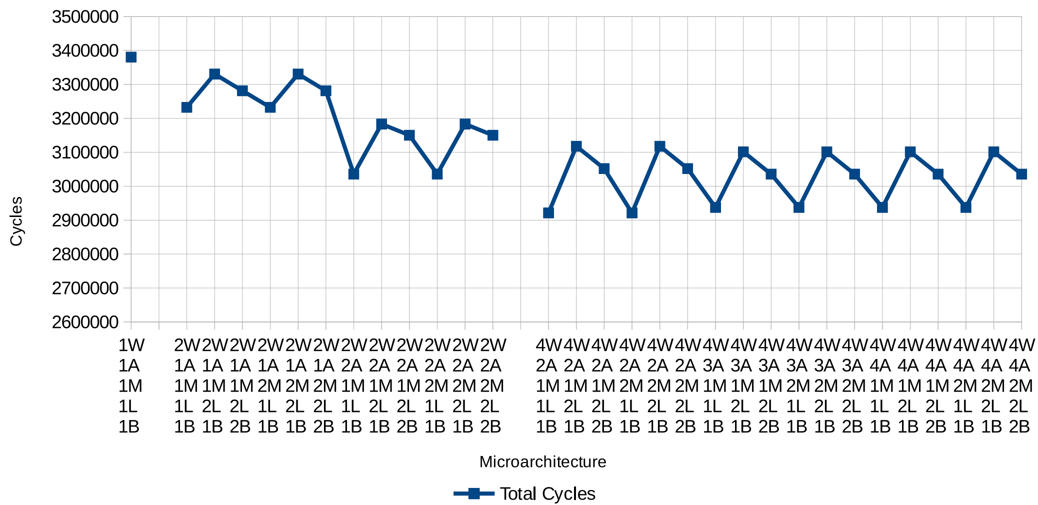

Figure 7.12: Total cycles for Reduction using 1 context across varying microarchitecture configurations.

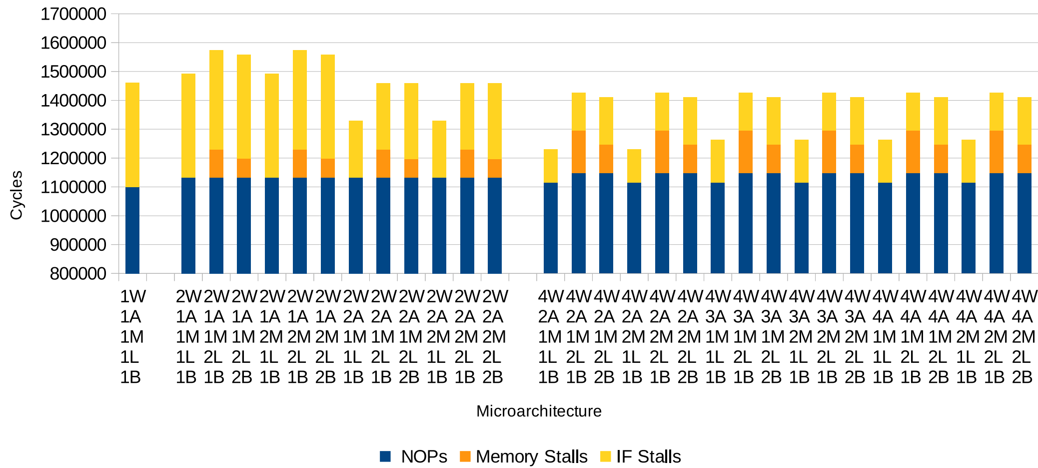

Figure 7.13: Total average stalls and NOP cycle count for Reduction using 1 context across varying microarchitecture configurations.

7.4.2 TLP Performance

This section presents the results of systems which contain more than one context with the speedup calculated against the scalar, single CU, device.

7.4.2.1 Binary Search

Figure 7.14 depicts the performance of Binary Search across the maximal microarchitectures and with multiple contexts. The graph shows that the problem scales almost perfectly for the scalar and 4-wide devices up to 4 CUs, as long as there is a DRAM bank for every four LSUs in the system. As the problem does not scale as well up to the full eight contexts, it suggests that the value is found just as fast when using six or seven contexts. The effect of the IF stalls is still apparent in the 4 CU device as the 2-wide devices still noticeably underperform both the 4-wide and scalar configurations. The most efficient configuration for this benchmark would be to use a scalar microarchitecture and a simple memory configuration of one bank between four CUs.

7.4.2.2 Breadth-First Search

The performance of Breadth-First Search across all maximal microarchitectures is presented in Figure 7.14, showing that both kernels generally scale well across the available contexts. The performance of BFS_1 varies little with the microarchitectures and scales nearly linearly up to 4 CUs as the memory subsystem then starts to limit the performance. With a memory

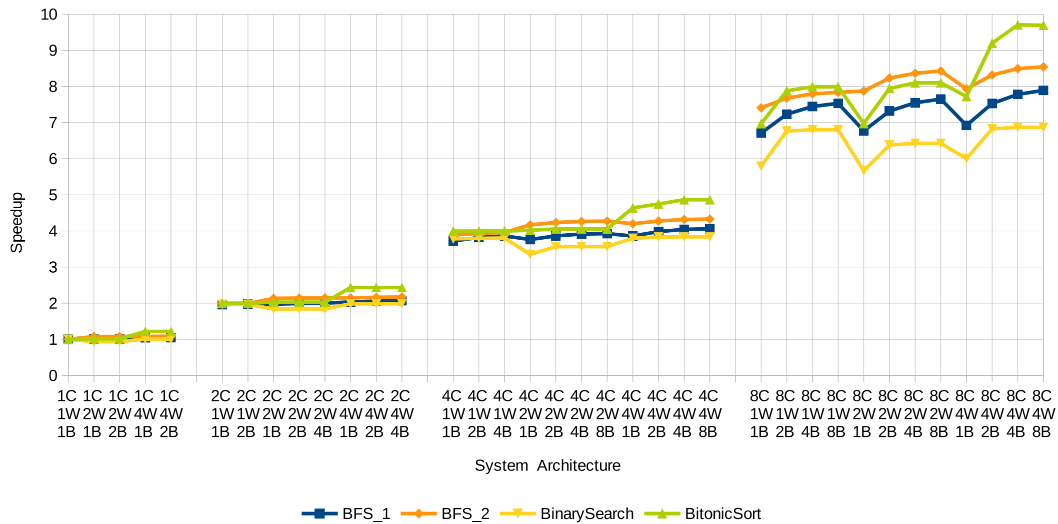

Figure 7.14: Speedup of Breadth-First Search, Binary Search and Bitonic Sort across a selection of multi-context, maximal microarchitecture, configurations.

bank per context, the performance approaches a 8x speedup compared to 7x with only a single bank. The threaded performance of BFS_2 scales better than BFS_1, being less dependent on the memory system and also gaining benefits from ILP but only up to the 2-wide machines; these achieve 8-8.5x speedup. The most effective configuration for these two kernels would be the 8 CU, 2-wide device with 8 banks.

7.4.2.3 Bitonic Sort

The performance of Bitonic Sort across all maximal configurations is depicted in Figure 7.14. The algorithm scales perfectly across the contexts as long as there's one DRAM bank for every four LSUs in the system, but there is very little performance difference between the scalar and 2-wide configurations. The performance increases from ILP in the 4-wide configurations are multiplied as the number of contexts are increased, achieving 9.5x speedup with eight LSUs and four banks. Though the difference in performance of the single- and dual bank is particularly clear in these devices, and further performance is gained from using four banks but this is continued to eight banks.

7.4.2.4 Fast Walsh Transform

The multi-context performance of the Fast Walsh Transform is presented in Figure 7.15. The graphs shows that the memory configuration does not affect performance, even when

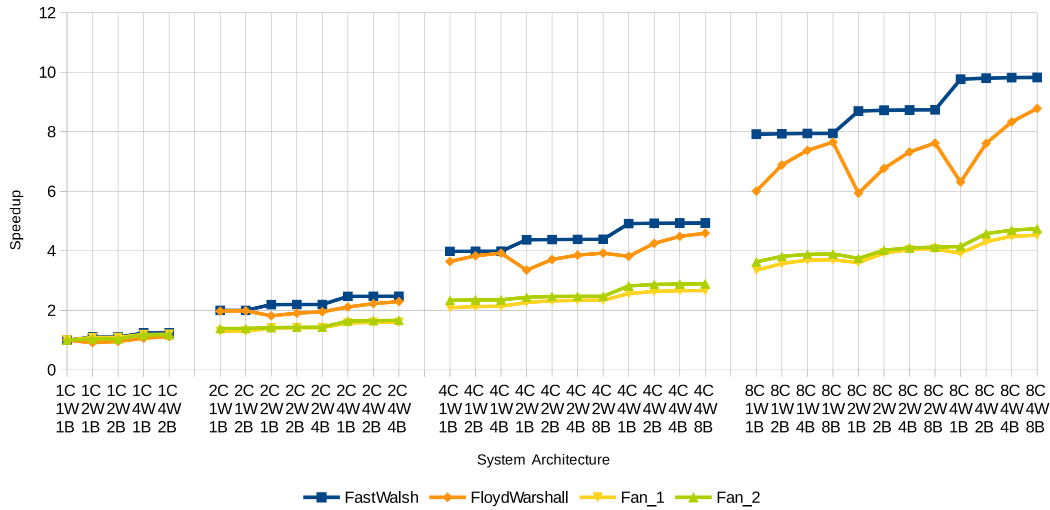

Figure 7.15: Speedup of Fast Walsh Transform, Floyd Warshall and Gaussian Elimination across a selection of multi-context, maximal microarchitecture, configurations.

there is sixteen LSUs in the system with only a single DRAM bank. It also shows the problem scaling perfectly with the scalar devices achieving 2, 4 and 8x speedup compared to the single CU scalar machine. The gains from ILP also scale near perfectly too: with 2.2, 4.4 and 8.7x for the 2-wide configurations, and 2.5, 4.9 and 9.8x for the 4-wide.

7.4.2.5 Floyd Warshall Pass

The results from the Floyd Warshall benchmark are also shown in Figure 7.15. The performance degradation in the 2-wide devices, due to IF stalls, continues to be evident in all the systems with the scalar configurations outperforming them; with the problem compounded by a reliance on the memory configuration. The memory configuration proves to be the bottle neck in the 8 CU systems, with all configurations achieving ∼6x speedup when only a single bank is used. This increases to 7.7 for the scalar, 7.6 for the 2-wide and 8.8x for the 4-wide devices with eight memory banks.

7.4.2.6 Gaussian Elimination

Complete system performance for the two kernels from Gaussian Elimination are depicted in Figure 7.15. It shows that neither Fan_1 or Fan_2 scale well across the multi-context systems due to the variable number of workgroups that are instantiated throughout the execution of the program. The increase in FUs and memory banks leads to marginal performance

improvements throughout the systems for both kernels, the smaller memory configurations only become a bottleneck in the 8 CU systems but it is not significant. The single CU performance of Fan_1 is slightly greater than Fan_2, with 1.2x compared to 1.18x for the 4-wide, but the problem scales better over multiple contexts for Fan_2 which achieves a 4.74x speedup in the largest system compared to 4.5x for Fan_1.

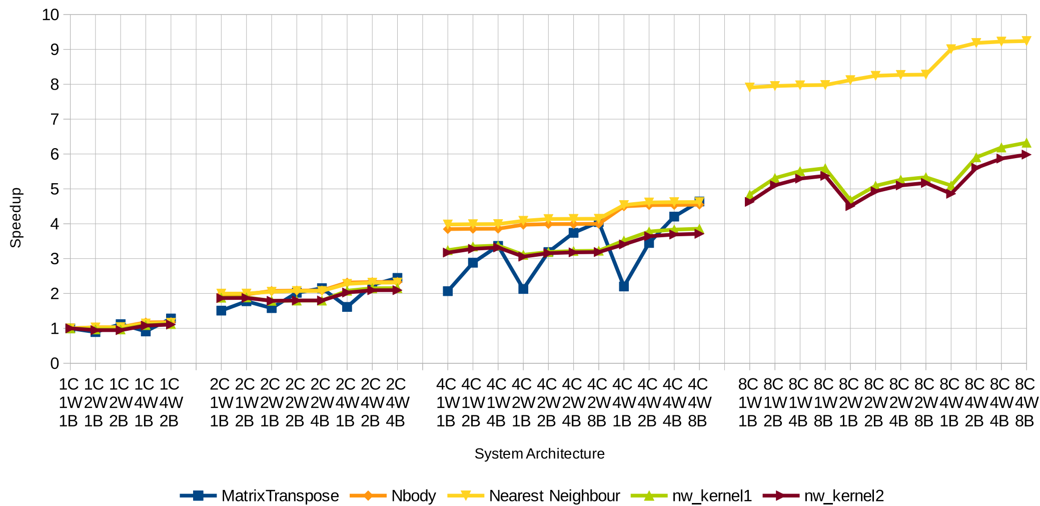

Figure 7.16: Speedup of Matrix Transpose, NBody Simulation, Nearest Neighbour and Needleman-Wunsch across a selection of multi-context, maximal microarchitecture, configurations.

7.4.2.7 Matrix Transpose

The size of the dataset only required the use of four workgroups, so the results from the 8-context device are not shown, the rest are shown in Figure 7.16. The graph shows that because the kernel is memory bound even in a single CU device, the problem is exasperated as more CUs are used in the system and the problem does not scale very well. Though there is a significant amount of ILP available, with the largest single CU device achieving 1.3x speedup, the 2- and 4 CU systems only achieve 2.4 and 4.6x speedups respectively. The performance also more than doubles in the 4-wide microarchitectures when increasing the number of banks from one to eight.

7.4.2.8 NBody Simulation

Like MatrixTranspose, the NBody simulation was only split into four workgroups so only single-, dual- and quad-CU devices are shown in Figure 7.16. Unlike the matrix transpose

kernel, NBody scales very well with the memory configuration having little effect on system performance; The IF stalls continue to reduce the effectiveness of the 2-wide configurations, but they still achieve 2- and 4x speedup over the scalar, single CU device. While the largest scalar system achieves a 3.85x speedup and the largest 4-wide device achieves 4.55x. This algorithm is more parallel than these results show, if the maximum workgroup size had been set to 128 instead of 256, eight contexts could have been instantiated and most likely very nearly further double the performance with a single memory bank.

7.4.2.9 Nearest Neighbour

The complete set of results from the Nearest Neighbour kernel is presented in Figure 7.16. The performance is similar to NBody because they both contain complex emulated floating-point operations, but this scales to the full 8 CUs. Again, performance from the 2-wide configurations is disappointing but the 4-wide machines are capable of multiplying their ILP gains across all the cores without being hindered by simple memory configurations; with the largest microarchitectures scaling at 1.16, 2.31, 4.62 and 9.24x speedup over the scalar, single CU, machine. The performance of the scalar systems also scales perfectly with 2, 3.99 and 7.98x over the single CU system.

7.4.2.10 Needleman-Wunsch

The two kernels from Needleman-Wunsch are shown in Figure 7.16, which both exhibit very similar charateristics. Neither kernel scales particuarly well due to the varying number of workgroups that are executed during the iterations of the program. It also shows all the 2-wide configurations perform worse than their scalar counterpart for both of the kernels. The memory configuration has a negligible effect on performance for both the single and dual CU devices, with performance only seriously affected in the 8 CU devices where, for both kernels, the largest configurations vary by 23.5%.

7.4.2.11 Radix Sort

The permute and ScanArraysdim1 kernels both operate upon a single workgroup so their results are not shown in this section, the results of the other kernels are presented in Figure 7.17. The multi-context results for the FixOffset kernel is uninteresting until eight contexts are instantiated. For the other systems, there is little difference in performance in relation to the microarchitecture and the problem scales linearily to two contexts, but then falls short for most of the 4 CU machines. The response for the systems with eight contexts exhibits a completely different response with the microarchitecture varying the performance by 2.5x. The total stalls across the systems in shown in Figure 7.18 and it helps explain the different

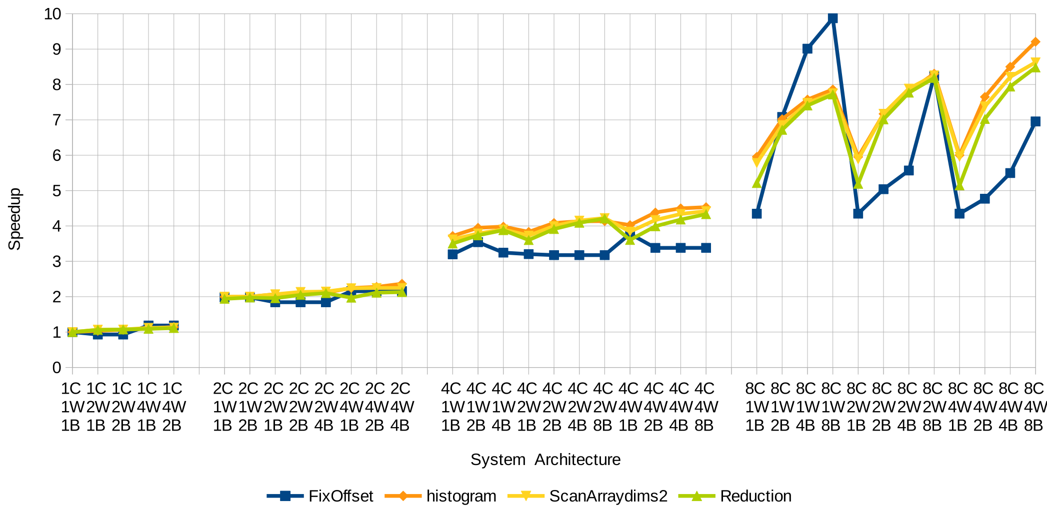

Figure 7.17: Speedup of the kernels of Radix Sort and Reduction across a selection of multi-context, maximal microarchitecture, configurations.

overall performance. The total number of NOPs and stalls encountered by the dual CU systems is almost the same as the single CU machines, but the total work has been halved between the contexts which results in double the performance. In the 4 CU systems, with multi-issue microarchitectures, the number of memory stalls greatly increases but generally is not related to the number of banks in the system. It was shown in the previous section that memory stalls were encountered whenever there was two LSUs in the device, regardless of the number of banks; here that continues but the issue is compounded by the number of contexts. The memory stalls then vary throughout the various microarchitectures of the 8 CU machines, with all configurations benefitting from a higher number of banks. The scalar devices suffer far fewer memory stalls than the multi-issue microarchitecture, by ∼6x in the greatest extreme, keeping the stalls inline with the scalar, single CU, machines. This would suggest that this configuration would achieve 8x the performance of the single CU system, instead it is nearer 10x; this is because the inner loop gets unrolled and results in an 8-fold reduction in the number of branches taken.

The performance of both the histogram and ScanArraysdim2 are very similar, with all microarchitectures scaling perfectly from single to dual context systems. The memory configuration begins to play a part in the performance of the 4 CU systems. For histogram, the single- and dual- CU devices continue to scale perfectly with enough banks to support and the 4-wide devices only fall short very slightly. ScanArraysdim2 still scales very well, but just behind the rate of histogram. This pattern continues for the 8 CU, 8 banked, machines

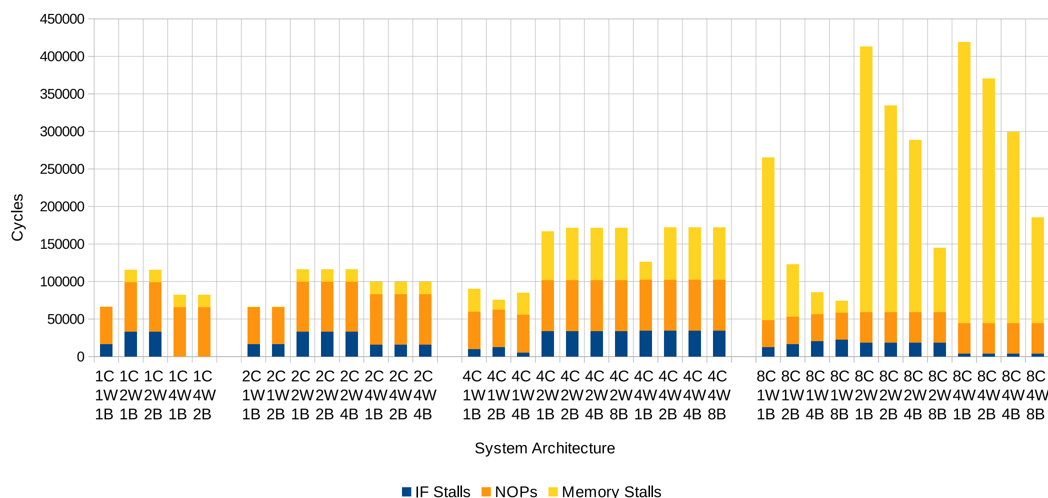

Figure 7.18: Total stalls and NOPs of FixOffset, from Radix Sort, across maximal microarchitecture systems.

with histogram achieving 7.86, 8.31 and 9.21x, while ScanArraysdim2 achieves 7.75, 8.25 and 8.63x.

7.4.2.12 Reduction

The speedups of maximal microarchitecture, multi-context systems for the Reduction benchmark are shown Figure 7.17. The problem scales well across the multiple contexts, but is sensitive to the memory configuration; with all the 8 CU microarchitectures being bottlenecked at 5x speedup when there is only a single DRAM bank. The wider machines benefit from ILP gains too with the 8 CU, 8 banked, machines achieving: 7.72, 8.18 and 8.48x speedup for the scalar, 2-wide and 4-wide respectively.

7.4.2.13 TLP Performance with Silicon Data

This subsection uses silicon data, in Table 7.3, together with the simulated results to gain a greater understanding of how the LE1 systems operate as more contexts are instantiated. The data was acquired using the UMC65nm process with the LE1 configured to have a 256KB data RAM and 16KB instruction RAM. The speedup presented in the following graphs is relative to the single context device.

Figure 7.19 shows that the slight decrease in Fmax results in less than 2x speedup, especially for the kernels of gaussian elimination as these have already been shown not to scale well. Most of the kernels achieve ~1.65-1.77 speedup while FixOffset achieves

Table 7.3: LE1 silicon data.

Contexts	Microarchitecture	Fmax (MHz)	Power (nw)
1	2W-2A-1M-1L-4B	413.1	35571698
2	2W-2A-1M-1L-4B	357	51893081
4	2W-2A-1M-1L-4B	384.9	85317717
6	2W-2A-1M-1L-4B	278.8	121000000
8	2W-2A-1M-1L-4B	377.6	146053150
10	2W-2A-1M-1L-4B	317.1	210468449

1.85x. The increase in Fmax for the 4CU machines results in better scaling for most of the benchmarks: ten benchmarks achieve over 3.5x, five achieve 3x while Fan1 and Fan2 achieve 1.87 and 2.15 respectively; as shown in Figure 7.20.

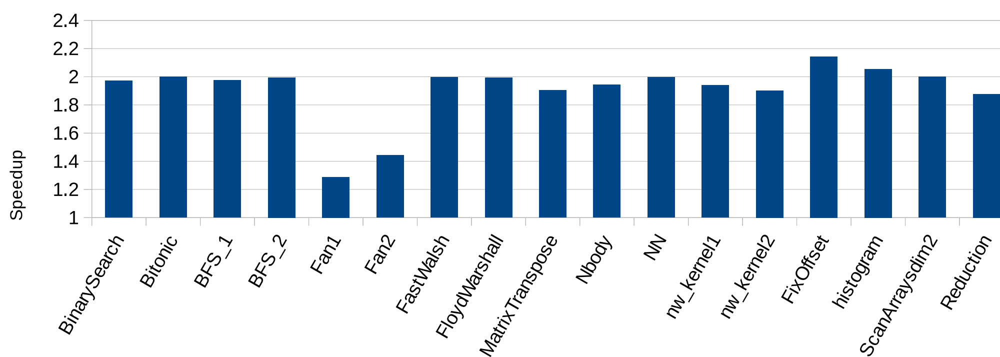

Figure 7.19: Speedup of a 2W-2A-1M-1L-4B LE1 system with two contexts using silicon data.

The achievable clock frequency for the 6 CU machine is significantly less than the single context, by 1.48x, which is evident from the results in Figure 7.21 as most perform slower than with the 4 CU machine. Neither NBody or MatrixTranspose scale beyond four contexts, but the results have been presented to show the detrimental affect that the system configuration can have. As neither of these kernels benefit from the extra two contexts, the significant decrease in clock frequency results in both performing 1.38x slower than the 4CU device. From the simulated results, NN scales perfectly to achieve a 6x speedup with FloydWarshall, Reduction, histogram and ScanArraysdim2 all achieving between $\sim 5.4 - 5.7$; these are all reduced to under 4x in real-world terms. FixOffset even super scales to achieve 9.1x speedup, which is cut to 6.1 once the clock frequencies are considered. Both BinarySearch and FastWalshTransform see performance decreases as the kernels do not utilise the extra two cores.

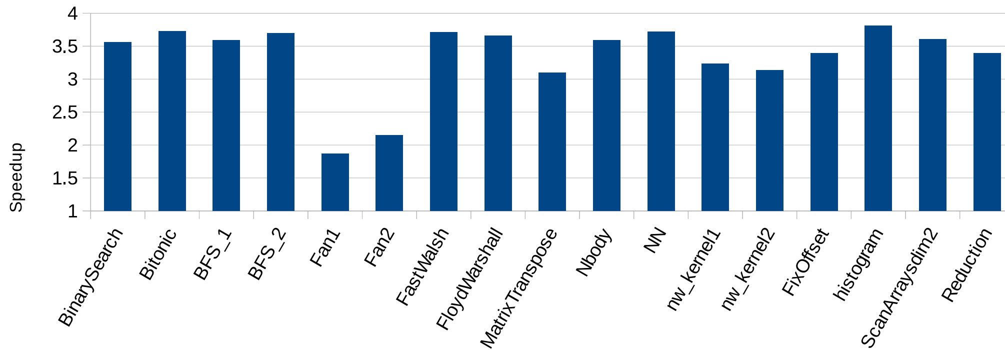

Figure 7.20: Speedup of a 2W-2A-1M-1L-4B LE1 system with four contexts using silicon data.

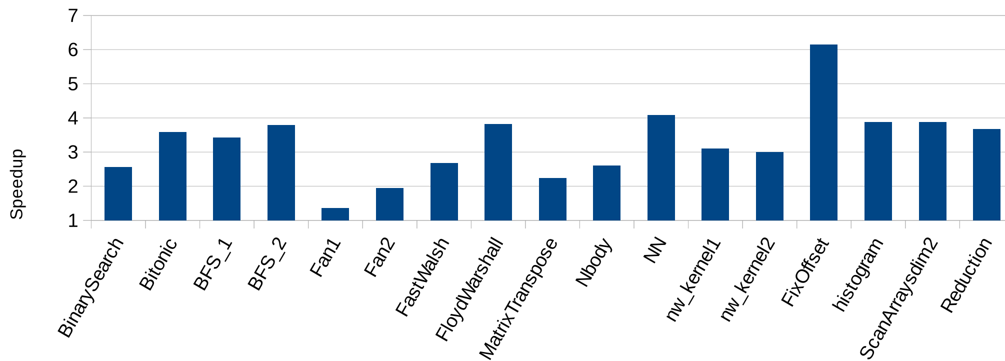

Figure 7.21: Speedup of a 2W-2A-1M-1L-4B LE1 system with six contexts using silicon data.

The clock frequency of the 8 CU device is increased to a more competitive level; allowing both BinarySearch and FastWalshTransform to execute over 2.5x faster than the 6 CU configuration, and enables most benchmarks to execute over 6x faster than the single context machine. The results are presented in Figure 7.22. The gaussian elimination kernels continue to gain no benefit from the extra cores and Needleman-Wunsch kernels continue to scale worse than the others. The two kernels from breadth-first search equal out to a 7x speedup and both BitonicSort and FastWalshTransform execute over 7.2x faster. The multi-context kernels from RadixSort also perform very well, with FixOffset performing 9x faster and histogram 6.8x. The reduction in clock frequency for the 10 CU machine results in all but three benchmarks executing slower than the 8 CU device; as depicted in Figure 7.23. Reduction stays constant while NN and ScanArraysdim2 achieve 1.06 and 1.04x speedups. These results show that the largest system is not necessarily the best, especially in the cases

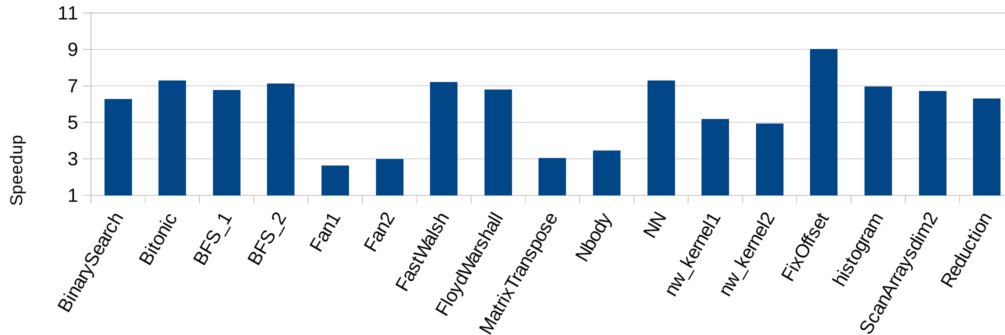

Figure 7.22: Speedup of a 2W-2A-1M-1L-4B LE1 system with eight contexts using silicon data.

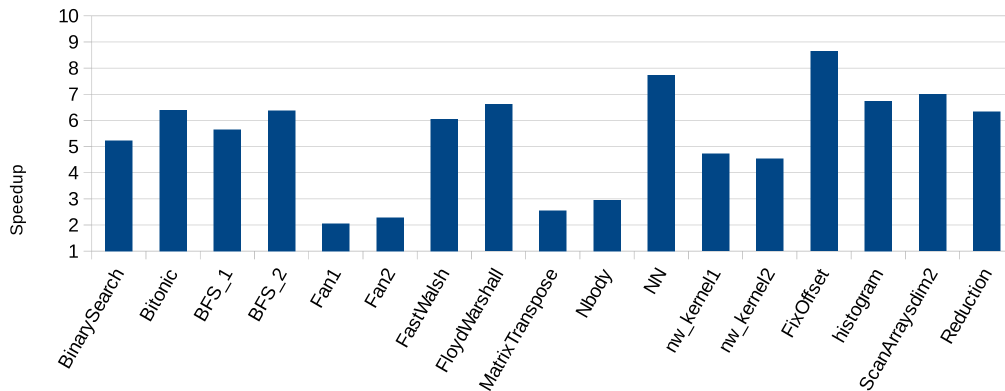

Figure 7.23: Speedup of a 2W-2A-1M-1L-4B LE1 system with ten contexts using silicon data.

where the algorithm does not scale across all the contexts and so performs better with fewer cores, but clocked at higher frequencies.

7.4.3 Development Branch

This subsection presents a comparison of the results from four benchmarks using between the original baseline compiler (3.2), using loop unrolling (3.2 + unroll) and the updated compiler (3.4) with a combination of optimisations. Loop unrolling was also forced for the updated compiler along with utilising the custom optimisation to remove the IF stalls that were identified in the previous section. Both the loop unrolling and alignment increase code size: loop unrolling duplicates the loop body a number of times which has to increase code size proportion to the number of bodies unrolled, while alignment inserts unnecessary

instructions and will vary dependent on the width issue and ILP. Table 7.4 summarises the average size of the IRAM across these four benchmarks and compiler optimisations, showing that LLVM 3.4 produces smaller code than 3.2 for these benchmarks and that the alignment optimisation, on average, roughly doubles the size required by the IRAM.

Table 7.4: Average IRAM sizes across all configurations and compilers.

Compiler	BinarySearch	BitonicSort	FloydWarshall	Reduction
3.2	508	567	679	911
3.2 + unroll	1127	1437	5607	10654
3.4	416	476	568	726
3.4 + unroll	1109	1166	6445	9430
3.4 + align	926	969	1282	1590
3.4 + unroll + align	2683	2118	15796	20982

7.4.3.1 Binary Search

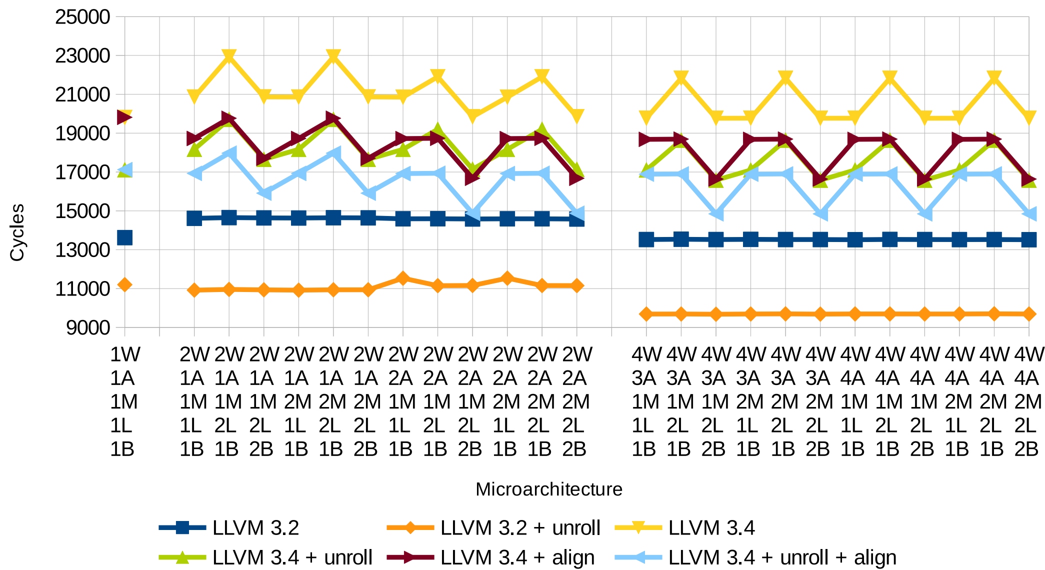

Figure 7.24: Total cycles of BinarySearch, on the single CU devices, using both compilers and optimisations.

Figure 7.24 shows the total cycle performance of BinarySearch using both the compiler backends and the applied optimisations. Loop unrolling with LLVM 3.2 results in an average improvement of 25.86% and this derived from the reduction in misprediction penalties as

less branches are executed due to the unrolling. The performance remains largely unaffected by the microarchitecture except for a small improvement in the 4-wide devices compared to the 2-wide machines.

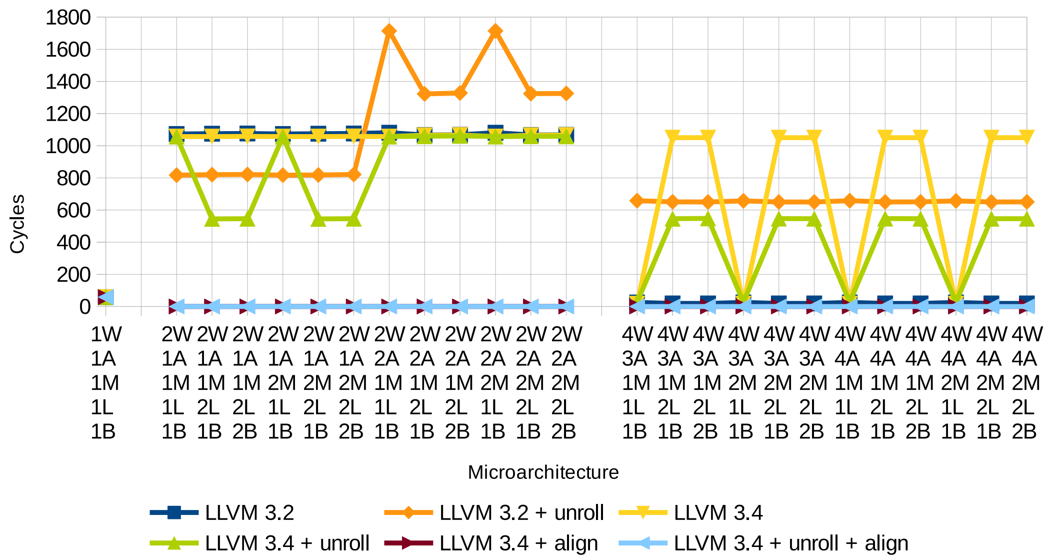

Figure 7.25: Total IF stall cycles of BinarySearch, on the single CU devices, using both compilers and optimisations.

The results of LLVM 3.4 however show a much different response than with the original backend, and only for two configurations and with maximum optimisations does the newer compiler manage to match the original compilers performance. The variations in the performance of the newer compiler is explained by the variations in both the memory and IF stalls, which are presented in Figures 7.25 and 7.26. The updated compiler suffers from memory stalls for each configuration with two LSUs with a single bank, with additional IF stalls in the 4-wide configurations when two LSUs are used. The alignment optimisation is however successful in removing the IF stalls when invoked, and this optimisation results in the 4-wide configurations performing no better than the 2-wide devices. The performance of the updated compiler is also decreased by the number of NOPs it creates, it executes ∼3000 more NOPs than the original. On average the updated compiler performs 49.69% worse than the original, brought closer to 17.48% when all the optimisations are switched on. The average improvement from unrolling is also lower than the orignal compiler, with 14.87%, though the average reduction of stalls and NOPs is the same at ∼4000 cycles combined; but this is a lower percentage of the total cycles.

The multi-context, maximal microarchitecture performance speedups are presented in

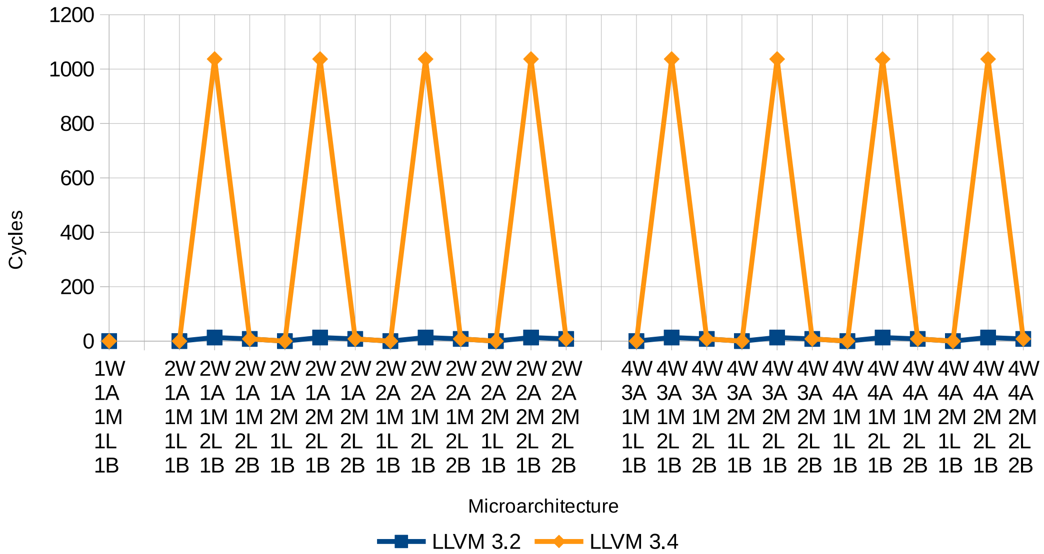

Figure 7.26: Total memory stall cycles of BinarySearch, on the single CU devices, using both compilers.

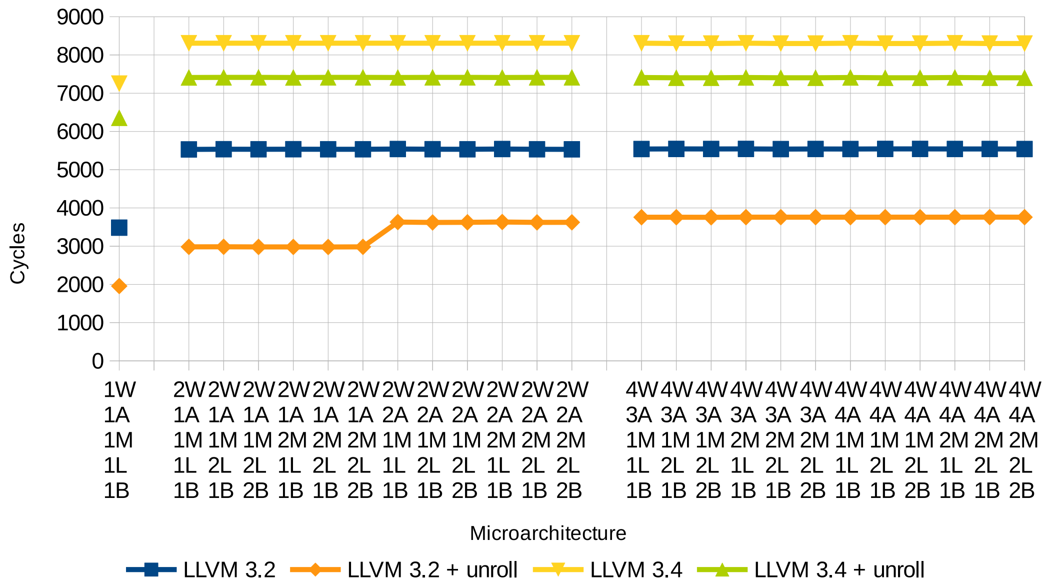

Figure 7.27: Total NOP cycles of BinarySearch, on the single CU devices, using both compilers.

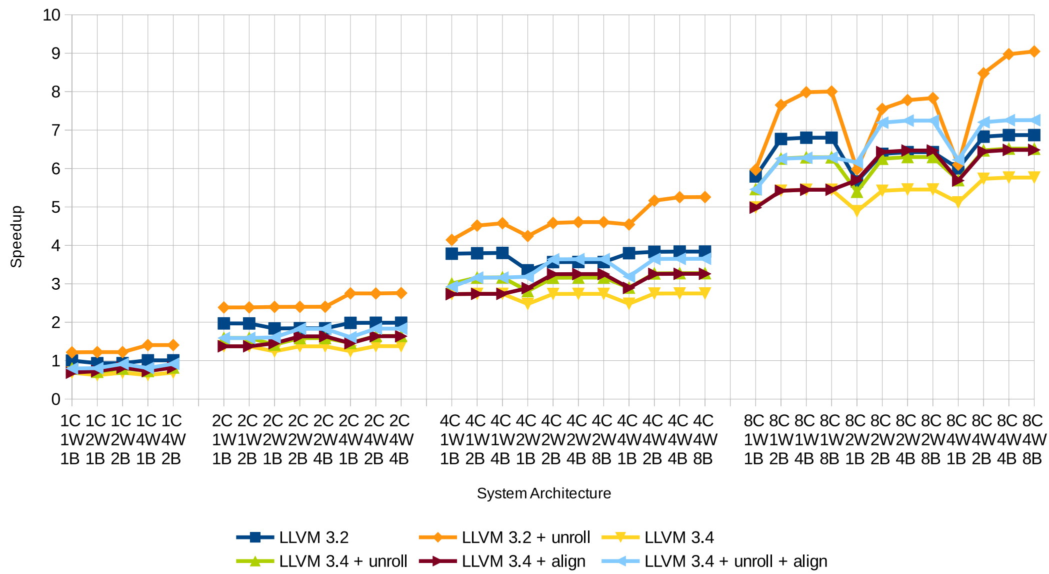

Figure 7.28: Speedup of BinarySearch, relative to a single CU ,scalar device, and LLVM 3.2, across the maximal microarchitecture configurations.

Figure 7.28. The performance advantage of loop unrolling in the original compiler is highlighted as it constantly outperforms the other compilers, except for when the memory configuration bounds performance. The updated compiler with unrolling and alignment is able to compete with the baseline original compiler more as more contexts are instantiated. The performance of the 2-wide devices match the baseline for two and four contexts, and is faster in the eight context machine.

7.4.3.2 Bitonic Sort

The single CU performance of bitonic sort comparing the original compiler to the development branch, with optimisations, is depicted in Figure 7.29; the stalls are in Figures 7.30 and 7.31. The updated compiler performs better in the more narrow devices compared to the original baseline compiler. The response to the microarchitecture is similar, however where the original suffers from IF stalls, the updated suffers a large increase in memory stalls for devices with two LSUs. The performance of the 4-wide devices matches the original with the memory stalls persisting, and the single LSU configuration suffers from sharp increases in IF stalls. The overall average performance gain over the original baseline compiler is 5.23%.

Using loop unrolling with LLVM 3.2 results in less variation across the microarchitectures, because of more uniform IF stalls, and leads to an average improvement of 17.42%. Loop

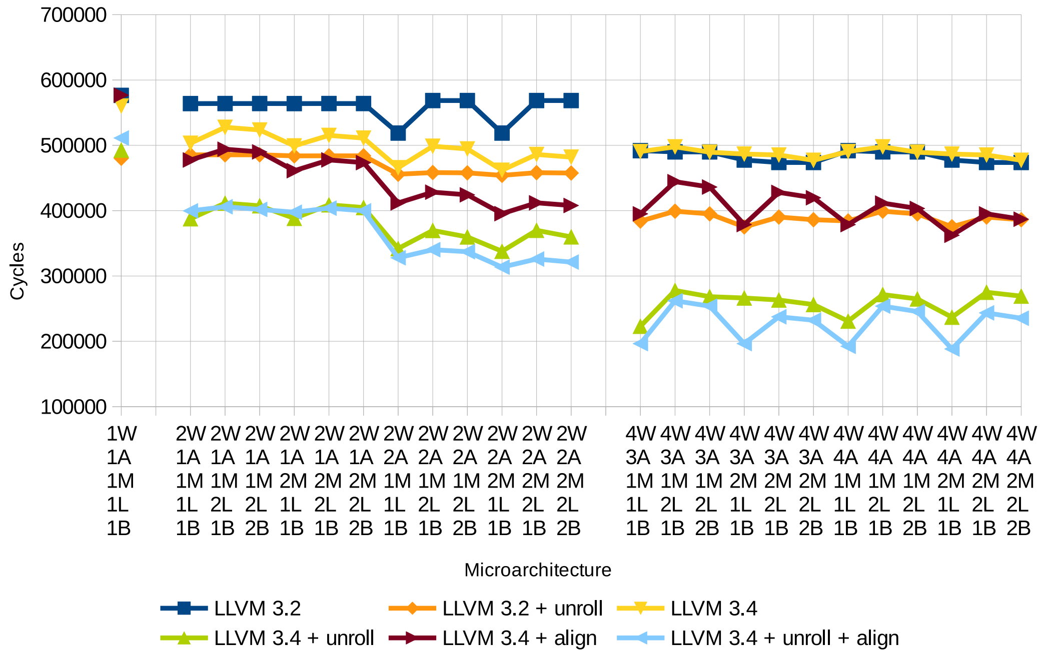

Figure 7.29: Total average cycles of Bitonic Sort, on the single CU devices, using both compilers and varying optimisations

unrolling is more effective on the development branch, improving upon the original baseline by 37.66% and on it's own baseline by 34.22%. This improvement comes from an increase in ILP as well as the reduction in miss predict penalties and modest reductions in both IF and memory stalls. The alignment optimisation successfully removes the IF stalls which enables the compiler to perform as well or better than the original with unrolling except for four configurations; these suffer from too many memory stalls. With IF stalls eradicated, the performance gain in the 4-wide devices is purely from ILP gains, which is possible from the enlarged basic block from the unrolling and partially predicted ISA being utilised effectively. The alignment optimisation enables the compiler to outperform the original by 17.56% and by it's own baseline by 13.01%. The alignment optimisation also helps to boost the performance of the loop unrolling, particularly in the 4-wide configurations, for an average improvement of 41.64% over LLVM 3.2 and 38.42% over LLVM 3.4. The best performing microarchitecture for LLVM 3.4 with unrolling is the 4-wide device with three ALUs and one of each for the rest, this performs 61.32% faster than the scalar device with LLVM 3.2. With unrolling and alignment, the fastest device uses four ALUs, two MULs and a single LSU and bank; this achieves 67.34% improvement over the original scalar result.

The multi-context performance of the maximal microarchitecture and both compilers,

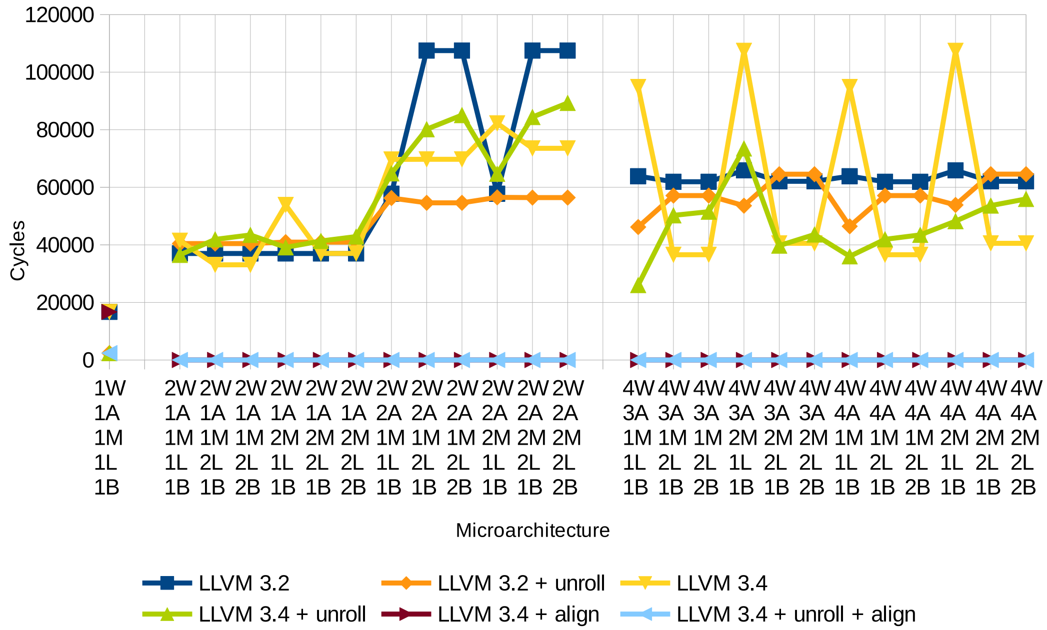

Figure 7.30: IF stalls of Bitonic Sort.

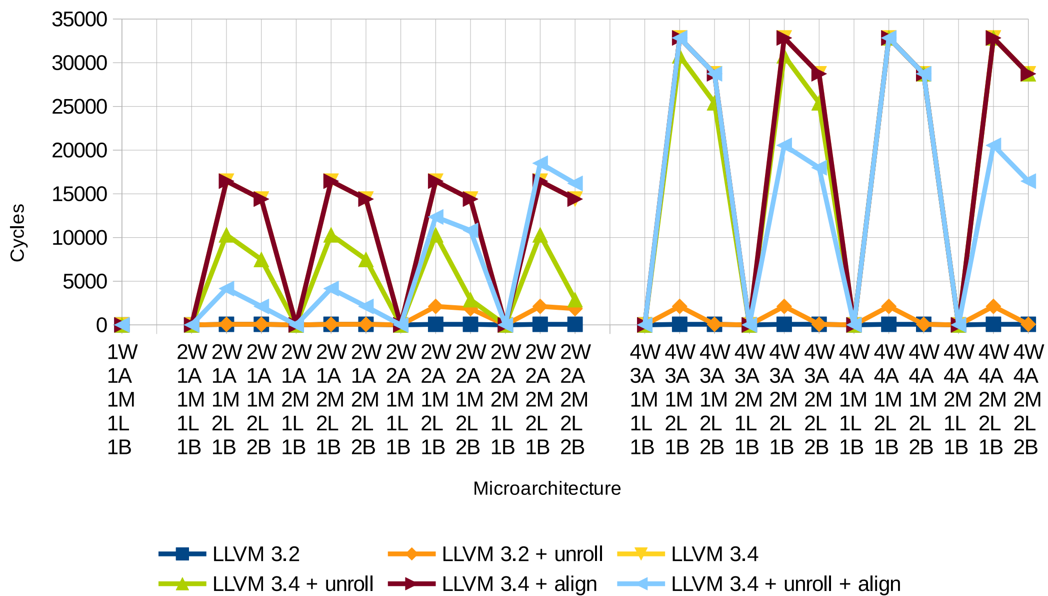

Figure 7.31: Memory stalls of Bitonic Sort across maximal microarchictectures with different compiler backends.

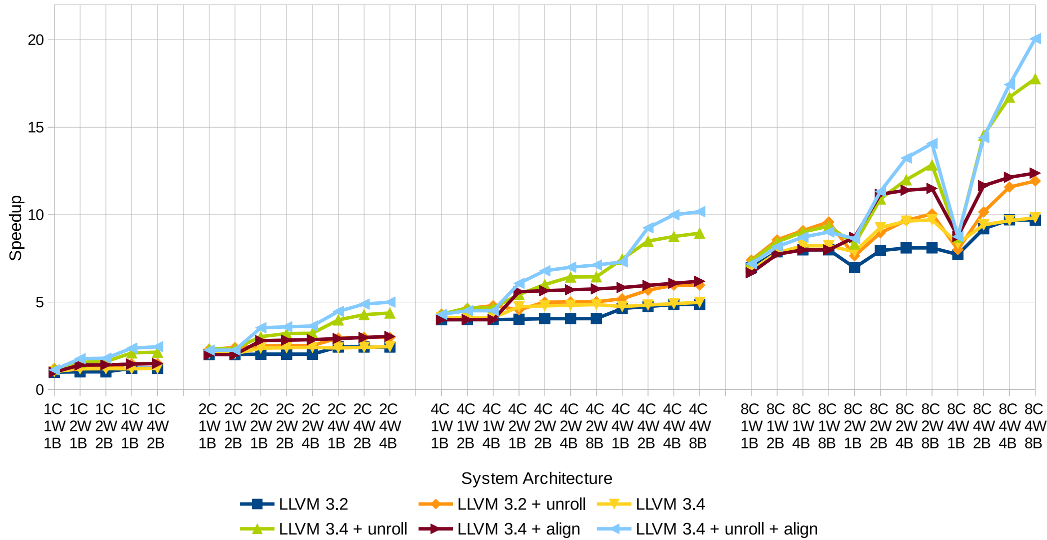

Figure 7.32: Speedup of Bitonic Sort, relative to a scalar device with LLVM 3.2, across multi-context, maximal microarchitecture, devices, using both compilers and varying optimisations.

with optimisations are presented in Figure 7.32. It shows that the updated compiler continues to outperform the original, and matches the unrolled version, for the 2-wide devices. All the 4-wide machines benefit greatly from the increase in ILP gained from loop unrolling, but the updated compiler with alignment continues to outperform the original with loop unrolling enabled. There is a clear dependence on the memory configuration in the 8 CU devices; this is particularly true for the machines that are exploiting more ILP where performance is at least halved by limiting the system to a single bank. Also, the single context analysis showed that the maximal microarchitecture was not the most effective, due to memory stalls, and this continues for the multi-context machines. Though it is not shown on the graphs, the 8C-4W-3A-2M-1L-8B device achieves a 22.95x speedup against the original compiler on the scalar device.

7.4.3.3 Floyd Warshall

Figure 7.33 shows the total average cycles of the maximal microarchitecture configurations for the two compilers; the IF and memory stalls are depicted in Figures 7.34 and 7.35. Loop unrolling is an effective optimisation for the original compiler, with the unrolled code being 1.29x faster on average. Further performance is gained in the machines with four ALUs as the devices with three still encounter IF stalls. The performance of the development branch

compiler is very similar to the original except that far more IF stalls are encountered in the 4-wide machines which makes most of these configurations perform worse. Loop unrolling is less effective for the 3.4 compiler, being 1.21x faster than it's baseline and 1.17x faster than the baseline 3.2 compiler. The unrolling is unable to counteract the memory stalls in the 2-wide devices with two ALUs and LSUs. The alignment removes the IF stalls resulting in the aligned and unrolled code running 1.28x faster than the 3.4 baseline, but only 1.24x faster than the 3.2 baseline.

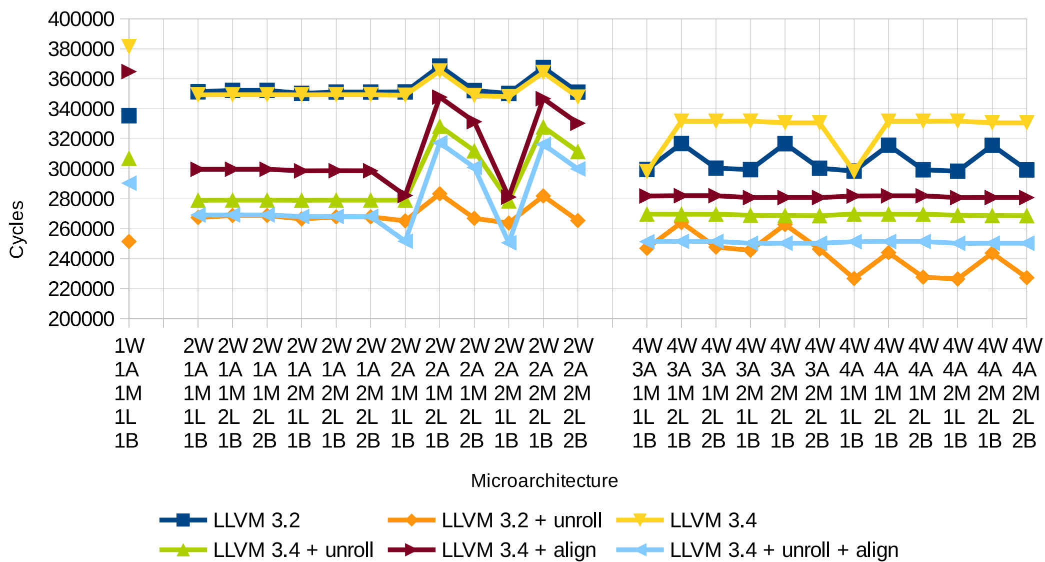

Figure 7.33: Total average cycles for FloydWarshall using the different compilers and optimisations, for the single CU devices.

The multi-context comparison is presented in Figure 7.36. The baseline development compiler comes out as the slowest performer with it's highest gain below 8x and does not gain any benefit from using 4-wide configurations. For the scalar microarchitectures, the baseline 3.2 compiler and it's unrolled version outperforms the 3.4 baseline and the aligned version however they all perform very similarily for the multi-issue machines. The highest performing is the original compiler with loop unrolling enabled, though the 3.4 compiler with full optimisations comes a close second. But in the 8 CU devices, with fully optimised development compiler, the 2-wide devices perform no better than the scalar machines as the number of memory stalls encountered using two LSUs per CU is counterproductive; configurations with a single LSU should be capable of gaining the 10x speedup of the largest device.

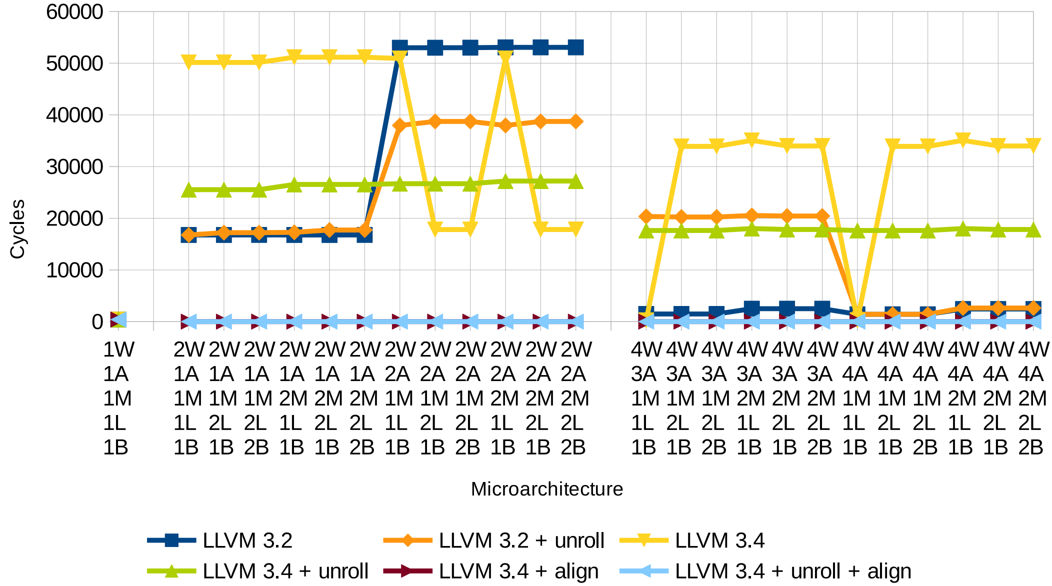

Figure 7.34: Total average IF stalls of FloydWarshall for the single CU devices, using both the compiler backends.

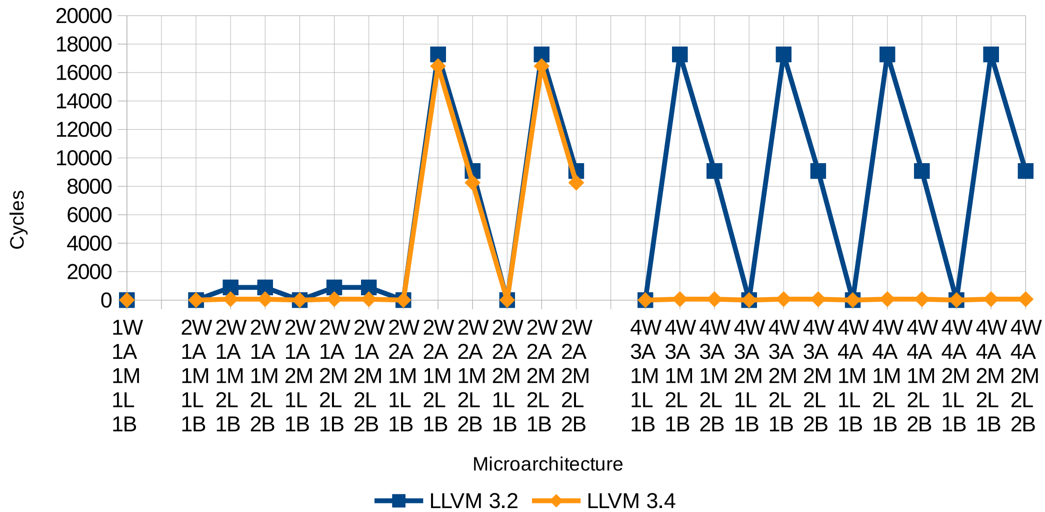

Figure 7.35: Total average memory stalls of FloydWarshall for the single CU devices, using both the compiler backends.

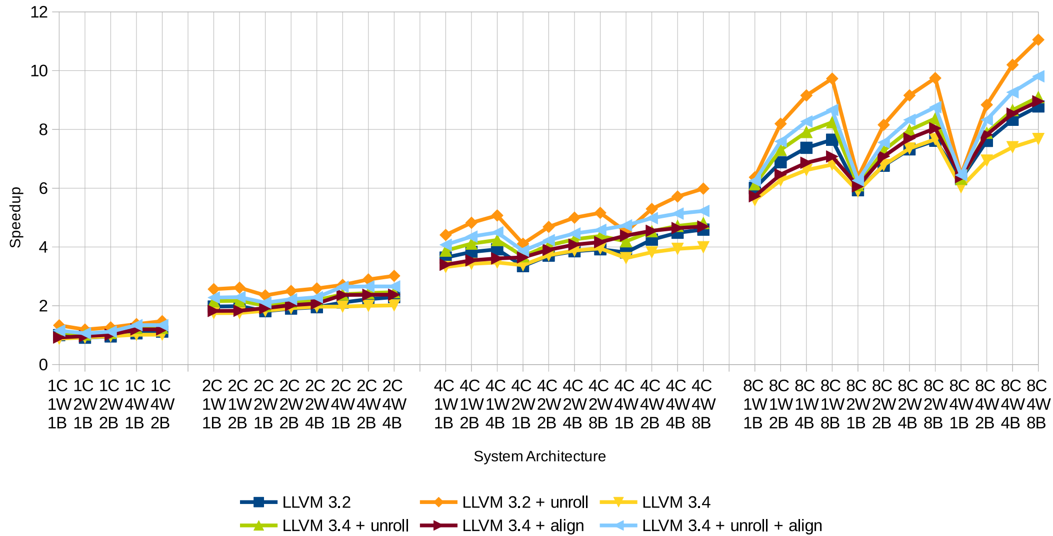

Figure 7.36: Speedup, with respect to a scalar device and LLVM 3.2, for FloydWarshall using different compilers and optimisations across multi-CU systems of maximal microarchitecture configurations .

7.4.3.4 Reduction

The total cycles of the Reduction benchmark across maximal microarchictecture devices is shown in Figure 7.37 with the stalls in Figures 7.38 and 7.39. The baseline 3.4 compiler performs better than the baseline 3.2 for every microarchitecture, with an average speedup of 1.18x over it which can be attributed to better scheduling. No compiler or optimisations are capable of preventing the memory stalls when there is a mismatch between the LSUs and banks, but the optimisations are successful in improving performance overall. Loop unrolling with the 3.2 compiler leads to an average speedup of 1.38x which is the same average gain from using 3.4 and the alignment optimisation and close to the 1.44x speedup with the 3.4 compiler and unrolling. This result means that the unrolling is relatively not as effective in the updated compiler, again this can be attributed to the higher quality of scheduling in LLVM 3.4 as it does not need to enlarge the area to improve ILP as much. The fully optimised development compiler achieves an average speedup of 1.64x over the scalar device on 3.2.

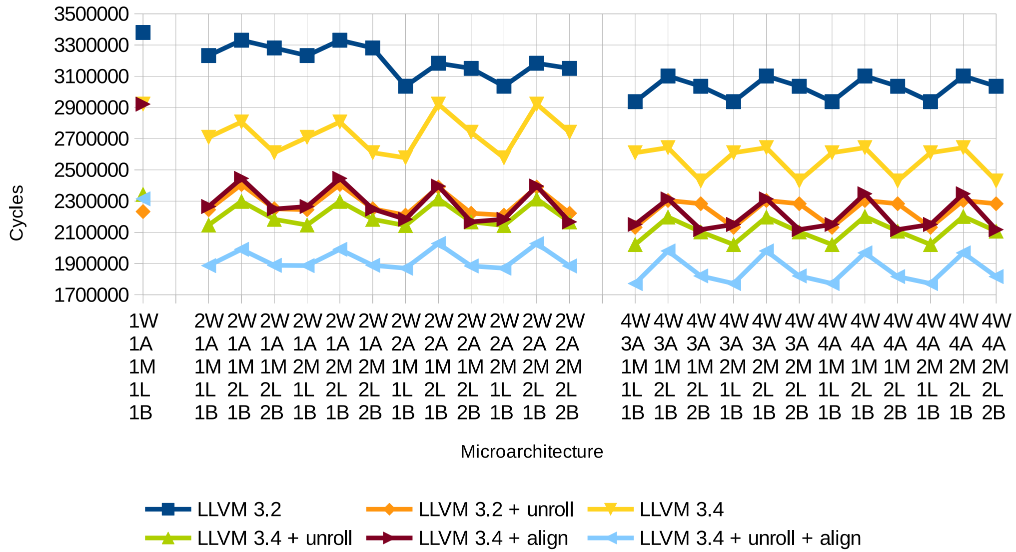

Figure 7.37: Total average cycles for Reduction using LLVM 3.2 and loop unrolling.

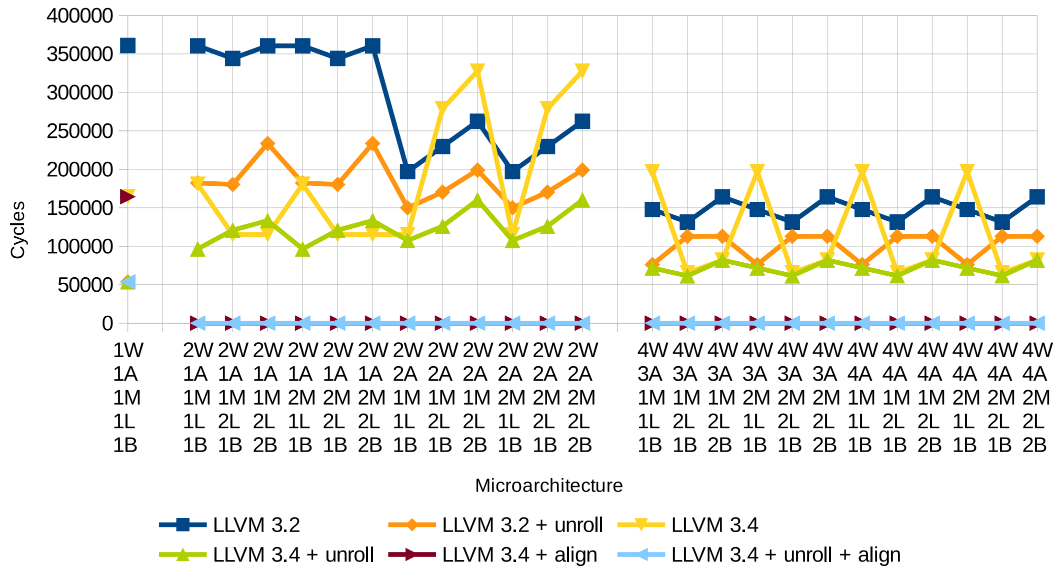

Figure 7.38: Total average IF stall cycles for Reduction on the single CU devices using both compilers and optimisations.

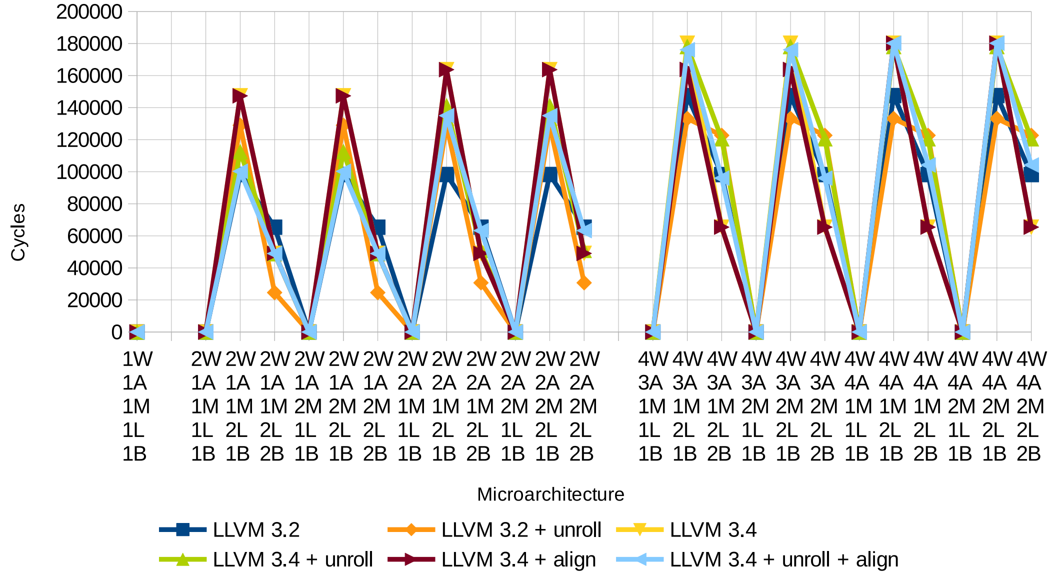

Figure 7.39: Total average memory stall cycles for Reduction on the single CU devices using both compilers and optimisations.

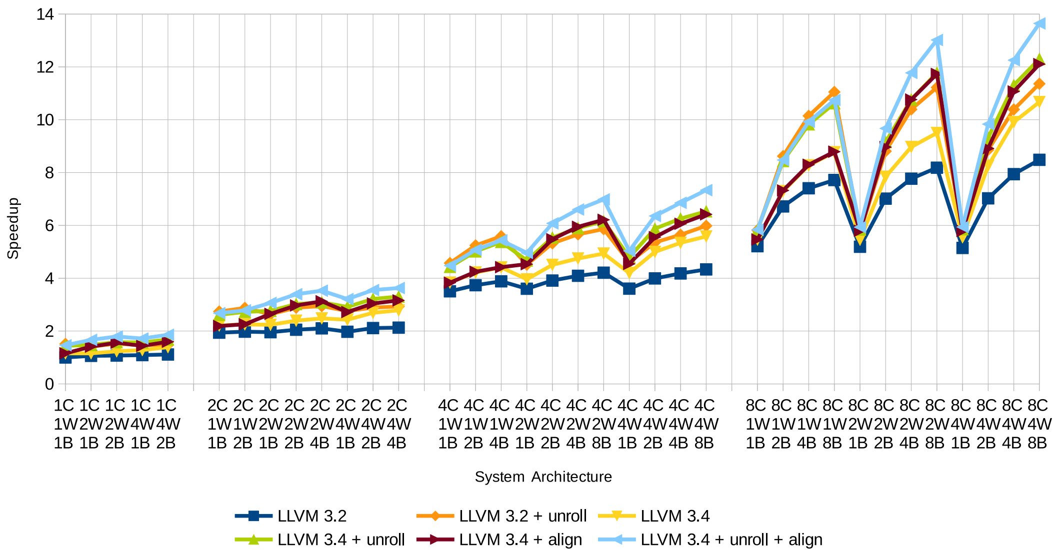

Figure 7.40: Speedup, relative to a scalar device with LLVM 3.2, of Reduction on the single CU devices with maximal microarchitectures using both compilers and optimisations.

The multi-context results for Reduction are depicted in Figure 7.40 and shows that the improvements in the single CU device are carried through, and multiplied, through the multi-context systems. Though for multi-CU, scalar, machines, the alignment optimisation has no benefit and so all the compilers that employ unrolling perform very similarily with the 3.2 compiler being the marginally faster. The alignment optimisation becomes very useful in the 2-wide configurations, enabling the compiler to generally outperfom both 3.2 and 3.4 with unrolling. The graph also shows the baseline 3.4 compiler is much more effective at utilising the wider devices than the original 3.2 compiler; performing 25% faster in the largest system. The difference between the optimisations becomes smaller in the largest configuration due to little more ILP being discovered past 2-wide and the memory system bounding performance; code generated using fully optimised compiler spends over 10% of it's total cycles stalling for memory, compared to under 5% for just the aligned program.

7.5 Summary

This chapter has presented the results from 12 benchmarks, including a total of 19 kernels, over 240 system configurations using the stable toolchain. The results from 4 benchmarks, over 193 system configurations, have also been reported using 5 other compiler and optimisation combinations; including the development branch of the compiler. The selected kernels have real world applications and some require both intra- and inter-kernel communication, and many run over multiple iterations. Some also operate in two dimensions and require synchronisation techniques to be in place.

The results from the stable compiler show that generally it is not capable of exploiting the VLIW capabilites of the LE1. No region enlargement techniques have been used and so scheduling is performed upon basic blocks which do not present many chances for significant ILP. In many of these benchmarks some ILP was discovered in the 2-wide machines, but not enough to overcome the two cycle pipeline latencies which resulted in an increase in NOPs; which negated the ILP gains. The 2-wide devices also suffered from significant increases in IF stalls which hindered their performance further, often resulting in them performing slower than the scalar device. The increased performance of the 4-wide machines over the 2-wide were because of the reduction in the IF stalls. The general lack of ILP resulted in a general performance invariance to the microarchitecture, with only the memory configuration being a real differentiator. For some of the benchmarks, the memory stalls incurred from using two LSUs resulted in slower performance than using a single LSU, though an increase in banks did help alleviate the problem. This suggests that the memory should be further split into more banks for the kernels which require more bandwidth and have more than one LSU.

However, the system is better at exploiting TLP by successfully scaling the problems over

the available contexts to improve performance. Again, the differences in microarchitecture have little affect on the overall performance, except for the memory configuration. For all of the kernels, apart from the floating-point based nearest neighbour and n-body simulation, the number of banks had a direct and significant influence on the achievable speedup once four or more contexts were instantiated. This problem was exacerbated for kernels that had already suffered from memory stalls in the single context tests. The results show a linear scaling across contexts, where bandwidth and the number of workgroups permit, which indicates that the static scheduling of the workgroups is sufficient in most cases. Additional performance gains are found in the kernels which exposed some useful ILP, which enabled speedups of 9-10x compared to a single, scalar, context. By using the silicon data, the extrapolated simulator results suggest that these gains would be similar on a real device too as long as it was symmetrical; as achievable clock frequencies on the 6- and 10-CU devices were too low.

The results from the stable compiler highlighted two key areas which could help improve the performance: (1) enlarge the scheduling regions to improve ILP and reduce branch mispredict penalties and (2) reduce IF stalls to increase the performance of the 2-wide machines. Loop unrolling was enabled to enlarge the scheduling regions and a custom optimisation was introduced to remove the IF stalls. Forcing loop unrolling had a very positive effect on all the tested kernels, with improvements of ∼15-35% which derived from increases in ILP and reduced number of branches executed. The increase in ILP did generally increase the dependence upon the memory configuration though.

The IF stalls are induced when the instruction words are split across memory locations that are accessible by the instruction fetch engine, when this occurs an additional cycle is required to fetch the whole word. The IF stalls were particularly prevalent in the 2-wide devices with two ALUs, and results suggested that these were the devices that were capable of the greatest ILP increases relative to the increase in hardware complexity. By padding the instructions to the issue width, the IF stalls have been removed; which has made the 2-wide much more competitive and leads to performance increases similar to the gains of unrolling of the original compiler. The newer compiler was generally better at scheduling, and this combined with efficient instruction selection (use of `slct` instructions to remove control-flow), loop unrolling and alignment allowed the system to perform twice as well as the original for bitonic sort.

Performance is not always better in the development compiler though, the results of binary search were particularly disappointing. In the two kernels that contained considerable control-flow (binary search and Floyd Warshall) the original compiler with unrolling outperformed the development compiler even with all of the optimisations enabled. This can be attributed to the different handling of compare instructions in the newer backend,

as they were lowered early on which prevents LLVM from performing some control-flow optimisations. Another version of the compiler would have to re-address this issue.

Chapter 8

Conclusion

8.1 Chapter Objectives

This chapter concludes the research and development that has been conducted during the course of this thesis. The contributions to knowledge are discussed along with possible further research which would complement that already undertaken. The work presented in this thesis includes an investigation into the suitability of the Single Program Multiple Data (SPMD) programming model for a unique VLIW CMP, the LE1. OpenCL was selected as the language, and platform, to base this research upon which required that a compiler, driver and runtime library was implemented to support the execution on the simulated LE1. Another aspect to this research was to investigate the configurable capabilities of the LE1 and the effect that differing micro- and system architectures would have on system performance.

8.2 Summary of Thesis Objectives

The aims as defined in Chapter 1 were:

- Enable OpenCL compilation for a custom VLIW CMP.
- Enable execution of OpenCL kernels on an FPGA platform.
- Extensively benchmark the CPU as this has not been previously performed.

After extensive research into parallel architectures and languages, LLVM was chosen to provide the base of a compiler that could be encorporated into an OpenCL driver. This system could then be used to investigate the suitability of the configurable VLIW CMP as an OpenCL accelerator device. The objectives defined in Chapter 4 were:

- Develop an LLVM compiler backend for the LE1, including compiler intrinsic functions to support OpenCL kernels.
- Develop a userland driver to encompass the compiler to enable automatic compilation of kernels, as well as controlling data transfers.
- Transform OpenCL kernels into another SPMD form, more suitable for the LE1 VLIW CMP.
- Implement a runtime library to support execution of the kernels on the LE1.
- Develop a method to schedule OpenCL workgroups across the multiple cores of the CMP.
- Investigate how the choices in microarchitecture and system architecture effect the performance of the system.

8.3 Contributions

After research into parallel architecture and languages, it was decided to primarily explore thread-level parallelism on the LE1 using OpenCL. Chapter 4 identified that to achieve this a toolchain would need to be implemented which could automatically compile OpenCL kernels and execute them upon the LE1. In the design and implementation of that toolchain, the following contributions to knowledge presented are:

- LLVM compiler backend for the unqiue VLIW CMP.
- Enabling OpenCL kernel execution on an FPGA via a fully programmable VLIW architecture.
- Detailed investigation into system and microarchitecture configurations of the VLIW CMP and their effect on performance.

The need of a compiler for the LE1 required identifying a pre-existing open source project to use so that development time would be greatly reduced. The two predominate open source compilers are GCC and LLVM. LLVM was chosen for its modular architecture and popularity within the research community, though it did limit the options of which parallel languages could be used later. GCC also produced faster executing binaries than that of Clang/LLVM as it is much more mature. At the beginning of the research, the only VLIW architecture supported in mainline LLVM was the Qualcomm Hexagon and they had introduced a framework to enable statically scheduled, multiple issue, targets. This framework was utilised by the LE1 backend to support its VLIW architecture, the only

issue being that the target generator script needed to create unique names for each of the functional units within all of the created targets. The use of that framework for the LE1 was replaced by a simpler resource table approach as the LE1's permissive architecture design did not require the complicated DFA approach that had been implemented for the Hexagon. The creation of the DFA for multiple targets could also make compiler compile-time significantly long. The resource table also enables the instruction packer to pack instructions together with their long immediates so to stop front-end stalls.

Research into parallel computing identified a major shift away from ILP exploitation and automatic parallelisation techniques to explicit use of DLP and TLP, the most recent advancement being heterogeneous computing with utilising the SPMD execution model. The primary choice of language and accelerator has been CUDA on NVIDIA GPUs. However, OpenCL was discovered to be an open specification to support heterogeneous computing and, as well as being designed to be target agnostic, it was also supported by Clang. Clang/LLVM's modular design enable the libraries to be easily incorporated into other tools, as was the case with the Clover project which used the compiler to execute OpenCL kernels on a x86 host. Clover was heavily modified so that it could operate as a heterogeneous compute driver, specifically for the LE1. For this to be possible the driver needed to control the transformation of the kernels, create a workgroup control function, transfer both instruction and data to the simulator and read the results back.

The OpenCL kernels first needed to be transformed to run on a non-multithreaded core, which was achieved using Clang libraries in two passes. This transformation is very similar to the ones reported in literature except that the output is valid OpenCL code and is not bounded by the number of nested loops within the kernel. Kernel instances are then launched by a target-specific control function that iterates through the number of workgroups. The code that is generated by the driver is compiled once for the target and executed by each LE1 context in the system, each one identifying themselves in the execution space by using their `cpuid` instruction. A runtime library was created, and the compiler modified, so that the LE1 could iterate through multiple parallel workgroups until completion without the intervention of the host. Though this research only obtained results from a simulated target, the API used to control the simulator is extendible to a FPGA platform and the target description file generated can be used to generate VHDL code for an LE1 system.

Finally, the extensive number of tests performed has provided a wealth of data about both the LE1 and the researched software system, highlighting areas of improvement for both. Initial results prompted the development of the updated compiler along with the optimisation to work around the front-end stalls of the hardware. It is hoped that the results can be later used as part of a training set to enable the driver to become a hardware/software codesign system. The repository of the full source code for the project is listed in Appendix

A.

8.4 Findings

This section introduces the findings generated from the research performed in this thesis. Firstly, it is explained how the SPMD programming model was found to be suitable for the LE1. This is followed by an overview of the features of the LE1 that made this research possible. Finally, heterogeneous computing, using OpenCL as the SPMD language and the LE1 as the target, is discussed.

8.4.1 Single Program, Multiple Data

Current high-performance programming languages and frameworks, such as OpenCL, have moved towards SPMD programming model to exploit both DLP and TLP. Though the OpenCL programming model has been designed around massively multi-threaded GPU architectures, inital tests show the developed system is capable of using the LE1 as a target device (as shown in Section 7.4). The execution on the LE1 contrasts to GPUs in two ways: (1) workitems are serialised on the LE1, and (2) the workgroups are statically scheduled in software, compared to the dedicated scheduling hardware of GPUs. The serialisation of the workitems into loops enables these 'thread' instances to be combined, via loop unrolling, which enlarges the scheduling region and so becomes more suitable for the VLIW architecture. This advantage is most notable in the bitonic sort benchmark, in which the development compiler is 100% faster than the original.

The threads have been coarsened into larger sets of parallel work so that the explicitly parallel nature of the SPMD model can still be exploited by the LE1 system. The coarsened kernels have also been modified in a generic way that could be used by other multi-core devices as well. The multi-core results show that the problems mostly scale linearly as more cores are instantiated in the LE1 system, even though the workgroup scheduling is performed in software. The results in Section 7.4.2 also show that the memory subsystem quickly becomes a bottleneck, for most of the benchmarks, in the multi-core systems leading to performance being up to halved, due to the limited number of DRAM banks.

8.4.2 Very Long Instruction Word Processor

The LE1 is designed as a shared memory multi-core accelerator, with each core executing from the same IRAM. Its architecture lends itself well to the SPMD execution model and is feasible due to its `cpuid` instruction. This single instruction enabled the core to identify the area within the execution space in which it should operate. The completely configurable

architecture of the LE1 was essential in the exploration of system and microarchitecture configurations and their effect on system performance. It was shown though that the compiler is unable to utilise the LE1 to its full potential, in wider configurations, as it generally uses too small scheduling regions. The results also highlighted the amount of time wasted due to stalls and mispredict penalties. These were due to the instruction fetch engine, the memory configuration and the disabled branch predictor. The IF stalls drove the inclusion of padding instructions to the machines issue width which improved performance, especially for 2-wide devices, at the expensive of code size increase.

In terms of compiler development, the decode hardware of the LE1 allowed the compiler to be more simple as the instruction packer does not have to worry about specific execution ports. Also, the relatively simple instruction set mapped well to the LLVM IR, which enabled simple, but effective, instruction selection; again shown through the compilers ability to remove all the control-flow within the bitonic sort benchmark. However the lack of floating-point hardware complicated the execution of OpenCL kernels as the applications are prodominately floating-point based and so required significant runtime support.

8.4.3 Towards Heterogeneous Computing

The results prove that hetergeneous computing could be supported by the developed software as it runs on an x86 host while the OpenCL kernels are compiled for, and run, on a simulated LE1. This means that data is successfully converted and transferred between the two different platforms. The execution of the kernel within the simulator required two binaries to be produced by the driver, including data and functions that were statically compiled into the binaries from a runtime library. This shows that the runtime is at least capable of supporting the execution of the presented kernels, including supporting the execution of emulated floating-point operations. The LE1 system is then capable of executing to completion without the host having to coordinate the execution of separate workgroups, so the static scheduling method works (for the two dimensions tested). To fully support heterogeneous computing, the system needs be able to seperate the work across the host and the LE1 device.

8.5 Limitations of Research

The primary limitation of the conducted research and presented results is that the driver overhead and time taken for data transfers has not been accounted for, this is mainly because the results were gained from a simulator. Although the simulator is cycle accurate, the method of data transfer between it and the LE1 driver is cumbersome and not representative

of a real system. Both the data and instruction binaries are read from the hard disk by the simulator and not transferred along a system bus. The data is also written in readable text format at the end of the assembly code, which is then assembled by the pre-existing assembler, requiring the data to be formatted and written twice. This also requires that the instruction binary is assembled each time that the kernel is run, and that the simulator always needs to be fed the complete program and data each time a kernel is executed. The proper method would have implemented JIT capabilities into the compiler which would negate the need of the assembler as well as the requirement to write the program to disk several times before the kernel begins executing. Data could also be fed straight to the simulator in an asynchronous manner.

The OpenCL specification defines an extensive platform, and though the use of the Clover project helped, the development time has limited the research to a subset of the specification. This resulted in no DSP kernels being used though the instruction set of the LE1 was designed for general purpose and integer DSP computing. The primary reason for this was that there was not enough time to implement support for the OpenCL `Image` types, and no time was used to enable the execution of kernels within a 3D space. The dependence on the runtime library is also a limitation, as very little development time was given to it and so it is not fully tested and will not be conformant with the OpenCL specification. Integer kernels were preferred to reduce the dependency on the runtime library, and as the other types of compute intensive kernels were floating-point based, the choice of kernels have not reflected the VLIW capabilites of the LE1.

As well as issues with the driver and runtime, the compiler has also not been fully tested as no testbench or coverage suite has been developed as this would have been a project in itself. The fundamental issue is that a developer toolchain is not available which makes debugging incredibly time consuming and difficult; even more so when trying to find errors produced from the compiler. The stable branch of the compiler was good enough to get a large set of valid results, but there are most likely issues still remaining within the compiler. This is particularly true for the development branch in which it only passed four of the tested benchmarks. Thus, research was greatly hindered as no part of the system has been fully tested and verified which made locating errors and bugs a task that often required too much time that would outweight the short-term benefits.

8.6 Further Research

As an extension to the previous section outlining the limitations of this research, a larger set of data from more suitable benchmarks could be acquired by improving the driver and the runtime library. Support could be added that would enable the execution of DSP kernels

which would suit the LE1, as too would encryption / decryption algorithms. These types of algorithms would better highlight the value of combining lightweight threads in multi-core DSPs. The compiler could also be modified to take advantage of the JIT framework within LLVM to be able to produce programs at runtime in the proper manner. This would then enable realistic data transfer to the target, and ideally the system could be run upon a small ARM + FPGA board, such as Zynq, to obtain real-world performance results.

The configurable and extensible nature of the LE1, combined with the runtime compilation model of OpenCL, opens many possibilites to further research. FPGA designs lend themselves perfectly to hardware / software codesign systems and the results presented in this thesis provide half of the data set required for the foundation of such a system. The presented results show how micro- and system architecture effects the performance reported by the simulator. This data could be combined with silicon data to produce results that would reflect real-world performance. This data could be then used as a training set by the driver to automatically select a system architecture for the kernel as well as other kernels that contain similar properties; this would require some analysis algorithms to be developed. The data collected from the silicon would not just to be limited to obtainable clock frequencies and die usage, but also the power requirements of individual functional units as well as the whole system. One aspect of the architecture that could also be explored is the size of the register file as it would contribute to a significant proportion of the area and power requirements of the core. Finer levels of customisation would also be possible by introducing custom instruction set extensions based on the requirements of the kernel. Again, analysis passes would need to be created that utilised profiling information as well as the data collected from previous experiments. The most obvious extensions, from an OpenCL perspective, would be the inclusion of floating-point hardware for both scalar and vector calculations.

References

[1] Robert R. Schaller. Moore's law: Past, present, and future. *IEEE Spectr.*, 34(6):52–59, June 1997.

[2] R.H. Dennard, F.H. Gaensslen, Hwa-Nien Yu, V.L. Rideout, Ernest Bassous, and Andre R. Leblanc. Design of ion-implanted mosfet's with very small physical dimensions. *Proceedings of the IEEE*, 87(4):668–678, April 1999.

[3] D.J. Frank. Power-constrained CMOS scaling limits. *IBM Journal of Research and Development*, 46(2.3):235–244, March 2002.

[4] Christian Martin. Post-Dennard Scaling and the final Years of Moores Law. Technical report, Hochschule Augsburg University of Applied Sciences, September 2014.

[5] R. Colin Johnson. Intel, IBM Dueling 14nm FinFETs. `http://www.eetimes.com/document.asp?doc_id=1324343`, Oct 2014.

[6] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. *Micro, IEEE*, 32(3):122–134, May 2012.

[7] HSA Foundation. *HSA Platform System Architecture Specification*, Jan 2015.

[8] M. Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, C-21(9):948–960, Sept 1972.

[9] Jun Liu, Yuanrui Zhang, Ohyoung Jang, Wei Ding, and Mahmut Kandemir. A Compiler Framework for Extracting Superword Level Parallelism. *SIGPLAN Not.*, 47(6):347–358, June 2012.

[10] Texas Instruments. *TMS320C55x DSP Library Programmer's Reference*, May 2013.

[11] ST. *UM0585 User Manual, STM32F10x DSP Library*, June 2010.

[12] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[13] C.L. Philip Chen and Chun-Yang Zhang. Data-intensive applications, challenges, techniques and technologies: A survey on big data. *Information Sciences*, 275(0):314 – 347, 2014.

[14] H. Nakano. Technology and applications of bluegene supercomputer and cell broadband engine. In *VLSI Technology, Systems and Applications, 2007. VLSI-TSA 2007. International Symposium on*, pages 1–4, April 2007.

[15] Henry Markram, Karlheinz Meier, Thomas Lippert, Sten Grillner, Richard Frackowiak, Stanislas Dehaene, Alois Knoll, Haim Sompolinsky, Kris Verstreken, Javier DeFelipe, Seth Grant, Jean-Pierre Changeux, and Alois Saria. Introducing the human brain project. *Procedia Computer Science*, 7(0):39 – 42, 2011. Proceedings of the 2nd European Future Technologies Conference and Exhibition 2011 (FET 11).

[16] Leonid Oliker, Jonathan Carter, Michael Wehner, Andrew Canning, Stephane Ethier, Art Mirin, David Parks, Patrick Worley, Shigemune Kitawaki, and Yoshinori Tsuda. Leading computational methods on scalar and vector hec platforms. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, SC '05, pages 62–, Washington, DC, USA, 2005. IEEE Computer Society.

[17] John E. Stone, David Gohara, and Guochun Shi. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *IEEE Des. Test*, 12(3):66–73, May 2010.

[18] Officially conformant OpenCL devices. `http://www.khronos.org/conformance/adopters/conformant-products#opencl`, Accessed May 2014.

[19] Top500 supercomputer site. `http://www.top500.org`, Nov 2014.

[20] Green500 supercomputer site. `http://www.green500.org`, Nov 2014.

[21] RenderScript API Guide. `"https://developer.android.com/guide/topics/renderscript/compute.html"`, 2014.

[22] R. Hundal and V.G. Oklobdzija. Determination of optimal sizes for a first and second level sram-dram on-chip cache combination. In *Computer Design: VLSI in Computers and Processors, 1994. ICCD '94. Proceedings., IEEE International Conference on*, pages 60–64, Oct 1994.

[23] James E. Thornton. The CDC 6600 Project. *IEEE Ann. Hist. Comput.*, 2(4):338–348, October 1980.

[24] G.F. Grohoski. Machine Organization of the IBM RISC System/6000 Processor. *IBM J. Res. Dev.*, 34(1):37–58, January 1990.

[25] K.C. Yeager. The Mips R10000 superscalar microprocessor. *Micro, IEEE*, 16(2):28–41, Apr 1996.

[26] Intel 64 and IA-32 Architectures Optimization Reference Manual. Technical report, Intel Corporation, Mar 2014.

[27] Travis Lanier. Exploring the Design of the Cortex-A15 Processor. Technical report, ARM.

[28] ARM. *ARM Cortex-A15 MPCore Processor, Technical Reference Manual*, June 2013.

[29] Brian Jeff. Enabling Mobile Innovation with the Cortex-A7 Processor. Technical report, ARM, October 2011.

[30] Youngmin Shin, Ken Shin, P. Kenkare, R. Kashyap, Hoi-Jin Lee, Dongjoo Seo, B. Millar, Yohan Kwon, R. Iyengar, Min-Su Kim, A. Chowdhury, Sung-Il Bae, Inpyo Hong, Wookyeong Jeong, A. Lindner, Ukrae Cho, K. Hawkins, Jae Cheol Son, and Seung Ho Hwang. 28nm high- metal-gate heterogeneous quad-core cpus for high-performance and energy-efficient mobile application processor. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2013 IEEE International*, pages 154–155, Feb 2013.

[31] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[32] Richard M. Russell. The cray-1 computer system. *Commun. ACM*, 21(1):63–72, January 1978.

[33] T. Watanabe. The nec sx-3 supercomputer system. In *Compcon Spring '91. Digest of Papers*, pages 303–308, Feb 1991.

[34] A Peleg and U. Weiser. Mmx technology extension to the intel architecture. *Micro, IEEE*, 16(4):42–50, Aug 1996.

[35] Intel Architecture Instruction Set Extensions Programming Reference. Technical report, Intel Corporation, Mar 2014.

[36] V. Parthasarathy, AV. Bharathi, and V. Rhymend Uthariaraj. Performance analysis of embedded media applications in newer arm architectures. In *Parallel Processing,*

2005. ICPP 2005 Workshops. International Conference Workshops on, pages 210–214, June 2005.

[37] Alex Peleg, Sam Wilkie, and Uri Weiser. Intel mmx for multimedia pcs. *Commun. ACM*, 40(1):24–38, January 1997.

[38] Power ISA Version 2.07. Technical report, IBM Corporation, May 2013.

[39] Srinivas K. Raman, Vladimir Pentkovski, and Jagannath Keshava. Implementing streaming simd extensions on the pentium iii processor. *IEEE Micro*, 20(4):47–57, July 2000.

[40] Introducing NEON Development Article. Technical report, ARM Limited, 2009.

[41] Bernard I. Witt. M65mp: An experiment in os/360 multiprocessing. In *Proceedings of the 1968 23rd ACM National Conference*, ACM '68, pages 691–703, New York, NY, USA, 1968. ACM.

[42] Douglas Hamilton. Multi-Core Multiprocessors in Embedded Applications. Technical report, Freescale Seminconductor, Inc, Jan 2005.

[43] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The directory-based cache coherence protocol for the dash multiprocessor. *SIGARCH Comput. Archit. News*, 18(2SI):148–159, May 1990.

[44] Mark S. Papamarcos and Janak H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. *SIGARCH Comput. Archit. News*, 12(3):348–354, January 1984.

[45] James Archibald and Jean-Loup Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Trans. Comput. Syst.*, 4(4):273–298, September 1986.

[46] The ARM Cortex-A9 Processors, v2.0. Technical report, ARM, Sept 2009.

[47] big.LITTLE Technology: The Future of Mobile. Technical report, ARM, 2013.

[48] Poonacha Kongetira, K. Aingaran, and K. Olukotun. Niagara: a 32-way multithreaded sparc processor. *Micro, IEEE*, 25(2):21–29, March 2005.

[49] D.M. Tullsen, S.J. Eggers, and H.M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Computer Architecture, 1995. Proceedings., 22nd Annual International Symposium on*, pages 392–403, June 1995.

[50] B. Sinharoy, R. Kalla, W. J. Starke, H. Q. Le, R. Cargnoni, J. A Van Norstrand, B. J. Ronchetti, J. Stuecheli, J. Leenstra, G. L. Guthrie, D. Q. Nguyen, B. Blaner, C. F. Marino, E. Retter, and P. Williams. IBM POWER7 multicore server processor. *IBM Journal of Research and Development*, 55(3):1:1–1:29, May 2011.

[51] M. Butler, L. Barnes, D.D. Sarma, and B. Gelinas. Bulldozer: An Approach to Multithreaded Compute Performance. *Micro, IEEE*, 31(2):6–15, March 2011.

[52] K. Gillespie, H.R. Fair, C. Henrion, R. Jotwani, S. Kosonocky, R.S. Orefice, D.A Priore, J. White, and K. Wilcox. 5.5 Steamroller: An x86-64 core implemented in 28nm bulk CMOS. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2014 IEEE International*, pages 104–105, Feb 2014.

[53] Joseph A. Fisher, Paolo Faraboschi, and Cliff Young. *Embedded computing - a VLIW approach to architecture, compilers, and tools.* Morgan Kaufmann, 2005.

[54] J. Fisher. Very long instruction work architectures and the eli-512. *Solid-State Circuits Magazine, IEEE*, 1(2):23–33, Spring 2009.

[55] John R. Ellis. *Bulldog: A Compiler for VLSI Architectures.* MIT Press, Cambridge, MA, USA, 1986.

[56] Ken Kennedy. Use-definition chains with applications. *Computer Languages*, 3(3):163 – 179, 1978.

[57] R.P. Colwell, W.E. Hall, C.S. Joshi, D.B. Papworth, P.K. Rodman, and J.E. Tornes. Architecture and implementation of a vliw supercomputer. In *Supercomputing '90., Proceedings of*, pages 910–919, Nov 1990.

[58] M. Schlansker and M. McNamara. The cydra 5 computer system architecture. In *Computer Design: VLSI in Computers and Processors, 1988. ICCD '88., Proceedings of the 1988 IEEE International Conference on*, pages 302–306, Oct 1988.

[59] B.R. Rau. Cydra 5 directed dataflow architecture. In *Compcon Spring '88. Thirty-Third IEEE Computer Society International Conference, Digest of Papers*, pages 106–113, Feb 1988.

[60] J. R. Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. Conversion of control dependence to data dependence. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '83, pages 177–189, New York, NY, USA, 1983. ACM.

[61] Vinod Kathail, Mike Schlansker, and Bob Rau. HPL PlayDoh architecture specification: Version 1.0. Technical Report HPL-93-80, Hewlett-Packard Laboratories, February 1993.

[62] V. Kathail, M. Schlansker, and B. R. Rau. HPL-PD Architecture Specification: Version 1.1. Technical Report HPL-93-80(R.1), Hewlett-Packard Laboratories, February 2000.

[63] Michael S. Schlansker, B. Ramakrishna Rau, and Multitemplate. Epic: An architecture for instruction-level parallel processors. Technical report, 2000.

[64] H. Sharangpani and H. Arora. Itanium processor microarchitecture. *Micro, IEEE*, 20(5):24–43, Sep 2000.

[65] C. McNairy and D. Soltis. Itanium 2 processor microarchitecture. *Micro, IEEE*, 23(2):44–55, March 2003.

[66] L. Codrescu, W. Anderson, S. Venkumanhanti, Mao Zeng, E. Plondke, C. Koob, A. Ingle, C. Tabony, and R. Maule. Hexagon dsp: An architecture optimized for mobile multimedia and communications. *Micro, IEEE*, 34(2):34–43, Mar 2014.

[67] Qualcomm Technologies, Inc. *Hexagon V5/V55 Programmer's Reference Manual*, August 2013.

[68] Texas Instruments. *TMS320C64x/C64x+ DSP CPU and Instruction Set, Reference Guide*, July 2010.

[69] Texas Instruments. *TMS320C6424 Fixed-Point Digital Signal Processor*, Dec 2009.

[70] M. Mattina. Architecture and Performance of the Tile-GX Processor Family. Technical report, Tilera Corporation, 2014.

[71] B.D. de Dinechin, R. Ayrignac, P.-E. Beaucamps, P. Couvert, B. Ganne, P.G. de Massas, F. Jacquet, S. Jones, N.M. Chaisemartin, F. Riss, and T. Strudel. A clustered manycore processor architecture for embedded and accelerated applications. In *High Performance Extreme Computing Conference (HPEC), 2013 IEEE*, pages 1–6, Sept 2013.

[72] B.D. de Dinechin. Dataflow language compilation for a single chip massively parallel processor. In *Multi-/Many-core Computing Systems (MuCoCoS), 2013 IEEE 6th International Workshop on*, pages 1–1, Sept 2013.

[73] Paolo Faraboschi, Geoffrey Brown, Joseph A. Fisher, Giuseppe Desoli, and Fred Homewood. Lx: A technology platform for customizable vliw embedded processing. *SIGARCH Comput. Archit. News*, 28(2):203–213, May 2000.

[74] Vassilios Chouliaras. *VThreads Programmer's Reference Manual v1.3.7*, Jan 2011.

[75] Nicolas Brunie, Sylvain Collange, and Gregory Diamos. Simultaneous branch and warp interweaving for sustained gpu performance. *SIGARCH Comput. Archit. News*, 40(3):49–60, June 2012.

[76] W.W.L. Fung, I. Sham, G. Yuan, and T.M. Aamodt. Dynamic warp formation and scheduling for efficient gpu control flow. In *Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on*, pages 407–420, Dec 2007.

[77] Veynu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu, and Yale N. Patt. Improving gpu performance via large warps and two-level warp scheduling. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pages 308–317, New York, NY, USA, 2011. ACM.

[78] NVIDIA Corporation. NVIDIA Tegra K1 A New Era in Mobile Computing. Technical report, Jan 2014.

[79] NVIDIA Corporation. NVIDIA GeForce GTX 750 Ti. Technical report, 2014.

[80] AMD Radeon Graphics Technology. AMD Graphics Cores Next (GCN) Architecture White Paper, June 2012.

[81] Travis Williams. AMD Embeds Intelligent, Interactive and Immersive Experiences with 2nd Gen AMD Embedded R-Series APUs and CPUs. "http://www.amd.com/en-us/press-releases/Pages/amd-embeds-intelligent-2014may19.aspx", May 2014.

[82] C. Severance. Eben upton: Raspberry pi. *Computer*, 46(10):14–16, October 2013.

[83] VideoCoreIV 3D Architecture Reference Guide. Technical report, Broadcom Corporation, Sept 2013.

[84] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. *IBM J. Res. Dev.*, 49(4/5):589–604, July 2005.

[85] B. Flachs, S. Asano, Sang H.Dhong, H.P. Hofstee, G. Gervais, Roy Kim, T. Le, Peichun Liu, J. Leenstra, J. Liberty, B. Michael, Hwa-Joon Oh, S.M. Mueller, O. Takahashi,

A. Hatakeyama, Y. Watanabe, N. Yano, D.A. Brokenshire, M. Peyravian, Vandung To, and E. Iwata. The microarchitecture of the synergistic processor for a cell processor. *Solid-State Circuits, IEEE Journal of*, 41(1):63–70, Jan 2006.

[86] E. Gardner. Is the Intel Xeon Phi coprocessor right for me? Technical report, Intel Corporation, Mar 2013.

[87] G. Chrysos. Intel Xeon Phi Coprocessor - the Architecture. Technical report, Intel Corporation, Nov 2013.

[88] Reza Rahman. Intel Xeon Phi Coprocessor Architecture for Software Developers. Technical report, Intel Corporation, May 2013.

[89] L. Gwennap. Adapteva: More FLOPs, Less Watts. Technical report, The Linley Group, Nov 2012.

[90] A. Olofsson. Parallella Reference Manual. Technical report, Adapteva.

[91] A. Olofsson. E64G401 Epiphany 64-core Microprocessor Datasheet. Technical report, Adapteva, Jun 2014.

[92] Stephen Trimberger. A reprogrammable gate array and applications. *Proceedings of the IEEE*, 81(7):1030–1041, Jul 1993.

[93] K. Kawana, H. Keida, M. Sakamoto, K. Shibata, and I Moriyama. An efficient logic block interconnect architecture for user-reprogrammable gate array. In *Custom Integrated Circuits Conference, 1990., Proceedings of the IEEE 1990*, pages 31.3/1–31.3/4, May 1990.

[94] J. Rose and S. Brown. Flexibility of interconnection structures for field-programmable gate arrays. *Solid-State Circuits, IEEE Journal of*, 26(3):277–282, Mar 1991.

[95] J. Rose, R.J. Francis, D. Lewis, and P. Chow. Architecture of field-programmable gate arrays: the effect of logic block functionality on area efficiency. *Solid-State Circuits, IEEE Journal of*, 25(5):1217–1225, Oct 1990.

[96] UltraScale Architecture and Product Overview. Technical report, Xilinx Inc, May 2014.

[97] S. Leibson, N. Mehta. Xilinx UltraScale: The Next-Generation Architecture for Your Next Generation Architecture. Technical report, Xilinx Inc, May 2014.

[98] Xilinx Artix-7 FPGAs: A New Performance Standard for Power-limited, Cost-sensistive Markets. Technical report, Xilinx Inc, 2013.

[99] E. Mohsen. Balancing Performance, Power, and Cost with Kintex-7 FPGAs. Technical report, Xilinx Inc, Sept 2013.

[100] Meeting the Performance and Power Imperative of the Zettabyte Era with Generation 10. Technical report, Altera Corporation, Jun 2013.

[101] Liang Wang and K. Skadron. Implications of the power wall: Dim cores and reconfigurable logic. *Micro, IEEE*, 33(5):40–48, Sept 2013.

[102] E.S. Chung, P.A Milder, J.C. Hoe, and Ken Mai. Single-chip heterogeneous computing: Does the future include custom logic, fpgas, and gpgpus? In *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on*, pages 225–236, Dec 2010.

[103] R.K. Gupta and G. De Micheli. Hardware-software cosynthesis for digital systems. *Design Test of Computers, IEEE*, 10(3):29–41, Sept 1993.

[104] Yuanrui Zhang and M. Kandemir. A hardware-software codesign strategy for loop intensive applications. In *Application Specific Processors, 2009. SASP '09. IEEE 7th Symposium on*, pages 107–113, July 2009.

[105] A Alomary, T. Nakata, Y. Honma, M. Imai, and N. Hikichi. An asip instruction set optimization algorithm with functional module sharing constraint. In *Computer-Aided Design, 1993. ICCAD-93. Digest of Technical Papers., 1993 IEEE/ACM International Conference on*, pages 526–532, Nov 1993.

[106] N. Clark, Hongtao Zhong, and S. Mahlke. Processor acceleration through automated instruction set customization. In *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, pages 129–140, Dec 2003.

[107] L. Jozwiak, M. Lindwer, R. Corvino, P. Meloni, L. Micconi, J. Madsen, E. Diken, D. Gangadharan, R. Jordans, S. Pomata, P. Pop, G. Tuveri, and L. Raffo. Asam: Automatic architecture synthesis and application mapping. In *Digital System Design (DSD), 2012 15th Euromicro Conference on*, pages 216–225, Sept 2012.

[108] H. Corporaal and H. Mulder. Move: a framework for high-performance processor design. In *Supercomputing, 1991. Supercomputing '91. Proceedings of the 1991 ACM/IEEE Conference on*, pages –, Nov 1991.

[109] H. Corporaal. Design of transport triggered architectures. In *VLSI, 1994. Design Automation of High Performance VLSI Systems. GLSV '94, Proceedings., Fourth Great Lakes Symposium on*, pages 130–135, Mar 1994.

[110] E. Aardoom and P. Stravers. An application specific processor for a multi-system navigation receiver. In *Computer Design: VLSI in Computers and Processors, 1992. ICCD '92. Proceedings, IEEE 1992 International Conference on*, pages 128–131, Oct 1992.

[111] Pekka Jääskeläinen, Vladimír Guzma, Andrea Cilio, Teemu Pitkänen, and Jarmo Takala. Codesign toolset for application-specific instruction-set processors. In *Electronic Imaging 2007*, volume 6507, pages 65070X–65070X–11. The International Society for Optical Engineering., February 2007.

[112] J. Shalf, D. Quinlan, and C. Janssen. Rethinking hardware-software codesign for exascale systems. *Computer*, 44(11):22–30, Nov 2011.

[113] N.P. Carter, A Agrawal, S. Borkar, R. Cledat, H. David, D. Dunning, J. Fryman, I Ganev, R.A Golliver, R. Knauerhase, R. Lethin, B. Meister, AK. Mishra, W.R. Pinfold, J. Teller, J. Torrellas, N. Vasilache, G. Venkatesh, and J. Xu. Runnemede: An architecture for ubiquitous high-performance computing. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pages 198–209, Feb 2013.

[114] J. Rattner. Extreme Scale Computing. ISCA Keynote, 2012.

[115] D. Donofrio, L. Oliker, J. Shalf, M.F. Wehner, C. Rowen, J. Krueger, S. Kamil, and M. Mohiyuddin. Energy-efficient computing for extreme-scale science. *Computer*, 42(11):62–71, Nov 2009.

[116] Cadence. Tensilica IP, XTensa. `"http://ip.cadence.com/ipportfolio/tensilica-ip/xtensa-customizable"`, 2014.

[117] AMD. High-Bandwidth Memory (HBM) Reinventing Memory Technology. `"https://www.amd.com/Documents/High-Bandwidth-Memory-HBM.pdf"`, 2015.

[118] Sumit Gupta. What is NVLink? And How Will It Make The World's Fastest Computer Possible? `"http://blogs.nvidia.com/blog/2014/11/14/what-is-nvlink/"`, Nov 2014.

[119] M. Milward, D. Stevens, and V. Chouliaras. Embedded uml design flow to the configurable le1 multicore vliw processor. In *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2012 7th International Workshop on*, pages 1–8, July 2012.

[120] Keith Cooper and Linda Torczon. *Engineering a Compiler, Second Edition.* Morgan Kaufmann, 2012.

[121] R. Steven Glanville and Susan L. Graham. A new method for compiler code generation. In *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '78, pages 231–254, New York, NY, USA, 1978. ACM.

[122] M. Anton Ertl. Optimal code selection in dags. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 242–249, New York, NY, USA, 1999. ACM.

[123] Philip B. Gibbons and Steven S. Muchnick. Efficient instruction scheduling for a pipelined architecture. *SIGPLAN Not.*, 21(7):11–16, July 1986.

[124] Steven J. Beaty, Scott Colcord, and Philip H. Sweany. Using genetic algorithms to fine-tune instruction-scheduling heuristics. In *In Proceedings of the Second International Conference on Massively Parallel Computing Systems*, pages 6–9. IEEE Computer Society, 1996.

[125] G. J. Chaitin. Register allocation & spilling via graph coloring. *SIGPLAN Not.*, 17(6):98–101, June 1982.

[126] J.K. Ousterhout. Scripting: higher level programming for the 21st century. *Computer*, 31(3):23–30, Mar 1998.

[127] Filip Pizio. Surfin' Safari. `"https://www.webkit.org/blog/3362/introducing-the-webkit-ftl-jit/"`, May 2014.

[128] Mozilla Developer Network. SpiderMonkey Internals. `"https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey/Internals"`, Aug 2014.

[129] V8 Javascript Engine Homepage. `"https://code.google.com/p/v8/"`, December 2014.

[130] Erik Meijer, Redmond Wa, and John Gough. Technical overview of the common language runtime, 2000.

[131] Mono Project. `"http://www.mono-project.com/docs/advanced/runtime/docs/"`, 2014.

[132] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. Design of the java hotspot™ client compiler for java 6. *ACM Trans. Archit. Code Optim.*, 5(1):7:1–7:32, May 2008.

[133] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–. [Online; accessed 2014-11-29].

[134] List of Python implementations. "https://wiki.python.org/moin/PythonImplementations", July 2014.

[135] Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and Alan Edelman. Julia: A fast dynamic language for technical computing. *CoRR*, abs/1209.5145, 2012.

[136] ART and Dalvik. "https://source.android.com/devices/tech/dalvik/index.html", 2014.

[137] John Wollenburg Sias. *A Systematic Approach to Delivering Instruction-level Parallelism in Epic Systems*. PhD thesis, Champaign, IL, USA, 2005. AAI3199143.

[138] John Whaley. Partial method compilation using dynamic profile information. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '01, pages 166–179, New York, NY, USA, 2001. ACM.

[139] GCC Documentation, Profile Information. "https://gcc.gnu.org/onlinedocs/gccint/Profile-information.html", 2014.

[140] R.D.-C. Ju, K. Nomura, U. Mahadevan, and Le-Chun Wu. A unified compiler framework for control and data speculation. In *Parallel Architectures and Compilation Techniques, 2000. Proceedings. International Conference on*, pages 157–168, 2000.

[141] S.A Mahlke, R.E. Hank, J.E. McCormick, D.I August, and W.-M.W. Hwu. A comparison of full and partial predicated execution support for ilp processors. In *Computer Architecture, 1995. Proceedings., 22nd Annual International Symposium on*, pages 138–149, June 1995.

[142] J. Fisher. Trace scheduling: A technique for global microcode compaction. *Computers, IEEE Transactions on*, C-30(7):478–490, July 1981.

[143] Wen mei W. Hwu, Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, John G. Holm, and Daniel M. Lavery. The superblock: An effective technique for vliw and superscalar compilation. *THE JOURNAL OF SUPERCOMPUTING*, 7:229–248, 1993.

[144] Pohua P. Chang, Scott A. Mahlke, and Wen-mei W. Hwu. Using profile information to assist classic code optimizations. *Softw. Pract. Exper.*, 21(12):1301–1321, December 1991.

[145] S.A Mahlke, D.C. Lin, W.Y. Chen, R.E. Hank, and R.A Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Microarchitecture, 1992. MICRO 25., Proceedings of the 25th Annual International Symposium on*, pages 45–54, Dec 1992.

[146] B.A Maher, A Smith, D. Burger, and K. McKinley. Merging head and tail duplication for convergent hyperblock formation. In *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*, pages 65–76, Dec 2006.

[147] W.A Havanki, S. Banerjia, and T.M. Conte. Treegion scheduling for wide issue processors. In *High-Performance Computer Architecture, 1998. Proceedings., 1998 Fourth International Symposium on*, pages 266–276, Feb 1998.

[148] *Unified Assign and Schedule: A New Approach to Scheduling for Clustered Register File Microarchitectures*, MICRO 31, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[149] M. Lam. Software pipelining: An effective scheduling technique for vliw machines. *SIGPLAN Not.*, 23(7):318–328, June 1988.

[150] B. R. Rau and C. D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. *SIGMICRO Newsl.*, 12(4):183–198, December 1981.

[151] B. Ramakrishna Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, MICRO 27, pages 63–74, New York, NY, USA, 1994. ACM.

[152] David R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[153] The Open Group. The Single UNIX Specification, Version 2. `http://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread.h.html`, 1997.

[154] Message P Forum. MPI: A Message-Passing Interface Standard. Technical report, Knoxville, TN, USA, 1994.

[155] OpenMP Architecture Review Board. OpenMP application program interface version 3.0, May 2008.

[156] OpenMP Architecture Review Board. Openmp application program interface, 2013.

[157] NVIDIA Corporation. *CUDA C Programming Guide, Design Guide, Version 6.5*, August 2014.

[158] Khronos OpenCL Working Group. *The OpenCL Specification, Version 2.0*, October 2014.

[159] PyOpenCL website. `http://mathema.tician.de/software/pyopencl/`, 2014.

[160] JOCL project website. `http://jogamp.org/jocl/www/`, 2014.

[161] The Khronos Group Inc. The spir specification. Specification, 2014.

[162] HSA Foundation. *HSA Programmer's Reference Manual: HSAIL Virtual ISA and Programming Model, Compiler Writer's Guide, and Object Format (BRIG)*, May 2013.

[163] RenderScript API Reference. `"http://developer.android.com/reference/android/renderscript/package-summary.html"`, 2014.

[164] JohnA. Stratton, SamS. Stone, and Wen-meiW. Hwu. Mcuda: An efficient implementation of cuda kernels for multi-core cpus. In JosNelson Amaral, editor, *Languages and Compilers for Parallel Computing*, volume 5335 of *Lecture Notes in Computer Science*, pages 16–30. Springer Berlin Heidelberg, 2008.

[165] Jayanth Gummaraju, Laurent Morichetti, Michael Houston, Ben Sander, Benedict R. Gaster, and Bixia Zheng. Twin peaks: A software platform for heterogeneous computing on general-purpose and graphics processors. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 205–216, New York, NY, USA, 2010. ACM.

[166] P. J. Plauger. *The Standard C Library.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 1991.

[167] Jaejin Lee, Jungwon Kim, Sangmin Seo, Seungkyun Kim, Jungho Park, Honggyu Kim, Thanh Tuan Dao, Yongjin Cho, Sung Jong Seo, Seung Hak Lee, Seung Mo Cho, Hyo Jung Song, Sang-Bum Suh, and Jong-Deok Choi. An opencl framework for heterogeneous multicores with local memory. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 193–204, New York, NY, USA, 2010. ACM.

[168] Jia-Jhe Li, Chi-Bang Kuan, Tung-Yu Wu, and Jenq Kuen Lee. Enabling an opencl compiler for embedded multicore dsp systems. In *Parallel Processing Workshops (ICPPW), 2012 41st International Conference on*, pages 545–552, Sept 2012.

[169] D. Chih-Wei Chang, I-Tao Liao, Jenq-Kuen Lee, Wen-Feng Chen, Shau-Yin Tseng, and Chein-Wei Jen. Pac dsp core and application processors. In *Multimedia and Expo, 2006 IEEE International Conference on*, pages 289–292, July 2006.

[170] A.G. Dean and J.P. Shen. Techniques for software thread integration in real-time embedded systems. In *Real-Time Systems Symposium, 1998. Proceedings., The 19th IEEE*, pages 322–333, Dec 1998.

[171] A. Papakonstantinou, K. Gururaj, J.A. Stratton, Deming Chen, J. Cong, and W.-M.W. Hwu. Fcuda: Enabling efficient compilation of cuda kernels onto fpgas. In *Application Specific Processors, 2009. SASP '09. IEEE 7th Symposium on*, pages 35–42, July 2009.

[172] Zhiru Zhang, Yiping Fan, Wei Jiang, Guoling Han, Changqi Yang, and Jason Cong. Autopilot: A platform-based esl synthesis system. In Philippe Coussy and Adam Morawiec, editors, *High-Level Synthesis*, pages 99–112. Springer Netherlands, 2008.

[173] I. Lebedev, Shaoyi Cheng, A. Doupnik, J. Martin, C. Fletcher, D. Burke, Mingjie Lin, and J. Wawrzynek. Marc: A many-core approach to reconfigurable computing. In *Reconfigurable Computing and FPGAs (ReConFig), 2010 International Conference on*, pages 7–12, Dec 2010.

[174] Mingjie Lin, I. Lebedev, and J. Wawrzynek. Openrcl: Low-power high-performance computing with reconfigurable devices. In *Field Programmable Logic and Applications (FPL), 2010 International Conference on*, pages 458–463, Aug 2010.

[175] Pekka Jskelinen, Carlos S. de La Lama, Pablo Huerta, and Jarmo Takala. Opencl-based design methodology for application-specific processors. In Fadi J. Kurdahi and Jarmo Takala, editors, *ICSAMOS*, pages 223–230. IEEE, 2010.

[176] M. Owaida, N. Bellas, K. Daloukas, and C.D. Antonopoulos. Synthesis of platform architectures from opencl programs. In *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*, pages 186–193, May 2011.

[177] Konstantis Daloukas, Christos D. Antonopoulos, and Nikolaos Bellas. Glopencl: Opencl support on hardware- and software-managed cache multicores. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, HiPEAC '11, pages 15–24, New York, NY, USA, 2011. ACM.

[178] Hee-Seok Kim, Minwook Ahn, John A. Stratton, and Wen mei W. Hwu. Design evaluation of opencl compiler framework for coarse-grained reconfigurable arrays. In *FPT*, pages 313–320. IEEE, 2012.

[179] Altera Corp. Implementing fpga design with the opencl standard. Technical report, Nov. 2012.

[180] D. Chen and D. Singh. Fractal video compression in opencl: An evaluation of cpus, gpus, and fpgas as acceleration platforms. In *Design Automation Conference (ASP-DAC), 2013 18th Asia and South Pacific*, pages 297–304, Jan 2013.

[181] Xilinx. Vivado design suite user guide. User guide, May 2014.

[182] K. Shagrithaya, K. Kepa, and P. Athanas. Enabling development of opencl applications on fpga platforms. In *Application-Specific Systems, Architectures and Processors (ASAP), 2013 IEEE 24th International Conference on*, pages 26–30, June 2013.

[183] J. Coole and G. Stitt. Fast, flexible high-level synthesis from opencl using reconfiguration contexts. *Micro, IEEE*, 34(1):42–53, Jan 2014.

[184] J. Coole and G. Stitt. Intermediate fabrics: Virtual architectures for circuit portability and fast placement and routing. In *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2010 IEEE/ACM/IFIP International Conference on*, pages 13–22, Oct 2010.

[185] David Stevens. *On the Automated compilation of UML notation to a VLIW Chip Multiprocessor*. PhD thesis, Loughborough University, Dec 2013.

[186] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *CGO*, pages 75–88, San Jose, CA, USA, Mar 2004.

[187] Chris Lattner. *The Archicture of Open Source Applications*, volume i, chapter 11. Self published, March 2012.

[188] Clang website. `http://clang.llvm.org`, 2014.

[189] Clover source homepage. `http://cgit.freedesktop.org/mesa/clover/`, 2014.

[190] The Gallium3D website. `http://www.freedesktop.org/wiki/Software/gallium/`, 2014.

[191] C-Reduce website. `"http://embed.cs.utah.edu/creduce`, 2014.

[192] Ken Kennedy and John R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[193] Libclc website. `http://libclc.llvm.org`, 2014.

[194] Compiler-rt website. `http://compiler-rt.llvm.org`, 2014.

[195] Softfloat homepage. `http://www.jhauser.us/arithmetic/SoftFloat.html`, 2014.

[196] AMD. AMD APP SDK v2.7, Getting Started. Technical report, 2012.

[197] Shuai Che, M. Boyer, Jiayuan Meng, D. Tarjan, J.W. Sheaffer, Sang-Ha Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54, Oct 2009.

Publications

The following are publications that have resulted from the work presented in this thesis. The paper submitted to the IEEE Transactions on Computers, in July 2014, is currently under peer review.

[1] Samuel J. Parker, Vassilios A. Chouliaras. Enabling OpenCL on a configurable, VLIW Chip-Multiprocessor. *Systems Division Mini-Conference*, October 2013.

[2] Samuel J. Parker, Vassilios A. Chouliaras. An Automated OpenCL FPGA Compilation Framework Targetting a Configurable, VLIW Chip Multiprocessor. Submitted to *Computers, IEEE Transactions on*, July 2014.

Appendix A

Source Code Repository

Full source code of the project can be found at `https://github.com/grubbymits/esdg-opencl`.

Appendix B

Compiler and Simulator Target Generator Script

```
 path_to_driver = "/home/sam/src/esdg-opencl/"
path_to_llvm_backend = path_to_driver + "llvm-3.2/lib/Target/LE1/"
path_to_simulator_models = path_to_driver + "install-dir/machines/"
final_device_array = ""
driver_devices = ""
total_devices = 0
sim_output = True;
compiler_output = True;
mod_driver = True;

for context in [1, 2, 4, 8] :
  for width in [1, 2, 4]:
    issue_width = str(width)
    for alus in range(1, width+1):
      if ((issue_width != 1) & (alus < (width / 2))):
        continue
      num_alus = str(alus)
      for muls in [1, 2] :
        if (muls > width) :
          continue
        num_muls = str(muls)
        for lsus in [1, 2] :
          if (lsus > width) :
            continue
          num_lsus = str(lsus)
          for banks in [1, 2, 4, 8]:
            if (banks > (lsus * context)):
              continue
            num_banks = str(banks)
             # Define string simulator model
            simulator_target = """
              <galaxy>
              <systems>1</systems >
              <type>homogeneous</type>
              <system>
                <contexts>""" + str(context) + """</contexts>
```

```
   <SCALARSYS_PRESENT>1</SCALARSYS_PRESENT>
  <PERIPH_PRESENT>0</PERIPH_PRESENT>
  <DARCH>DRAM_SHARED</DARCH>
  <DRAM_BLK_SIZE>16</DRAM_BLK_SIZE>
  <DRAM_SIZE>0x1000</DRAM_SIZE>
  <STACK_SIZE>0x100</STACK_SIZE>
  <DRAM_BANKS>""" + num_banks + """</DRAM_BANKS>
    <context>
      <ISSUE_WIDTH_MAX>""" + issue_width + """</ISSUE_WIDTH_MAX>
      <ISA_PRSPCTV>VT32PP</ISA_PRSPCTV>
      <IARCH>IFE_SIMPLE_IRAM_PRIV</IARCH>
      <CLUST_TEMPL>1</CLUST_TEMPL>
      <HYPERCONTEXTS>1</HYPERCONTEXTS>
      <IFETCH_WIDTH>""" + issue_width + """</IFETCH_WIDTH>
      <IRAM_SIZE>0x100</IRAM_SIZE>
        <clusterTemplate>
          <name>Cluster0</name>
          <SCORE_PRESENT>1</SCORE_PRESENT>
          <VCORE_PRESENT>0</VCORE_PRESENT>
          <FPCORE_PRESENT>0</FPCORE_PRESENT>
          <CCORE_PRESENT>0</CCORE_PRESENT>
            <INSTANTIATE>1</INSTANTIATE>
            <INSTANCES>1</INSTANCES>
            <ISSUE_WIDTH>""" + issue_width + """</ISSUE_WIDTH>
            <S_GPR_FILE_SIZE>64</S_GPR_FILE_SIZE>
            <S_FPR_FILE_SIZE<0</S_FPR_FILE_SIZE>
            <S_VR_FILE_SIZE>0</S_VR_FILE_SIZE>
            <S_PR_FILE_SIZE>8</S_PR_FILE_SIZE>
            <IALUS>""" + num_alus + """</IALUS>
            <IMULTS>""" + num_muls + """</IMULTS>
            <LSU_CHANNELS>""" + num_lsus + """</LSU_CHANNELS>
            <BRUS>1</BRUS>
        </clusterTemplate>
        <hypercontext>
          <name>HyperContext0</name>
          <cluster>0_0</cluster>
        </hypercontext>
    </context>
  </system>
</galaxy>"""
```

```
 if (mod_driver) :
  driver_devices += " Coal::LE1Device( " + str(context) + ", " + issue_width + ", " + num_alus
    + ", " + num_muls + ", " + num_lsus + ", " + num_banks + ")"
  total_devices += 1
  if ((context == 8) & (width == 4) & (alus == width) &
      (muls == 2) & (lsus == 2) & (banks == 8)):
    driver_devices += " // " + str(total_devices) + "};"
    final_device_array = "static Coal::LE1Device LE1Devices[" + str(total_devices) + "] = {"
    final_device_array += "// cores, width, alus, muls, lsus, banks"
    final_device_array += driver_devices
    output_file = open("devices.h", 'w')
    output_file.write(str(final_device_array))
    output_file.close()
  else :
    driver_devices += ", // " + str(total_devices)
config_name = issue_width + "w_" + num_alus + "a_" + num_muls + "m_" + num_lsus + "ls"
if (sim_output) :
  simulator_target_filename = path_to_simulator_models + str(context) + "-core/"
  simulator_target_filename += config_name + "_" + num_banks + "b.xml"
  output_file = open(simulator_target_filename, 'w')
  output_file.write(str(simulator_target))
  output_file.close()
if not compiler_output :
  continue
if ((width == 1) | (banks != 1) | (context != 1)):
  continue
compiler_target = ""
for compiler_alu in range(alus) :
alu = str(compiler_alu)
compiler_target += "def ALU_" + alu + "_" + config_name + " :  FuncUnit;"
for compiler_mul in range(muls) :
  mul = str(compiler_mul)
  compiler_target += "def MUL_" + mul + "_" + config_name + " :  FuncUnit;"
for compiler_lsu in range(lsus) :
  lsu = str(compiler_lsu)
  compiler_target += "def LSU_" + lsu + "_" + config_name + " :  FuncUnit;"
compiler_target += "def BRU_0_" + config_name + " :  FuncUnit;"
```

```
 compiler_target += "def LE1" + config_name + "Itineraries :  ProcessorItineraries<[
for compiler_alu in range(alus) :
  alu = str(compiler_alu)
  compiler_target += " ALU_" + alu + "_" + config_name + ", "
for compiler_mul in range(muls) :
  mul = str(compiler_mul)
  compiler_target += " MUL_" + mul + "_" + config_name + ", "
for compiler_lsu in range(lsus) :
  lsu = str(compiler_lsu)
  compiler_target += " LSU_" + lsu + "_" + config_name + ", "
compiler_target += " BRU_0_" + config_name + " ], [ ], ["
compiler_target += " InstrItinData<IIAlu, [InstrStage<1, ["
for compiler_alu in range(alus) :
alu = str(compiler_alu)
compiler_target += "ALU_" + alu + "_" + config_name
if (compiler_alu != alus-1) :
  compiler_target += ", "
else :
  compiler_target += "]>], [3, 1]>, "
compiler_target += " InstrItinData<IIMul, [InstrStage<1, [ "
for compiler_mul in range(muls) :
  mul = str(compiler_mul)
  compiler_target += "MUL_" + mul + "_" + config_name
  if (compiler_mul != muls-1) :
    compiler_target += ", "
  else :
    compiler_target += "]>], [3, 1]>, "
compiler_target += " InstrItinData<IILoadStore, [InstrStage<1, [ "
for compiler_lsu in range(lsus) :
  lsu = str(compiler_lsu)
  compiler_target += "LSU_" + lsu + "_" + config_name
  if (compiler_lsu != lsus-1) :
    compiler_target += ", "
  else :
    compiler_target += "]>], [3, 1]>, "
compiler_target += " InstrItinData<IIBranch, [InstrStage<1, [BRU_0_" + config_name + "]>],
                                    [6, 1]> ]>; "
```

```
 compiler_target += "def LE1Model" + config_name + " : SchedMachineModel {"
compiler_target += " let IssueWidth = " + issue_width + ";"
compiler_target += " let Itineraries = LE1" + config_name + "Itineraries;}"
compiler_target_filename = path_to_llvm_backend + "MachineModels/LE1" + config_name + ".td"
output_file = open(compiler_target_filename, 'w')
output_file.write(str(compiler_target))
output_file.close()
output_file = open((path_to_llvm_backend + "LE1.td"), 'a')
output_file.write("def : Processor<" + config_name + ", LE1" + config_name + "Itineraries,
                               []>; ")
output_file.close()
output_file = open((path_to_llvm_backend + "LE1Schedule.td"), 'a')
output_file.write("include "MachineModels/LE1" + config_name + ".td")
output_file.close()
```

Appendix C

Results

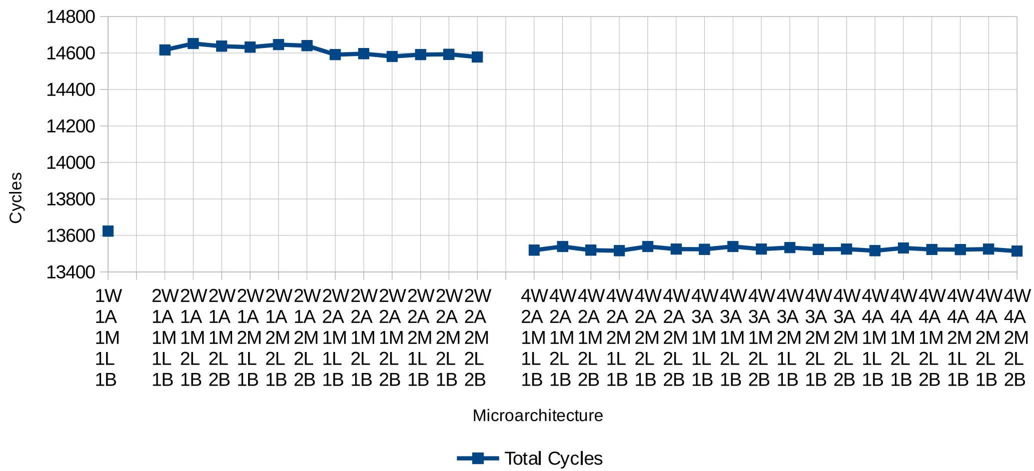

Figure C.1: Total cycles for BinarySearch using 1 context across varying microarchitecture configurations.

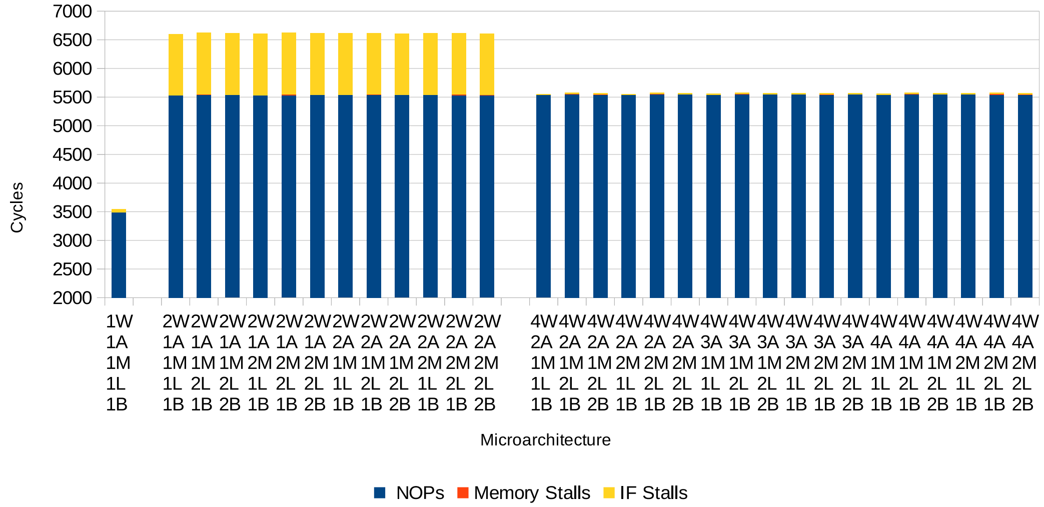

Figure C.2: Total IF stall cycles for BinarySearch using 1 context across varying microarchitecture configurations.

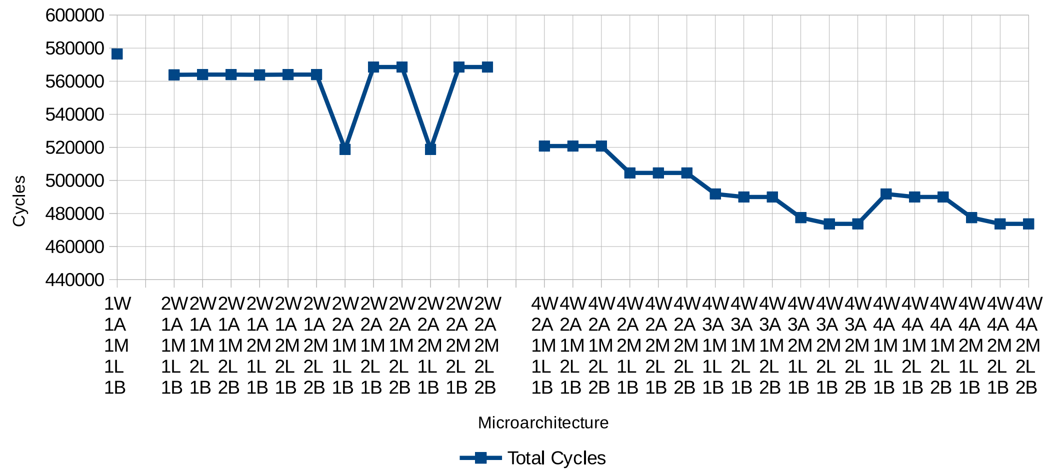

Figure C.3: Total average cycle count for BitonicSort using 1 context across varying microarchitecture configurations.

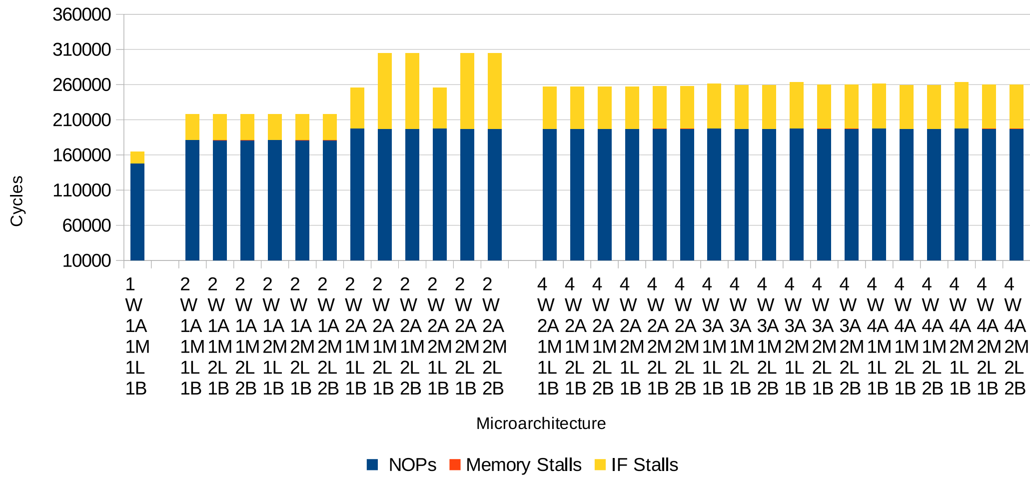

Figure C.4: Total average IF stall cycle count for BitonicSort using 1 context across varying microarchitecture configurations.

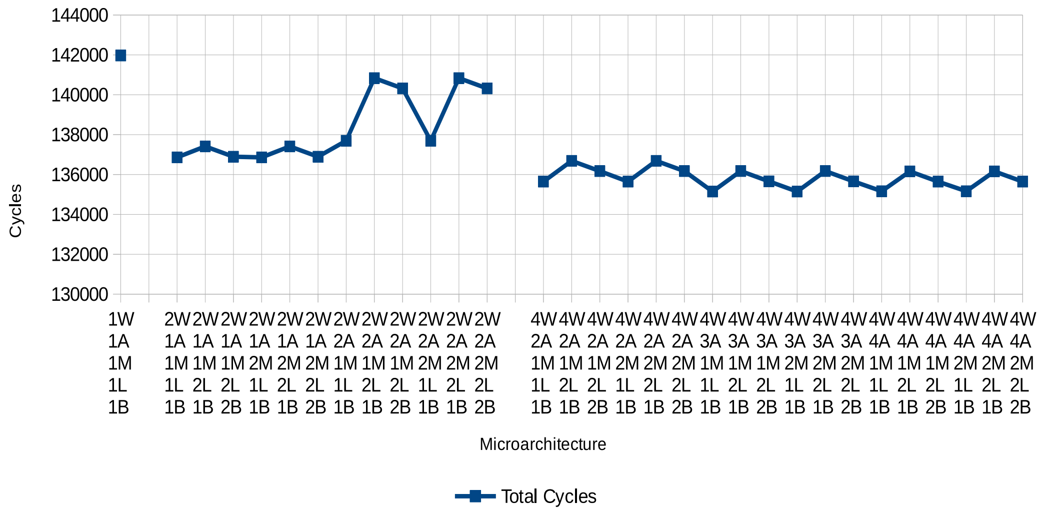

Figure C.5: Total average cycle count for BFS_1 using 1 context across varying microarchitecture configurations.

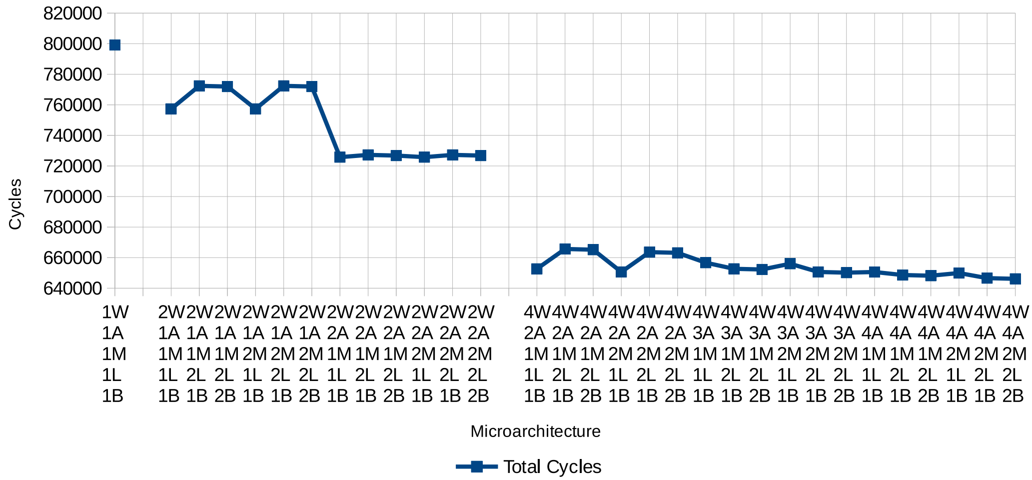

Figure C.6: Total cycles for FastWalshTransform using 1 context across varying microarchitecture configurations.

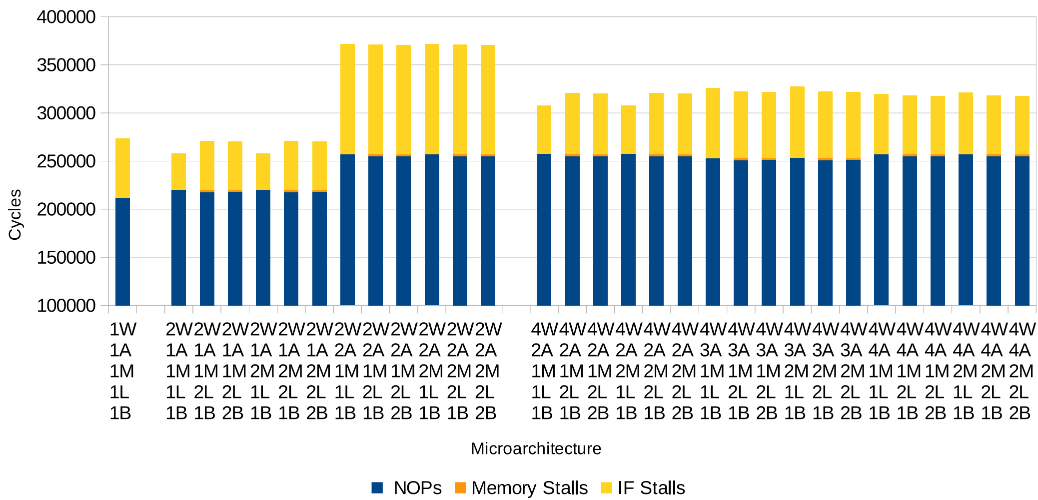

Figure C.7: Total stalls and NOP cycles for FastWalshTransform using 1 context across varying microarchitecture configurations.

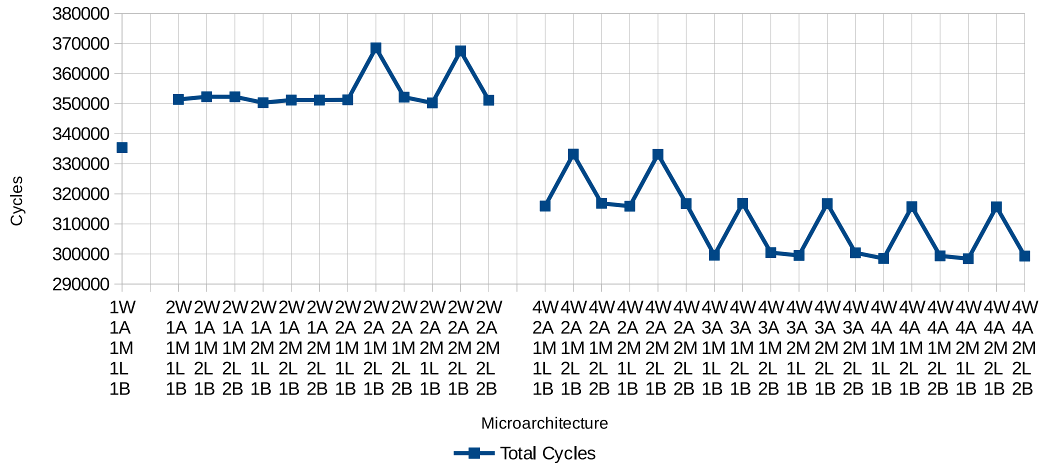

Figure C.8: Total cycles for FloydWarshall using 1 context across varying microarchitecture configurations.

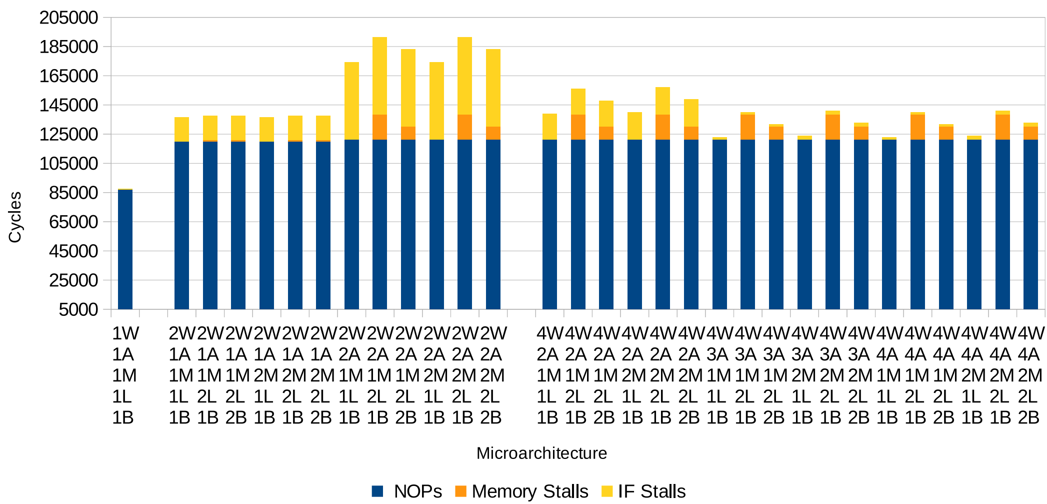

Figure C.9: Total stall and NOP cycles for FloydWarshall using 1 context across varying microarchitecture configurations.

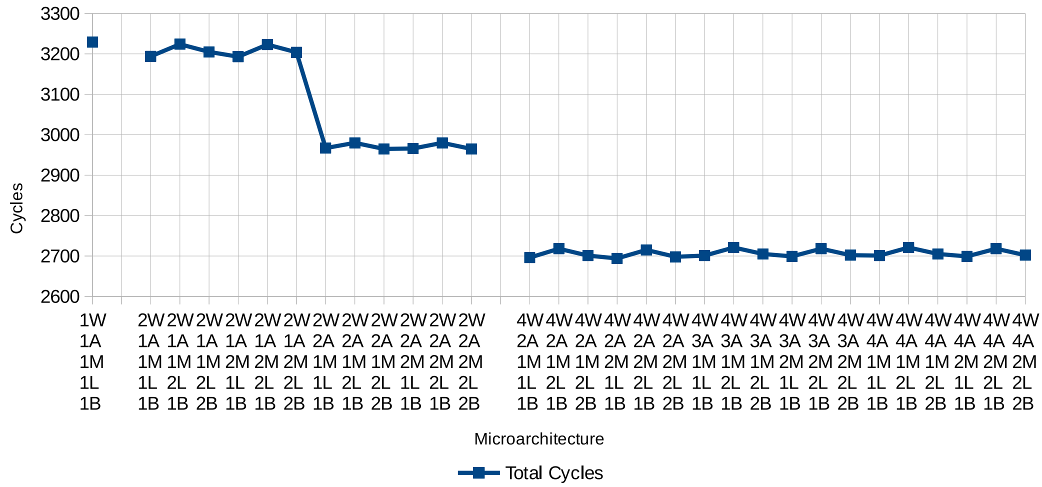

Figure C.10: Total average cycles for Fan1 of Gaussian Elimination using 1 context across varying microarchitecture configurations.

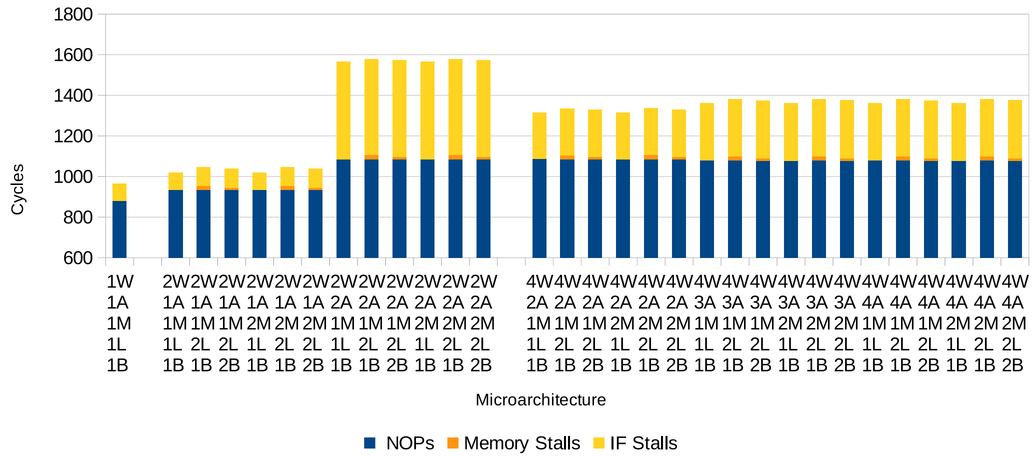

Figure C.11: Total average stalls and NOP cycles for Fan1 of Gaussian Elimination using 1 context across varying microarchitecture configurations.

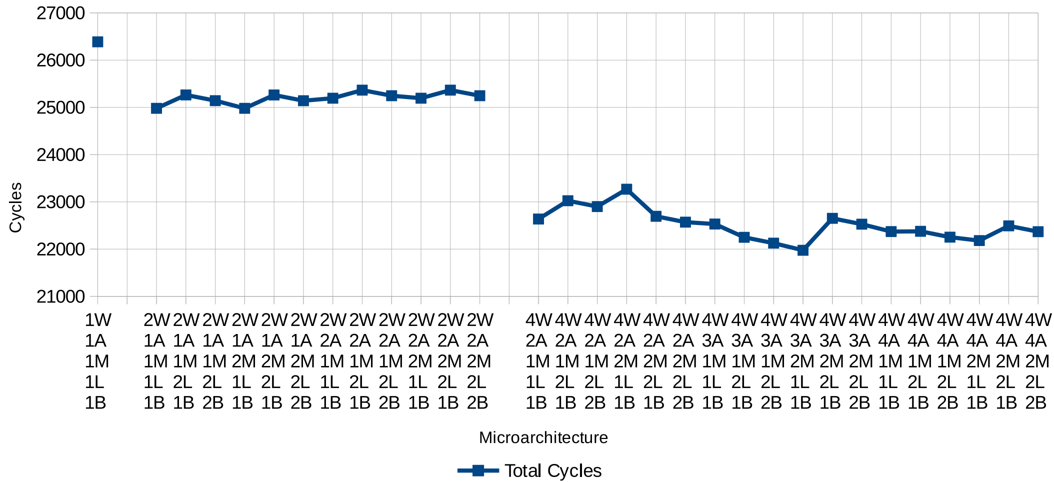

Figure C.12: Total average cycles for Fan2 of Gaussian Elimination using 1 context across varying microarchitecture configurations.

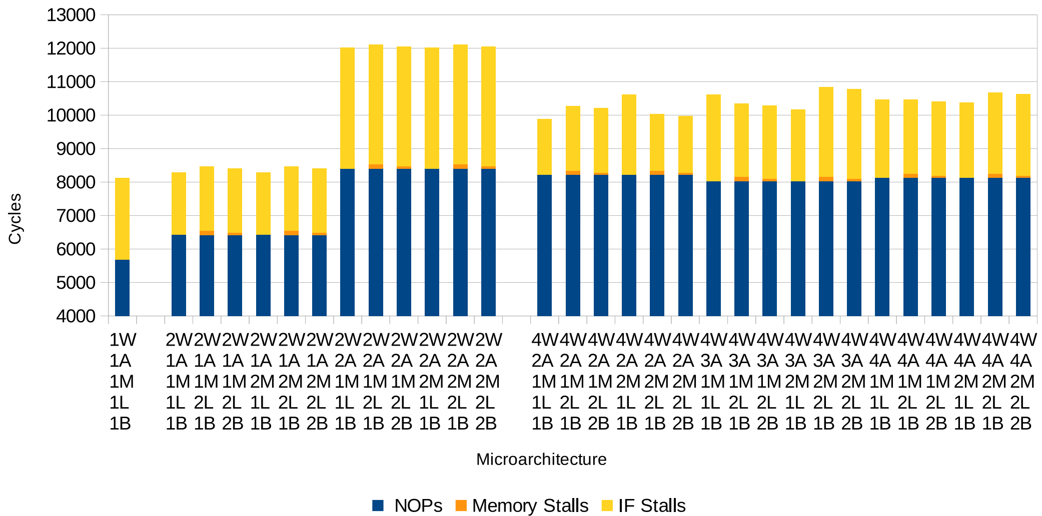

Figure C.13: Total average stalls and NOP cycles for Fan2 of Gaussian Elimination using 1 context across varying microarchitecture configurations.

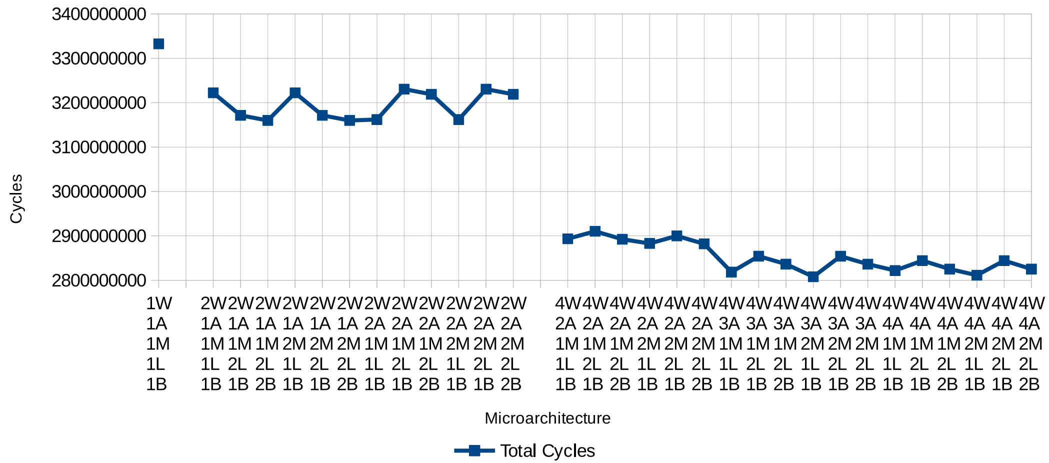

Figure C.14: Total cycle count for NBody using 1 context across varying microarchitecture configurations.

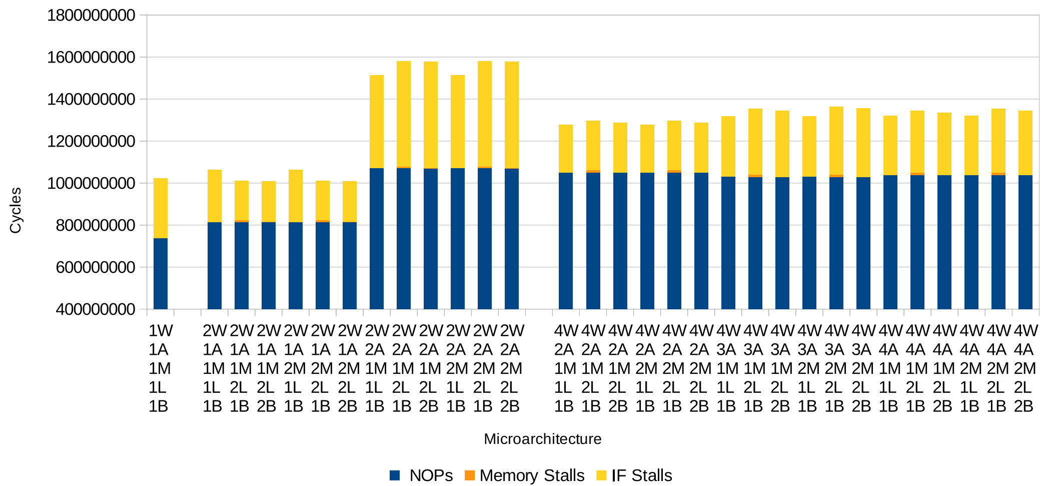

Figure C.15: Total stall and NOP count for NBody using 1 context across varying microarchitecture configurations.

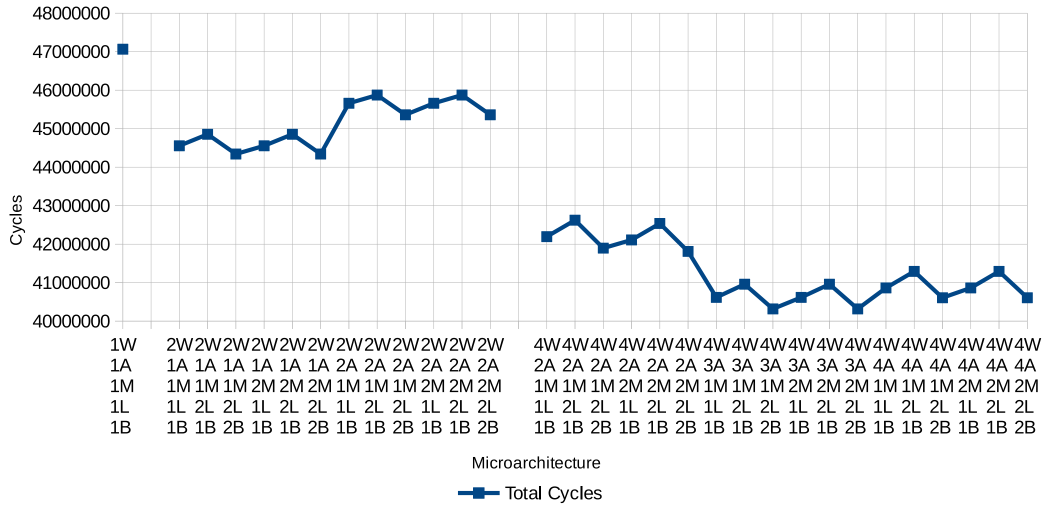

Figure C.16: Total cycle count for Nearest Neighbour using 1 context across varying microarchitecture configurations.

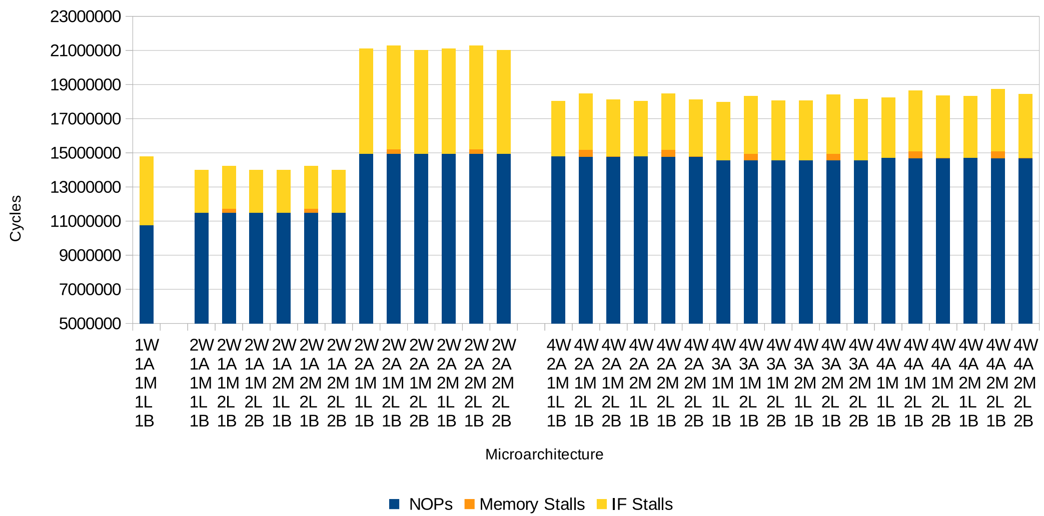

Figure C.17: Total stall and NOP cycle count for Nearest Neighbour using 1 context across varying microarchitecture configurations.

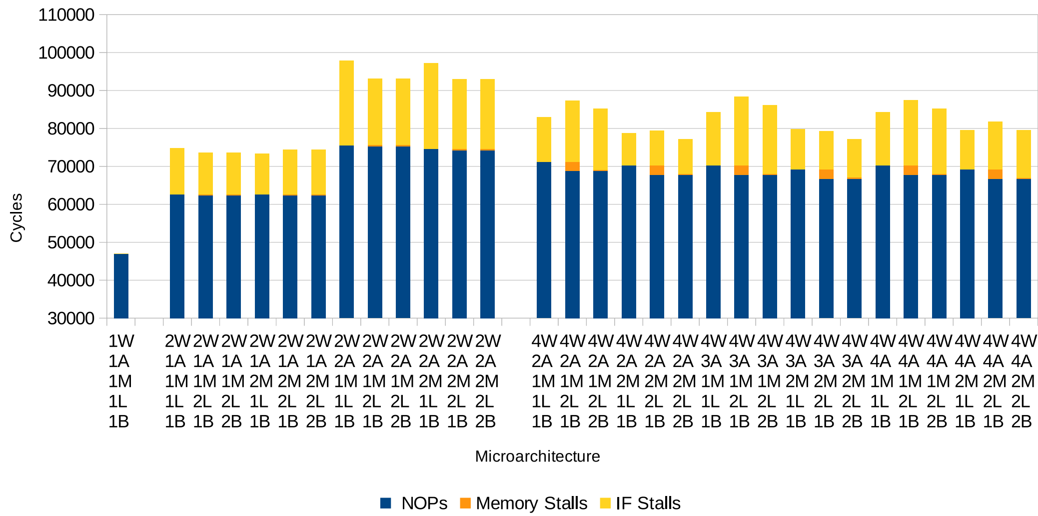

Figure C.18: Total average stall and NOP cycle count for nw_kernel1 from Needleman-Wunsch using 1 context across varying microarchitecture configurations.

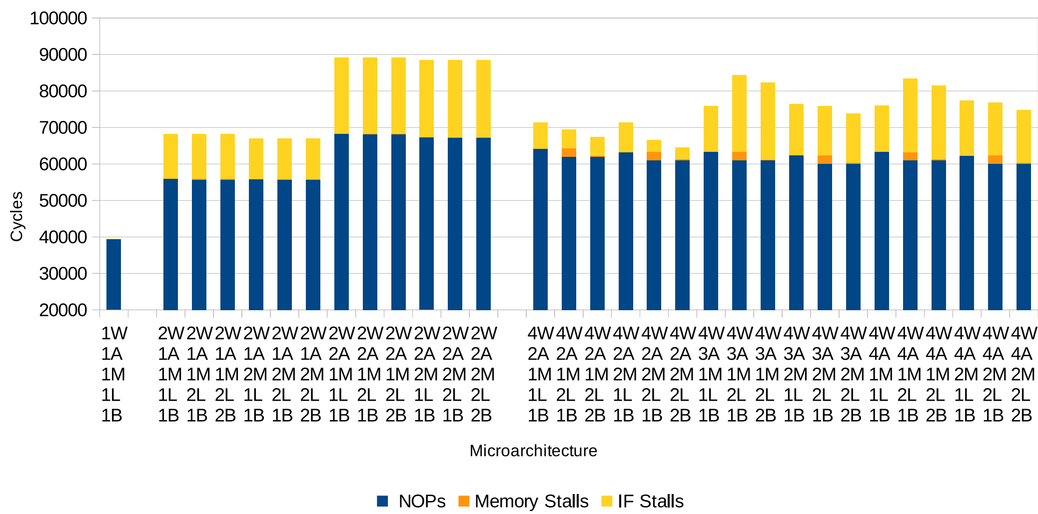

Figure C.19: Total average stall and NOP cycle count for nw_kernel2 from Needleman-Wunsch using 1 context across varying microarchitecture configurations.

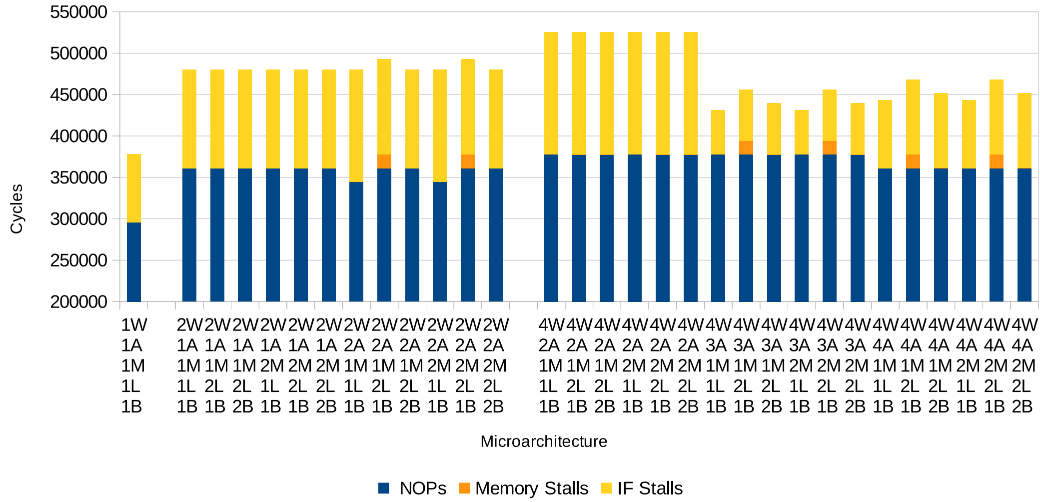

Figure C.20: Total average stalls and NOP cycle count for histogram from Radix Sort using 1 context across varying microarchitecture configurations.

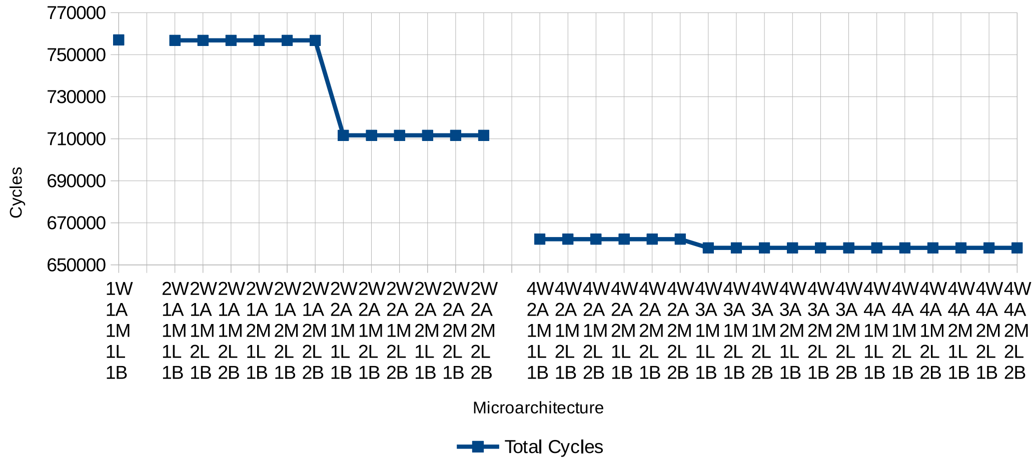

Figure C.21: Total average cycle count for permute from Radix Sort using 1 context across varying microarchitecture configurations.

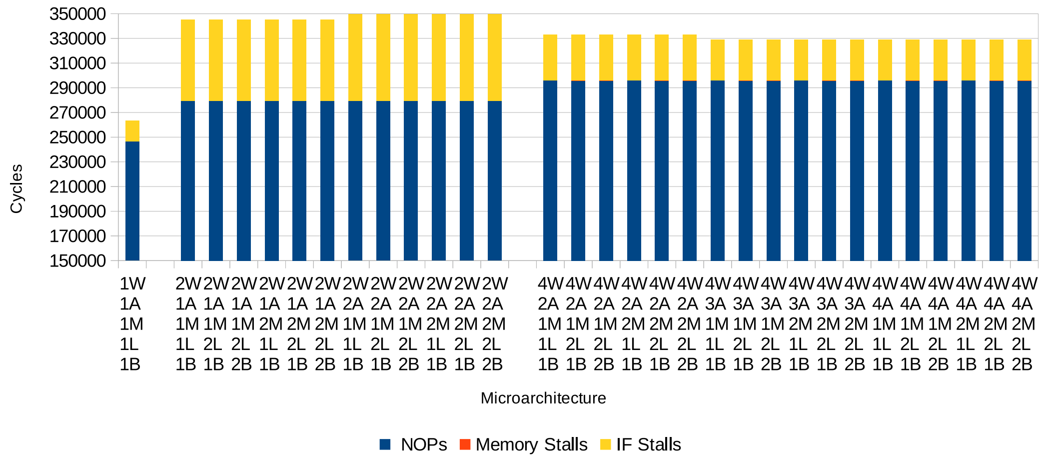

Figure C.22: Total average stalls and NOP cycle count for permute from Radix Sort using 1 context across varying microarchitecture configurations.

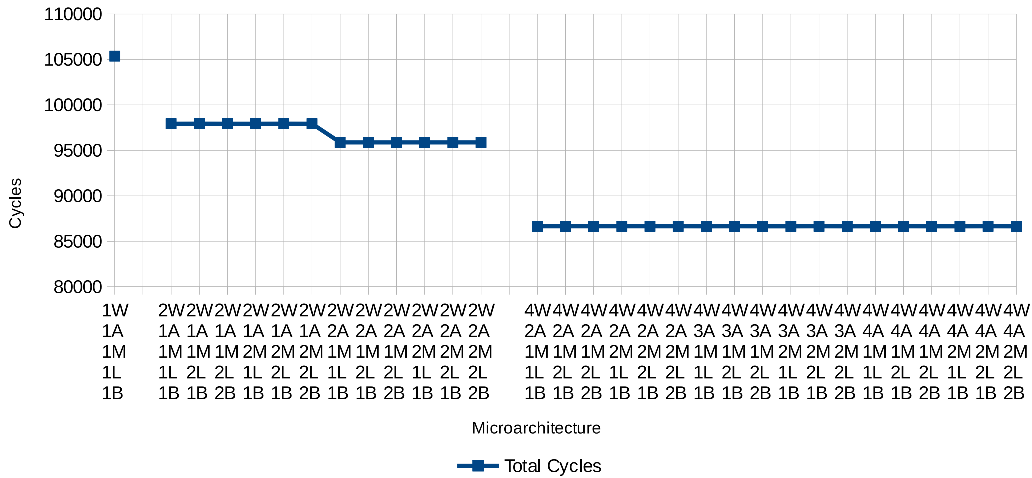

Figure C.23: Total average cycle count for ScanArraysdim1 from Radix Sort using 1 context across varying microarchitecture configurations.

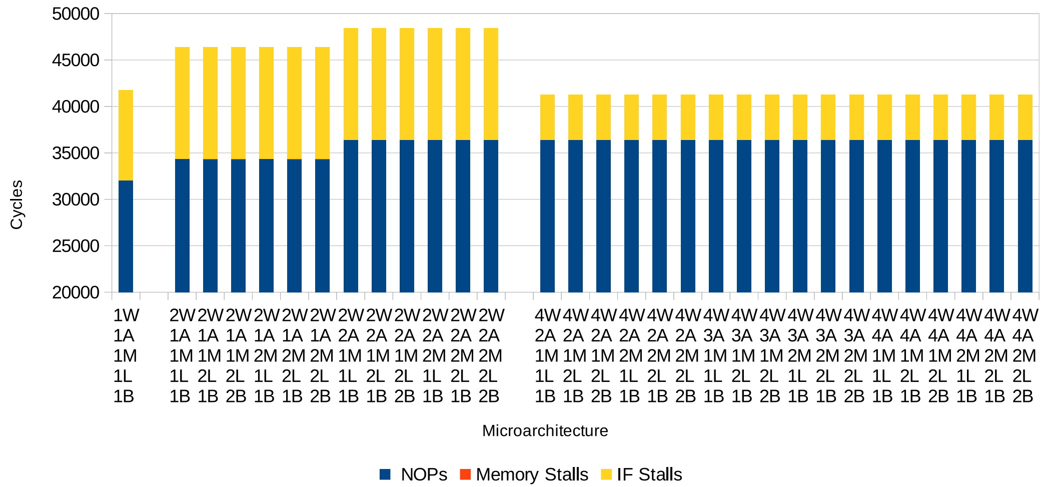

Figure C.24: Total average stalls and NOP cycle count for ScanArraysdim1 from Radix Sort using 1 context across varying microarchitecture configurations.

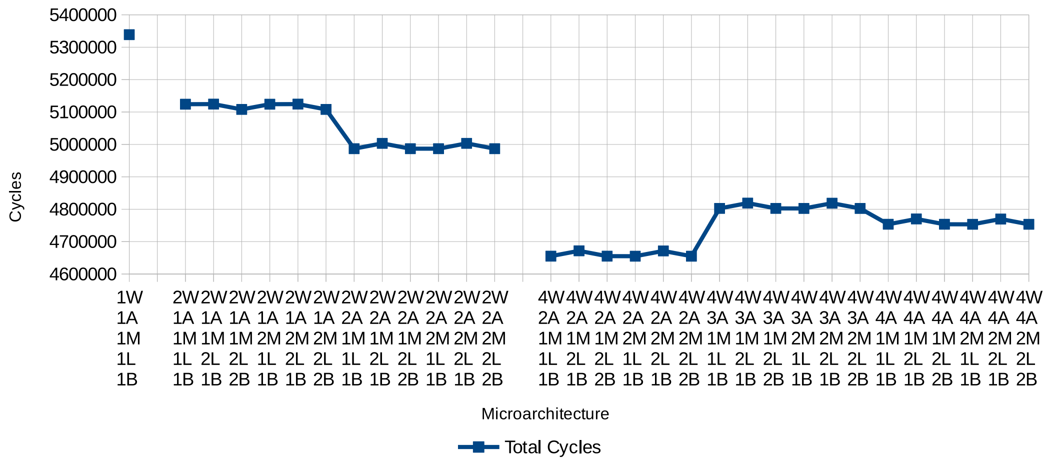

Figure C.25: Total average cycle count for ScanArraysdim2 from Radix Sort using 1 context across varying microarchitecture configurations.

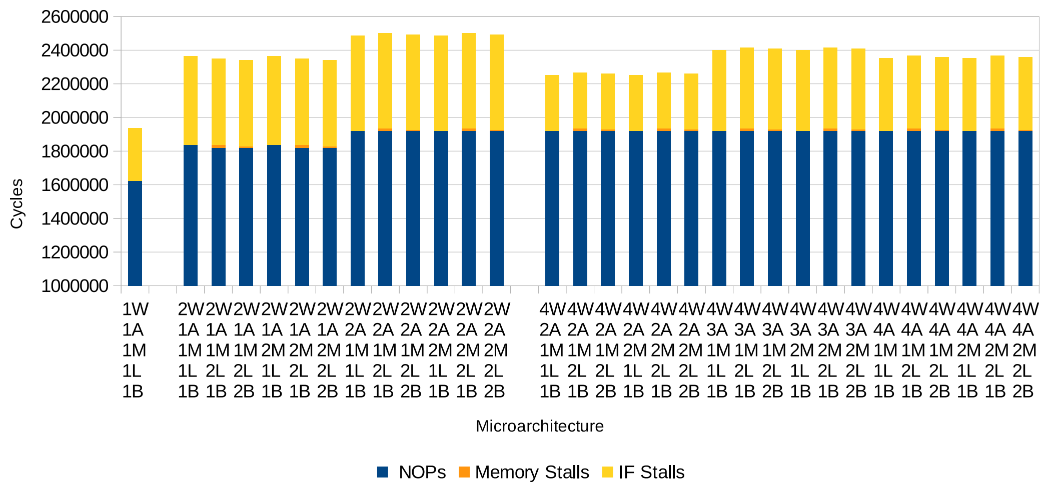

Figure C.26: Total average stalls and NOP cycle count for ScanArraysdim2 from Radix Sort using 1 context across varying microarchitecture configurations.

Appendix D

Complete Results from Experiments

Results can be found online at `https://github.com/grubbymits/thesis-results`