Loughborough University

This item was submitted to Loughborough University as a PhD thesis by the author and is made available in the Institutional Repository (https://dspace.lboro.ac.uk/) under the following Creative Commons Licence conditions.

creative commons

COMMONS DEED

Attribution-NonCommercial-NoDerivs 2.5

You are free:

- to copy, distribute, display, and perform the work

Under the following conditions:

Attribution. You must attribute the work in the manner specified by the author or licensor.

Noncommercial. You may not use this work for commercial purposes.

No Derivative Works. You may not alter, transform, or build upon this work.

- For any reuse or distribution, you must make clear to others the license terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder.

Your fair use and other rights are in no way affected by the above.

This is a human-readable summary of the Legal Code (the full license).

Disclaimer

For the full text of this licence, please go to:
http://creativecommons.org/licenses/by-nc-nd/2.5/

On the Automated compilation of UML notation to a VLIW Chip Multiprocessor

by
David Stevens

A Doctoral Thesis

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

December 2013

© David Stevens 2013

Loughborough University

Certificate of Originality
Thesis Access Conditions and Deposit Agreement

Students should consult the guidance notes on the electronic thesis deposit and the access conditions in the University's Code of Practice on Research Degree Programmes

Author……...……..…….

Title……..

I David Stevens, "the Depositor",
would like to deposit "On the Automated compilation of UML notation to a VLIW Chip Mulitprocessor", hereafter referred to as the "Work", once it has successfully been examined in Loughborough University Institutional Repository

Status of access OPEN / RESTRICTED / CONFIDENTIAL

Moratorium Period……years, ending……../……20…….

Status of access approved by (CAPITALS):……

Supervisor (Signature)……...……...

School of……...……

Author's Declaration *I confirm the following :*

CERTIFICATE OF ORIGINALITY
This is to certify that I am responsible for the work submitted in this thesis, that the original work is my own except as specified in acknowledgements or in footnotes, and that neither the thesis nor the original work therein has been submitted to this or any other institution for a degree

NON-EXCLUSIVE RIGHTS
The licence rights granted to Loughborough University Institutional Repository through this agreement are entirely non-exclusive and royalty free. I am free to publish the Work in its present version or future versions elsewhere. I agree that Loughborough University Institutional Repository administrators or any third party with whom Loughborough University Institutional Repository has an agreement to do so may, without changing content, convert the Work to any medium or format for the purpose of future preservation and accessibility.

DEPOSIT IN LOUGHBOROUGH UNIVERSITY INSTITUTIONAL REPOSITORY
I understand that open access work deposited in Loughborough University Institutional Repository will be accessible to a wide variety of people and institutions - including automated agents - via the World Wide Web. An electronic copy of my thesis may also be included in the British Library Electronic Theses On-line System (EThOS). I understand that once the Work is deposited, a citation to the Work will always remain visible. Removal of the Work can be made after discussion with Loughborough University Institutional Repository, who shall make best efforts to ensure removal of the Work from any third party with whom Loughborough University Institutional Repository has an agreement. Restricted or Confidential access material will not be available on the World Wide Web until the moratorium period has expired.

- That I am the author of the Work and have the authority to make this agreement and to hereby give Loughborough University Institutional Repository administrators the right to make available the Work in the way described above.

- That I have exercised reasonable care to ensure that the Work is original, and does not to the best of my knowledge break any UK law or infringe any third party's copyright or other Intellectual Property Right. I have read the University's guidance on third party copyright material in theses.
- The administrators of Loughborough University Institutional Repository do not hold any obligation to take legal action on behalf of the Depositor, or other rights holders, in the event of breach of Intellectual Property Rights, or any other right, in the material deposited.

*The statement below shall apply to **ALL** copies:*

This copy has been supplied on the understanding that it is copyright material and that no quotation from the thesis may be published without proper acknowledgement.

Restricted/confidential work: All access and any copying shall be strictly subject to written permission from the University Dean of School and any external sponsor, if any.

Author's signature……..Date……...……...……

user's declaration: for signature during any Moratorium period (Not Open work): ***I undertake to uphold the above conditions:***			
Date	Name (CAPITALS)	Signature	Address

Abstract

With the availability of more and more cores within architectures the process of extracting implicit and explicit parallelism in applications to fully utilise these cores is becoming complex. Implicit parallelism extraction is performed through the inclusion of intelligent software and hardware sections of tool chains although these reach their theoretical limit rather quickly. Due to this the concept of a method of allowing explicit parallelism to be performed as fast a possible has been investigated. This method enables application developers to perform creation and synchronisation of parallel sections of an application at a finer-grained level than previously possible, resulting in smaller sections of code being executed in parallel while still reducing overall execution time.

Alongside explicit parallelism, a concept of high level design of applications destined for multicore systems was also investigated. As systems are getting larger it is becoming more difficult to design and track the full life-cycle of development. One method used to ease this process is to use a graphical design process to visualise the high level designs of such systems. One drawback in graphical design is the explicit nature in which systems are required to be generated, this was investigated, and using concepts already in use in text based programming languages, the generation of platform-independent models which are able to be specialised to multiple hardware architectures was developed.

The explicit parallelism was performed using hardware elements to perform thread management, this resulted in speed ups of over 13 times when compared to threading libraries executed in software on commercially available processors. This allowed applications with large data dependent sections to be parallelised in small sections within the code resulting in a decrease of overall execution time.

The modelling concepts resulted in the saving of between 40-50% of the time and effort required to generate platform-specific models while only incurring an overhead of up to 15% the execution cycles of these models designed for specific architectures.

Acknowledgements

I would firstly like to take this opportunity to thank my supervisors Dr. Vassilios Chouliaras and Dr. Sijung Hu for their guidance and support throughout the course of this research.

The advice and direction of Dr. Vassilios Chouliaras has been critical from the start until the finish of this thesis.

I would also like to thank both the School of Electronic, Electrical and Systems Engineering at Loughborough University and the Electronics System Design Group within this school for the opportunity and financial support throughout the research stages of this Ph.D.

Table of Contents

List of Figures

List of Tables

1

Introduction

1.1 Problem Formulation

Modern embedded VLSI systems are more complicated than ever before, typically consisting of multicore, many-core, heterogeneous processors along with custom hardware blocks designed to efficiently compute specific tasks and optimise a particular metric such as power consumption or execution time. This leads to more complexity in terms of developing and deploying applications to execute on such systems while fully utilising all available computational resources.

A system can be comprised of multiple processors and hardware blocks, these processors can also be composed of multiple cores. Parallelism is defined as the ability to perform execution simultaneously across these processors, cores and hardware blocks. An application can be split up to perform computation in parallel; each parallel section of the application is referred to as a "thread" and multiple threads can be executed on a single core within a processor or as a custom hardware block designed to perform some specific tacks.

Theoretically, a system can be made up of many processors which in turn can be composed of many cores. These cores are then able to execute multiple threads. As more and more threads are active at the same time the complexity of managing these threads increases.

This parallel processing can be exploited both implicitly and explicitly. Implicit parallelism refers to the automated extraction of parallel computation by either compilers or other tools. Being implicit, it does not require the application developer to have any knowledge of the parallelism being exploited. This can be advantageous as no platform/tool specific pragmas or directives are required to be included in the program code, leaving the application developer to focus on the behaviour rather than the task of parallelising the computation. It can however provide draw-backs; implicit parallelism requires extremely intelligent tools

to extract parallelism from any program. This can lead to sub-optimal parallel code being produced when compared with code explicitly parallelised by the application developer.

Explicit parallelism is achieved through the inclusion of primitives and library calls which define sections of code that are suitable to be executed in parallel. The advantage of this approach is that the application developer has full control and visibility of where computation is parallelised. This requires no intelligent tools to perform the task and leads to easier debugging of code as no hidden optimisations are performed. The main disadvantage is that once code is written to include parallel primitives the code is not transferable to another platform or library without modifications being made to remove library specific primitives or calls.

Explicit parallelism is used most often in code due to the full view and control the application developer has of how computation is split across available hardware in a system. However, the result of multiple, heterogenous processors in a single system leads to an increase in the number of possible ways an application can be parallelised onto a particular architecture. As well as this, with more and more processors/computational components it becomes more difficult for the application developer to track all parallel computation.

One solution to this problem is to design the application at a higher abstraction level in a graphical environment. This graphical view of all computational elements aids the application developer in managing and viewing the explicit parallelism within the code. This can be achieved through the use of Unified Modelling Language (UML) [1] as the input medium for an application. A high level model, captured in UML, can then be refined (lowered) to be implemented on a physical hardware architecture. In this way the application developer can visualise both the hardware and software sections of a system and map them, in a graphical manner, as software to execute on one or more CPUs or as other hardware elements within the architecture.

Currently available tools and design processes which use UML as a graphical input language for multicore architectures require a platform-specific UML (PSUML) model to be captured before this can be refined to a physical architecture. This requires human interaction to make alterations directly to the UML model as well as an awareness of the best way to parallelise an application at design time.

One way around this is to design a platform-independent UML (PIUML) model which can then be refined to a platform-specific model dependent on the architecture being targeted. However, this solution requires a new set of tools to refine the PIUML model down to a multicore architecture.

The target architecture within the research conducted in this thesis is a statically customisable multicore processor. This processor can be instantiated with a variable number of cores in order to optimise for area and speed. Synthesis is directed by a machine model

file which defines the number of instances of the processor or cores to generate.

This forms the basis of the body of research discussed within this thesis; the methods for creating an application at a high abstraction level (in UML) which can be refined for a statically customisable multicore architecture. The application design takes place at a high level such that the application developer is unaware of the underlying architecture which is being targeted. This allows the application developer to focus mainly on the algorithmic aspect of a system while still being able to easily test and deploy the application on a physical hardware implementation.

Alongside the abstraction of the hardware from the application developer, investigations into exploiting various methods of parallelism within the architecture were conducted to find optimal methods of parallelism.

Throughout this thesis a "system" refers to a full implementation of an application and architecture. The application is the software designed to perform a specific task and the architecture is the physical hardware which is available to execute the application.

1.2 Aims and Objectives

The aims of the research conducted are:

- Develop a tool chain for a unique Very Long Instruction Word (VLIW) Chip Multiprocessor (CMP).
- Simplify the UML system design for a statically customisable multicore platform.

In order to meet the aims a set of objectives have been defined:

- Investigate parallelism methodologies.
- Design and Implement tool chain for a unique VLIW CMP.
- Implement parallel execution in a VLIW CMP.
- Investigate high level system design in UML.
- Develop UML modelling technique for statically customisable multicore platforms.
- Develop UML applications to utilise UML modelling techniques and tool chain.

The objectives are based around implementing a tool chain for a VLIW CMP which exploits multiple levels of parallelism in order to increase performance in terms of execution time. This VLIW CMP is then used as an architecture being targeted from a high level design language. This high level design language requires new modelling techniques to be developed to target this statically customisable architecture platform. Finally the creation of applications to test and display the tool chain and modelling techniques is required.

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

Parallelism, in terms of computing, is a method of simultaneously processing tasks with the main aim to decrease computation time. This thesis focuses on three main forms of parallelism; Instruction Level Parallelism (ILP), Thread Level Parallelism (TLP) and Task Parallelism (TP).

In 1972 Michael Flynn proposed a classification of computer architectures [2] which defines the number of instruction and data streams available in an architecture. The various forms of parallelism which are the focus of this thesis fall into multiple classifications depending on their implementations.
Flynn's Taxonomy describes four classifications of architectures:

- Single Instruction, Single Data (SISD) - This describes a conventional, single CPU architecture where each instruction is executed serially on a single data stream.
- Single Instruction, Multiple Data (SIMD) - In this classification the same instruction is executed across multiple data streams, this is useful for repetitive tasks over a data set.
- Multiple Instruction, Single Data (MISD) - Taking into account fault tolerance; multiple instruction streams compute over a single data set in order to reduce errors in computation.
- Multiple Instruction, Multiple Data (MIMD) - Both instruction and data streams are separate in this architecture classification; independent CPUs operate on separate data streams.

The MIMD category has been extended to include more detailed classifications for modern computing:

- Single Program, Multiple Data (SPMD) - A single program computes across multiple data streams; the same program operating across multiple processors each at a different point in the program.
- Multiple Program, Multiple Data (MPMD) - Multiple programs executing across multiple data streams; multiple processors execute multiple programs.

1.3.1.1 Instruction Level Parallelism

Instruction Level Parallelism (ILP) is a method of extracting instructions which can be executed simultaneously by an ILP pipeline without affecting the overall execution of the program [3]. In terms of Flynn's Taxonomy, this form of parallelism can be available in all classifications so long at the CPU within that clasification can execute multiple instructions (from a single stream) in a single cycle.

ILP can be performed using both static and dynamic scheduling. Both methods require an intelligent scheduling algorithm to ensure correct execution is performed while exploiting the maximum Instructions Per Clock (IPC). The IPC metric is used to determine the ILP of the code being executed, with the IPC being as close to the number of issue slots as possible being the optimal result. The difference between static and dynamic scheduling is where this scheduling takes place, static scheduling is performed at compile time by a software process and dynamic scheduling is performed at run time in hardware.

Static scheduling relies on a software algorithm to re-arrange instruction into packets to be executed. These packets are groupings of instructions which can be executed in parallel with no data-dependencies across the instructions. Both Very Long Instruction Word (VLIW) and in-order Superscalar processors use this form of scheduling to maximise IPC within code.

Dynamic scheduling is performed at run time within the processor executing the code, out-of-order Superscalar processors use dynamic scheduling to improve execution time. Hardware within the processor queues instructions and only performs execution once input registers are made available from previous instructions. This method enables instructions which do not contain data-dependencies to be executed in parallel, this allows subsequent instructions within the queue to be executed out-of-order if they can be computed without the use of values currently being computed. The Tomasulo algorithm [4] is a hardware implementation which supports full, out-of-order execution.

This method of parallelism is dependent on the type of application being executed as well as the architecture on which is it being executed. Algorithm 1 shows three instructions, instruction 2 can be executed at the same time as instruction 1 due to the fact it does not require the result of 1 (E) to perform its own calculation. Instruction 3 on the other hand requires both the calculation results from 1 and 2 to produce the correct result. This example, without the concept of ILP, would require 3 cycles to complete, ignoring any latencies of functional units performing computations. In an architecture which exploits ILP it could be executed in 2 cycles due to the availability of extra computational resources and the lack of data-dependency between instructions 1 and 2. This is a very simple example: the exploitation of ILP is heavily dependent on the number of computational resources available

Algorithm 1 Example of Instruction Level Parallelism. When executed on an ILP system instructions 1 and 2 can be executed in parallel as there are no data-dependencies between the operands. Instruction 3 requires the output of 1 and 2 and so must stall until these values are available.

1. $E = A + B$
2. $F = C + D$
3. $G = E * F$

to parallelise instructions as well as the nature of the program being compiled, the compiler and the micro-architecture of the hardware.

An application with an abundance of data dependencies or control flow changes will result in a low level of ILP being extracted. This is due to the CPU having to stall in order for the results of previous calculations to be computed before being able to perform current calculations using these results. Control flow changes also limit ILP due to the pipelined nature of processors; each time the control flow of an application changes new instructions from a different region of the instruction code need to be fetched and decoded to be executed, resulting in empty clock cycles during this pipeline restart process.

1.3.1.2 Thread Level Parallelism

Thread Level Parallelism is the process of creating and simultaneously/concurrently executing software threads of a single application on separate hardware resources available within an architecture [5]. Threading occurs at two levels; simultaneous and interleaved. Simultaneous Multi-threading (SMT) refers to multiple threads performing computation at the same time. These simultaneous threads can be executed on the same core through the inclusion of extra registers to enable this parallel execution or on separate cores, referred to as Chip Multiprocessing (CMP). Interleaved Multi-threading (IMT) is performed on a single processor or core where a single thread is able to execute at any one time. This is achieved primarily through the use of software libraries which create new threads and perform monitoring and management of created threads. Algorithm 2 shows a pseudocode example of creating two new threads, performing some computation and then synchronising with these created threads. The thread executing the code shown in Algorithm 2 is referred to as "master" and the threads it creates as "slave". *Number_of_Threads* is a variable used within loops to create and synchronise the slave threads. In this example this is set to two, resulting in two new threads of execution being created, and a total of three threads executing at the same time (including the master thread). The *thread_id[]* array is used to store identifiers of the created slave threads. *work()* is a function, undefined in this example, which performs some form of computation.

Algorithm 2 Example of Thread Level Parallelism, Master thread creates two new slave threads, executes work() and synchronises with slave threads.

```
Number_of_Threads ← 2 // Define number of Threads
thread_id[Number_of_Threads] // Global array to store thread ids of created threads
work() // Function which performs some computation
for i = 0 → (Number_of_Threads − 1) do
    // Perform execution of a function in a separate thread of execution
    thread_id[i] ← create_thread(work)
end for

// Master thread performs execution
work()

for i = 0 → (Number_of_Threads − 1) do
    // Synchronise with thread i
    join_thread(thread_id[i])
end for
// All created threads of execution are now synchronised with Master thread
```

Each time the first *for* loop is iterated a new slave thread is created and its identifier is stored in *thread_id[]*. In most threading libraries a function pointer is passed to the thread creation operation and it is this function which the slave thread begins execution. In this example the *work()* operation is to be computed in each slave thread.

After the first *for* loop the master thread also executes the *work()* operation. At this point in the code all three threads are executing this *work()* operation. Once the master thread has finished executing this operation the second *for* loop is iterated through. This *for* loop is used to perform synchronisation between the master and slave threads. Using each slave threads identifier from *thread_id[]* the master thread is stalled until the specified thread has completed executing the *work()* operation.

Once both slave threads have completed execution only the master thread is left executing. TLP can be used for splitting workloads across multiple threads to decrease execution time as well as to designate specific tasks to each thread, for example an application with a Graphical User Interface (GUI) could have a thread performing background computation and another drawing the windows required by the GUI. This allows both tasks to be executed at the same time which results in the GUI not becoming unresponsive when the background task performs computation.

TLP can fall under various definitions within Flynn's taxonomy. In architectures where

multiple threads can execute simultaneously (SMT) it would fall under the MIMD section as multiple processes are executed in parallel on separate data items. It is also possible to execute TLP on a single processor which interleaves multiple executing threads (IMT), in this case only a single instruction stream is executing at any given time and so the SISD or SIMD classification may be more appropriate.

1.3.1.3 Task Parallelism

Task Parallelism (TP) is achieved through multiple processors in a shared-memory system executing different process. In Flynn's Taxonomy this falls under the MIMD classification where multiple processors simultaneously execute different instructions on separate data items. An example of TP is shown in Algorithm 3. In this example the task being performed by each processor executing the code is dependent on its unique identifier (CPUID). The processor with CPUID *0* performs task zero, CPUID *1* performs task one and all other processors perform a default task. This method of splitting workloads over separate processors allows all available processors to use a unified instruction RAM and results in less complicated compilation as only a single instruction and data RAM is required to be generated.

Algorithm 3 Example of Task Parallelism, tasks are performed based on CPUID (unique identifier of CPU).

```
if CPUID == 0 then
    // Perform task 0
else if CPUID == 1 then
    // Perform task 1
else
    // Perform default task
end if
```

An alternative implementation of TP is based on distributed-memory systems, this implementation is usually performed across different processors. These processors and tasks being parallelised do not require to have anything in common and each has its own memory system. For distributed TP a method of communication is required to exchange data between the executing tasks. The Message Passing Interface (MPI) [6] was designed for this purpose and provides and means of communication between tasks. Distributed TP is not within the scope of this thesis as the research conducted is based around an embedded approach to parallelism.

1.3.2 Very Long Instruction Word Processor

Statically scheduled ILP architectures, such as VLIW processors, are a natural target for an ILP compiler. Originally conceived at HP Labs [7], the idea was that the best way to speed up execution was to perform instructions in parallel (in a long packet) instead of serially. VLIW processors [8, 9] execute multiple instructions in a single clock cycle, compared to a scalar processor which executes only a single instruction. The advantage of being able to execute multiple instructions is that overall execution time is reduced. Some noteworthy VLIW processors include the Intel Itanium [10], the Trimedia media processor from Philips [11] and the TMS320C62x series of DSPs from Texas Instruments [12].

VLIWs usually have a variable number of functional units and issue slots. Functional units are computational elements within the processor, such as arithmetic logic units and multiply units, whereas the issue slots refer to the number of instructions which can be fetched and decoded at any time. With the ability of consuming variable number of instructions per clock cycle the aim is the keep the available functional units active at all time. For example a VLIW with an issue width of four is being fully utilised when all four issue slots are filled each cycle.

The ability to execute multiple instructions in parallel while maintaining correct execution implies extra complexity in both hardware and software to track data-dependencies across instructions and registers. Without this, registers containing incorrect data could be used in calculations resulting in data path execution issues. VLIWs rely on a compiler, which performs scheduling of instructions for parallel execution, to be aware of their hardware configurations. This includes all computational resources and latencies to efficiently schedule instructions and fully utilise the CPU. As the instruction schedule is built statically by the compiler, VLIW processors offer significant computational power with less hardware complexity than is associated with out-of-order superscalar processors which perform scheduling of instructions at run time.

1.3.3 High Level Design

The current industry standard for modelling software-intensive systems is the Unified Modelling Language (UML) [1]. It was created by the Object Management Group [13] and is a general-purpose modelling language for object-oriented software engineering. UML is used to create visual models of applications using a set of graphical notations and diagrams. The UML standard is extended to support the development of embedded systems using the Modelling and Analysis of Real Time and Embedded systems (MARTE) profile [14, 15]. The MARTE profile consists of stereotypes which are used to label UML elements with various hardware and software concepts. Using these stereotypes it is possible to capture an under-

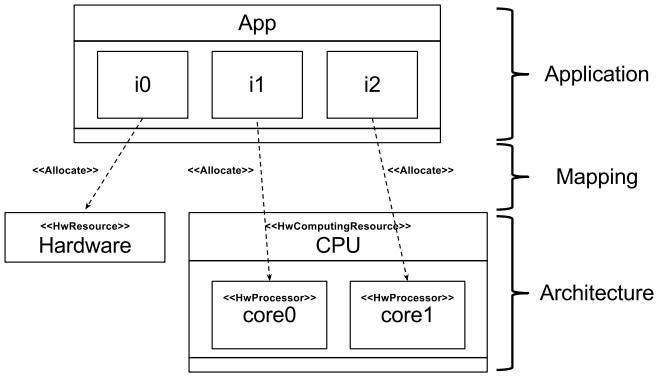


Figure 1.1: Example model showing Application mapped to both a hardware block (*HwResource*) and cores within a processor (*HwProcessor*)

lying architectural view and specify how an application should be mapped to a hardware architecture.

The exchange of UML models between tools vendors is encapsulated in the XML Metadata Interchange (XMI) standard [16]. This standard is an XML-based description language in which the behaviour and structure of the UML model is represented. XMI is supported by most UML tools, however, different tools generate different flavours of XMI models which leads to compatibility issues in transporting these different models between tools.

It is standard across multiple UML design methods to split the a model into three distinct sections captured in a single UML model shown in Figure 1.1:

A) Application

B) Architecture

C) Mapping

The application is captured in Class Diagrams in UML which contain the static structure and behaviour of the model. This static structure is then used as the basis in the Composite Structure Diagrams; these diagrams capture the deployment of objects and the communication structure of the application. It is at this level which the architecture is also captured using the MARTE profile to stereotype classes as either a "hardware resource", a "processor" or a "processor core" and then, using allocate arrows, to map the application elements to the architecture. An example of this design specification is shown in Figure 1.1; it shows an application with three objects where object *i0* is mapped to a hardware block (*HwResource*) and objects *i1* and *i2* are mapped to separate cores within a processor (*HwProcessor* within *HwComputingResource*).

Using a modelling language, such at UML, the design being captured should be kept as

implementation-independent as possible. By being keep implementation-independent, designs are able to be translated between different architectures and implementations without the need for the application developer to modify or refine the initial model.

1.4 Contributions of Thesis

The work presented in this thesis focuses on Thread Level Parallelism. Firstly, explicit TLP is investigated in a statically configurable VLIW processor which implements thread management in hardware. This is then further investigated using UML notation to map tasks to available computational resources within the VLIW processor.
The main contributions made within this thesis are as follows:

Design Methodologies: UML design notations enabling a single platform-independent model to be created which can then be transformed to fully utilise a statically configurable multicore architecture. Wildcard multiplicities and adjustable mappings are used while defining the UML model and then translated when required to implement a statically mapped, platform-specific UML model.

Translation Processes: Scripts, using the UML design methodologies, were developed to perform translations between platform-independent and platform-specfic UML models.

Software Tool Chain: A tool chain was developed for a VLIW CMP along with a full tool flow from C and UML to execution on both hardware and cycle-accurate simulation.

Hardware threading: The conception and validation of a hardware threading mechanism, using a subset of the PThread library to perform fine-grain parallel execution within applications.

Adoption of Tools and Methodologies: A selection of the UML design methodologies and the software tool chain are in use by commercial companies as part of an FP7 project.

1.5 Thesis Outline

The remainder of this thesis is organised as follows:

- Chapter 2 is a detailed survey of research and development including various level of parallelism and high level system modelling.

- Chapter 3 summarises the background literature survey and identifies the direction and area of research which will be undertaken.

- Chapter 4 presents, in detail, the experimental framework which has been developed as an initial building block to fully implement and evaluate the research contributions.

- Chapter 5 provides a detailed view of the novel areas of research as well as introducing areas/projects in which concepts presented in this thesis and tools developed throughout the research are being adopted.

- Chapter 6 describes a selection of applications (C and UML) developed along with results and comparisons to provide investigations into explicit parallelism and high level modelling concepts.

- Chapter 7 concludes this research undertaken and offers possibly extensions to complement the results presented here.

- Chapter 8 presents a list of journal, conference and project publications generated throughout the course of the research undertaken.

2

Background

2.1 Chapter Objectives

The objective of this chapter is to present a review of previous studies and research into key areas of interest defined in chapter 1. These areas include:
1. UML based High Level Modelling
2. Parallelism
3. VLIW Processors
This background research was used to find a research area in which to focus the efforts of this thesis to best contribute to scientific knowledge.

2.2 High Level Modelling

2.2.1 Object-Oriented Programming

Object-Oriented Programming (OOP) consists of partitioning applications into a set of abstract data types, called objects. The objects are instantiations of behavioural elements called classes. Classes consist of a set of private attributes and behaviour defined to perform specific tasks. They have a public interface which enables other sections of an application to interact with their internal behaviour. Communication between objects takes the form of messages passing across the public interfaces. The main benefit of OOP is the avoidance of redundant code and the easier management of software complexity. Redundant code is limited through the use of inheritance, where base classes can be extended to fit a specific purpose. Software complexity is managed through the encapsulation of sections of code within objects, leading to ease of testing, as it is possible to test small sections of an ap-

plication separately. Once tested and confirmed correct these classes can be instantiated multiple times as well as reused in other applications as required.

2.2.2 Visual Programming Languages

Visual Programming Languages (VPL) use a selection of diagrams to define the structure and behaviour of applications. They aim to improve the accessibility of information between the application developer and the design and development tools. The authors of [17] present an in depth study into various visual programming tools and languages and conclude that visual programming reduces non domain-specific development time in an industrial setting due to the ease of communication between application developers and end clients. Through the use of visual programming the end clients can be involved more directly with the development of an application by being able to visualise the actual implementation rather than by traditional documentation practices. This enables end client interaction and feedback at an earlier time during design and development which can result in issues being caught before implementation takes place. Visual programming adds layers of abstraction from the underlying implementation and makes application development more accessible. The book [17] presents case studies of companies using visual programming languages and tools in real life development; The Measurement Technology Center (MTC) used LabVIEW [18] to assist in the redesign of flight software used for NASA's Galileo mission. This task was compared by one group using a VPL approach to another using a text-based programming language (C). The outcome of the study, based on the creation of 40 applications, is that visual programming increased productivity, independent of domain.

> *"MTC consistently finds that visual programming reduces development time by at least a factor of 4, and up to an order of magnitude"*
>
> – "Visual Object-Oriented Programming" [17], Chapter 2.7, Page 38

The authors state that one of the complexities of using VPLs is that by creating a visual language there is no requirement to be able to visualise all aspects of any other language definition. This leads to issues when attempting to refine a visual model down to a physical implementation.

> *"In creating a visual OOP language, the designer has the opportunity to devise a language that is designed specifically to mesh with the visual techniques he or she wishes to emphasize, and is not required to find ways to provide visual capabilities for a predefined language."*
>
> – "Visual Object-Oriented Programming" [17], Chapter 8.1, Page 162

2.2.3 Unified Modelling Language

The current visual design language of choice in software-intensive systems is the Unified Modelling Language (UML) [1]. UML is a general purpose object-oriented modelling language created in 1997 by the Object Management Group [13]. UML can be used at various levels within the software development life cycle, from requirements through design & implementation and validation & verification [19, 20, 21, 22]. At the conception stage application developers can create basic skeleton structures of applications along with requirements for elements through the use of use case and timing diagrams. UML is designed to be domain independent where no implementation details are included. This results in the ability to transform the UML model easily between domains and models of computation. By including domain-specific elements in a UML model, including behaviour in the form of state diagrams and action code, it is possible to refine the UML model into a real life working implementation.

There are various sources of methods of UML design for software engineering [23, 24, 25, 26]. All go into details around designing software models of applications using the UML specification and an object-oriented method of software development. They include case studies of real life applications captured in UML.

All concepts available within the UML specification are captured in an underlying meta-model. This meta-model is made accessible to the application developer through the Graphical User Interfaces (GUI) of UML tools and their available diagram types. There are a total of 14 different diagrams which model the structure and behaviour of an application. These diagrams are listed below along with descriptions of their usage. Different UML tools display these diagrams slightly differently and also allow multiple types of diagrams to be captured in a single diagram. For example, the *Class Diagram* displays the attributes of classes in a model, these attributes can be typed as other classes which is essentially similar information to that captured in the *Composite Structure Diagram*.

Structure Diagrams:

- ***Class*** : Applications Classes, Operations, Attributes and their relationship.
- ***Component*** : Split of application across Components and their connections.
- ***Object*** : Instantiations of Classes and their connections.
- ***Profile*** : Low level inheritance of UML model.
- ***Composite Structure*** : Internal structure (Objects, Ports and Connectors) of Classes.
- ***Deployment*** : Physical elements of system and their interactions.

- ***Package*** : Relationships between top level groupings of application

Behaviour Diagrams:

- ***Activity*** : Overall control flow of an application.
- ***Use Case*** : High level functionality of system.
- ***State Machine*** : Behaviour of Classes using states, transitions and actions.
- ***Interaction*** : Control and data flow.
  - *Communication* : Messages between Objects and Parts.
  - *Interaction* : Overview of Communication Diagrams.
  - *Sequence* : Message passing between Objects.
  - *Timing* : Similar to Sequence diagrams with the inclusion of the lifetime of Objects.

The UML standard is extended to support the development of embedded systems using the Modelling and Analysis of Real Time and Embedded systems (MARTE) profile [27]. The MARTE profile consists of stereotypes which are used to designate UML elements for specific hardware and software concepts. Using these stereotypes it is possible to capture an architectural specification of a hardware platform and map application elements to this architecture. The inclusion of the MARTE profile into a UML model allows architectures as well as applications to be captured and modelled using UML.

The MARTE profile consists of 5 sections which all serve a part in defining embedded systems in UML:

- ***Core Elements*** : Foundation elements of structural and behavioural in MARTE profile.
- ***Non Functional Property Modelling*** : Textual annotations to capture physical attributes of systems.
- ***Time Modelling*** : Real-time system modelling.
- ***Generic Resource Modelling*** : Extensions for modelling architectural elements.
- ***Allocation Modelling*** : Mapping of application to architectural elements.

The ability to model both hardware and software aspects of systems has led to a wide range of research and development into various means of synthesising hardware systems from

designs captured in UML, to creating multicore software applications and merging both of these concepts together to generate fully co-designed system.

Previous studies using UML as an input modelling tool to generate real life hardware and software systems include Unified Process for Embedded Systems (UPES) [28] and Hardware and Software Objects on Chip (HaSoC) [29]. These studies use UML profiles, including MARTE and UML for Real Time (UML-RT) [30], to describe partitioning UML models across the hardware and software domains inclusive of scheduling and communication mechanisms required for this partitioning. In [28] the authors focus on developing a design principle for embedded system co-design along with an environment for simulation of both hardware and software sections of the UML model. Similarly, in [29] the full life-cycle of the design is explored and emphasis is placed on early stage execution of the model along with re-use of previously developed hardware and software components.

The idea of application mapping on a configurable architecture followed by design space exploration was recently proposed in [31]. Here the authors present Gaspard, a design framework which exploits both data and task level parallelism. Using multiplicities of objects in the UML application which are mapped statically to a multi-core co-designed architecture using the MARTE profile. The approach is performed at a high abstraction level and is refined towards a lower level through iterations of deployment. The main problem with this approach is that all aspects of the application, architecture and mappings are required to be captured in the initial UML model. When a different mapping is required, for example, when an object is to be mapped to a dedicated hardware block rather than a software block, the application developer is required to manually make this alteration in the UML model.

Another study which uses the Gaspard design framework and focuses on alternative mappings is presented in [32]. Here the authors present the concept of defining alternative mappings between dedicated hardware blocks and multicore processors using finite-state machines within the UML model to switch between mapping modes. However, nothing is stated about how these mappings are generated and it reads as though they are required to be specified by the application developer in the initial UML model.

In [33] the authors present a new UML extension profile which defines a design rule and set of stereotypes to better model the architecture being targeted as well as the mapping of the application to this architecture. The application section is a collection of behavioural and structural components which define a platform-independent model. The architecture section consists of existing library architectural elements with parameterisations, such as frequency, area and power. Finally the mapping section connects the application and architecture sections together through the use of dependencies. The authors also present a profiling tool which uses the stereotypes and rule sets presented to optimise the communication overhead between application elements.

This work is discussed further in [34], where the authors define the Koski design flow. This is a full flow from specification down to prototyping on a Field Programable Gate Array (FPGA). The authors have developed a framework which splits the full design flow into smaller, more manageable design sections, including algorithm development, design entry and verification. They also address design space exploration using an iterative approach on these design sections to produce a model which meets the initial specifications provided. The design space exploration is achieved through automated tasks which are controlled through the Koski GUI.

In [35] the authors discuss direct hardware synthesis from UML. They present a brief overview of how UML model behaviour can be implemented in hardware using finite-state machines.

In [36] the author presents a prototype tool using UML to validate hardware design, using UML to define the correct execution of an embedded systems dependent on a design specification. The authors generate HDL assertions which can be used to verify the hardware produced adheres to the initial design specifications. This study is based on behaviour captured in state machine diagrams, which are translated to a hardware description language.

The authors in [37] present a method of translating UML behavioural models to hardware VHDL. This is achieved through the use of a high level language, named Semantic Model Definition Language (SMDL), which is used to interface between models of computation and translates concepts within the UML model into formal language specifications. These specifications are then translated into a low-level finite-state machine described in VHDL to be synthesised on a hardware architecture. As with the studies referenced above the authors focus solely on behaviour captured in state machines within the UML model. The authors note that there is an issue in terms of semantics of UML models from different UML tools which leads to ambiguities when translating behaviour to another language.

Presented in [38] is a methodology to use UML for System-on-Chip (SoC) design, from system level specifications down to partitioning between the hardware and software domains. In this study, only the application section is captured in the UML model, the architecture and mapping is included through a user guided stage of a predefined library of architectural templates. The authors define a method of extracting both structural and behavioural details from a UML model, using this information they then present the application developer with a list of objects and communication channels. Through a GUI the application developer is able to choose a predefined architecture to map the application to. This methodology allows a single UML model to be mapped in multiple configurations onto various architectures, leading to fully platform-independent UML modelling.

In [39] the authors present a subset of the MARTE profile to be used to target mul-

tiprocessor systems and perform system-wide analysis based on timing requirements and communications between objects prior to developing any behavioural implementations. Presented is a modelling system for fast design space exploration based on worst case scenarios of communication timing between the structural elements of an application. The benefit of this method is that it allows preliminary investigations to aid the application developer in the best methods of implementing behaviour. However, it is based on worst case communication scheduling and the authors do not discuss any physical implementation of the design presented to confirm whether these communications assumptions are correct.

In [40] the authors present a model driven engineering methodology to generate OpenCL [41] code from a UML model, using MARTE stereotypes, to be executed on a heterogenous hardware system. They present example applications which are mapped to hardware elements adhering to the OpenCL specification and use transformations to generate an OpenCL compliant application which will execute on OpenCL compatible systems.

There are commonalities which can be seen throughout the investigations into developing UML models and profiles for both application and architectural aspects of system. These are:
A) the split of the model into application and architecture sections along with a method of mapping objects within the application to specific elements within the architecture.
B) the trade off of keeping the abstraction level of the model high while including enough information to fully deploy all aspects of the application onto an underlying architecture.
C) the use and extensions of UML profiles, such as MARTE, to stereotype sections of the UML model to define architectural components of a system.

2.3 Parallelism

Parallelism in computing is the method of executing multiple instructions or tasks simultaneously. This can be applied at multiple levels of granularity within an application. At a coarse-grain, sections of an application can be split into independent tasks, or threads, which can be executed in parallel. It is also possible at a finer grain; at the basic block and instruction level. Basic blocks are sections of code with a single entry and exit point, leading to the basic block always being executed in its entirety. At a smaller scale still, instructions can be parallelised to be executed in the same clock cycle. The main aim of all forms of parallelism is to reduce the overall execution time of an application.

As has already been stated, there are two forms of parallelism; implicit and explicit. Implicit parallelism is defined as:

> *"The sequentially coded source program is translated into parallel object code by a parallelizing compiler. [...] With parallelism being implicit, success relies heavily*

on the "intelligence" of a parallelizing compiler."

– "Advanced Computer Architecture: Parallelism, Scalability, Programmability" [42], Chapter 1.1, Page 18

where the application developer is not required to have any knowledge of parallelism and the application can be run both serially and in parallel, depending on the compiler chosen to generate object code. As stated in [42], this method is highly dependent on the compiler and how much parallelism it can extract from the application code. Alternatively, explicit parallelism is defined as:

"Parallelism is explicitly specified in the user programs. [...] Special software tools are needed to make an environment more friendly to user groups."

– "Advanced Computer Architecture: Parallelism, Scalability, Programmability" [42], Chapter 1.1, Page 19

where the application developer must exploit parallelism within the application themselves. This is achieved using libraries, such as PThread [43], MPI [6] and OpenMP [44], or writing explicit parallel code specifically for the target architecture. However, for a large scale architecture with multiple processing elements the latter would become unmanageable very quickly. The use of libraries aids in managing explicit parallelism, however, there is a draw back of the application code being dependent on a certain library implementation and only compilable for architectures which support the library being used.

It is not possible to parallelise all sections of an application, there are many factors which affect the amount of parallelism able to be extracted. There are sections of applications which cannot be parallelised and must be run serially, including setup and sections which require the convergence of data or instruction streams. An optimal parallel application is defined as one where a linear execution speed up is seen by added extra computational resources. For example, adding X processing resources results in an overall execution time of $1/X$. Amdahl's law [45] specifies the potential speed-up through parallelisation of an application, as shown in equation 2.1.

$$S(N) = \frac{1}{(1-P) + \frac{P}{N}} \tag{2.1}$$

where P is the portion of the program which can be performed in parallel, resulting in $(1-P)$ being the remainder of the application which must be performed serially, and N is the number of computational resources. This results in a maximum possible speed up of $1/(1-P)$.

The limits of hardware are not taken into account in Amdahl's law, the law works using percentages of execution and shows there is an upper bound to the number of extra computational resources which will positively affect the execution time. The idea of memory

conflicts and register bank sizes are not factored into the equation but should be investigated. Memory systems within conventional architectures are banked, this means that they are split into sections to allow multiple memory accesses at any given time. There is a trade off of the number of banks and number of computational resources which access the memory. This is similar to the instruction level, where the number of registers available to perform computations affects the amount of parallelism which can be extracted.

Another factor which affects parallelism in terms of possibility and also speed up is data-dependencies. These dependencies at the basic block and task level relate to sections of an application executing in parallel requiring data which is being computed by other threads of execution and not being available at the time needed. This can lead to race conditions and incorrect execution as a result of old data being used in computations. To address this problem the application must either be created in a manner where the application developer is aware off all parallel execution state and timing to ensure there are no data race issues, or use extra hardware/software to guard memory sections using mutual exclusion primitives or transaction level (lock-free) shared data accesses.

The remainder of this section looks into previous research carried out into parallelism at instruction, basic block and task level.

2.3.1 Instruction Level Parallelism

The most widely understood form of fine-grain parallelism which can be exploited within an application is at the instruction level, referred to as Instruction Level Parallelism (ILP). This involves executing more than one instruction per clock cycle with the goal of shortening the execution time of the application. However, there are tradeoffs to be made as it requires additional hardware resources and a more complex bypass logic to retrieve the resulting computations and make them available for use by proceeding instructions. Also, the amount of ILP which can be extracted from applications is highly dependent on the nature of the program.

An extensive study into ILP was performed in [46]. The author investigated the upper limits the instruction parallelism which can be extracted through various methods, including speculative branch and jump predictions, register window sizes, loop unrolling, register file sizes and register renaming. A selection of systems was investigated from a completely optimal system with a perfect memory system, perfect register renaming, unlimited instruction fetching, and a very large number of functional units to an extremely memory and register limited system. Through simulation of a large number of benchmarks on these systems it was found that even on the optimal system the amount of ILP which could be extracted ranged between four and seven instructions per clock.

This work was extended in [47] where the authors took the results generated previously

to investigate the effects of control flow on parallelism. The authors investigated three alternative methods of control flow analysis; branch prediction speculation, control dependence analysis and following multiple flows of control. The paper concludes that all methods are important in parallelism and the nature of the program being executed plays a key role in the amount of parallelism which can be extracted. Speculating control flow is only beneficial if there are no data-dependencies through the section of the program attempting to be parallelised.

In [48] the authors present the Mutliflow compiler and trace scheduling algorithm. The aim of the scheduling algorithm is to exploit instruction level parallelism within applications compiled for a VLIW processor which can execute 28 instructions per clock cycle. This is achieved through static analysis of the instructions to be executed, using a large set of registers for storage it is possible to calculate data-dependencies at the instruction level. This is a form of implicit parallelism which the application developer is not required to be aware of. However, this method requires a very intelligent set of compilation tools to fully exploit the underlying architecture and produce highly parallel object code.

ILP exploitation within an out-of-order Superscalar (SS) processor is investigated in [49]. This methodology differs from that presented for VLIWs due to the SS performing scheduling solely in hardware rather than depending on an intelligent software compilation process to produce parallel code. Extra hardware is required to determine the relationships between instructions and prevent "hazards", which are defined as sections of code where instructions reference the same storage location. They can occur in three forms: read-after-write (RAW), write-after-read (WAR) and write-after-write (WAW).

The application developer is usually unaware of ILP and it is classified as implicit parallelism. The studies presented show that to exploit ILP there is a lot of computation required by either the compiler, producing the object code, or the hardware itself to prevent incorrect execution and decrease overall execution time.

2.3.2 Basic Block Parallelism

Basic Blocks, as described previously, are sections of a program which contain a single entry and exit point. Basic blocks are used to analyse control flow within a program as their entry and exit points can be used to investigate execution paths of an application.

Methods of parallelism which use this control flow analysis at the basic block level to speed up execution time have been investigated. Which basic block is executed depends on parameters computed within an application and conditional statements decide which execution path should be taken. This is the basis of investigations into a form of parallelism which exploits basic blocks.

In [50] the authors present the Weld architecture model. Based on a multithreaded

VLIW processor, this model aims to reduce run-time latency effects which are introduced through control flow changes during execution. This is achieved through a novel method of inserting custom operations into object code to "spawn" and "squash" threads of execution. This new operation, named "*bork*", which implies a branch and fork operation, is inserted at certain points throughout the program and upon execution a new thread of execution (referred to as the descendant thread) is generated, executing a forthcoming basic block. When the main thread (referred to as the ancestor thread) reaches the section of code where the speculative descendant thread has been spawned it checks to see whether the *bork* operation has performed the correct execution, if correct the two threads are merged and the ancestor continues execution based on the state of the descendant thread. Alternatively, if the descendant thread has been incorrectly speculated then it is disregarded (squashed). The ancestor and descendant threads of execution execute on a single processor with the two instruction streams "welded" together, hence the name. This is performed at run time and attempts to fully utilise all functional units within the VLIW processor. The authors present a selection of results gained from a benchmark suite and report a 27% increase in performance through using the Weld model with two to six threads versus conventional execution.

The Weld architecture is further investigated in [51] where the authors present a 34% increase in performance based on a dual thread VLIW processor. The authors report that in the previous study the inclusion of extra speculative threads led to more complicated synchronisation hardware and an operation "welder". Thus, the focus was on optimising the Weld algorithms for a single speculative thread.

A similar methodology is presented in [52] where the authors present the Single-Program Speculative-Multithread (SPSM) architecture. This methodology also uses extra instructions ("fork" and "suspend") to perform a similar concept to that of the Weld architecture, relying on an intelligent compiler to produce code containing the custom instructions specified. Similar to the ancestor and descendant threads the SPSM architecture defines a main and future thread where the main thread forks data independent code to the future thread and then merges the data back into itself once it reaches a suspend instruction within the object code.

These methods present an implicit form of parallelism where tools generate the speculative threads and the application developer is not required to be aware of this extra functionality, resulting in previously written code not requiring any alterations to take advantage of these provided extensions. In both cases instruction set architecture extensions and custom hardware resources are required to accommodate these methodologies. However, both require no software processes to perform thread synchronisation and thread management.

2.3.3 Thread Level Parallelism

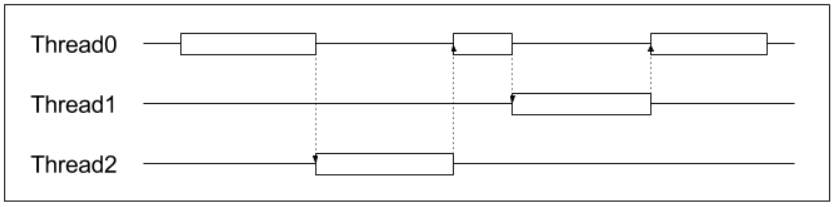
The highest level of application parallelism performed is defined as Thread Level Parallelism (TLP). This methodology requires the use of software libraries and/or unique identifiers of threads to distribute execution over multiple threads of execution. Control or data flow can be distributed across multiple threads using software libraries to create and synchronise threads of execution executing in parallel or using unique identifiers to perform parallel tasks by hardcoding sections of code to be executed on specific computational resources.

These libraries abstract away the complexity of threading by allowing the application developer to create and synchronise threads without an explicit knowledge of how or where the threads execute. This requires extra computation overhead to manage the created threads and is usually performed by an operating system or other dedicated resource. There have been studies which present the implementation of these software libraries in hardware as a means of decreasing this overhead [53, 54]. Both studies cite that the overall advantages of performing certain tasks in hardware rather than software routines results in the ability to perform parallel management and communication at a finer granularity than with software implementations due to lower latencies of operations.

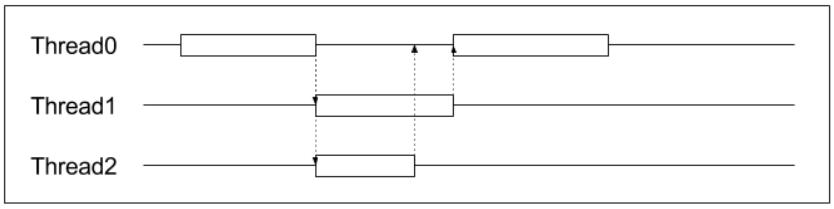
There are two main methods of threading at the task/thread level: simultaneous and temporal (blocked). The difference between the two is the number of threads which can execute at any point in time; in temporal threading only a single thread is able to execute at any time and simultaneous threading refers to when more than one thread executes at any time. Temporal threading aims to fully utilise all computation resources by performing thread switches in the event of a stall produced by an access to memory or control flow change. By switching between threads in this manner it is possible to keep instruction pipelines full. An example of Temporal Threading is shown in Figure 2.1(a), there is only ever one thread active at any time. (Dotted arrows display a context switch between threads.)

Simultaneous threading, on the other hand, is aimed at architectures with the availability of more than one computation resource. An architecture with multiple instances of a processor could use simultaneous threading to fully utilise the architecture. An example of Simultaneous Threading is shown in Figure 2.1(b), where execution on $Thread1$ and $Thread2$ is performed at the same time.

Two example of temporal threading are Interleaved Multithreading (IMT) and Blocked Multithreading (BMT). IMT is a method of switching threads after each cycle or specified time interval, this results in minimal empty slots seen in the pipeline from memory and control flow latencies as other threads of execution mask this overhead [55] [56]. BMT, on the other hand, switches to alternative threads on memory stalls and control flow changes as a method of filling the pipeline where it would otherwise be empty [57, 58].



(a) Temporal Threading - Only a single thread executing at any time, context switches shown with dotted arrows



(b) Simultaneous Threading - More than one thread executing at a given moment in time

Figure 2.1: The difference between Temporal and Simultaneous threading. This simple example shows only a single concurrent thread in the Temporal example and multiple parallel threads in the Simultaneous example.

The difficulties of parallel programming and the limitations of implicit parallelism are discussed in [59]. The author presents the idea that threading is complicated and must be done explicitly at a low level by an application developer who is aware of the underlying architecture being targeted. However, emphasis is placed on the nondeterministic nature of threaded programming. The number of possible outcomes from executing a threaded program is too great for the application developer to fully visualise and with increased number of threads this problem is increased exponentially.

> *"Threads continue to dominate the parallel programming landscape despite the existence of alternatives. Many obstacles prevent these alternatives from taking root, probably the most important being that the very notion of programming, and the core abstractions of computation, are deeply rooted in the sequential paradigm to which most widely used programming languages adhere. Syntactically, threads provide either a minor extension to these languages, as in Java, or just an external library. Semantically, they thoroughly disrupt the languages' essential determinism."*
>
> – "The problem with threads" [59], Page 8

POSIX threads (PThreads) is an IEEE standard which defines a Application Programming Interface (API) for creating and managing threads of execution along with methods of synchronisation between threads [43]. One of the benefits of using the PThreads library is the portability of applications due to numerous platforms supporting the PThreads primitives. This results in applications only requiring recompilation rather than rewriting for each platform.

Another standard for thread parallel programming is the Message Passing Interface (MPI). This is an API which is focused on the communications between processes running in separate threads on distributed-memory architectures. It provides a model for communication between processes running on a distributed memory system [6]. Unlike the PThread library there is no concept of the creation of threads, this results in the possible use of both libraries together when developing a parallel application. For example, the PThread library is used to instantiate multiple threads (across shared-memory and distributed-memory systems) and MPI is used for communication and synchronisation between these active threads.

OpenMP is another widely used threading library consisting of a set of functions and also preprocessor directives for the application developer to use to generate parallel applications for shared-memory machines [44]. The OpenMP model consists of a single master thread which, through the use of the preprocessor directives, spawns slave threads to execute tasks in parallel. Using a barrier mechanism the master thread is blocked until all slave threads complete their specified computations.

The threading libraries and approaches discussed above are compared in [60] which offers an in-depth view into the current implementation of multithreading within applications and parallel programming models.

Research into task level parallelism includes [61] which presents a rule named Kill If Less Than Linear (KILL). The authors investigate the trend of increasing the number of cores within a multicore system and introduce a design approach based on performance and area of computation resources. They suggest "with x% increase in core area there must be a greater than x% increase in performance" as a baseline of whether adding extra cores is a viable approach to increasing performance.

In [62] the authors present the superthreaded architecture, which combines compiler-directed thread-level speculation of control and data-dependences with run-time data dependence verification hardware to exploit both TLP and ILP. This architecture attempts to split the control flow of applications across multiple thread processing elements to diminish the effects of latencies introduced by memory accesses. This is performed at loop level where at the top of a loop body a global index variable is updated and a new thread of execution created. This method is implemented using instruction set extensions to create new threads of execution, perform memory operations and the synchronisation of threads. These extra

instructions are included implicitly by the compiler. It was concluded that the success of this methodology was highly dependent on the type of program being targeted; programs with data-dependencies did not benefit from using the superthreading technique.

[5] presents an investigation into simultaneous multithreading (SMT), utilising the Multiflow trace scheduling compiler [48], and comparing against alternative processor organisations (a wide superscalar, a fine-grain multithreaded processor, and single-chip, multiple-issue multiprocessing architectures). The benefit presented is the ability to fully utilise all function units within a processor through ILP and TLP, where a processor can issue multiple instructions from multiple threads each cycle. The authors conclude that SMT outperforms superscalar and fine-grained multithreading by four and two times, respectively.

The authors of [63] present an in-depth investigation into explicit multithreading. They conclude that the use of threading and multiple processors/cores with a small issue width offers a greater execution speed up than a single processor/core with a wide instruction fetch capability due to the limitations of compilers in scheduling instructions to all processor resources each cycle.

The authors of [64] developed a hardware implementation of the Message Passing Interface (MPI) used in multithreading for communication and synchronisation between active threads. They implemented a set of Remote Memory Access (RMA) primitives which supported the underlying functionality required by MPI. The reason for the hardware implementation was due to the high latency of a software based mechanism of communications. Allowing the main CPU to offload the communication tasks to a hardware processor to not waste valuable CPU cycles in communication activities.

SMT in VLIWs is investigated in [65]. The authors present a method of exploiting SMT at the cluster level of a VLIW. Clusters are a grouping of registers and functional units within the processor, which are used to aid parallelism as the VLIW compiler is able to perform instruction scheduling individually within these multiple groups. The deployment of instructions is based on the cluster level rather than the functional units, which results in register conflicts being more easily resolved. This method removed horizontal waste by filling the instruction pipeline with other available clusters instructions, however as with other methods of VLIW multithreading there are scalability issues relating to the compilers which attempt to produce very wide instruction bundles.

2.4 Very Long Instruction Word Processor

The benefit of a Very Long Instruction Word (VLIW) processor is the exploitation of Instruction Level Parallelism (ILP) through wide instruction packets. Such processors contain multiple instances of functional units to allow the execution of multiple instructions in

parallel. The benefit of a wide instruction is the decrease of execution time as multiple, data-independent instructions can be executed in a single clock cycle. This requires an intelligent compiler to be able to take advantage of the multiple functional units and schedule instructions to fill the pipeline.

> *"The VLIW processor design philosophy is to open up to the program not only the operations, as in RISC, but the ILP itself. Just as there are constituent parts of a CISC operation that are not visible in the program, superscalar ILP hardware can arrange the parallelism in ways not specified in the code. The VLIW design philosophy is to design processors that offer ILP in ways completely visible in the machine-level program and to the compiler.*
> *Examples of the principles of VLIW design are: don't allow the hardware to do things you cannot see when programming; don't waste silicon on said hardware; avoid hardware that computes anything other than the intended computation on the critical path of every instruction; have only clean instructions; and don't count instruction bits."*

– "Embedded computing - a VLIW approach to architecture, compilers, and tools" [8], Chapter 2.2, Page 59

The above quote was taken from [8], this book gives an indepth overview of ILP and VLIWs, going into details of the Instruction Set Architectures (ISA), instruction encoding, micro-architecture and compiler techniques required for VLIWs. The book also introduces the VEX system [66] (VLIW EXample) which is a VLIW compiler and tool chain made available from HP Labs [7]. The book focuses on and uses the VEX system as a reference point throughout its examples.

There are many VLIW compliers available [67, 68, 66, 69] and the use of VLIWs is increasing with the new ARM GPU architectures being based on VLIW architectures [70]. The Trimaran [67] compiler is a full compiler and simulation environment for VLIW architectures which allows the application developer to fully customise and optimise the VLIW architecture and object code being generated. The Stanford University Intermediate Format (SUIF) [68] compiler is a research compiler which explicitly generates parallel object code from serial inputs. Although not specifically a VLIW compiler there have been studies which built upon it to generate code for VLIW processors [71, 72]. The VEX compiler [66] is a research orientated compiler which offers a full compiler and simulator environment and enables the alterations of the base architecture and the inclusion of custom instructions to simulate speed up achieved through performing computation in custom hardware accelerators. LLVM is gaining ground in the compiler space at the moment and has recently included a VLIW backend to its compiler collection [69]. However, at the start of the re-

search presented in this thesis this was unavailable. The ARM Mali-400 [70, 73] processor is an OpenCL compatible device which is based on a VLIW architecture, unfortunately due to it being a commercial product information regarding the compiler and physical architecture is limited.

The utilisation of VLIW processors is measured though the use of horizontal and vertical waste metrics. Horizontal waste is empty issue slots within wide instruction packets, unable to be filled due to data-dependencies and control flow changes and vertical waste is empty instruction packets, or no-op instructions, where no computation is performed as a result of waiting for computation in functional units to complete or data to be read from external memory.

The horizontal waste is filled by the compiler and as previous research has shown is usually limited between four and seven instructions per cycle [46]. Vertical waste is limited using techniques such as basic block and task/thread parallelism where context switching occurs to hide these latencies.

A rich history of ILP exploited by VLIW and Superscalar (SS) processors is archived in [74]. The authors discuss the beginning of ILP in the 1940s and 50s, where the idea of being able to execute more than one instruction at the same time was first discussed, to the conception of VLIW processors in the late 1970s and early 1980s with the availability of more space within silicon chips.

However, as discussed in [46], the extent to which this ILP can be exploited is limited (five instructions per clock, median) depending on the type of application being executed and extra methods of parallelism are required to further increase execution speed up.

This section looks at previous research into VLIW processors and comparisons with other types of architectures.

Superscalar (SS) processors are similar in nature to VLIW processors, they consist of multiple function units and can execute multiple instructions in parallel to achieve reduced execution times. There are two types of SS processors, out-of-order and compile time. Compile time SS processors are similar to VLIW processors and require instructions to be scheduled at compile time. Where as out-of-order SSs perform this scheduling in hardware, resolving data-dependencies between instructions at run time. This results in a simpler compiler required to generate machine code for out-of-order SSs and leads to a more complex hardware design to perform this scheduling at run time.

In [75] the authors present a comparison based on a H.263 video encoder around compiler and application developer optimisations. The one-way SS outperforms an eight wide VLIW (C600) with only compiler optimisations by a ratio of 7.5:1. Using memory and code optimisation techniques for the VLIW the authors are able to increase the performance of the VLIW in such a way to achieve a 14% speed up compared to a 256 wide SS imple-

mentation and a speed up of 61 times that of the VLIW with only compiler optimisations. The memory optimisations were performed manually after statically analysing the H.263 code and included moving frequently accessed memory items into on-chip memory areas to reduce load/store latencies. The code optimisations included writing custom parallel assembly code for computationally expensive functions within the application. This experiment heavily favoured the VLIW implementation and showed a limit to the amount of parallelism gained when solely using compiler optimisations to implicitly exploit ILP within an application. Unfortunately the authors do not provide the overall instructions per clock figures gained from their optimisations, it would be interesting to see the extent to which the customisations performed fully utilised the eight wide pipeline of the VLIW.

The authors of [76] compare the performance between VLIWs and SSs compared to a SIMD architecture. Using a range of benchmarks, including kernels and audio codecs, the authors present a study into the exploitation of parallelism focused around branch prediction and out-of-order execution to fully utilise wide architectures. They conclude that the speed up offered by VLIWs is completely dependent on the application being executed, seeing relative speed ups from 0.63 to 9.0 times over their range of benchmarks; experimenting with perfect branch prediction increases this ten fold. The SS did not see a greater than four times speed up and this was traced to the extra resources required to schedule the instructions at run time.

The TRACE VLIW and Trace Scheduling compiler are presented in [77], a seven wide TRACE processor provides a speed up in the order of five compared with minicomputers available at the time, such as the VAX 8870, when running scientific code. A background of VLIW architectures and designs is described along with the mechanisms of the TRACE Scheduling compiler. This compiler produces instruction parallel code through statistical analysis of source code and performing branch prediction to expose larger basic blocks and implicitly exploit the maximum amount of parallelism within an application.

A complementary approach to TRACE for scheduling code for both VLIW and SS architectures is presented in [78]. The authors present an analytical method of moving operations around loops and conditional statements to extend the reach of ILP which can be exploited. This results in up to 20% run time improvement versus a single issue machine, although this is entirely dependent on the code being executed.

Another approach in exploiting ILP in non-numeric code is introduced in [79]. The authors suggest a logarithmic execution speedup up to five-fold is achievable through their methods, even on realistic devices. This methodology parallelises code across different basic blocks in a single cycle and performs checks at the end of each cycle to confirm which basic block was the correct one to execute, the results of operations from this basic block are then written back and other results disregarded.

The customisable nature of VLIWs, in terms of issue width and the number of function units, results in fine tuned architectures for specific domains. This leads to a large design space to explore in terms of all the different options to choose from when generating suitable VLIW architectures. [80] presents a insight into trade-offs relating to different application domains. Introducing the Lx VLIW processor, the authors conclude that enabling the customisation of an architecture is very effective in term of cost/area trade offs, they present a four to eight times speed up in a selection of benchmarks. They also report that the amount of ILP which can be exploited is very application specific.

The authors of [81] present a homogeneous multicore VLIW system accompanied by hardware co-processors accessible as custom instructions within the VLIW. This configuration gains a speed up of 10 and 30 times in relation to two scalar pipelined architectures, Intel's XScale and StrongARM, respectively. The compilation flow performs profiling to extract loops from application code and perform behavioural synthesis to seamlessly integrate hardware blocks which implement the extracted loops. A set of media benchmarks were targeted and compared using a single issue and four issue VLIW against the scalar processors.

A fully customisable VLIW processor and tool chain is presented in [82]. The mAgic architecture allows the application developer to specify various parameters of the VLIW processor, including the issue width, number of functional units and size and types of memory available to the processor. A single machine description is used both to customise the hardware and generate a custom compiler to generate object code to execute on the hardware.

A hardware multithreading technique using VLIW processors is presented in [83], the authors report a speed up of between 4.5% and 27%, comparing a single threaded VLIW processor running legacy code with a multithreaded VLIW. This is performed using an interleaved method of threading, where the processor begins executing another thread of execution on the event of an instruction with a long latency. When executing a hand optimised MPEG-2 decoder a performance increase of only 4.5% was recorded compared with 27% when using an unoptimised MPEG-2 decoder. The authors conclude that while the hand optimised code performed better it is not always feasible to optimise all applications, in this case hardware multithreading can be used to reduce overall execution time.

2.5 Summary

This chapter presented previously conducted research and development in the areas of high level modelling, various methods of parallelism and VLIW processors.

The overview of Visual Programming Languages showed that they aid design of appli-

cations in terms of development time, showing a factor of four improvement over direct code approaches. The main drawback being that there are many ways to visualise what is essentially the same concept. UML was used as a basis of the investigation due to its current dominance in the field of VPL; UML allows a high level of abstraction from architecture specific design. However, to be able to implement a design either intelligent tools or experts are required to refine UML models to an implementable level. One of the key aspects discovered through this background research was the various different ways researchers have approached the concept of capturing applications and architectures through UML and defining how they relate to one another.

Three main methods of parallelism were covered, including instruction, basic block and task/thread level. ILP is achieved through finding data independent instructions which can be executed simultaneously to reduce the execution time of applications. This method shows varying efficiency based on the type of application being targeted, numerical applications which work on large data sets in loops can be parallelised efficiently whereas control flow based applications are more difficult. Basic block parallelism is mainly exploited through branch prediction to reduce the amount of time a processor spends refilling its pipeline. Examples of both temporal and simultaneous threading have been investigated and achieve high speed up, although as with ILP the type of application affects the amount of parallelism achievable. Finally, TLP was investigated, this is mainly implemented through the use of libraries which perform the management of threads to implement parallel tasks. These methods often incur an overhead of requiring operating system or another executive to perform thread and communication management.

VLIW processors and compilers implicitly exploit some parallel aspects of applications through static analysis of code and scheduling. VLIW compilers are able to implicitly extract fine-grained parallelism at the instruction level to decrease execution time of applications. There seems to be a limit to the amount of ILP which can be exploited, many studies converge on the figure of between four and seven being the maximum ILP exploited even when simulating on perfect architectures. A notable trend in studies which report higher amounts of ILP being achievable is that scientific code, which is based mainly on loop intensive calculations, is used for testing theories and methodologies. In real world applications this is not the case and applications are more control flow based, these types of applications limit the idea of ILP due to small basic blocks resulting in fewer areas of the code to parallelise. The result of using non-scientific code is that alternative levels of parallelism need to be exploited alongside ILP which leads onto the research and development to be carried out in this thesis.

The following chapter reviews these previous studies and uses them to identify an area of research which will be investigated as a basis of this thesis.

3

Identification of Research Area

3.1 Chapter Objectives

This chapter analyses work and research previously conducted, reviewed in chapter 2, to find a specialisation area to focus the main research and development activities in this thesis.

3.2 Parallelism

Instruction Level Parallelism (ILP) has been extensively researched and documented over the previous 40 years, from its first conception in the late 1970s and early 1980s to modern day wide architectures. ILP has been found to be limited to between four and seven instructions per clock [46, 59] even when ignoring the limitations of physical systems. There has also been research into many forms of Thread Level Parallelism (TLP) investigating how context switching and control flow speculation [50, 51, 52] can eliminate execution stalling due to memory accesses and pipeline refills. Along side this, multiple threading libraries have been developed to give the application developer full visibility and access to multiple execution threads with which to perform parallel tasks [43, 6, 44].

Context switching occurs in two forms: interleaved and blocked. Interleaved is wherein each clock cycle, or given time frame, an alternate thread is executed, and blocked is where if the currently active thread is stalled a waiting thread is executed. Control flow speculation, used to mask stalls incurred through memory access and pipeline refill latencies, requires extensions to hardware systems and software tool chains to include and execute look ahead instructions to create and destroy speculative threads where control flow is dependent on calculations made at runtime. Threading libraries impose an overhead of extra hardware or software processes to manage executing threads.

A multicore Very Long Instruction Word (VLIW) processor targets both of these forms of parallelism, ILP in the scheduling of instructions and TLP in the availability of multiple cores to perform execution. The statically customisable LE1 VLIW Chip Multiprocessor (CMP) [84], which is described in section 4.3.1, has been chosen as the basis of investigations due to this. The processor is currently in the early development stage with only a basic Instruction Set Architecture (ISA) which enables modifications and extensions to be introduced during this development process. Also, a software tool chain is yet to be produced resulting in the ability to cherry pick implementation details from previous research. This results in the development of a tool collection and hardware processor aimed at fully exploiting multiple forms of parallelism with focus on low-cost hardware thread management.

From investigations into VLIW compilers the VEX compiler [66] has been chosen due to the ability to compile complex code and easily modify the architecture which is being targeted through the use of a plain text machine description file. Another plus is that the VEX compiler is still under active development by HP Labs whereas other compilers which were investigated are no longer maintained. VEX is aimed towards a research audience in ILP and introduces a simple method of including custom instructions to examine the theoretical speed up of custom hardware blocks to perform computationally expensive sections of an application. Due to the vast amount of research into ILP it was decided not to extensively investigate alternative methods in parallelisation at this level. The use of VEX allows the exploitation of ILP without it being the main focus of the research.

The TLP which is to be focused on during the course of this research will be based around a subset of the PThread [43] library as well as a Single Process Multiple Data (SPMD) style threading mechanism. The PThread library allows the application developer to fully control the creation, deletion and synchronisation of threads to decrease execution time and thus, speed up execution. However, this requires a thread management process, which uses computational resources, to allow threads to be created and synchronised. The VEX complier does not include any notion of TLP, which allows full control of threads to be implemented and managed through the LE1 processor and tool chain. As a method of reducing the overhead of the thread management the inclusion of a hardware PThread management unit which is accessible through custom instructions will be investigated and implemented.

The SPMD method can be achieved through the inclusion of a single custom operation to return a unique identifier to differentiate between executing threads. The management of these threads is solely performed in the application created by the application developer which leads to a simple threading mechanism with little to no overhead.

These two methods provide the building blocks for the remainder of the proposed research and allow parallelisation of applications using two alternative threading mechanisms.

3.3 Visual Programming

Using a visual programming language (VLP) adds a layer of abstraction from the application developer which enables focus during implementation to remain on the application rather than physical architecture being targeted. The current VLP standard in both research and industry for modelling systems is the Unified Modelling Language (UML) and this will be investigated as a means of application creation within this thesis.

Previous research using UML for system modelling involves splitting the UML models into three distinct sections: Application, Architecture and Mapping. One of the key drawbacks to this concept found in literature is the requirement to directly modify the UML model or generate at design time the possible mapping options of how an application should be implemented on a specific architecture. In [31] the application developer is required to manually alter the mappings, and in [32] the use of state machines to loop through possible mappings is used. Both of these methods are specific to an architecture available at the time the UML model is created and although the model is not entirely a platform-specific UML (PSUML) model the level of abstraction introduced from the use of UML is slightly lost. Both methods require human interaction to include these mapping options, this human interaction results in the inability to use UML models as inputs for automated processes which could explore and optimise designs for a given metric, for example speed or area.

A concept of generic architectures and mappings will be investigated in order to create a fully platform-independent UML (PIUML) model with the aim of a single design being valid for multiple architecture configurations as well as architectures unavailable at the time of creation. This will aid in the reuse of UML models for various architectures, different methods of this will be investigated including:

A. Solely generating the application section of a UML model and having the architecture and mapping automatically generated and included. This should result in a fully PIUML model which can be retargeted without the requirement of designing an architecture.

B. Allowing the application developer to specify an architecture and mapping in which to explore around.

C. Design notations to allow parts of an application to be tagged as able to be parallelised across multiple processors within the architecture.

Depending on the size of the application and architecture the first method could result in a very large design space to explore around so a fully automated flow for generating and modifying PIUML models would be ideal. The second method would reduce this design space and focus around a certain user guided architecture/mapping of a system.

3.4 Application Flow

Bringing both of the proposed research areas together results in using UML to capture applications designed for multicore systems. With the inclusion of the customisable VLIW this leads to a modifiable base architecture which would currently require the direct modification of a UML model in order to target different instantiations of the architecture. Using the VLIW tool chain along with the concept of generic architectures and mappings, the idea is to take a PIUML model and generate a PSUML model for a specified multicore architecture.

This also leads to the capture of alternative TLP models within UML and their interactions with the underlying architecture. By creating concepts which can be captured in UML to specify that certain components and data flows can be executed in parallel it is possible to use a single UML system design to target and fully utilise a modifiable hardware architecture. This is achieved by taking sections of the application which have been specified as parallel and including multiple instantiations of these objects based on the number of computational elements available within the architecture. This is performed through modifications to both the behaviour and structure of the application as well as the structure of the architecture.

There are various UML modelling tools available, both commercial and free to use. All vary slightly in their implementations of Graphical User Interface, underlying UML Meta-Model and XMI generation. XMI is used as a container for data exported from the UML tools to be used by other tools. Due to these problems there is a lack of fully implemented applications captured in UML models which can be used for benchmark purposes.

A behavioural synthesis tools, FalconML [85], will be used to translate XMI containing UML models to C code which is required by the VLIW tool chain. As the research interest is in methods of capturing parallelism within UML models the use of an external tool to provide this behavioural synthesis functionality allows the focus to remain solely on the concept of modelling.

Due to the lack of benchmark UML models available, various applications are required to be developed to test the suggested research ideas. Both small kernels and full applications will be created and tested to present a valid research output.

In order to perform analysis on the benefit of designing UML models for a generic architecture and having an intelligent program implement the final design, a method of quantisation in terms of work required by the application developer is required. This will be based around the amount of user interaction required to generate and modify a UML model from one architecture to another, this metric can then be used along with comparisons of the execution times from both user performed and automatically generated methods to provide insight into the effectiveness of the proposed design methods.

3.5 Summary

This chapter presents the areas of research which will be the focus of this thesis. Initially, work will be carried out to provide a full tool flow for the statically customisable LE1 VLIW CMP, followed by research into various forms of modelling parallelism in UML. Finally, application development will be performed to provide a comparison point of all the work conducted around this thesis. The key areas of work are:

- Investigation into hardware threads in the LE1 VLIW CMP.
- The capture of Platform-Independent UML models.
- Fully modifiable mappings for application mapping to statically customisable multicore platforms in UML.
- Intelligently threading applications in UML.
- Development of UML benchmarks.

Initially the conception and creation of the LE1 tool chain and threading mechanisms will be investigated, this will result in an architectural specification which can then be used by UML models as a base architecture to investigate the generic mapping and architecture modelling.

4

Experimental Framework

4.1 Chapter Outline

This chapter presents the theoretical and practical aspects of the experimental framework which has been implemented in order to conduct research into parallelism and modelling within this thesis. It is split into two main sections, firstly the theory behind the implementation is presented and then the physical implementation details are described.

Full details regarding the novel modelling rules which have been created, along with their usage and implementation are also presented.

The underlying architecture which is being targeted is introduced, including details of the Very Long Instruction Word (VLIW) processor and System-on-Chip (SoC) generated as a base platform to execute applications. This is followed by a detailed description of the software tool chain developed to generate machine code for executing on the heterogeneous processors within the architecture.

The architecture and software compilation are required to be synchronised so applications are compiled for the correct underlying architecture. The method of keeping these two sections in sync using a single, extensible machine model is then described.

4.2 Theory

This section introduces the theoretical aspects of parallelism and high level modelling constructs which are of importance for the design flow from UML to a multicore system. Firstly, parallel aspects of the multicore VLIW are presented and discussed, followed by the novel modelling semantics required to capture this multicore architecture in UML and methodologies to create applications to utilise this modifiable architecture from a single design.

4.2.1 Very Long Instruction Word Parallelism

Instruction Level Paralleism (ILP) is a form of parallelism which is intrinsic within VLIW processors where wide instruction packets execute in parallel to decrease execution time. There is a large amount of research in which ILP has been investigated with the conclusions converging on the figure of between four and seven instructions per clock being the saturation point of ILP [46, 59]. Due to this previous work the research presented within this thesis does not investigate improving ILP and instead looks for alternate methods of parallelism with which to complement ILP.

The main area of interest is in Thread Level Parallelism (TLP) within the statically customisable VLIW multicore system to reduce thread management latencies and exploit the maximum amount of parallelism available within an application. This is achieved through the use of an explicit threading library with the thread management being controlled through custom hardware in order to not tie the processor up with thread management and leaving it to execute only the application code.

The LE1 VLIW Chip Multi Processor (CMP) [84] is being used as the target architecture, as it was made available and was still under development at the beginning of the research. Only a basic Instruction Set Architecture (ISA) was implemented and there was no software tool chain to target the processor. This allowed full control during the development of the software tool chain to customise the ISA for the needs and requirements of the hardware threading management.

4.2.1.1 Instruction Level Parallelism

ILP is the primary form of parallelism exploited by VLIW processors. An intelligent compiler performs instruction scheduling and register allocation to allow multiple instructions to be executed in parallel, resulting in an overall reduced execution time. The tool chain developed in this thesis uses the VEX research compiler [66]. This compiler implements implicit ILP based on the machine model of a target VLIW.

4.2.1.2 Task/Thread Level Parallelism

Task and thread level parallelism is harnessed through the availability of multiple cores within the VLIW architecture being used as a base of this research. TLP is explored two alternative ways:
A) Using the PThread library.
B) An SPMD approach.

The PThread library implements an explicit form of threading, requiring the application developer to specify sections of an application which can be performed in parallel. Current implementations of the PThread library rely on software processes to maintain the state of active threads within a system. These software implementations require an operating system or other form of task scheduler to perform this management. As this is implemented in software it requires processing power from the executing processor to manage the active threads. The proposed PThread approach performs thread management using hardware state tables. This removes the requirement of a task scheduler to be implemented in software. This will be achieved using hardware tables which are accessible through software calls and custom instructions from code executing on the processor. These hardware tables maintain the state of all active threads and custom hardware performs the task of thread management. This hardware implementation is proposed to reduce the latencies of thread management as well as removing the requirement of a software process executing on the processor to perform this task.

The SPMD approach, referred to throughout as the CPUID approach, performs parallel execution based on a unique identifier of each thread. Using this unique identifier the task performed by each active thread can be specified within the application, resulting in different threads performing different computation. This CPUID approach requires the target processor to include a software call to return a unique value based on the thread requesting the identifier.

On the LE1 system in both PThread and CPUID approaches a physical core can execute a single thread at any time. This results in a maximum number of parallel threads equal to the number of cores within the physical architecture, however, once a thread has completed execution the core can be re-used by a new thread of execution. Both approaches require different execution environments, for PThread a single core is required to be active at the start of execution, this then creates new threads of execution to perform tasks. This differs from the CPUID approach which requires all available cores to be active at run time, each core begins execution from the entry point of an application and then performs computation dependent on its unique identifier.

Due to the ISA of the LE1 being under development extensions were included to incorporate both PThread and CPUID instructions along with hardware which service these custom instructions. The aim of including a hardware PThread implementation is to reduce the requirement of software running on the VLIW itself to perform thread management tasks leaving the VLIW to only execute the application code produced by the application developer, hence reducing the amount of work performed, leading to a reducing of execution time.

4.2.1.3 Statically Customisable VLIW Processor

Larger FPGA devices enable multiple cores and functional units to be included in designs. With more and more context, hypercontexts and functional units the customisation of the LE1 processor became unmanageable and as a result a method of easily customising the target VLIW processor was required. A machine description file, readable and modifiable by both human and computer, was created to allow modification to the underlying architecture. Using a standardised markup language which allows a hierarchical structure it was possible to capture multiple instances of cores within a processor along with alternative architectural templates for each core. XML was therefore chosen as a medium for the LE1 configuration file. The availability of XML parsers and writers in multiple programming languages enables use by multiple programs/scripts and the XML structure is easily readable and modifiable by hand.

The XML machine description file is used as a synchronisation point within both the hardware and software sections of the LE1 tool chain to eliminate software and hardware compilation inconsistencies. Due to VLIW processors requiring implicitly scheduled machine code, the machine code produced only executes correctly when used with the hardware architecture specified at compile time.

4.2.2 Modelling

This section describes the high level modelling of both application and architecture using UML. It introduces both the existing methods of capturing systems within UML along with the novel techniques and approaches which are one of the key contributions of the research presented.

The methodologies presented use previous work into modelling for configurable architectures [31, 29, 34] and builds upon them to create a fully dynamic modelling procedure to fully exploit all available system parallelism. From the previous research, cited above, the aspect of three separate sections within a system is used. These sections are: Application, Architecture and Mapping. The Application section includes the structural and behavioural aspects of a system; the Architecture details the structure of the underlying hardware being targeted and the Mapping links the application and architecture together to specify how the application is implemented on the architecture. The MARTE profile is used to define the target architecture of the UML model.

An example system captured in UML using these three sections and displayed in a Composite Structure Diagram is shown in Figure 4.1. It shows a top level class (*App*) containing three objects (*i0*, *i1* and *i2*) which are instantiations of classes defined in the application section of the UML model (not shown). This is the Application section of the

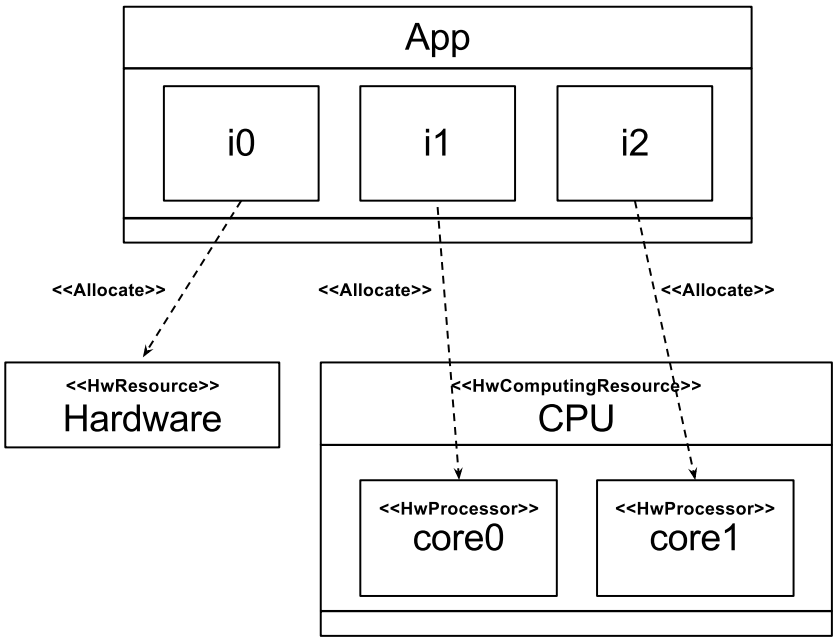


Figure 4.1: Full System captured in UML

system. The Architecture is captured in a similar way, using classes and objects along with stereotypes from the MARTE profile to represent hardware blocks (*HwResource*) and a CPU (*HwComputingResource*) which is composed of two processor cores (*HwProcessor*). The Mapping then links the Application and Architecture with MARTE *Allocate* dependencies to specify how objects in the application section are mapped on the architecture. The example shows object *i0* mapped to the *HwResource*; this is used to define a hardware implementation of this object, possible via the FalconML behavioural synthesis tool [85], whereas and *i1* and *i2* are mapped to separate cores within the CPU.

This results in the behaviour of object *i0* being implemented in VHDL by FalconML which can then be synthesised as a hardware block on an FPGA device. Objects *i1* and *i2* are implemented in C code by FalconML, this C code is then compiled for the LE1 VLIW CMP with each object being executed on separate cores. FalconML also generates the wrapper code for all objects to perform communications between the hardware and cores within the LE1.

While this method of system capture in UML is not novel itself there is definite novelty when capturing a system to be mapped on an statically customisable architecture, like the LE1 VLIW CMP. In order for a software system to map correctly on a given architectural instance the UML model must be altered for each variance; this leads to the requirement

of the mappings changing to accommodate this customisable architecture. For example a system generated to use three LE1 cores is only valid for instances of the architecture which have three or more cores instantiated; any fewer and the system defined in the UML model is incorrect; any more and cores are wasted with no computation being mapped to them.

A final issue occurs when capturing parallelisable applications when the target architecture is either unknown or able to be modified to offer more or fewer cores for parallel sections of the application to execute on. The existing methodologies require a statically defined UML model to translate to low level abstraction to implement as either hardware (RTL VHDL) or software (C or C++). This static nature results in limitations for the application developer who is tasked with refactoring the UML model whenever the application or architecture changes. The novel approach, presented here, removes this limitation through the use of wildcard notations and adjustable mappings to allow the translation of models to be performed through an automated process.

The full system is captured through the use of a UML modelling tool and then exported in XMI. This is an XML-based description language in which the behaviour and structure of the UML model is represented [16]. This XMI structure contains all of the implementation details of the system but does not include any notion of the diagrams which were created in the UML modelling tool. This XMI structure is used for the methods defined here; it is read, modified and then rewritten, finally it can be read back into a UML modelling tool or other tool which uses the XMI schema.

The exported XMI structure is used by FalconML to generate hardware and software aspects of the system along with the communication layer between processors and hardware. FalconML allows the mapping of objects to hardware resources and generates hardware descriptions (RTL VHDL) of the objects mapped to these sections. This thesis does not use this aspect of the tool, although the design principles introduced are transferable to these hardware blocks.

In order to perform alterations to the UML model to be usable by other tools the exported XMI structure is directly modified. By pre-processing the UML model in this XMI structure and utilising the modelling rules, introduced below, it is possible to transform a platform-independent UML (PIUML) model to a statically-mapped platform-specific UML (PSUML) model which can be then synthesised to a hardware/software co-design system by the behavioural synthesis tool. In this flow the application developer creates the architecture and application along with a single mapping of the application to that architecture which is then processed to automatically generate a selection of valid, statically-mapped PSUML models. The main concepts of interest here are:

A) the methods of defining a generic model using wildcard multiplicities.

B) adjustable mapping to guide the pre-processing tools to generate a valid UML model.

These concepts, without the pre-processing stage as part of the tool flow to translate from the PIUML would result in undefined behaviour in the tools performing behavioural synthesis as they require PSUML models in order to generate hardware and software implementations of the UML model ready for synthesis or compilation onto physical architectures.

4.2.2.1 Architecture

The existing approach for UML modelling for a configurable architecture is to go back to the UML modelling tool, redefine the available architecture and then export the XMI for this new model. This method requires human interaction in order to make any modifications to the architecture section of the UML model. When creating models targeting the configurable VLIW CMP this results in multiple instances of the UML model to capture all possible implementations of the underlying hardware.

To allow for the architecture within a system to be defined as modifiable/configurable the use of wildcard multiplicities (*) is proposed. Architectures are captured in UML similarly to applications, using class and object elements from the UML specification. These UML elements are then stereotyped using the MARTE profile to tag them as architectural elements. In this case we use the *HwProcessor*, *HwComputingResource* and *HwResource* stereotypes to define available hardware elements. Multiplicities are used to define the number of instances of UML elements, the wildcard multiplicity defined here allows the *HwProcessor*s to be defined as "expandable", where more than a single *HwProcessor* may be available. Similarly with the adjustable mapping, a transformation process will be required to convert the UML model utilising the wildcard multiplicities to a static model which includes the correct number of *HwProcessor*s for the current architecture being targeted.

For example, Figure 4.2 shows an architecture section of a system using wildcard multiplicities. A single CPU is defined containing three *HwProcessor*s; without using the wildcard multiplicity this defines a three core CPU but with the use of the wildcard multiplicity, as shown in *core2*, it defines a three or more core CPU. This results in the PIUML model, containing the wildcard multiplicity, being reusable across multiple architectural instances. This idea is the second major contribution of this research in the space of high level modelling for a customisable underlying architecture and is complemented by a similar notion introduced in the following section.

4.2.2.2 Application

As presented in the previous section, when developing a UML model for a configurable architecture, such as the LE1, the application developer is required to modify the architec-

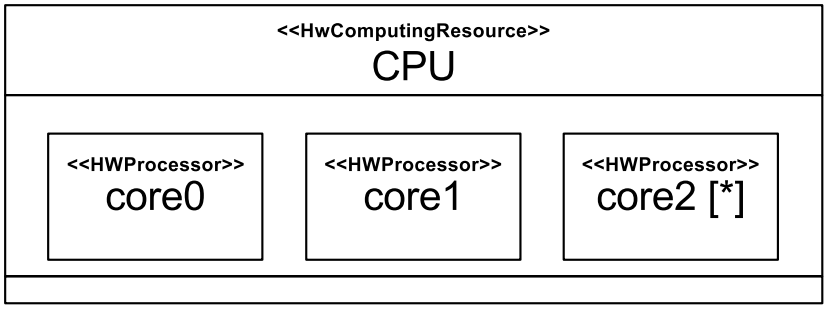


Figure 4.2: Example architecture captured using MARTE and wildcard multiplicity notation

tural elements within the model to match that of the architecture being targeted. These modifications would also require modifications to the application section of the UML model. Removing or adding architectural elements results in either application elements (objects) being mapped to cores which no longer exist in the architecture or cores which do not have any application elements mapped to them at all.

This limitation is addressed by utilising the wildcard multiplicity and transformation process, similarly to the architecture modelling, to define application elements which can be instantiated multiple times and which are able to perform computation in parallel. Within the application section this design notation and transformation process is used to replicate application elements to fully utilise the current architecture.

A complex transformation process is required by the application section in order to replicate application elements while still achieving a functionally correct UML model. This is due to the nature of UML design and how communications between elements is achieved. The surrounding structure and behaviour of the UML model needs to be aware of the replicated application elements to communicate with them. Methods for performing this transformation are presented within this section.

UML is an object-oriented design language and as such uses class's and objects to define the behaviour and structure of applications. A class is constructed of state and methods which define the behaviour of an application. Classes are then instantiated as objects which implement the classes behaviour, these objects are then connected to one another to define the structure of an application.

Classes are composed of attributes, operations, state machines and ports. Attributes are local variables available within a class used to store state (data). In UML the behaviour of a class can be defined both graphically, using state machines, and textually, using operations. Ports are used to present the public functionality of a class and a means to pass data into and out of classes.

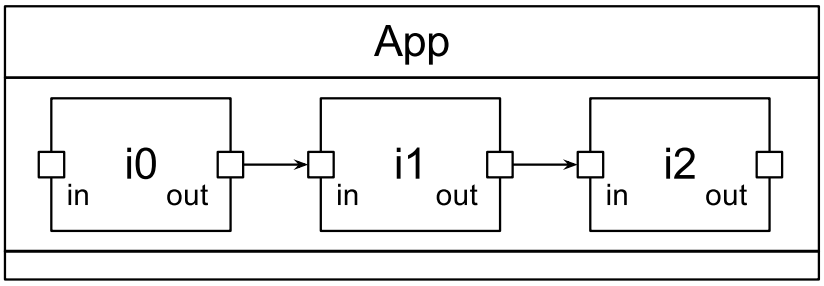


Figure 4.3: Example application captured in UML

An interface is a UML element which simply defines a set of public operations. Interfaces are realised by classes in order to define their public functionality to the application. Within a class a set of ports, which also implement the interface realised by the class, are used as the communication points within the application.

When a class is instantiated as an object all internal state, behaviour and ports of the class are replicated and used to define the object. The ports within objects are connected to ports on other objects and define the communication structure of an application. Figure 4.3 shows an example application section where each object has two ports, *in* and *out* with the output port of each object connected to the input port of the next object. This allows data to flow from left to right across the application, resulting in object *i0* being able to call operations within *i1* made available through the interface which is realised by the base class of *i1*.

Using the same wildcard multiplicity notation idea as in the architecture section, it is possible to define objects within an application which can be executed in parallel. The use of a wildcard multiplicity in an application section is shown in Figure 4.4, the only difference between this and Figure 4.3 is that object *i1* has been tagged with a wildcard multiplicity (*), meaning this object can be replicated multiple times.

The use of the wildcard multiplicity to specify objects which can be replicated results in a requirement of the surrounding objects and ports to be aware of these replicated objects. This results in complicated decisions regarding the surrounding structure of the application. Two different approaches to this decision are possible where one modifies the structure of the application and the other modifies both structure and behaviour. Each approach replicates objects within the model based on the number of cores available in the architecture, this replication of objects is performed at compile time and produces static mappings of objects to cores. Both methods are described below:

A) The first method requires extra classes and objects to be created within the UML model.

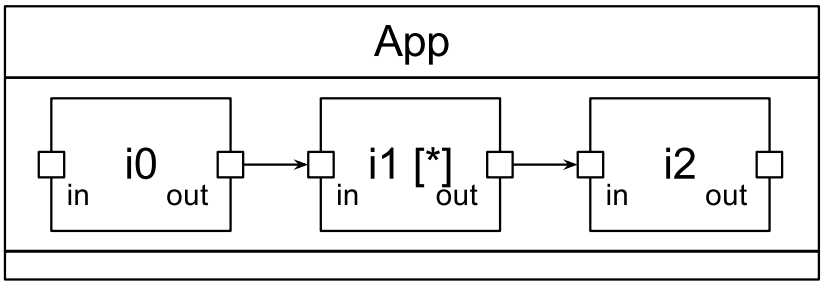


Figure 4.4: Example modifiable application captured in UML using wildcard multiplicity

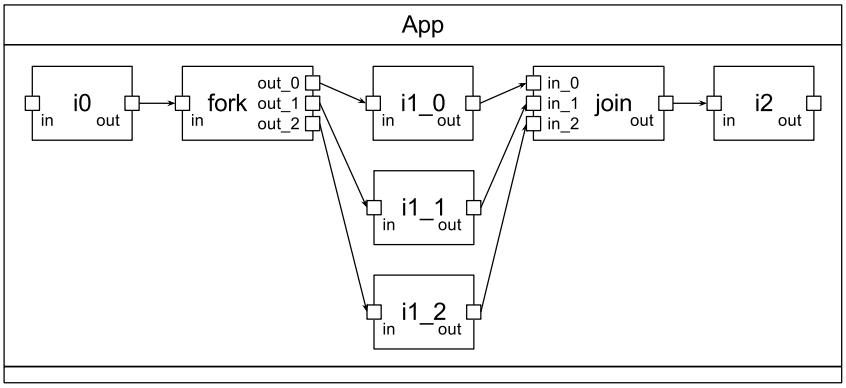


Figure 4.5: Example application extension using fork and join objects

These extra classes surround the object being replicated to provide "fork" and "join" points without the need to directly modify the surrounding objects. These new classes provide synchronisation points for the replicated objects and result in the surrounding application being unaffected by any modifications. Figure 4.5 shows an example of this method using Figure 4.4 as the input UML model and replicating object *i1* three times.

The creation of new classes and objects is shown Figure 4.5 in which objects *fork* and *join* have been created along side three instantiations of object *i1*. The *fork* object implements its own internal behaviour which passes calls destined for *i1* to each instantiation in turn (in a round robin method). For example, the first time an operation within *i1* is called from *i0* this is passed to *fork*, which then passes this to object *i1_0*, the second time this will be directed to *i1_1* and so on. The *join* object performs synchronisation and simply forwards the outputs from each *i1* instantiation to object *i2*. This methodology allows operations within *i1* to be performed in parallel as each instantiation implements the same internal behaviour. The *fork* and *join* objects connect in-between the existing objects in the

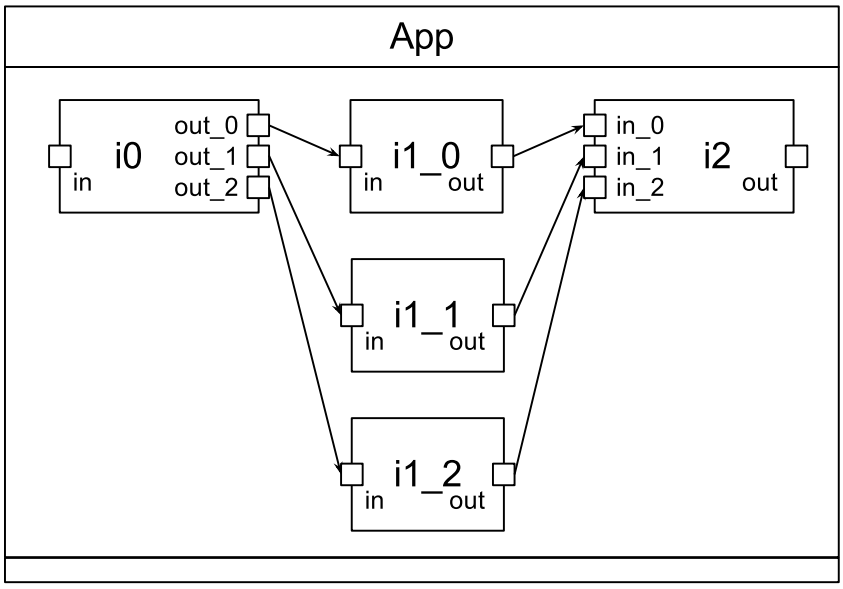


Figure 4.6: Example application extension by modifying surrounding structure

UML model and behave as split and synchronisation points. There are some limitations to this methodology, the operations need to be non-blocking (they do not have return values), otherwise this removes the parallelism which could be exploited due to only a single object being active at any time. Also, if the operation(s) being performed in the replicated objects are not computationally expensive this can result in a negative performance in including these objects due to them not being fully utilised for computation.
B) The second method requires the modification of both structure and behaviour in the UML model. The objects connected directly to the object tagged with the wildcard multiplicity are modified instead of including the *fork* and *join* objects. The structural modifications include instantiating multiple ports within these objects in order to connect to the replicated objects. The behaviour within the surrounding objects is then required to be modified to be aware of these newly instantiated ports.

Once again using Figure 4.4 and replicating object *i1* three times Figure 4.6 shows the resulting UML model generated from using this second method.

The method is shown in Figure 4.6 where *i0* and *i2* have newly instantiated ports connected to the replicated *i1* objects. This method requires the behaviour within *i0* and *i2* to be modified to be aware of these new ports. For example, a call to an operation in *i1* from the *out* port of *i0* (as seen on the right hand side of *i0* in Figure 4.4) will pass arguments across the connection between *i0* and *i1*. In the application generated by the

```
/* Before */
...
        function(out, /* begin */ 0, /* end */ 12);
...

/* After */
...
        function(out_0, /* begin */ 0, /* end */ 4);
        function(out_1, /* begin */ 4, /* end */ 8);
        function(out_2, /* begin */ 8, /* end */ 12);
...
}
```

Figure 4.7: Calls to an operation named *function* where the first argument is the port which is used along with a list of arguments to pass. After modification the initial call is replaced by 3 calls to the same operation across all newly generated ports with the workload split across each call.

application developer this will reference port *out*, however, this port is no longer available in the modified application as shown in Figure 4.6. The port has been replaced with new instances (*out_0*, *out_1* and *out_2*) this then requires any behaviour which references the *out* port to be modified to pass data to the newly instantiated ports. An example of this modification is shown in Figure 4.7 where a call to *function* using the *out* port is replicated three times to reference the new ports.

The method targeted in the transformation process instantiates multiple instances of objects is the one which generates the *fork* and *join* objects around the wildcard object. The alternate method requires direct modification of the surrounding classes and objects which alters their behaviour and structure (action code and ports). This would require a parser to modify the action code used to pass data over the newly available ports and include a mechanism with which to split the data across the newly created ports.

Using the method which generates *fork* and *join* classes allows all other objects around that tagged with a wildcard multiplicity to remain unmodified. By limiting the modifications to a single object, rather than modifying all surrounding objects to accommodate these changes, the possibility of introducing errors into the UML model is reduced. The *fork* and *join* objects will be instantiated from classes generated specifically for the purpose required.

Using this method there are two alternative threading mechanisms which need to be implemented:

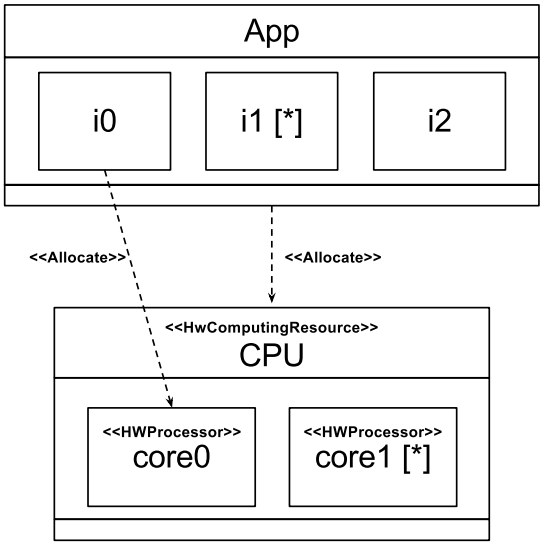


Figure 4.8: Example usage of Design Rules showing wildcard notations along with static and dynamic mapping.

Firstly a simple round-robin mechanism in which the *fork* object passes the calls it receives to subsequent port and instantiations of the wildcard objects and the *join* object simply forwards any data it gets from any input port to the single output port. Using Figure 4.5 as an example the *fork* object would pass calls to *i1_0*, *i1_1*, *i1_2*, *i1_0*, *il_1*, et cetera. and the *join* object would take calls from *in_0*, *in_1* and *in_2* and pass data directly to its *out* port. This is similar to the PThread library [43] where each call creates a new thread of execution to perform computation.

The alternative mechanism is to split a single computation across available objects; this would require the *fork* object to pass data items to the wildcard objects where the amount of data is dependent on the number of other objects performing computation. The *join* object would then be required to process the output data and only pass this onto the next object once all computation has completed. This is similar to the threading techniques seen in the OpenMP library [44], where a master thread spawns multiple slave threads and uses a barrier mechanism to wait for all slave threads to complete execution before continuing.

4.2.2.3 Mapping

Firstly, the concept of Mapping is investigated. Mapping is used to define where objects in an application are to be deployed within an architecture and enables parallel threads to be mapped to parallel computational resources.

Table 4.1: Mapping type definitions: CPU refers to a *HwComputingResource* and Core refers to a *HwProcessor*

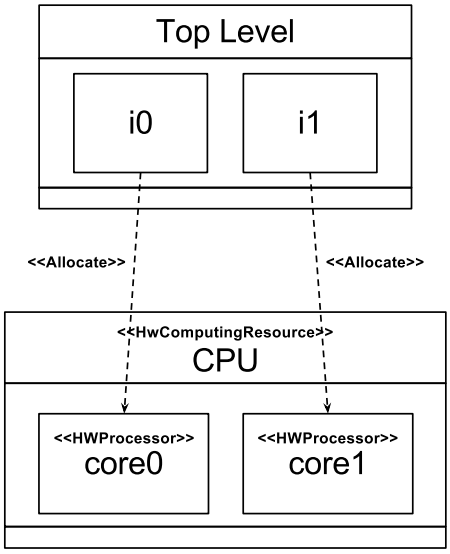
Application	**Architecture**	**Mapping**	**Details**
Application	CPU	Adjustable	Any un-mapped Objects will be mapped to any free Cores in CPU
Application	Core	Static/Adjustable	Any un-mapped Objects will be mapped to specific Core (depending on the number of mappings from the Top Level)
Object	CPU	Adjustable	Object is mapped to any free Core in CPU
Object	Core	Static	Object is mapped to specific Core

The main concept revolves around two types of mapping; **static** - a form of required mapping where a thread must execute on a specified resource, and **adjustable** - a form of constrained mapping where a thread can execute on one of (possibly) many specified resources. A full definition of static and adjustable mappings is shown in Table 4.1 along with examples and descriptions of their usage in Figure 4.9.

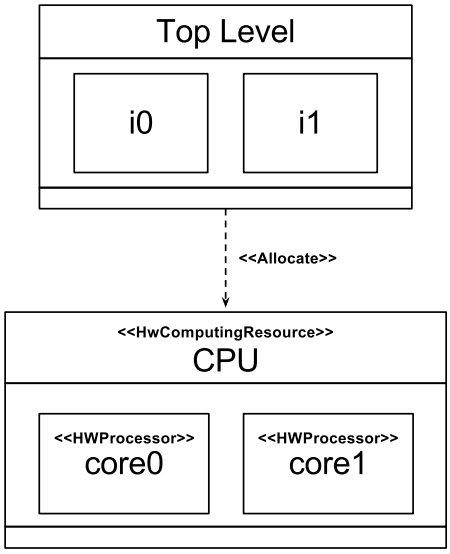
Currently, FalconML requires *static* mapping to process and correctly implement an application across available architectural elements, for example CPU cores or a custom hardware block (RTL VHDL). The application developer must specify at design time the relationship between the application and architecture. Through the inclusion of *adjustable* mappings a model can be modified to be architecturally independent as it no longer includes just required mappings but also includes constrained mapping which can be used to focus the final mappings once an architecture is known.

Figure 4.9(a) defines a fully static model in which both objects (*i0* and *i1*) map directly to *HwProcessor*s. This results in a single mapping option available for this configuration, the underlying architecture being targeted must have at least two cores to be able to execute this application.

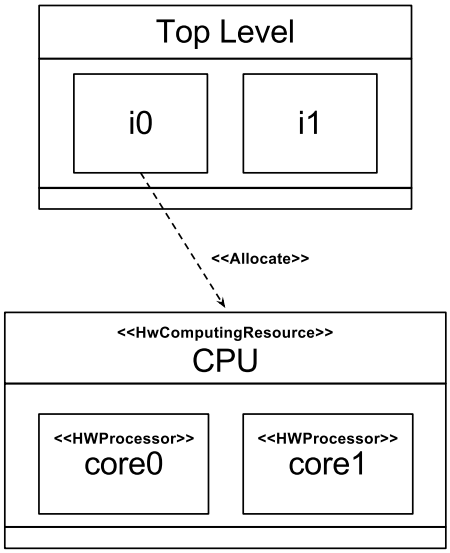
A fully adjustable model is shown in Figure 4.9(b), a single mapping from the application to the CPU (*HwComputingResource*) defines that any object within the application can be mapped to any core in the CPU. This results in four possible static mappings from this example; both *i0* and *i1* are mapped to *core0* or *core1* and *i0* is mapped to *core0* while *i1* is mapped to *core1* and vice versa.



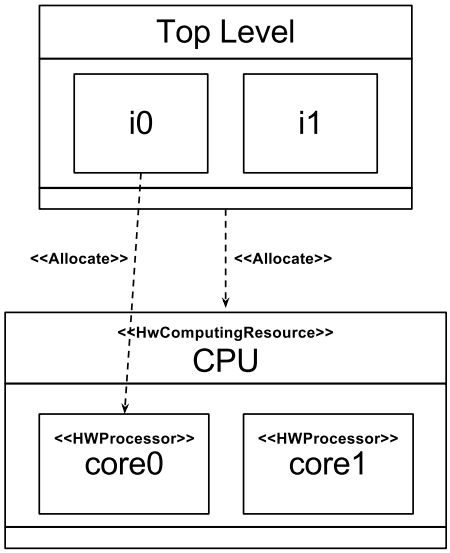
(a) Static Mapping - Objects directly mapped to *HwProcessors*



(b) Adjustable Mapping - Application mapped to *HwComputingResource*



(c) Adjustable Mapping - Object mapped to *HwComputingResource*



(d) Static and Adjustable Mapping - Objects mapped to both *HwComputingResource* and *HwProcessor*

Figure 4.9: Examples of Static and Adjustable mapping

Figure 4.9(c) shows another adjustable mapping option in which object *i0* is mapped to the CPU, resulting in two mapping options as it can either be mapped to *core0* or *core1*. In this example object *i1* is not mapped to any architectural resource as there is no higher level mapping options included.

Both static and adjustable mappings are shown together in a single system in Figure 4.9(d). Object *i0* is mapped statically to *core0* and will always be mapped to this specific core within the CPU. The second mapping is shown from the application to the CPU, this defines that any unmapped objects within the application can be mapped to any available core within the CPU. In this example there are two permutations, *i0* always maps to *core0* and *i1* can be mapped to *core0* or *core1*.

The combination of static and adjustable mappings within a UML model results in multiple mapping permutations of the application to the architecture. FalconML requires a PSUML model as an input, so a transformation process in-between XMI export from the UML tool and input into FalconML is required in order to produce a PSUML model from the PIUML model. This idea of adjustable mapping which is refined to static mappings is the first major contribution of this research in the space of high level modelling for a customisable architecture.

4.2.2.4 Design Rules

The adjustable mapping and wildcard multiplicity design rules introduced above do not require extensions of the UML specification. They allow the application developer to create a PIUML model which can then be refined to a PSUML model once the target architecture is known.

These design rules, used in both the application and architecture section of the UML model, are one of the major contributions of the research presented in this thesis and result in an approach which allows the application developer to create a single UML model which can later be refined to fully utilise a target architecture. Figure 4.8 shows an example of a system using these design rules, an application named *App* containing three objects (*i0*, *i1* and *i2*). Object *i1* is tagged with a wildcard multiplicity meaning that multiple instances of it can be instantiated within the application. The architecture consists of a single CPU with two cores (*core0* and *core1*), with *core1* being tagged with a wildcard multiplicity. This defines a CPU with two or more cores, which is dependent on the architecture being targeted. There are two mappings displayed, one from object *i0* to *core0* and one from *App* to *CPU*. The first mapping is a static mapping, *i0* will always be instantiated on *core0* and the second is adjustable. Any unmapped objects within *App* can be statically mapped to any available core within *CPU*.

This results in a different number of mapping permutations which is dependent on the

Table 4.2: Possible permutations of statically mapped UML models based on model shown in Figure 4.8.

Permutation	**Core0**	**Core1**
0	i0, i1_0, i1_1, i2	
1	i0, i1_0, i1_1	i2
2	i0, i1_0	i1_1, i2
3	i0	i1_0, i1_1, i2
4	i0, i1_0, i2	i1_1
5	i0, i2	i1_0, i1_1
6	i0, i1_1, i2	i1_0
7	i0, i1_1	i1_0, i2

number of cores and times that *i1* is replicated. For example, if there were two cores within the CPU and *i1* is replicated twice there would be a total of 8 permutations available. These possible permutations are shown in Table 4.2, where *i1* is replaced with *i1_0* and *i1_1* to represent the two instances. By increasing the number of available cores the number of permutations increases, for example, with three cores there would be 27 permutations and for four there would be 81.

The next section explains the implementation details required to perform the task of refining a PIUML model using the proposed design rules to fully utilise a specific architecture.

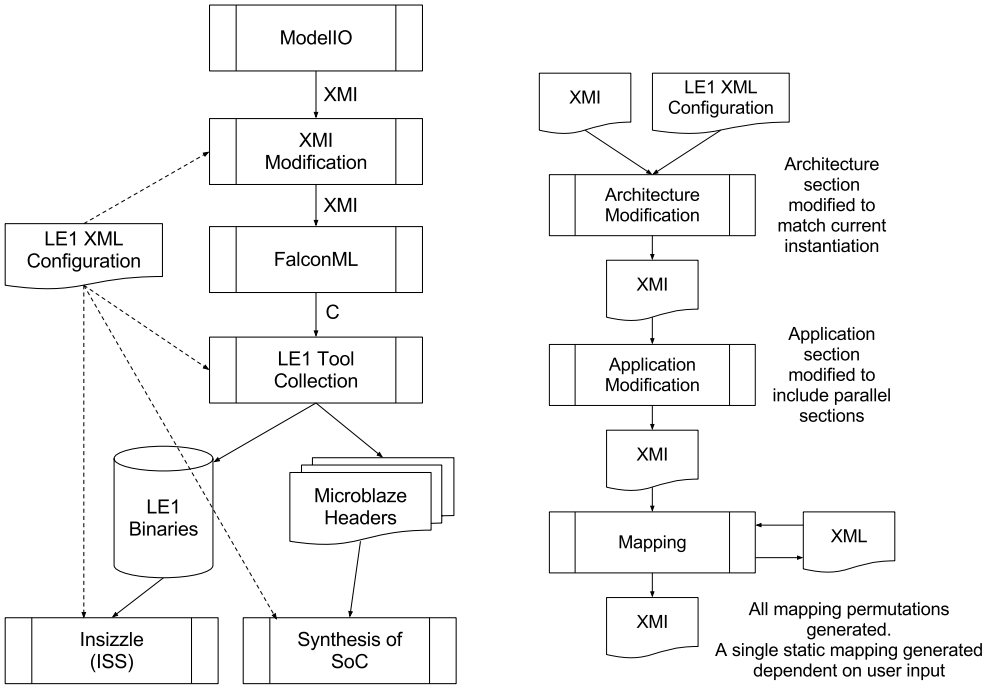
4.3 Practice

This section introduces the implementation details of the hardware system used as a platform for the research within this thesis. Also introduced is the software tool collection developed to transform PIUML models down to machine code to execute on the statically customisable VLIW platform. An overview of the tool flow is shown in Figure 4.10.

4.3.1 Very Long Instruction Word Processor

This section presents all of the elements which make up the hardware system being targeted. The hardware system is implemented on a Xilinx ML605 Evaluation Board [86] with a Virtex-6 LX240T device, allowing multiple VLIW processor to be instantiated.

The LE1 VLIW CMP [87] is highly parametrisable in both architectural and micro-



(a) Full tool flow showing UML to hardware and simulation

(b) Process of modifying XMI from platform-independent to platform-specific model

Figure 4.10: Tool flow from UML model to execution. The application developer generates a UML model in Modelio, this is then exported and processed to generate a statically mapped PSUML model which is synthesised by FalconML. This output is then compiled and assembled through the LE1 Tool Collection and the resulting output can be executed in simulation or on the physical LE1 device.

architectural views. It presents architectural parameters to the application developer to fully customise the hardware being produced. Table 4.3 shows the high level parameters of the LE1, these customisations are available to exploit both ILP and TLP while making trade offs between area and performance.

The base Instruction Set Architecture (ISA) is an amalgam of the partially-predicated Multiflow TRACE architecture [88] and the fully-predicated EPIC architectures [89] augmented with substantial Single Instruction Multiple Data (SIMD) support [90]. This model (ISA, state) can be extended with additional registers and single/multi-input, multi-output custom instruction extensions.

Table 4.3: Micro-Architectural Configuration settings available within the LE1 VLIW CMP.

Architectural Parameter	**Description**
ISSUE_WIDTH	Architectural width (LIW) of the processor. This is the number of RISC operations (syllables) dispatched every on clock
IALUs	Number of integer ALUs per processor
IMULTs	Number of integer multipliers per processor
IRAM_SIZE	Size of closely-coupled instruction (code) RAM. The application code is loaded in this memory prior to execution
DRAM_SIZE	Size of closely-coupled data RAM. The initialised data segment is loaded in this memory prior to execution. Serves as the stack area for all active hardware threads
DRAM_BANKS	Number of banks of the Data RAM. Accesses to disjoint banks incur no cycle penalty
LSU_CHANNELS	Number of channels to the Data RAM per processor
LE1_CONTEXTS	Number of LE1 contexts (cores) in the multiprocessor (CMP) configuration.

4.3.1.1 Configurability

The LE1 hardware and software tool chain is configured using a single XML configuration file. This is used to specify the number of contexts, hypercontexts and clusters along with the number of functional units and sizes of register files. It is used to generate the physical hardware as well as produce software which runs on specific implementations of this hardware. The LE1 structure is captured and the hierarchical view is displayed in Figure 4.11, this structure is then captured in an XML schema used throughout the tool collection. This configuration file is used to generate a main configuration VHDL file which is used to synthesise the LE1 system in hardware and is read directly by the Instruction Set Simulator to populate the software simulation of the LE1 system. In both cases the number of function units, registers and architectural elements as defined within the XML file are instantiated. Finally the configuration file is used to generate a VEX machine model file based on the micro-architectural configurations, this machine model file is used by VEX to perform in-

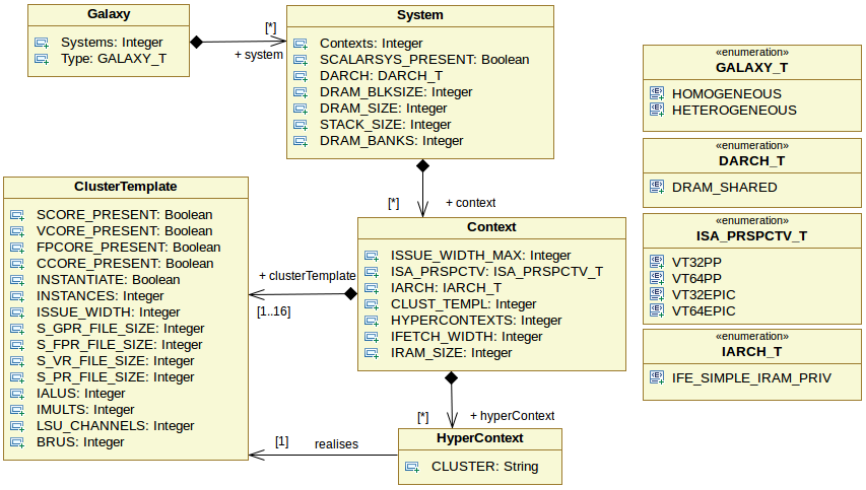


Figure 4.11: Structure of LE1 XML configuration

struction scheduling and optimally utilise the available functional units and registers. The use of a single configuration file throughout the tool flow maintains a synchronisation point across all tools and results in hardware and software sections being synthesised and compiled for the same target LE1 system.

An example XML config file is attached in Appendix A. This shows a two context, four issue wide homogenous LE1 system. A homogenous system implies that all contexts within the LE1 are similar and as such only a single cluster template definition is required. In this case the cluster template is instantiated by the single hypercontext and this will be instantiated twice due to the number of contexts being specified as "2" within the XML file.

4.3.1.2 Processor Core Organisation

The LE1 has an 8-stage pipeline, shown in Figure 4.12. This shows the internal structure and a break down of the internal pipeline stages of the LE1. The *Pipe Control* block depicts the primary control mechanism which schedules the full flow from instruction fetch and decode to data execution. It is responsible for initialising internal registers and memory sections through a debug mechanism from a host machine as well as maintaining the control space of the LE1 through a collection of state machines which schedule the overall system execution. As well as the *Pipe Control*, the CPU is composed of an *Instruction Fetch Engine* (IFE), *Load Store Unit* (LSU) and the main execution core (*LE1_CORE*). The IFE maintains the instruction cache (IRAM) and associated state machine. Each long instruction word can be up to two times *ISSUE_WIDTH* operations wide due to the inclusion of 32-bit immediates for large integers and addresses which results in the IFE controlling interlocks to

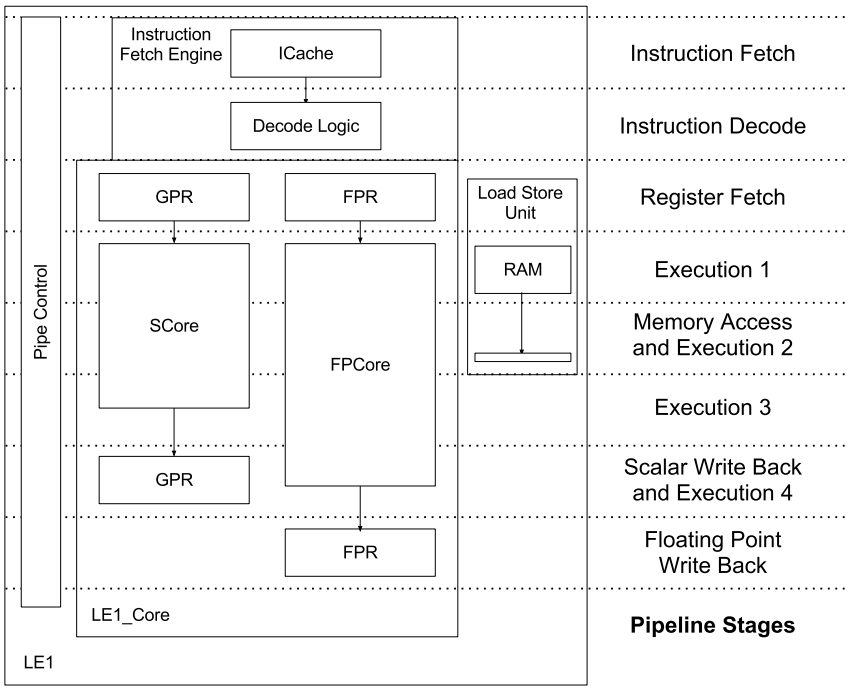


Figure 4.12: LE1 CPU core schematic

retrieve all required instructions when they span more than one IRAM location. The banked, shared memory is accessed from the *LE1_CORE* through the LSU. The number of channels (*LSU_CHANNELS*) to the memory along with the banking system (*DRAM_BANKS*) of the memory are both configurable. Finally, The *LE1_CORE* includes the main execution data paths of the CPU each with its own register set, integer (*SCore*) and floating point (*FPCore*) data paths. The *FPCore* is not instantiated in this study (all floating point calculations are simulated through library calls). The configuration of the *SCore* is dependent on the *IALU*s and *IMULT*s parameters shown in Table 4.3 which define the number of functional units available for executing integer based arithmetic operations.

4.3.1.3 Multiple Contexts

Increasing the value of the *LE1_CONTEXTS* parameter within the LE1 XML configuration file (Table 4.3) results in multiple instances of the *LE1_CORE* being instantiated. Figure 4.13 shows a dual context, shared memory LE1 CMP. Each *LE1_CORE* consists of functional units, instruction RAM and decode logic with a shared memory accesses to the common, banked DRAM. This shared DRAM enables communications between contexts as well as the ability for both contexts to perform computation on a shared data set.

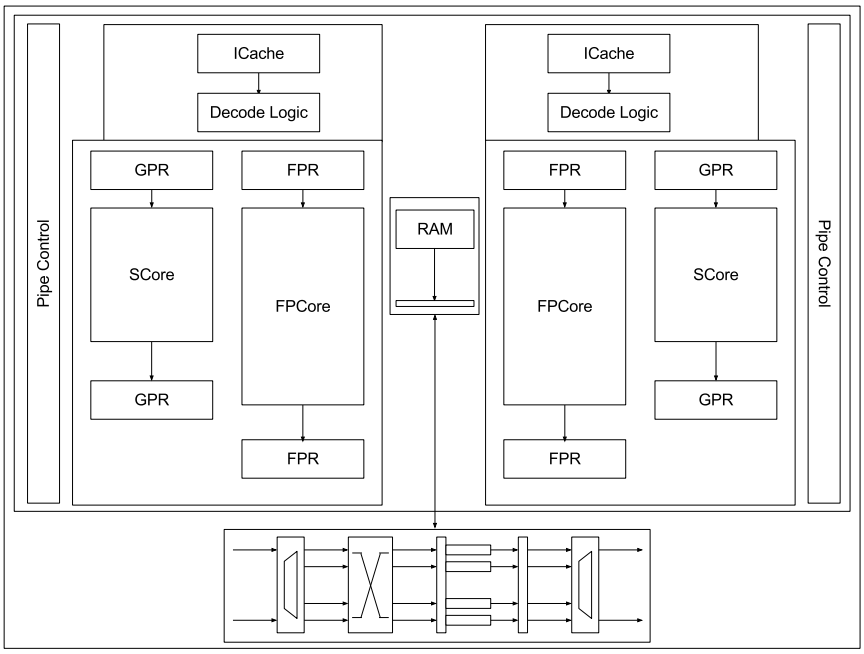


Figure 4.13: Two-way multiprocessors consisting of two instances of a 4-wide, single-cluster LE1 core, the common data memory and the thread control unit

4.3.1.4 Synthesis and System-on-Chip

The LE1 hardware is synthesised to match the configuration which is specified through the LE1 XML configuration file. This is translated into a VHDL file which sets configuration constants to generate the specified hardware.

Once the LE1 CMP is synthesised it is connected to the PLB system of a Xilinx FPGA design. This system contains a MicroBlaze processor along with memory blocks and peripherals. The MicroBlaze is a soft-core processor designed by Xilinx [91]. The processor itself is highly configurable to be optimised for either area or performance. In the usage with the LE1 we use an area optimised MicroBlaze core as it is simply used as a bus master to load instruction and data RAMs on the LE1 and no computationally intensive code is executed on it.

A single MicroBlaze processor is instantiated to act as a boot loader for the LE1. An API to the LE1 has been created which allows full access to the LE1 hardware for reading and writing registers, memory and control state of the LE1. This API consists of a collection of low level functions which are used along with data produced by the LE1 software tool chain, presented below. All data required by the LE1 is compiled through MB-GCC [92] (using the Xilinx SDK) with instruction and data RAMs being included as header files and

accessible through the MicroBlaze. These low level functions make it possible to compose more general functions to load instruction and data RAM and communicate with all available LE1 contexts. A default setup has been generated which requires a single LE1 instruction and data RAM to be included into the MicroBlaze binary. Once this is compiled and executed on the MicroBlaze it automatically loads instruction and data RAMs and starts execution on the LE1. The available API functions and descriptions of their functionality are included in Appendix B.

The MicroBlaze is also used to retrieve instrumentation data from the LE1 after execution has completed. All global data variables can be extracted from the LE1 data RAM which allows confirmation that correct execution was performed.

4.3.1.5 Task/Thread Level Parallelism

Thread management and dynamic allocation to hardware contexts takes place in the Thread Control Unit (TCU) within the LE1. This is a set of hierarchical state machines, responsible for the management of software threads and their allocation to execution resources. It accepts PThread operation requests from either the host or any of the executing hypercontexts. It maintains a series of hardware (state) tables, and is a point of synchronisation amongst all executing hypercontexts. Due to the need to directly control the operating mode of each hypercontext while having direct access to the system memory, the TCU resides within the debug interface (DBG_IF) where it makes use of the existing hardware infrastructure to start and stop execution and modify memory. A critical block in thread management is the Context TCU which manages locally (per context, in the PIPE_CTRL block) the distribution of PThread operations to the centralised TCU. Each clock, one of the active hypercontexts in a context arbitrates for the use of the context TCU; When granted access, the command requested is passed on to the TCU residing in the DBG_IF for centralised processing. Upon completion of the PThread operation, the Context TCU returns (to the requesting hypercontext) the return values, as specified by that command.

A subset of the PThread library is implemented, namely: *pthread_create*, *pthread_join* and *pthread_exit*. These three functions enable the creation, synchronisation and termination of threads within the LE1 system to provide an explicit method of application threading.

The PThread hardware implementation is not instantaneous, the setup time for the management of threads through the TCU carries latencies which block execution in the calling thread until the operation has been serviced. The latency of a *pthread_create* operation is 20 clock cycles. The 20 clock cycles allow the TCU to locate an available hypercontext within the LE1 system and initialisation of registers required for execution.

The *pthread_create* operation takes four arguments and returns the status of the operation. These arguments are shown in Figure 4.14. The operation is serviced by the TCU.

```
int
pthread_create( pthread_t *restrict thread,
                const pthread_attr_t *restrict attr,
                void *(*start_routine)(void *),
                void *restrict arg);

thread:         global memory pointer to set ID of created thread
attr:           unused
start_routine:  function pointer used to set program counter
arg:            pointer to global memory used to set as register 3
```

Figure 4.14: Arguments for pthread_create Operation

Firstly an available hypercontext within the LE1 system is located, this is a hypercontext currently not performing any execution. Once found, the ID of this hypercontext is stored in the location specified by *thread*, the hypercontext then begins execution of the function pointed to by *start_routine* using the argument pointed to by *arg*. The *pthread_create* operation also takes an *attr* argument to define the priority and scheduling of the created thread. In the LE1 hardware PThread implementation this is currently unused, it has been left in for compatibility reasons but is currently unimplemented and any thread attributes will be ignored.

When more than one *pthread_create* is issued at any time, or if a second is issued while the TCU is already servicing a request the second thread of execution is stalled until the current request is serviced, this is then stalled for 20 clock cycles to service its own request.

The *pthread_join* operation is a blocking operation. The execution of the calling thread is suspended until the target thread is terminated. The latency of this operation in the LE1 hardware PThread implementation is five clock cycles in a best case scenario. If the target thread has already completed execution and terminated the calling thread is stalled for five clock cycles, this is the time required by the TCU to perform a check and return the status of the target thread to the calling thread.

The *pthread_exit* function is implemented as a HALT instruction to terminate execution of the hypercontext as well as updating state within the TCU to allow other threads to be aware that the thread is no longer active.

In this implementation of the subset of the PThread library the passing of return values from the *pthread_exit* and *pthread_join* functions is not implemented. Due to this global variables are required to pass data across threads as well as careful programming to ensure that the executing code is thread-safe.

The CPUID method is simpler and returns a 32 bit control register consisting of a unique identifier for each hypercontext in the LE1 system. This allows the application developer to specify in their code how each hypercontext should execute. The CPUID instruction is encoded directly into the ISA. Within the C code a call to a function named *getCPUID()* is replace in the generated assembly to the *CPUID* instruction. For example:

```
int my_cpuid = getCPUID(NULL);
```

is replaced with

```
CPUID r0.5
```

where the variable *my_cpuid* is mapped to register 5 in the current hypercontext. This operation has a latency of 1 cycle, so the register is able to be used in the following clock cycle. This allows the returned value to be used to define the control flow of the program.

4.3.2 Software Tool Chain

The tool chain developed for the LE1 VLIW consists of the research compiler VEX, developed by HPLabs [7], along with a collection of scripts which perform transformations of the assembly generated by the compiler. These transformations extend the instruction set to include threading mechanisms and libraries required by both the physical implementation and the cycle accurate instruction set simulator. Figure 4.15 displays an overview of this tool chain starting with C code and resulting in the instruction and data RAM sections of the LE1 system.

4.3.2.1 Compiler

The compilation section of the tool chain is provided by a research tool available from HP Labs. VEX, Vliw EXample [66], is a VLIW compiler which extracts ILP from C code. VEX provides a full compilation and simulation environment, allowing C code to be executed as if on a VLIW described through the use of a machine model to define the number of functional units and cache sizes available. A compiled simulator is produced from the input C code which produces multiple heuristics based on the execution run.

In this tool chain VEX is used solely for its instruction scheduling and assembly generation. A dedicated LE1 assembler and simulator were developed to exploit and enable TLP provided by the LE1.

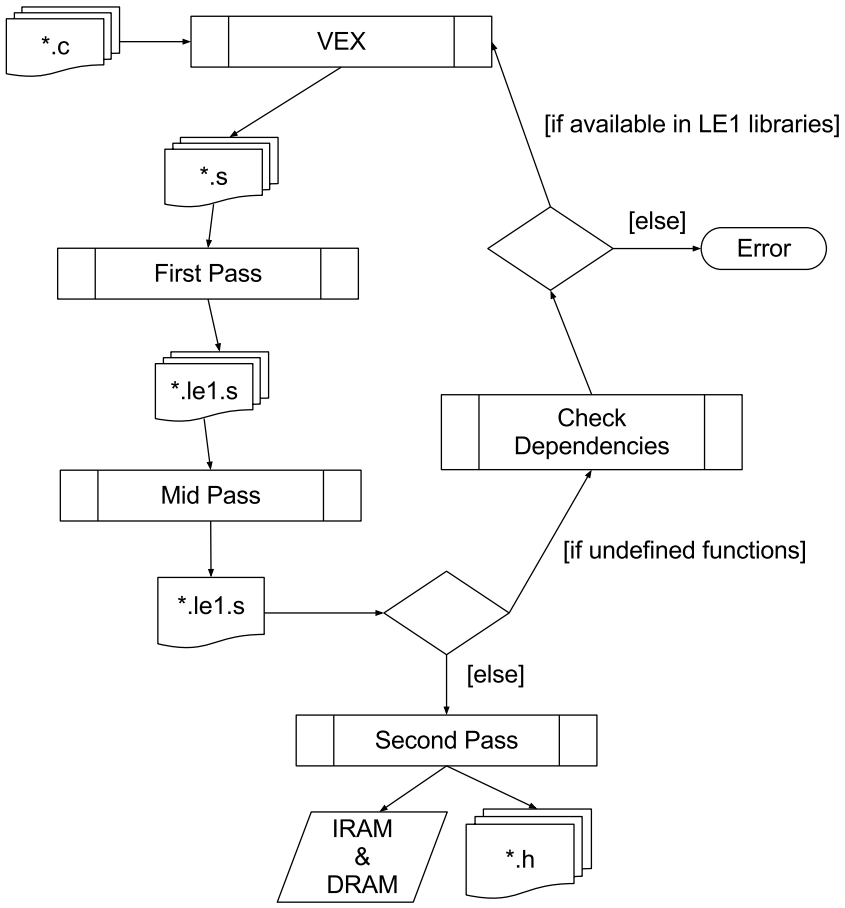


Figure 4.15: LE1 Tool Chain overview

4.3.2.2 Assembler

The Assembler section of the LE1 Tool Collection takes the scheduled assembly produced by the VEX compiler and transforms this to match the required ISA of the LE1 VLIW. These transformations include operation modifications and renaming, reordering of the code at function level and producing a single self contained assembly file. The Assembler is split into three sections which are performed in sequence to produce the final machine code. These three sections are described below; the order of execution is dependent on the output of the previous stages and is controlled by a scripted process to produce the final set of files used by various other sections of the tool collection.

The Assembler scripts and libraries are made available in [93].

4.3.2.2.1 Implementation Details

First Pass

The first stage of the LE1 Assembler takes the VEX assembly generated and translates it to create a new assembly file which adheres to the LE1 assembly specification. The assembly which is generated at the end of this pass is split into distinct sections: Operations, Imports, Data Labels, BSS Labels and Data Section.

The operations section contains all instructions, including scheduling, to implement the C code input. Imports displays a list of functions and variables which are required by the current file but not implemented within; these could be external and located within other assembly files from the same application or required from libraries which need to be included into the application. The data and BSS labels contains a list of all global variable names along with their location within the data section. The data section is stored in 32-bit hex format within the file. This pass also checks for a main function; as this is the start function for all applications the main function is required to be at the first instruction as the default behaviour of the LE1 is to start execution with a program counter of 0x0. If a main function is found it is moved to the top of the operations section.

Mid Pass

The mid pass takes multiple assembly files populated in the first pass and concatenates them together to produce a single assembly file in the same format as the files created by the first pass, taking care to preserve memory addressing. The order in which the files are concatenated requires that any file containing a main function be first, as this maintains the requirement of starting execution from 0x0. A check is also performed to make sure only a single main function is included. The data section is simply concatenated to create a single contiguous memory block, the labels within this data are incremented so that all labels point to correct data sections. The import section is then processed to remove any functions and data items which have been included through the concatenation of these assembly files.

This pass is only required when more than one C file is compiled and the main orchestration script controls whether this is perfomed or not. After the mid pass is run the orchestration script checks the list of imported functions which are required and if there are any which are available within the LE1 library section they will be compiled through VEX for the current machine model and included into the final single assembly file.

Second Pass

This is the final pass of the assembly and requires a single self contained assembly file with no external dependencies. This pass finalises all instruction and data labels and produces

the machine code based on the LE1 ISA. The output of this stage consists of a collection of files as follows:

binaries/

dram.bin	Binary file containing initialised LE1 data section
iram0.bin	Binary file containing full instruction area of LE1

microblaze/

data.h	Full data section stored in C byte array
inst.h	Full instruction section stored in C byte array
le1_obj.h	All instruction and data labels

output/

output.data.txt	Text file showing data section
output.inst.txt	Text file showing instruction section
output.le1.s	Self contained LE1 Assembly file

The IRAM and DRAM binaries are used by the LE1 interpreted simulator to confirm correct execution as well as produce execution heuristics. For execution on hardware three C files are generated to be compiled for the MicroBlaze processor. These files include the IRAM and DRAM stored in a byte array within C header files along with another file which contains the addresses of functions and variables within these arrays. These addresses aid in debugging the hardware system as global variables within the LE1 hardware can be read and written using the names given in the application being executed. This allows the use of variable names from the compiled application to be accessible from within the MiroBlaze system to get/set global variables within the LE1 system. Some other text files are generated to help debugging and matching assembly instructions to functions and program counters. These include breakdowns of the instruction and data RAMs in plain text including the assembly instructions, machine code syllables and program counters.

4.3.2.2.2 Libraries

A selection of libraries have been implemented for use with the LE1 tool chain. These are implemented in C and are included in the current application depending on whether functionality is required. Currently the math library, string library and some dynamic memory allocation are supported. Each function (for example, sin(), cos(), strcpy(), ...) is implemented in separate files which are only included if required by the application. Each such file needs to be compiled for the current hardware configuration. This is done automatically through the orchestration script as shown in Figure 4.15. Separating each function

means only required functions are available within the IRAM and helps in minimising the IRAM size. A list of dependencies and available base functions is used by the scripted assembly process; this list of available functions is queried with a list populated from the Imports section of the assembly file. If available they are compiled for the current hardware implementation and included in the IRAM. This method allows the users to define their own library functions if required or to use the base implemented functions.

4.3.2.3 Orchestration

The compilation and assembly stages are fully automated through a single scripted process. The transitions in Figure 4.15 are controlled through an orchestration script which copies files into required locations and invokes the processes (displayed as rectangles) depending on the outcome of previous stages. This simplifies the tool chain and also aids in a single machine model being used throughout the flow to configure all aspects of the tool chain to target a single design point.

4.3.2.4 Simulator

The LE1 Instruction Set Simulator (Insizzle) is a customisable, interpreted simulator which executes instructions as the LE1 hardware would. It is used to confirm correct execution as well as producing various output heuristics which can be used to inspect/profile the code being executed. Insizzle was produced during the course of this thesis and was used for extracting heuristics of application execution under alternative configurations of the LE1 processor.

Insizzle requires the instruction and data binary files produced by the Compilation and Assembly stage along with the machine description in the LE1 XML configuration file. Initially, the XML configuration file is read which sets up the simulator to match the hardware being targeted. There are various arguments which can be passed to Insizzle which alter execution and provide more information output. These are displayed in the Appendix in Table D.1.

Once execution is completed heuristics of the run are printed to the output along with a file name “memoryDump.dat” which contains a full data section dump. This can be used to retrieve global data variables to confirm correct execution as well as compare against a known correct output. The heuristics printed to screen are displayed in the Appendix in Table D.2.

The Insizzle source code is made available in [93].

4.3.2.4.1 Implementation Details

Insizzle was produced along side the creation of the LE1 hardware implementation and aided in verification and validation of the LE1 RTL. It was also used to implement various threading techniques prior to them being implemented in hardware, as it was easier to fault find and finely tune ideas. It also functioned as a test bench for the XML configuration file to confirm the implementation of this fully incorporated all aspects of the hardware design.

The simulator implements a set of control registers identical to that of the hardware as specified in the programmer's reference manual [84]. The simulator was written in C as a single executable which is passed the instruction and data binaries along with a machine model description (LE1 XML configuration file). The machine model is used to define the simulator's internal system registers which are in turn used to dynamically populate internal storage and data structures. The simulator executes the IRAM as the hardware would, taking into account interlocks when fetching instruction bundles and servicing memory requests. Each available hypercontext is queried each cycle to check its internal state, this internal state defines whether it is **ready**, **running**, **blocked** or **terminated**. A hypercontext with a state of **ready** defines that it is available to perform execution, **running** defines that the hypercontext is currently executing instructions, **blocked** and **terminated** are used in threaded mode to define states relating to synchronisations between threads and when execution has completed within a hypercontext. The simulator has extra states to mimic the execution of the hardware; these include a stall count state which can be set and is decremented each cycle until zero to stop execution of instructions in a given hypercontext.

The instruction fetch section of the simulation uses a look ahead within the IRAM to stall the execution for the correct number of cycles before allowing the instructions within a packet to be executed. Through the use of the simulator, the instruction fetch has been found to be a critical path within the LE1 execution, with around 10% of execution cycles being the result of instruction fetch engine stalls. These stalls are incurred due to packing of instructions to reduce the size of the IRAM. Instructions are grouped into packets and each instruction within the packet can be executed in parallel. In order to minimise the size of the IRAM a single bit within each instruction is used to denote the end of a packet. This enables instruction packing within the IRAM and reduces the space required for the IRAM as no-op instructions are not required to denote empty instruction packet slots. This instruction packing results in the possibility of packets spanning multiple instruction rows and the instruction fetch engine is tasked with extracting full packets from the IRAM before instruction decoding is performed. Large immediates are encoded into the instruction RAM as full 32 bit values with each instruction able to include a 32 bit immediate. This results in a worst case of two additional cycles for the instruction fetch engine to retrieve a full

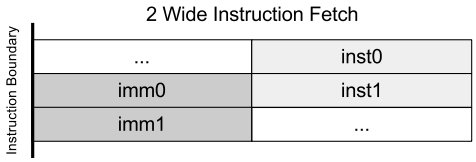


Figure 4.16: Instruction packing and fetching example in LE1 system.

instruction packet from the IRAM. An example of this instruction fetching is shown in Figure 4.16.

The example shows a worst case scenario using a two wide LE1 instruction fetch engine. Three cycles would be required to retrieve the full instruction. Each instruction (*inst0* and *inst1*) include a 32 bit immediate (*imm0* and *imm1*) resulting in a full instruction packet of four 32 bit words. In the first cycle only *inst0* would be retrieved, the instruction fetch engine would know this included a 32 bit immediate along with the fact the "packet end bit" not being set. The second cycle would retrieve *imm0* and *inst1*, this time the "packet end bit" is set within *inst1*, however, this also includes a 32 bit immediate. The final cycle would retrieve *imm1* which completes the packet, this is then passed to the next stage of the pipeline to be decoded. This process incurred a two cycle stall due to the extra fetches required to generate the full instruction packet.

In order to partially hide stalls incurred by the the instruction fetch engine the memory system and instruction fetch engine run in parallel. If execution is stalled due to the memory banking system, where more than one memory operation attempts to read or write to a single bank, the instruction fetch engine can continue to fetch the next instruction packet to make the next instruction available for execution once the memory is retrieved.

If running, and not stalled, the simulator retrieves its current packet of instructions and executes them in the order they are specified in the IRAM. This is the only difference between the simulator and hardware, as the hardware would execute the instructions in parallel. However, this does not affect execution due to there being no data-dependencies between registers used within the instructions. Loads and stores are pushed onto a memory access queue which is serviced at the end of each system-wide cycle. This servicing of the memory request queue takes into account the number of memory banks available in the system and either provides data to the requesting hypercontext or stalls the hypercontext until the memory operation can be serviced by the load/store unit.

The simulator also implements a set of system calls, such as file I/O, which allow a wider range of application code to be compiled and run under simulation to retrieve execution heuristics before having to modify code to work in the hardware implementation without file I/O.

There are two modes of execution available:
A) only the state of a single hypercontext within the system is set to **running** and is executing instruction at time zero, the other available hypercontexts are set as **ready**. This is used in PThread mode to allow the creation of threads through the ISA extensions based on the PThread library.
B) all available hypercontexts have a **running** state and begin execution at 0x0 within the instruction RAM, this is used in both single core and CPUID mode.

The Insizzle simulator produces an overall cycle count of an application for a given LE1 XML configuration file and this figure can be used to predict the actual execution time of the LE1 system in hardware.

In order to confirm the heuristics output of Insizzle were inline with the hardware implementation, instrumentation was included within the hardware to allow clock cycles to be measured. Various test applications and system configurations were executed and the results compared from both simulator and hardware execution. With only a single load/store unit instantiated, resulting in only one memory operation per cycle, the total cycles reported by both the simulator and hardware are exact. When more than one memory operation is performed each cycle, which can be due to either multiple load/store units in a single core or multiple cores executing in parallel, the results differ; with Insizzle reporting a higher number of cycles than the hardware.

This difference in the reported cycle counts in simulation and execution on hardware is a result of how the memory system is simulated. As cores are simulated in a specific order each cycle (first to last) with memory operations pushed onto a queue when executed there is an order bias which is not present in the hardware. This results in a slight deviation in cycle counts produced by Insizzle, with Insizzle reporting a worst-case cycle count.

From the heuristics generated from Insizzle an estimate of the hardware execution cycles can be made. Insizzle reports the number of stall events as the result of memory conflicts, using this value along with the total execution cycles gives a margin of error in which the hardware execution cycles resides. Although the simulation of the memory system is not perfect it allows for initial conclusions to be drawn from heuristics regarding execution time which can be used to guide optimal hardware generation for specific applications.

The simulator executes at around 1.8MHz (per hypercontext) and is only 3.5% of the speed of the hardware running at 50MHz. However the ability to display each executed instruction allows deeper investigation of the executed cycles and a richer amount of information regarding execution. The simulator is also configurable and allows investigations into various LE1 configurations without the need to synthesise the hardware system, which can be a very time-consuming process.

4.3.3 Modelling

The three methods for system modelling in UML for a statically customisable multicore architecture were described previously in section 4.2.2. They focus on the three sections of a system from the modelling perspective, application, architecture and mapping and can be used singularly or all together to allow the application developer to produce a PIUML model.

This section describes the implementation details of the proposed methodologies. Finally, an example using all three modelling methods together is presented and discussed. One of the limitations of UML modelling tools when exporting the UML model to other applications is the loss of diagrammatic information. Each UML tool vendor uses different ways to store diagram information and there is no standard way of exporting this data. The behavioural synthesis tool requires its input in the form of an XMI structure, this file is exported from the UML modelling tool and it is this file which is modified by the tools and processes to transform a PIUML to a PSUML model. This XMI structure includes only the underlying UML model which has been captured in a UML modelling tool by the application developer. Modifications are made directly to this XMI structure through the use of XML parsers and scripts to perform the tasks required. Due to the lack of diagram import option in the UML modelling tools some extra work was performed to visualise possible static mappings available based on the input UML model. This is described in more detail below.

The methodology used to design applications in UML uses a subset of the UML specification which is outlined below. The architecture which the application is mapped to is also captured in UML using a similar style along with elements from the MARTE profile to stereotype classes and objects as hardware elements.

The behaviour of an application is created using UML classes and Class Diagrams. A class is seen as a containing element and includes variables and operations which are available within the class itself. Ports are also included within classes, and are used to transfer data into and out of these classes. The Class Diagram is used to specify various object-oriented aspects of the application: associations, dependencies, etc. Ports are required to have a type which is in the form of an interface. An interface in UML is similar to a class which includes only operation definitions, it defines the public functions available within a class. The functions defined within the interface are accessible to external objects through ports on the class. The behaviour is included using both UML state machines and action code. State machines capture behavioural flow in a graphical form and are used to schedule behaviour depending on the current state of an object. Action code can be included within the state machines where it can be tagged onto transitions and used to perform entry and exit actions within states. Action code can also be included within operations in a class similarly to

conventional programming languages. Behaviour within state machines can be modified through the use of signals and signal events. A signal event is used to trigger a transition from one internal state to another within the state machines. Signals are linked to operations within an interface and allow non-blocking calls to be performed between objects.

Composite Structure Diagrams are used to define the communication structure of an application. Classes are instantiated in the form of objects which contain the behaviour and ports of the class. Using connectors the ports are then connected together to specify the channels which objects communicate through. This results in a complete application view in UML.

It is at this level which the architecture is also captured using the MARTE profile to stereotype classes as either a CPU (*HwComputingResource*) or a processor core (*HwProcessor*) and then using *Allocate* arrows to map the application elements to the architecture.

4.3.3.1 Architecture

Using the LE1 XML configuration file presented in 4.3.1.1 along with the XMI structure of the UML model it is possible to modify the underlying architecture within the model to match that of the physical architecture being targeted. The steps for modifying the architecture section of XMI structure are displayed below. The process has been implemented in Perl and can be found in [93].

1. Read the LE1 XML configuration file and store the number of available context in the architecture.
2. Read XMI structure.
3. Located the *HwComputingResource* with in the stored structure.
4. Within the *HwComputingResource* count the number of *HwProcessor*s and check for items tagged with wildcard multiplicity.
5. If any *HwProcessors* are tagged with a wildcard multiplicity and the total number is less than is available within the physical architecture insert new cores and stereotype them using MARTE profile.
6. Else If the number of available cores within the physical architecture is less than that specified in the XMI structure return an error.
7. Write XMI structure to file.

This results in a new XMI structure where the number of *HwProcessors* within the UML model matches that of the underlying multicore architecture being targeted.

4.3.3.2 Application

The modification of the application within the model requires additions to both the structure and behaviour of the model. Two methods, as described in section 4.2.2.2, are referred to here as RoundRobin and Split, where RoundRobin passes calls to subsequent replicated objects and Split executes a single operation across multiple available objects and waits for all to finish before proceeding. The overall process of this modification is shown below; for both RoundRobin and Split this process is similar with only step 6 being different. An implementation of this process, written in Perl, can be found in [93].

1. Read XMI structure.
2. Record the number of objects within the application and cores within the architecture.
3. Find objects within the architecture tagged with a wildcard multiplicity.
4. Calculate the number of times to replicate this object.
5. Find the class which is the base of the object to be replicated.
6. Perform RoundRobin or Split.
7. Create new XMI model containing new objects.

Step 6 using the RoundRobin method:

1. Find the Interface which the input port(s) of the object to be replicated uses.
2. Create a new class (*Fork*) which implements all operations in this interface.
   - Generate input port(s), same number as on the object to be replicated.
   - Generate output port(s), number of input ports on object to be replicated multiplied by the number of times to replicate the object.
   - For each operation create an attribute to define the last object which was called.
   - Create behaviour for each operation to perform a round robin calling mechanism.
3. Find the Interface which the output port(s) of the object to be replicated uses.
4. Create a new class (*Join*) which implements all operations in this interface.
   - Generate input port(s), same number as output ports in *Fork*.
   - Generate output port(s), same number as output ports in object to be replicated.
   - Create behaviour for each operation to pass data directly from input to output port.
5. Instantiate both *Fork* and *Join* classes within the application.
6. Reconnect all connections from object to be replicated to the *Fork* and *Join* objects.

7. Remove object to be replicated from application.
8. Create multiple instances of class which should be replicated.
9. Connect output ports of *Fork* to input ports of new objects and their output ports to the input ports of *Join*.
10. Make sure there is a mapping from the application to the CPU, this allows full mapping permutations.

Step 6 using the Split method, this requires the operations being split to have a predefined set of arguments. These arguments are used to split the computation across all replications of the wildcard object. They include the data set, the size of each item to be processed and the number of items to be processed. Using these arguments the fork object splits computation across each available object.

1. Find the interface which the input port(s) of the object to be replicated uses.
2. Create a new class (*Fork*) which implements all operations in this interface.
   - Generate input port(s), same number as on the object to be replicated.
   - Generate output port(s), number of input ports on object to be replicated multiplied by the number of times to replicate the object.
   - Create an attribute to define the number of available objects to perform computation.
   - Each Operation is then implemented to include a call to each available object using the start, size of data items, number of available objects and their id.
3. Find the Interface which the output port(s) of the object to be replicated uses.
4. Create a new class (*Join*) which implements all operations in this interface.
   - Generate input port(s), same number as output ports in *Fork*.
   - Generate output port(s), same number as output ports in object to be replicated.
   - Create an attribute to define the number of available objects to perform computation.
   - Create an attribute for each operation to be used as a counter to increment for each call received.
   - Create behaviour for each operation to increment the operations counter. If all are finished pass data directly from input to output port, else do nothing.
5. Instantiate both *Fork* and *Join* classes within the application.
6. Reconnect all connections from object to be replicated to the *Fork* and *Join* objects.
7. Remove object to be replicated from application.

8. Create multiple instances of class which should be replicated.
9. Connect output ports of *Fork* to input ports of new objects and their output ports to the input ports of *Join*.
10. Make sure there is a mapping from the application to the CPU, this allows full mapping permutations.

The use of either RoundRobin or Split is dependent on the type of application being modified. The inclusion of the *Fork* and *Join* objects result in an overhead to perform computation which alters control flow. However, when investigating the amount of work required to modify the UML models for specific architectures the trade off between design time and computation overhead will be investigated.

The usage of both RoundRobin and Split methods for processing data within real applications is presented in chapter 6.

4.3.3.3 Mapping

The process of translating a PIUML model using adjustable mappings to that of a statically mapped PSUML model is split into two processes. Firstly a process reads the XMI structure and generates a list of all possible static mapping permutations along with a set of images displaying these static mappings. The second process then takes this list of static mappings along with a number defining which mapping to implement. The original XMI structure is then modified and a single statically mapped PSUML model is generated.

Implementation details of these two processes is described in detail below. Both are made available in [93].

4.3.3.3.1 Read Allocations

The read allocations process which generates all possible static mapping permutations:

1. Read XMI structure.
2. Create a list of all UML classes.
3. Traverse XMI structure to find the Application section, store each object within the application.
4. Find MARTE *HwComputingResource* stereotypes, produce a list of available *HwProcessor*s contained.
5. Find MARTE *Allocate* stereotypes, find the UML dependency they link to.

6. Using identified dependencies find which level of the application and architecture sections they use.

7. Calculate which level of the Application the allocation arrow starts at:
   - Object: create a reference to either a single core or all cores within an CPU.
   - Application: create a reference for each available object to either a single core or all cores within a CPU.

8. This results in a list for each object within the application of the possible cores within the CPU it could be mapped to.

9. Remove any duplicate entries of cores in the references.

10. Using this list loop through and generate all possible permutations of objects to cores.

11. Generate an XML file containing all of these possible permutations.

12. If specified create an SVG image of each possible mapping, showing the application, architecture and mapping section in a graphical manner.

4.3.3.3.2 Write Static Allocation

After the Read Allocations process a single XML file is generated containing all of the possible static mappings of the UML model. This output is then used by this process, along with a user specified design to create a new XMI structure containing a single, statically mapped system.

The Write Static Allocations process generates a statically mapped PSUML model:

1. Read XMI structure.

2. Remove all instances of MARTE *Allocate* stereotypes from XMI structure and the UML dependencies these link to.

3. Read XML file generated in the read allocations process.

4. Use passed mapping id to specify an available mapping.

5. Generate new UML dependencies to match the given mapping and attach to XMI structure.

6. Stereotype these new UML dependencies with MARTE *Allocate* tag.

7. Write XMI structure to file.

4.4 Example Flow

Using all three methods, described in the previous section, together results in a PIUML model which can be modified to target multiple architectural instantiations of a hardware template. The processes must be run in the correct order to fully utilise the underlying architecture. Firstly the architecture section should be expanded, replacing any cores tagged with wildcard multiplicities with extra cores to match that of the architectural instance being targeted. Following this, the application section is expanded to generate extra objects to exploit parallelism within the application. The number of times the object is replicated is calculated using the number of unmapped cores within the architecture section. Finally, with a fully expanded architecture and application section of the system, the mapping process is performed in order to generate a list of all possible static mappings. Then, using one of these possible mappings, a PSUML model is produced which aims to fully utilise the current instantiation of the architecture.

An example of the tool flow from UML, using the design rules presented in this section, down to simulation of a multicore LE1 processor in Insizzle is included in Appendix F. This tool flow is used as a basis for investigations using various applications in chapter 6.

4.5 Summary

This chapter introduced the hardware architecture and the software tool chain which targets it. The hardware system, which is implemented on a single FPGA, is composed of a configurable multicore VLIW processor which communicates with a scalar CPU through a bus. The scalar CPU acts as a master on the bus to load instructions and data RAMs into the VLIW and control the execution of all available cores. The LE1 VLIW CMP includes a subset of the PThread library which is implemented in hardware extensions to perform thread management tasks. The software tool chain targeting this hardware system and PThread implementation is comprised of multiple compilation and assembly sections. These work together to produce a single binary which executes on the scalar CPU and automatically loads and executes the VLIW CMP.

Also introduced are the novel UML design rules which provide a means to produce high level models to target the multicore hardware system resulting in a single PIUML model able to be refined to generate a PSUML model for a specific architecture without the requirement of the application developer redesigning the initial UML model.

The UML design notations include:

Architecture Wildcard: This enables the application developer to tag architectural elements with a wildcard multiplicity to define possible expansions. For example, a core

within a CPU with a wildcard multiplicity represents one or more cores. This notation is used along with a physical implementation where the number of cores is known to replicate the cores tagged with wildcard multiplicities so that the architecture matches that being targeted.

Application Wildcard: Similar to the architecture wildcard, application objects tagged with a wildcard multiplicity define a section of an application which can be replicated. This represents a section of an application which can be performed in parallel. Two methods of using this notation are shown which are based on different thread library techniques for performing parallel tasks.

Adjustable Mapping: This allows the application developer to specify the mapping of an application to an architecture at a higher abstraction level which can then be refined to a PSUML model.

To link the system generated in UML to the underlying architecture and tool chain, which only accepts C as an input language, another tool is required to transform the application generated in UML to a usable format for the compiler. A behavioural synthesis tool is used for this job, it takes a UML model exported into an XMI structure and generates both hardware (VHDL or Verilog) and software (C or C++) implementations of the model. This is a key tool used to verify that the UML design rules which are defined produce correct models as well as to investigate the difference in performance compared with hand crafted UML models.

The work presented in this chapter is expanded on in chapter 5 which introduces the tool chain of an FP7 project. This project uses the tools and a selection of the design rules presented here, along with a Design Space Exploration (DSE) tool. The adjustable mapping design notation is used in the UML models as an input to the tool chain and the DSE interacts with the generated static mappings. The DSE tool specifies which static mapping to generate and uses the resulting statically mapped PSUML model to gather heuristics regarding design sizes and execution time.

The concepts of both hardware threading and UML modelling described in this chapter are then investigated in chapter 6 and results are compared with alternative platforms and implementations.

5

Implementation and Usage

5.1 Chapter Objectives

This chapter introduces a tool flow which is currently implemented and being used within the ENOSYS FP7 project (intEgrated modelliNg and synthesis tOol flow for embedded SYStems design) [94].

The ENOSYS project tool flow is focused on using UML as a high level input language to model applications destined for hardware/software co-design. Tools and modelling methodologies developed within this thesis are used in the tool flow to enable automated design space exploration. The design space exploration aims to generate optimised UML models using alternative mappings across the hardware and software domains based on an initial UML model. These mappings are synthesised, compiled and executed guided by a design space exploration tool which uses area and execution metrics to prune the available design space.

The UML design notations presented in this thesis have been extended and modified to aid in this design space exploration across both the hardware and software domain. These modifications are introduced and discussed in this chapter.

5.2 ENOSYS Project

The main objective of the ENOSYS project is to reduce design and development cost as well as shortening the time to market of electronic products [94]. The project proposes the modelling and synthesis of embedded systems through a novel design methodology and enhancement of existing tools presented within a tool flow based on both commercial and research tools. The project consortium consists of four commercial companies and two

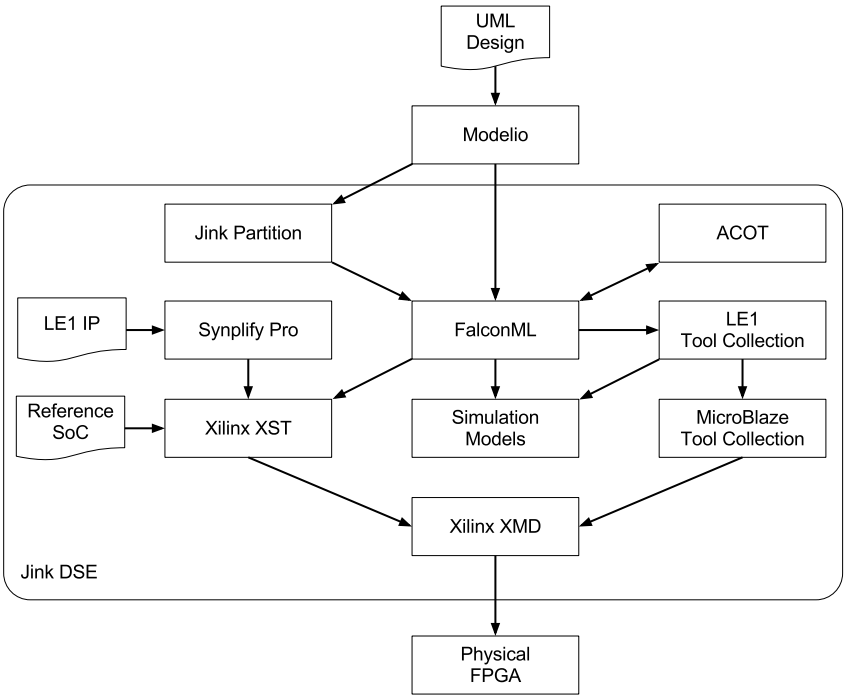


Figure 5.1: High-level block diagram of the ENOSYS tool flow [95]

universities; two of the four commercial companies offer tools and the other two design case studies to present the tool flow and ENOSYS methodology. The two universities also offer tools used within the project.

5.2.1 Tool Flow

A high level block diagram of the ENOSYS tool flow is show in Figure 5.1, the arrows represent data transfer between tools, which are displayed as boxes. The Jink Design Space Explorer (DSE) works as an orchestrator to control this flow of information exchanged between tools. The functionality of the tools incorporated within the project are described below.

5.2.1.1 Design Space Explorer

The design space exploration of the ENOSYS flow is conducted by Jink [96]. The DSE initialises necessary files and invokes tools in the correct order, as defined within the flow as well as performing housekeeping to ensure a single architectural template is used through out a single design iteration.

Jink performs design space exploration around a platform-independent UML model (PI-UML). This design uses the static and adjustable mapping design notation presented as a contribution of this thesis.

The Jink tool evaluates alternative application mapping across architectures to produce best case designs based on end-user requirements. This is achieved through interaction with the PIUML to platform-specific UML (PSUML) model translation process presented in chapter 4 to generate a list of all possible static mappings which allows the intelligent DSE to iterate through all PSUML models based on the adjustable mappings of the original PIUML model.

Jink also uses the LE1 XML configuration file to produce the physical hardware of the LE1 system which is included into a System-on-Chip (SoC) and the machine code for executing on the multicore LE1 system.

Finally, the DSE parses the output logs and report files from tools within the flow and extracts design characteristics and metrics associated with compilation, synthesis and execution. This data is stored in a database which is available to the user as well as being used by the exploration engine to prune the search space and automatically produce an ideal static mapping of the UML model.

5.2.1.2 Modelling

The ENOSYS project presents a subset of the UML and MARTE specification to be used for system design. The ENOSYS modelling language uses two concepts from UML: Composite Structure and State Machine Diagrams. This selection enables the capture of structure and behaviour of an application, respectively. Three concepts from the MARTE profile are also included, two to capture the architecture of a system: the *Allocate* and *Hardware Resource Modelling* (HRM), which includes *HwProcesor*, *HwResource* and *HwComputationResource*, and the *Value Specification Language* (VSL) used to capture non-functional properties of a system used for requirements and traceability throughout the tool flow.

The Modelio [97] UML capture tool, provided by one of the commercial partners within the ENOSYS project, is used along with the ENOSYS modelling language for design entry.

5.2.1.3 Behavioural Synthesis

The behavioural synthesis tool used within the project uses UML models captured using the ENOSYS modelling language and produces custom hardware blocks, multicore C code and the communication protocols for cross-domain message passing based on the mapping defined in the UML model.

FalconML [85], provided by Axilica, another commercial partner within the ENOSYS

project, provides this functionality and produces both VHDL RTL and C code to be synthesised and compiled to produce an implementation of the input UML model.

The split of the application captured in UML into either hardware or software is dependent on the mappings defined in the UML model. FalconML requires a PSUML model to perform the RTL and/or C code generation correctly.

The C code generated includes FalconML wrapper code which implements event queues for each available core within an architecture. The mapped application objects then execute on specific cores. If more than a single object is instantiated on a single core the objects are interleaved using these FalconML internal event queues.

5.2.1.4 Source Code Optimisation

An optimisation tool is also included in the ENOSYS tool flow. ACOT [98] is a source-to-source transformation tool which optimises the C code generated by FalconML. High and low level optimisations are performed including loop unrolling, intra-block vectorization and algebraic simplification. These optimisations are managed by the Jink DSE which configures the level of optimisation performed throughout the code.

This section of the tool flow takes code generated by FalconML and produces optimised C code which is then compiled for the LE1 CMP.

5.2.1.5 Compilation

The compilation stages of the ENOSYS flow consist of the LE1 Tool Collection and the MicroBlaze GCC compiler [92]. The LE1 system, which is the target programmable platform within the ENOSYS project, is implemented as part of the SoC shown in Figures 5.2 and 5.3. The LE1 tool collection is required to produce machine code for this hardware implementation. The SoC consists of a MicroBlaze processor which is a master on the PLB along with the LE1 and FalconML hardware blocks, connected as slaves. The MicroBlaze initialises these slaves and initiates execution in both the hardware and software sections of the synthesised UML model. Other peripherals are also connected to the SoC, including a UART for console logging from the MicroBlaze as well as memory blocks and controllers.

The C code generated by FalconML and ACOT is required to be compiled for the LE1 VLIW CMP. This produces machine code to execute on the LE1 in two formats; binary files and C header files.

The binary files are used by the LE1 instruction set simulator to perform initial execution to ensure correct execution and extract performance metrics (including execution time). These metrics are stored and used by the DSE to perform exploration around both LE1 configurations and ACOT optimisations.

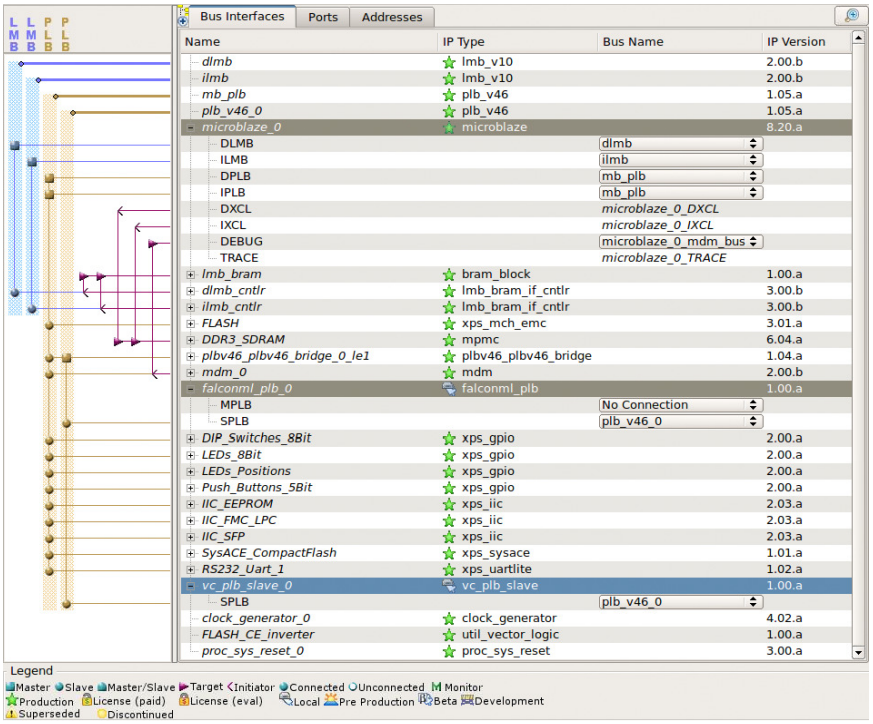


Figure 5.2: ENOSYS System-on-Chip consisting of MicroBlaze (*microblaze_0*), FalconML (*falconml_plb_0*) and LE1 (*vc_plb_0)* hardware blocks (highlighted).

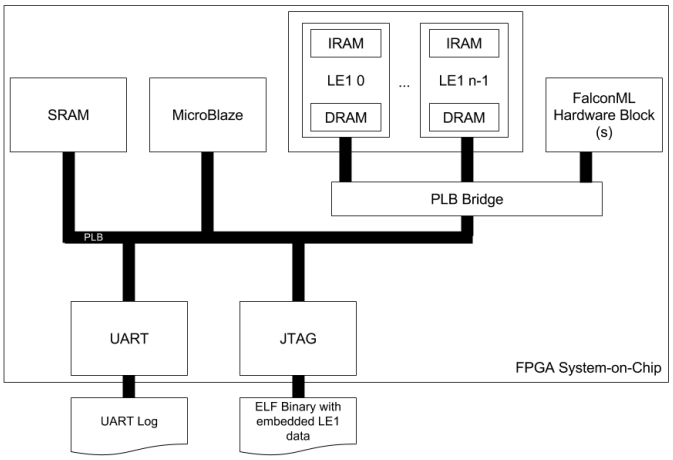


Figure 5.3: Block diagram of ENOSYS System-on-Chip consisting of MicroBlaze, FalconML Hardware Block, multiple LE1 and other peripherals; UART, JTAG, SRAM and PLB Bridge to connect LE1 and FalconML to PLB Bus

The C header files are included into a MicroBlaze program used to initialise the LE1 VLIW CMP executing in hardware. The MicroBlaze program loads the instruction and data RAMs into the LE1 processor(s) and modifies control registers to start execution of the code on the LE1.

This program is also used to extract execution metrics from the physical LE1 implementation once execution has ceased. The MicroBlaze program is pre-written and uses the low level LE1 API included in Appendix B.

5.2.2 Contributions

The ENOSYS project targets a hardware/software co-designed system consisting of a multicore VLIW processor and custom hardware accelerators. The behavioural synthesis tool generates the hardware blocks, multicore software and the communication process to perform message passing between these. The project utilises both the LE1 Tool Collection and the mapping methodologies developed in this thesis.

The static and adjustable mapping methodology developed for multicore processors is extended to target the hardware blocks available through the ENOSYS project tool flow.

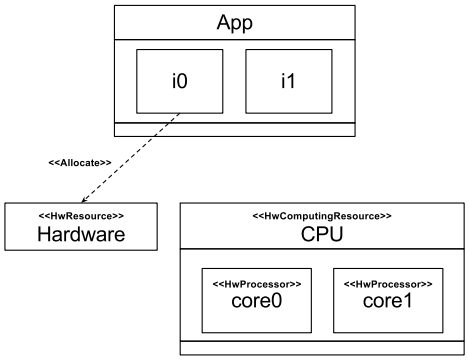
The process of adjustable mapping as described previously is extended to include the concept of *HwResource*s. These MARTE stereotyped objects within the architecture section of the system are treated similarly to that of the processor cores (*HwProcessor*) where mappings of objects to a *HwResource* defines that the object is always instantiated as a hardware block and mappings from the application to a *HwResource* define a mapping which is dependent on any other mappings within the system.

Figure 5.4 shows examples of static and adjustable mappings used across the hardware and software domain.

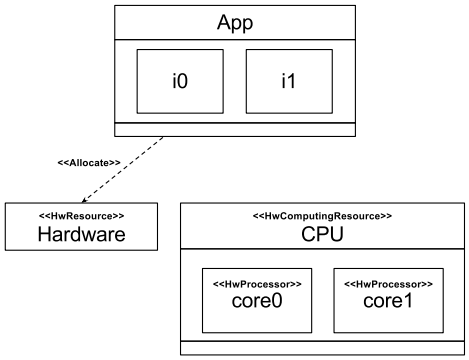
Figure 5.4(a) shows a static mapping from an object within *App* to a *HwResource*, object *i0* will always be instantiated as a hardware block.

An adjustable mapping is shown in Figure 5.4(b) where a single mapping from *App* to a *HwResource* defines that any unmapped objects within *App* will be instantiated as hardware blocks. In this example there are no other mappings resulting in both objects *i0* and *i1* being instantiated as hardware blocks.

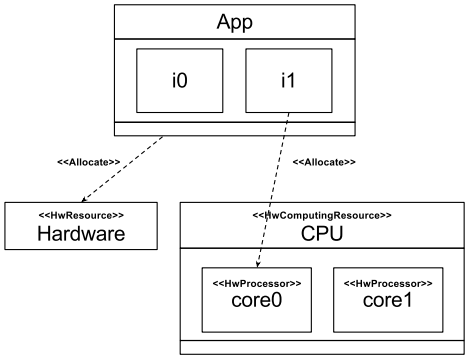
When mappings based around the processor and cores as discussed in chapter 4 are included in the UML model, as shown in Figure 5.4(c), which includes a static mapping from object *i1* to a *HwProcessor*, the adjustable mapping to the *HwResource* results in only object *i0* being instantiated as a hardware block. A final example, Figure 5.4(d), shows adjustable mappings from the application to both a *HwResource* and *HwComputingResource*. Both objects within *App* can be statically mapped to the hardware block or either core within the CPU.



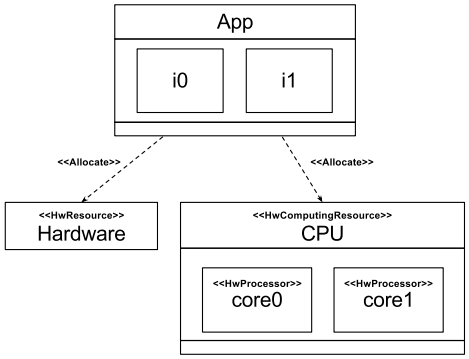
(a) Static Mapping - Object directly mapped to *HwResource*



(b) Adjustable Mapping - Application mapped to *HwResource*



(c) Static and Adjustable Mapping - Application mapped to *HwResource* and Object mapped to *HwProcessor*



(d) Adjustable Mapping - Application mapped to both *HwResource* and *HwComputingResource*

Figure 5.4: Examples of Static and Adjustable mapping used in ENOSYS project

The difference occurs in how FalconML treats mappings to a *HwResource* and a *HwProcessor*, with the behaviour of objects being generated in VHDL RTL or C code, respectively. The mappings shown in Figure 5.4 would not be recognised by FalconML as the UML model must be statically mapped PSUML to produce a valid output in both hardware and software.

The research conducted in this thesis around the adjustable mapping does not include investigations into hardware/software co-design as performed in the ENOSYS project. The reason for this is due to scope and implementation. The architecture and tool chain developed in this thesis consists of the multicore VLIW processor with research conducted around parallelism. The inclusion of custom hardware blocks to perform computation in parallel with the multicore processor increases the complexity of the implemented UML model. This results in difficulties in simulation and hardware execution as a full SoC must be simulated

along with the communication bus to retrieve heuristics with which to perform analysis.

The adjustable mapping notation presented in this thesis has been extended within the ENOSYS project to include the concept of hardware blocks. This extension to the notation provides the ENOSYS project a process with which to create PIUML models for the target hardware/software co-design systems which can be explored by an automated tool. Along with this a transformation processes to create PSUML models from a PIUML model using these adjustable mappings within the ENOSYS project was generated and is being utilised by the DSE flow.

The concentration of this thesis on only the software domain enables conclusions to be drawn regarding both the modelling techniques and VLIW processor which are the contributions of this thesis.

5.3 Summary

This chapter introduced a European project which utilises contributions of the research and development performed in this thesis. The ENOSYS FP7 project presents a tool flow which uses UML as a high level input language to model applications destined for hardware/software co-design.

The contributions from this thesis utilised in this project are:

LE1 Tool Collection: The LE1 VLIW CMP forms the multicore processor of the target architecture and the tools developed are required for producing machine code to execute on the hardware processors. The XML configuration file which defines the LE1 structure is also a key part of the tool flow as the DSE tool modifies and creates new instances and configurations of the LE1 processors when performing exploration around the available architecture platform.

Adjustable Mapping: The design methodology of specifying a mapping at a higher level of abstraction to define adjustable mappings is used to allow single UML models to be explored without the requirement of modifying the initial model. The DSE also invokes the translations processes developed (see Appendix E) and uses the generated static mapping permutations to explore valid PSUML models mapping the original application across the hardware and software domains.

The use of these concepts and tools within the ENOSYS project presents a level of dissemination and displays that the research contributions of this thesis provide addition and improvement to real-world tool flow implementations.

6

Experiments

6.1 Chapter Objectives

This chapter presents investigations into both hardware POSIX Thread (PThread) management and novel UML notations presented previously within this thesis. Firstly, two soft-core processors which support the PThread library are introduced, the are used as comparison points with the LE1 VLIW hardware PThread implementation. The MicroBlaze and Leon3 processors are introduced in section 6.2 and an investigation in the overhead of utilising the PThread library to perform thread management is presented. Both processors support Interleaved MultiThreading (IMT) which time-multiplexes multiple threads on a single core/CPU; this is extended, in the case of the Leon3, to implement a Chip Multiprocessor (CMP) platform for a more direct comparison with the LE1 VLIW CMP.

A selection of benchmarks written in C was executed on the available architectures for various hardware configurations, ranging up to eight cores. The benchmarks are threaded using the PThread library at different levels and both fine and coarse-grained threading is investigated. The fine-grained threading includes multiple calls to the PThread library to perform tasks in parallel whereas the coarse-grained threading generates a single thread per available core/CPU and specifies which data should be computed within each active thread. The results of execution are evaluated in relation to a single core/CPU to compare the overhead of the PThread library used to perform thread management.

The coarse-grained threaded benchmarks are also executed using the CPUID threading method. The results from these runs are then compared with the PThread execution and the usability of this method discussed with respect to the PThread library implementations.

The novel UML notations are then presented and investigated. The main benefit of the wildcard multiplicity and adjustable mapping UML notations is to reduce application devel-

oper interaction with the UML modelling tool by producing a single platform-independent UML (PIUML) model that can be transformed to platform-specific UML (PSUML) models to utilise a statically configurable architecture. Due to the nature of UML modelling tools requiring the application developer to input and modify the behaviour and structure of a system through a Graphical User Interface (GUI) a method of quantifying this work has to be defined to allow comparisons and conclusions to be made regarding the possible benefits of the proposed UML notations. This is performed by capturing the mouse and keyboard interactions of the application developer when generating a UML model through the UML tool GUI. A custom program was developed to capture mouse and keyboard events performed by the application developer when generating both architecture specific models as well as utilising the UML design notations within a model. An overview of this process of collecting and evaluating this data is presented in section 6.5.1.

A selection of base UML models have been generated to perform comparisons of the proposed UML notation with respect to the standard process of manually modifying UML models to fully utilise a modifiable architecture. The underlying behavioural of the UML models are kept constant across the various benchmarks with the structure being modified to capture configurable architectures and to specify parallel sections of an application. The UML systems generated for these comparisons are detailed in section 6.5.

Benchmarks created in both C and UML include parallelism defined statically at compile time. The number of threads explicitly created is dependent on the number of cores/CPUs available, this allows investigations in the benefits of performing parallel execution at a fine-granularity but does not touch on the issue of load-balancing. In some of the benchmarks the amount of work performed in each thread is not equal and a method of interleaving the computation to spread the load across all threads is used, this is described further in section 6.4.1. Ideally a dynamic scheduler able to balance the computation across all available threads would have been used for this purpose but was out of the scope of this research.

6.2 Comparison Platforms

Four programmable architectures are used to perform comparisons with the parallelism methodologies proposed in this thesis. The MicroBlaze [91] and Leon3 [99] are two scalar soft-core processors; the Leon3 also has a multiprocessor configuration, named the Leon3MP. Finally the LE1 VLIW CMP as introduced in chapter 4. All architectures are synthesised for a Xilinx ML605 FPGA development board.

The MicroBlaze is a scalar, soft-core processor available from Xilinx [100] and is included in the Xilinx XPS tools used for the FPGA device. The processor is highly configurable and

can be optimised for either area or performance. The MicroBlaze used in this study is the performance optimised design consisting of a five stage instruction pipeline. This processor includes PThread library support through the inclusion of the Xilinx Kernel which is used to perform timing and context switching through IMT.

The Leon3 is a soft-core microprocessor based on the SPARC-V8 RISC architecture available through Gaisler Research [101]. The Leon3 is a single core processor which can also be synthesised in a multiprocessor configuration, named the Leon3MP.

Both the single and multiprocessor configurations are used as comparison points. The Leon3 includes a thread library developed by Florida State University (FSU-pthread) [102] which implements general PThread standards in the form of thread management, mutual exclusions and conditional variables. Similar to that of the MicroBlaze this is an IMT implementation and uses context switching to execute multiple threads. The PThread library does not support the Leon3MP, however, an implementation has been developed to provide similar thread management functionality to that in the LE1 CMP.

The Leon3MP system allows up to sixteen CPUs to be instantiated within a system; there are some limitations in this architecture which require CPU0 to be the master CPU, all other CPUs can be activated at any time through the use of an interrupt register. However, if any CPU exits or throws an exception the whole system hits a software breakpoint which ceases all execution. Using a set of global arrays and data structures a method of performing thread creation and synchronisation across the multiple CPUs has been implemented. A subset of the PThread library operations are implemented (*pthread_create*, *pthread_exit* and *pthread_join*) to make this Leon3MP threading implementation similar to that of the LE1 CMP PThread hardware implementation. At the start of execution a single CPU (CPU0) is active. This CPU uses system configuration registers to ascertain the number of available CPUs within the current system. Using this information a global state table is initialised; this includes the state of each CPU along with function and argument pointers. The other available CPUs in the system are then started. However, they do not execute this initialisation routine and instead are directed to a function where they poll the state table entry relating to their CPU identifier. On the execution of a thread create operation the global state table is used to locate a currently inactive CPU; once found the global state of the selected CPU is updated to a busy status and the function and argument pointers are initialised based on values passed to the create operation. Once a function pointer is set the selected CPU begins execution of that function and once complete the CPU updates the global state table, to allow subsequent threads to be allocated to it and returns to the polling loop. This global state table allows PThread operations to poll the states and either send tasks to an available CPU or wait for a specified CPU to finish execution. This very basic functionality allows the implementation of a *pthread_create* operation which initialises

Table 6.1: Processor configurations used within the experiments in this chapter.

Processor	**Number of cores**	**Threading Library**	**Threading Type**
MicroBlaze	1	PThread	Interleaved
Leon3	1	PThread	Interleaved
Leon3MP	up to 16	custom PThread subset & CPUID	Chip-Multiprocessing
LE1 CMP	up to 8	hardware PThread subset & CPUID	Chip-Multiprocessing

required threads on separate CPUs within the system and a *pthread_join* operation which checks the global status of all CPUs to perform synchronisation.

This configuration allows the comparison of the LE1 CMP hardware PThread implementation with a CMP system able to execute the same benchmarks. The modifications to the Leon3MP initialisation along with the implementation of global state and the subset of PThread operations are included in [93].

Also available within the Leon3MP system is some custom assembly to return the current processor identifier; this allows a comparison with applications executing on the LE1 CMP in CPUID mode with the Leon3MP and is also used to only allow CPU0 to perform the initialisation routine in the PThread library implementation. A list of the available processor systems and their configurations is depicted in Table 6.1. These configurations form the basis of comparison data into parallelism in the LE1 CMP using both a hardware PThread and the CPUID technique.

The PThread libraries available for the MicroBlaze and Leon3 provide a valid comparison point as both implementations include a full PThread library on readily available soft core processors. The main difference between these and the LE1 and Leon3MP PThread implementation is that the latter provides CMP parallelism and the former IMT. The LE1 and Leon3MP however require a larger portion of the FPGA fabric due to multiple cores/CPUs instantiated. The development of the software PThread library subset for the Leon3MP system allows a more direct comparison point as this, similarly to the LE1, supports CMP.

The execution time of each application is collected from simulations and execution on hardware in order to perform comparisons between the different processors and library implementations as well as investigate the speed up of instantiating extra cores to perform execution. All software PThread implementations require initialisation prior to using PThread operations; this setup time is not included within the extracted execution time. This allows the time spent performing the computation for each benchmark to be the main focus of the

results generated leading to conclusions into the overhead of the alternative implementations of thread management.

The following sections make use of the architectural platforms introduced. Initially the timing overhead of using the PThread library to perform simple thread creation and synchronisation is investigated across all platforms and then benchmark applications are executed to perform a more in-depth comparison.

6.3 Micro-Benchmarks

The PThread library operations include a time penalty for the management of threads. To perform an initial investigation in the use of software threading libraries versus the proposed LE1 CMP hardware implementation three tests were created. These tests are executed on all platforms to enable comparisons between the alternative PThread implementations. The tests perform thread creation and synchronisation and use available timing mechanisms to retrieve either the number of clock cycles or absolute times to perform the specified task.

The three tests performed investigate the overhead of creating and synchronising threads in the alternative systems. These figures can be used to evaluate the granularity of the parallelism which can be extracted from an application. The three tests are described below:

A) **Create**

The results of this test show the time elapsed between the master thread issuing a *pthread_create* operation and the slave thread beginning execution. A timer is started prior the the *pthread_create* operation and stopped within the slave thread.

B) **Join**

The elapsed time from the slave thread completing to the master thread being aware of the termination of the slave thread through a *pthread_join* operation is calculated in this test. A timer is started and the slave thread exits; the master thread performs a *pthread_join* operation and, once it is made aware of the termination of the slave thread, the timer is stopped from the master thread.

C) **Create & Join**

The slave thread in this test executes an empty function and the results show the accumulated time taken for both previous tests. A timer is started, the master thread performs a *pthread_create* operation directly followed by a *pthread_join*; once the *pthread_join* is completed the timer is stopped.

The PThread libraries used on the processor platforms is as follows:

- **MicroBlaze**: Software PThread implementation, Xilinx Kernel PThread library.
- **Leon3**: Software PThread implementation, Florida State University PThread library.
- **Leon3MP**: Custom PThread implementation, see Appendix G.
- **LE1 CMP**: Hardware PThread implementation.

The MicroBlaze figures are generated using the *XPS_Timer* module and library calls [103]; this provides the number of clock cycles taken to perform a given task. The Leon3 and Leon3MP figures are extracted using the *clock()* function available through the sparc-elf-gcc compiler and Leon hardware. This function returns the number of microseconds since the start of execution. This method provides figures which include a margin of error; the higher the frequency of the implementation the higher the possible error. In this study frequencies of 75 and 120 MHz were used, resulting in possible errors of ± 37.5 and ± 60 cycles, respectively. Finally, the LE1 results are generated through the Instruction Set Simulator (Insizzle) using the *clock()* function which returns the number of clock cycles since the start of execution. The simulator executes a worst-case cycle-accurate simulation, in terms of memory access and instruction fetching as described in the chapter 4. The LE1 used in this test is a dual context system with both contexts consisting of a single hypercontext with two issue width, two ALUs, two multiply units and a single LSU channel. A common shared DRAM is also included with a single memory bank.

Due to the non-deterministic nature of the software implementations of the PThread library and the context switching required in the IMT systems the tests were executed 1000 times and the mean time taken to execute used. For each of the systems investigated the creation and synchronisation times have been extracted and are shown in Table 6.2 with derived figures displayed in italics. These derived figures convert clock cycles to time taken in μseconds to enable easy comparisons between the different methods of heuristic extraction across the available platforms.

All tests are carried out as best case scenarios consisting of at most two threads being active at any time in the user space; a single executing master thread creates a single slave thread and then performs a synchronisation on this slave thread.

Table 6.2: Execution time of PThread libraries available on various architectures.

Processor	**Implementation**	**Create** (μ **Seconds**)	**Join** (μ **Seconds**)	**Create & Join** (μ **Seconds**)
MicroBlaze	IMT @ 75MHz	*29.133*	*18.907*	*47.227*
MicroBlaze	IMT @ 150MHz	*21.693*	*12.913*	*34.193*
Leon3	IMT @ 75MHz	208.000	4788.000	4987.000
Leon3	IMT @ 120MHz	153.000	4844.000	4989.000
Leon3MP	CMP @ 75MHz	2.000	1.000	3.000
Leon3MP	CMP @ 120MHz	2.000	0.000	2.000
LE1	CMP @ 50MHz	*0.640*	*0.640*	*1.280*
Leon3MP	CMP @ 75MHz based on MicroBlaze	29.000	18.000	47.000
Leon3MP	CMP @ 120MHz based on MicroBlaze	18.000	11.000	30.000
Leon3MP	CMP @ 75MHz based on Leon3	206.000	4781.000	4988.000
Leon3MP	CMP @ 120MHz based on Leon3	129.000	2988.000	3118.000

The MicroBlaze, Leon3 and Leon3MP systems were executed at both 75MHz and at the highest frequency available for each configuration. The 75MHz results are used to perform comparisons and the 150MHz and 120MHz are shown to provide the shortest PThread management times possible. The LE1 system is simulated at 50MHz in order to incorporate up to eight cores (used for later benchmarks) and the SoC required to perform execution on a Virtex 6 FPGA device.

Unlike the software implementations the times gathered from the hardware implementation of the LE1 PThreads were always consistent due to the management being performed in hardware without any conflicts for computational resource. This resulted in an overall time of 1.280 μs to create and synchronise with the created thread.

As can be seen from the results the MicroBlaze and Leon3 implementations which perform the thread management in software using IMT perform a factor of 25 and 3,800 times worse than the LE1 hardware CMP implementation, respectively. This is due to only a single thread able to execute at any time, resulting in the interleaved execution of the master and slave threads on the single CPU. This is due to only a single thread able to execute at any time, resulting in the interleaved execution of the master and slave threads on the single CPU. A thread performing execution management is also active (Xilinx Kernel on the MicroBlaze and FSU-pthread Kernel on the Leon3) leading to three threads of execution being interleaved in the benchmarks. This results in higher latencies between the creation and synchronisation routines as they are not executed immediately and have to wait until they are scheduled to execute. This differs from the CMP implementation where each thread is executed on a separate CPU with no contention for execution.

It is interesting to point out that due to the *clock()* function of the Leon3MP system calculating in μs and the speed of which the PThread functions tested performed the results are not exact. As a result of this the Leon3MP implementation clocking at 120MHz displays a time of 0 seconds for the Join test. This is one of the drawbacks of using the Leon3MP timing method instead of being able to extract cycle counts directly from the hardware processor(s).

The MicroBlaze and Leon3 processors execute their PThread libraries under IMT. A single core/CPU performs execution with all active threads time multiplexed on the single core. Due to this, it is not valid to perform comparisons between the MicroBlaze or Leon3 with CMP threading implementations, such as the Leon3MP and LE1. The execution time of IMT is not reduced with the inclusion of more threads, instead more computation in the single core is required as thread management must be performed to interleave the computation of each thread onto the single core.

In order to "simulate" the MicroBlaze and Leon3 performing CMP threading their thread create and join times recorded in this micro-benchmark study are replicated in the Leon3MP

threading library. Stalls were added to make the latencies of the Leon3MP thread create and join operations match those of the MicroBlaze and Leon3 processors and enable investigations into the advantages of performing thread management in hardware. The bottom four rows of Table 6.2 display the Leon3MP system including these stalls in the thread create and join operations to match the times the MicroBlaze and Leon3 processor systems reported.

Parallel threading can lead to a decrease in execution time, although this is only true when the amount of work performed in the created thread is greater than the amount of work required to perform creation and synchronisation of the thread. If this is not true the possible performance increase by performing a task in parallel is cancelled out by the thread management overhead. Shorter thread creation and synchronisation times enable exploration into the granularity of parallelism. In most real-life applications it can be difficult to identify sections of code which can be executed out of order and have no data-dependencies. With the LE1 hardware PThread implementation as long as the section of the application being parallelised executes for more than 1.280 μs the use of the thread library results in an overall execution time decrease. Compared with that of 34.193 and 4987.000 micro seconds on the MicroBlaze and Leon3 systems, respectively.

6.4 C benchmarks

This section introduces a set of C applications used to perform comparisons across the alternative hardware platforms and software implementations of PThreads. The applications have been chosen for their ease of threading due to the limited subset of the PThread primitives they use (only create, join and exit).

The first section presents applications which use a subset of the PThread library to perform explicit Thread Level Parallelism using both fine and coarse-grained thread creation and synchronisation. This is performed to highlight the differences between the alternative implementations of thread management and how the latencies of the thread creation and synchronisation operations affect execution times. This is followed by a selection of the PThread applications modified to use the CPUID instruction; this implementation differ as there is no predefined synchronisation methods. This requires either the use of hand written, application specific synchronisation techniques or the use of fully data-independent algorithms.

All results are extracted from executing the benchmark applications on four platform configurations:

A) Leon3MP: with custom software PThread implementation & CPUID.

B) Leon3MP: with software thread management modified to simulate the thread creation and synchronisation times of the MicroBlaze.

C) Leon3MP: with software thread management modified to simulate the thread creation and synchronisation times of the Leon3.
D) LE1 VLIW CMP: with custom hardware PThread implementation & CPUID.
The CMP architectures are evaluated with a varying number of physical cores/CPUs (from one up to eight) and the benchmarks themselves perform alternative methods of threading.

6.4.1 POSIX Thread

These benchmarks are threaded using PThread library primitives at two levels of granularity: fine and coarse-grained. In the coarse-grained category a single *pthread_create* operation is executed per available core/CPU at the beginning of the application. Each thread then performs the computation tasked to it and finally *pthread_join* operations are used to synchronise the active threads. In each case the number of *pthread_create* and *pthread_join* operations executed is equal to:

$$(Number\ of\ Cores - 1) \tag{6.1}$$

This allows the benchmarks to be threaded without the overheads of utilising the PThread library management interfering with the overall execution times, or more truthfully, with only a negligible amount of overhead.

The fine-grained threaded benchmarks are designed to use the PThread library as much as possible. Within loops and other data independent sections of code multiple threads are created and then synchronised. Where the number of *pthread_create* and *pthread_join* operations executed is equal to:

$$(Size\ /\ Number\ of\ Cores) * (Number\ of\ Cores - 1) \tag{6.2}$$

Size is different across the benchmarks and will be defined within the descriptions of each. This aggressive use of the PThread operations demonstrates how using a threading library with long latency leads to a degradation in overall execution time when the computation being performed within a thread takes less time than the overall management time of the thread.

Each benchmark is executed across multiple configurations of the architectures, where the number of available cores/CPUs is modified from one up to eight. Each benchmark is written to modify the number of threads created based on the number of physical cores/CPUs available as shown in Equations 6.1 and 6.2. For example, the coarse-grained benchmarks will always create a thread of execution for each core/CPU within the architecture. Each thread will then compute an equal, if possible, fraction of the workload. This implementation, in best case scenario should lead to a linear decrease in the total execution time. The

fine-grained benchmarks differ in that they are not threaded at the highest level; at certain points within the code multiple threads are created to utilise available resources within the architecture. These threads compute their fraction of the workload and are then synchronised, this may occur multiple times within a loop, resulting in the PThread operations being called numerous times within a single benchmark. In a best case scenario this would lead to a speed up as described in Amdhal's Law which takes into account both serial and parallel sections of a program.

In both cases the best case scenario does not take into account the overhead of requiring thread management to perform creation and synchronisation of threads. The following benchmarks and execution results are used to perform an investigation into the effect of this on execution time.

6.4.1.1 Mandelbrot Set

The Mandelbrot Set is a mathematical set of points whose boundary is a distinctive and easily recognisable two-dimensional fractal shape named after the mathematician Benoit B. Mandelbrot [104, 105].

The Mandelbrot Set was chosen as a benchmark as it is highly parallel and truly data-independent. Each point within the set can be calculated in parallel based on a predefined magnification setting and origin co-ordinates. A second set of co-ordinates is then passed to calculate the pixel value at a specific position. Due to this the Mandelbrot code is used through all of the example parallel techniques presented in this thesis. In this PThread example a 160 by 480 Mandelbrot fractal is computed using a 10 colour palette and a maximum iteration value of 100. The computation required is not equal per pixel; with the maximum iterations per pixel set at 100 there is a possible difference of x100 depending on the pixel position within the image. Originally the work load was split into contiguous sections of the image which resulted in large differences between the computation performed by each thread. As a result of this the computation is split on a row basis throughout the output image. This results in an interleaved output with the computation performed across each thread more equally balanced.

In this benchmark the coarse-grained threading is performed by the master thread generating a new thread within each available core/CPU. This thread is then tasked to compute a section of the output image. The speed up of including extra threads on the available platforms is shown in Figure 6.1. The cycle counts extracted while executing this benchmark for the available platforms are shown in Table I.1 along with the speed up of adding extra cores/CPUs and threads compared with the single core/thread execution in Table I.3.

The fine-grained threading uses multiple PThread operations; the number of instances is derived from Equation 6.2 where *Size* is equal to the number of pixels within the output

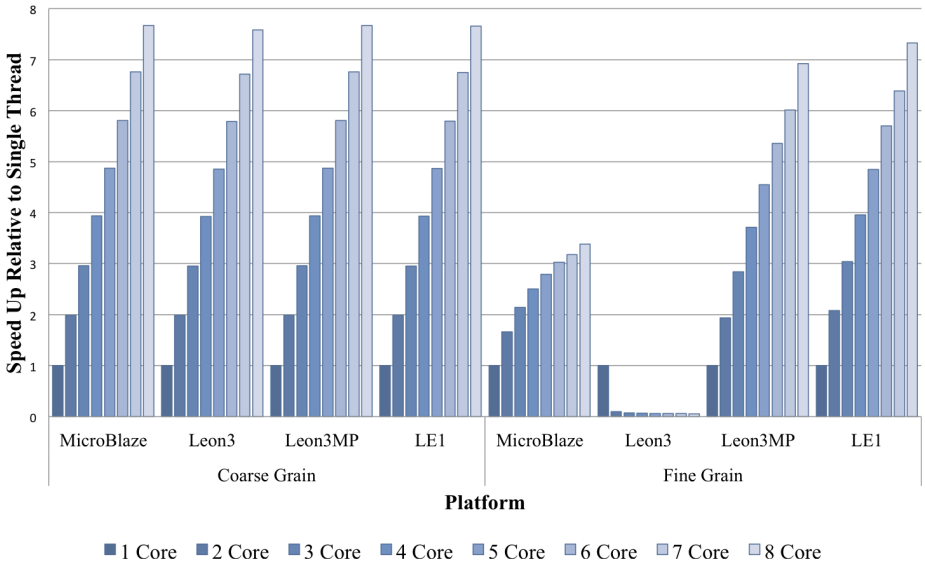


Figure 6.1: Mandelbrot PThread Execution Results

image which in this case is 76,800 (based on a 160 x 480 image). The master thread iterates through the full set of output pixels and creates new threads for each. It then computes a pixel value itself and synchronises with all other threads. This is performed in a loop until all output pixels have been generated. This results in zero PThread operations being executed in the single core/CPU benchmark and up to 67,200 PThread operations pairs (both *pthread_create* and *pthread_join*) in the eight core/CPU benchmark. The speed up of including extra threads on the available platforms is shown in Figure 6.1. The cycle counts extracted while executing this benchmark for the available platforms is shown in Table I.2 along with the speed up of added extra cores/CPUs and threads compared with the single core/thread execution in Table I.4.

To show that overall execution time increases when using IMT this benchmark was executed on the MicroBlaze and Leon3 processors. This is the result of only a single computation unit performing execution with each thread and the thread management being time multiplexed. There is no benefit in this method of threading and an overall overhead is incurred by this multiplexing. Tables I.1 and I.2 show the results obtained from execution on these platforms. As can be seen, adding additional threads on the MicroBlaze results in a worst case increase in overall execution time of 0.029% compared with the single thread execution. The Leon3 performs worse, due to the larger latencies of the *pthread_create* and *pthread_join* operations, with a worst case of 6.321% increase in execution time compared with single thread execution of the benchmark. The fine-grained benchmarks display an

even worse trait; the number of PThread operations executed made it unable to retrieve heuristics from the Leon3 platform. The cycle counts from the MicroBlaze were captured from this fine-grained benchmark and showed an execution time increase of 16.085% compared with a single thread. These results show what was discussed previously. Due to only having a single core/CPU available for computing the benchmark the IMT implementations result in an increased overall execution time. The use of IMT in these instances is well suited to performing context switching to check for user inputs and display outputs, rather than the parallel computation which is being performed by the LE1 CMP and Leon3MP.

6.4.1.2 JPEG Decode

This JPEG decoder is based on a small C implementation called NanoJPEG [106]. This implementation was modified to remove dynamic memory allocation and file I/O to enable it to be executed on the embedded processors used for comparisons in this study. A 64 x 64 pixel JPEG image of Lena was used as an input data set for this benchmark.

The JPEG decoder is a good example of a real-life application. Due to the nature of decoding within the JPEG specification there are data-dependencies between sections of the encoded image; macro blocks are used within the decode process where pixel values being computed may rely on the computation of other pixels within the same or adjacent macro blocks.

The coarse-grained benchmark takes the form of decoding eight JPEG images, with a JPEG decoder being instantiated in each thread. These eight instances all decode a separate image which allow the computation to be parallelised across the available cores/CPUs within the architecture. The speed up of including extra threads on the available platforms is shown in Figure 6.2. The execution cycle counts over the multiple cores/CPUs is shown in Table I.5 along with the speed up compared with a single core/CPU in Table I.7.

A single function was found to be independent of order; at the end of decoding a single macro block an inverse discrete cosine transform is performed on each column within a macro block (*colIDCT*). This function is called eight times in a loop for each column within the macro block. This loop was modified to split the work over all available cores and then use a unique identifier along with the total number of parallel threads to specify whether or not to perform the computation. This resulted in the fine-grained threading benchmark, using a slightly modified Equation 6.2, where "*Size / Number of Cores*" is set to the number of macro blocks within the image. This fine-grained threading is performed on a single 64 by 64 image being decoded resulting in 96 macro blocks and up to 672 PThread operations pairs (both *pthread_create* and *pthread_join*) in the eight core/CPU benchmark. As the amount of computation performed within the function is small it is a perfect example to show the advantage of having a threading library with a small time overhead for thread management.

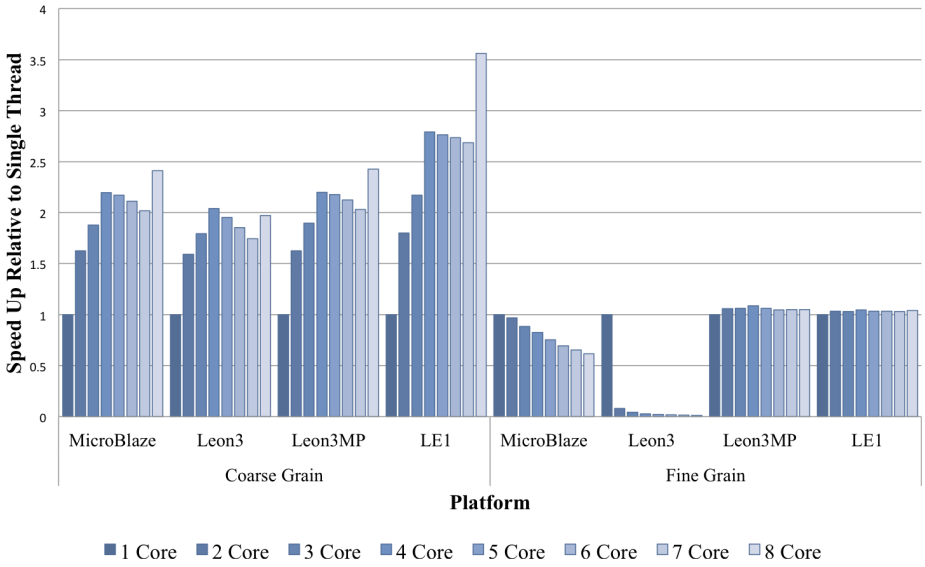


Figure 6.2: JPEG Decode PThread Execution Results

The speed up of including extra threads on the available platforms is shown in Figure 6.2. The results for the available platforms is shown in Table I.6 along with the multicore speed up with respect to single core/CPU execution in Table I.8.

6.4.1.3 Sobel Filter

The Sobel Filter is an image processing algorithm used in edge detection. The input is an image and the output is a greyscale image displaying "edges" within the image, this type of filtering would be used as the first stage in feature detection to preprocess images. Each pixel within the input image is used in calculations with its surrounding pixels and two masks are used to find horizontal and vertical transitions to detect edges and corners. This algorithm requires a lot of data reading, however, the algorithm itself has no data-dependencies and each pixel value in the output image can be calculated independently. The inputs are an image of size 640 by 480 and two 3 by 3 item mask arrays which are used for each pixel.

The coarse-grained threading is performed based on Equation 6.1, resulting in-between zero and seven PThread operation pairs (*pthread_create* and *pthread_join*). This example is split in an interleaved fashion similarly to the Mandelbrot Set; each thread executes a full row of the input image and then moves down a set number of rows. The speed up of including extra threads on the available platforms is shown in Figure 6.3. The cycle counts extracted while executing this benchmark are shown in Table I.9 along with the multicore speed up compared with the single core/CPU execution in Table I.11.

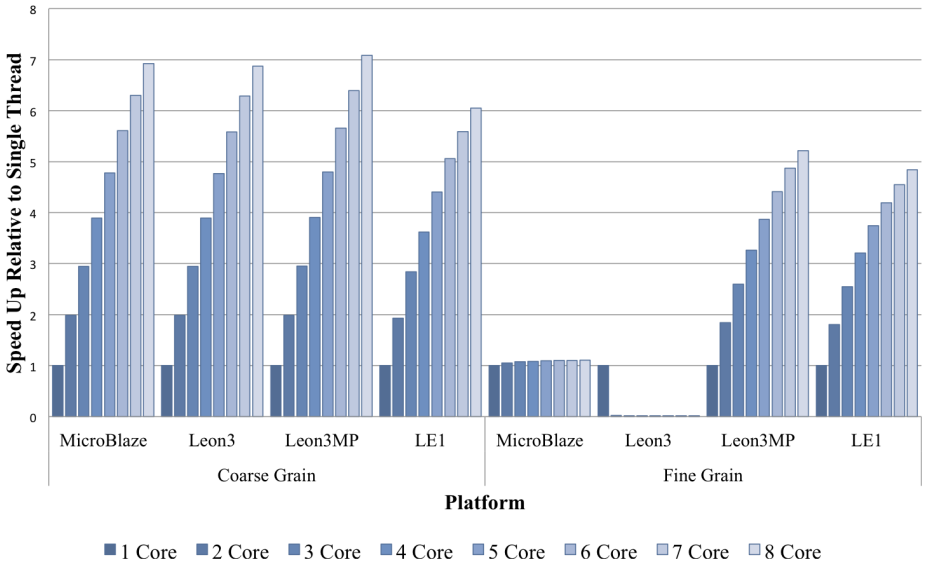


Figure 6.3: Sobel Filter PThread Execution Results

The fine-grained threading is also performed similarly to the Mandelbrot Set benchmark. The only difference being the number of PThread operation pairs executed due to the difference in image sizes. *Size* is defined using a 640 by 480 pixel image, however, the actual algorithm ignores a single pixel border which results in 304,964 pixels (based on a 638 x 478 image). This leads to up to 267,768 *pthread_create* and *pthread_join* operations when eight cores/CPUs are available. The speed up of including extra threads on the available platforms is shown in Figure 6.3. The cycle counts extracted while executing this benchmark are shown in Table I.10 along with the multicore speed up compared with the single core/CPU in Table I.12.

6.4.1.4 Data Encryption Standard

The Data Encryption Standard (DES) was developed in the early 1970s and published as an official standard in 1977. It has since been surpassed by other such standards as it is considered insecure due to the key size being small enough to be susceptible to brute force attacks. The algorithm takes a 64bit key to generate a set of 16 48bit sub-keys which are used to encrypt 64bit blocks of plaintext, through a 16 stage Feistel Network, into 64bit blocks of ciphertext. This benchmark was chosen due to the possibility of parallelising the DES algorithm across multiple threads with each 64bit plaintext and ciphertext pair being data-independent from any other block.

Similarly to the previous benchmarks this is parallelised using both the coarse and fine-

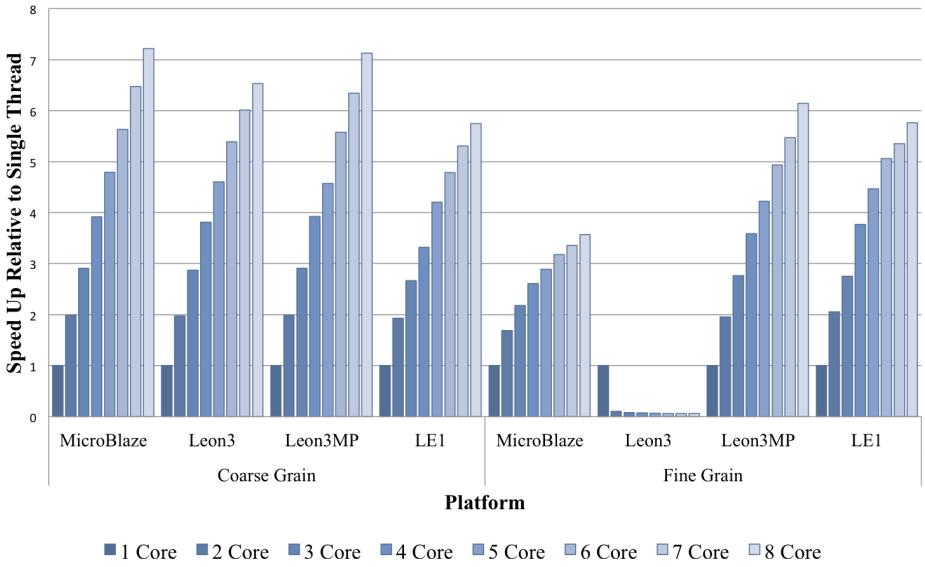


Figure 6.4: Data Encryption Standard PThread Execution Results

grained threading. The DES benchmark processes 64KB of data, resulting in 8,192 blocks of 64bits to encrypt. In the coarse-grained benchmark a thread is instantiated within each available core/CPU and then works across the input data using its knowledge of the total number of threads and the size of the data to be encrypted. The speed up of including extra threads on the available platforms is shown in Figure 6.4. The cycle counts extracted while executing this benchmark for the available platforms is shown in Table I.13 along with the CMP speed compared with the single core/CPU execution in Table I.15.

In the fine-grained threading the maximum number of PThread library operations pairs executed is 7,168, based on 8,192 blocks. The speed up of including extra threads on the available platforms is shown in Figure 6.4. The cycle counts extracted while executing this benchmark for the available platforms is shown in Table I.14 along with the speed up of added extra cores/CPUs and threads compared with the single core/CPU execution in Table I.16.

6.4.1.5 Execution Results

The above benchmarks were executed across two of the architectures defined in section 6.2. The first architecture is the Leon3MP which included eight CPUs, a 32KB instruction cache and a 16KB data cache per CPU and a system clock of 75MHz. The second architecture is the LE1 VLIW CMP clocked at 50MHz consisting of eight homogeneous contexts, each comprised of a two issue width VLIW with two ALU, two multiply units and a single

Load/Store channel. A common shared DRAM is also included split into two memory banks.

The benchmarks were compiled through both sparc-elf-gcc and the LE1 Tool Collection with optimisation levels -O3 using simulated floating point support where required.

Due to the overhead of the PThread libraries on both processors using IMT, retrieving the execution times for the benchmarks was troublesome as a result of timer registers overflowing and the way which timers were implemented. When clocked at 75MHz the MicroBlaze timer overflowed after 57 seconds, as a result of a 32bit register being used to track clock cycles. For the benchmarks executed in this study this was not sufficient. To enable executions to be timed a separate process, running as a thread, was required to track this overflow and increment a second counter. Introducing this extra thread would add extra computation and interleaving of threads, resulting in the MicroBlaze performing more work than the other processors. The Leon3 timers in IMT resulted in inconsistent numbers being returned. It was discovered that the timer only incremented for CPU0 while it was executing resulting in the interleaving of computation to other threads not being counted. Due to available threads not necessarily performing the same amount of work this resulted in unusable data.

As a result of these timer limitations in the MicroBlaze and Leon3 these architectures are not included. Instead the Leon3MP executing the custom PThread library to simulate the *pthread_create* and *pthread_join* times recorded in the micro-benchmarks (Table 6.2) for the MicroBlaze and Leon3 were used. This enables investigations into how the latencies of the thread management operations affects overall execution time of the benchmarks. Removing the work required to multiplex threads on a single processor when threading using IMT also enables more direct comparisons to be made as all implementations are based on CMP architectures. This results in timing differences being due to the different latencies of the thread management operations, which is of interest in this study.

Each bar chart below (Figures 6.5 - 6.9) shows all benchmarks being executed on a single architecture and configuration, with each bar displaying the speed up for the given number of cores/CPUs relative to the same benchmark being executed on single core/CPUs without the use of PThread library operations.

The results from running the above benchmarks on the Leon3MP architecture using the custom PThread library are shown in Figure 6.5. The Mandelbrot Set, Sobel Filter and DES benchmarks show a near linear speed up when including extra CPUs in both the fine and coarse grained threaded mode. There is a slight difference in the implementations which utilise the fine-grained parallelism; for example directly comparing the two, the speed up in the fine-grained benchmark is a factor of 0.05 less than the coarse-grained benchmark. This is due to the increased number of PThread library calls and thus the increased overhead incurred for thread creation and synchronisation.

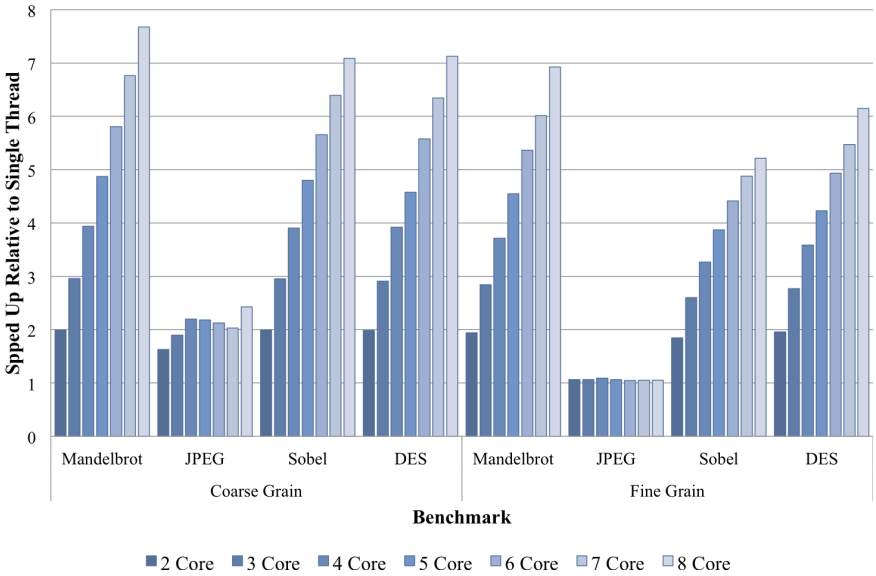


Figure 6.5: Benchmarks run on Leon3MP architecture displaying speed up relative to a single CPU when extra CPUs are available to perform computation.

Table 6.3: Example of how JPEG images (a - h) are distributed across available CPUs in execution of JPEG coarse Grain benchmark.

Benchmark Threads	CPU							
	1	2	3	4	5	6	7	8
4	a, e	b, f	c, g	d, h	-	-	-	-
5	a, f	b, g	c, h	d	e	-	-	-
6	a, g	b, h	c	d	e	f	-	-
7	a, h	b	c	d	e	f	g	-
8	a	b	c	d	e	f	g	h

The JPEG Decode benchmark displays different behaviour and is a result of the data-dependent computation within the JPEG decoder. In the coarse-grained JPEG benchmark a total of eight separate JPEG images are decoded. Each image must be decoded in its entirety on a CPU and due to this the full workload of eight images is not able to be split equally across all CPUs. This is evident from the data displayed in the bar chart as the execution on 5, 6 and 7 CPUs perform worse than on 4 CPUs. This is due to available CPUs idling as a result of the uneven split of the workload over the available architecture. This split of execution over the available CPUs is displayed in Table 6.3. This shows the JPEG images (a - h) and which CPU performs its decode; as can be seen in the execution with 5, 6 and 7 CPUs the distribution of the workload is not equal.

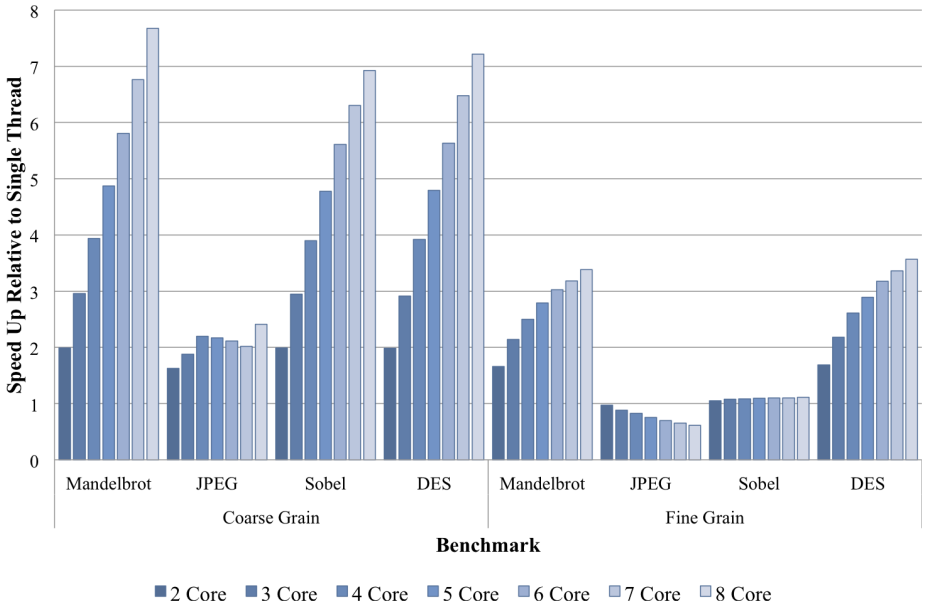


Figure 6.6: Benchmarks run on Leon3MP architecture simulating the *pthread_create* and *pthread_join* operation times of the MicroBlaze PThread library. Graph shows speed up relative to a single CPU when extra CPUs are available to perform computation.

The fine-grained JPEG benchmark displays a similar trait to that of the coarse-grained JPEG where 5, 6 and 7 CPUs perform worse than the 4 CPU implementation. This is the result of a similar uneven split of computation across the CPUs. Within the *colIDCT* function being parallelised a loop iterates 8 times over a macro block. This benchmark also displays that the amount of work performed within the created threads very closely matches that of the overhead of the PThread operations and so the inclusion of the thread management does not offer great benefits. All execution runs including the PThread creation and synchronisation operations show a speed up of between 0.919 and 0.956 times compared to a single CPU. Taking into account the FPGA resources required to implement up to eight CPUs versus a single CPU and the increase in performance gained it would be more favourable to perform the execution on a single CPU.

The execution speed up of the Leon3MP including stall cycles to simulate the PThread library calls of the MicroBlaze is shown in Figure 6.6. The results of the coarse-grained threading of all benchmarks, are consistent with those of the Leon3MP execution results. This is due to the negligible overhead of the thread management in comparison to the amount of computation performed within the created threads. This overhead is shown to have an effect on the fine-grained benchmarks on this architecture. The Mandelbrot Set and DES

benchmarks display a similar trend as previously seen on the Leon3MP, however, the speed up factor is not as great due to this overhead. Alternatively the Sobel Filter results display a minimal execution speed up with the addition of extra CPUs. This is due to the amount of work being performed being very close to the total overhead of the thread management and thus resulting in a stalemate where creating new threads does not provide any great speed up.

The JPEG Decode results display a negative trend to that seen previously. This again is due to the work being performed in the generated threads compared to the overhead of creating and synchronising those threads. In this case the thread management time outweighs the time taken to perform computation and thus the inclusion of extra threads results in a degradation of total execution time.

Executing the benchmarks on the Leon3MP with the custom PThread library while simulating the thread management times of the Leon3 FSU PThread library requires two graphs. This is due to the performance difference between threading at the two levels of granularity. The coarse-grained threading is shown in Figure 6.7 where similar trends are seen as in the previous results. The fine grained threading is shown in Figure 6.8 where in all benchmarks the inclusion of extra threads results in a significant increase of execution time. This is due, once again, to the overhead of using the PThread library to perform thread management. In the Leon3 PThread library, as shown in Table 6.2 the times recorded for this implementation were a factor of 100 and 1600 times greater than that of the MicroBlaze and custom PThread subset implementation, respectively. Due to this large overhead all benchmarks suffer when fine-grained threading is implemented.

The final architecture on which the PThread benchmarks are implemented on is the LE1 VLIW CMP using the hardware PThread support. As the PThread operation timings (shown in Table 6.2) are very close to those of the Leon3MP custom PThread library the results in the LE1 speed up shown in Figure 6.9 are very similar to that of the Leon3MP results. There are some fluctuations in the LE1 results and these are due to the nature of the VLIW instruction packing and fetching. There are times when the VLIW requires to be stalled to fully fetch an instruction packet, as the instruction fetch within the LE1 can take a maximum of three cycles to produce a complete instruction packet when the latter spans three IRAM lines. This can be seen in the fine-grain Mandelbrot Set results where the use of two threads takes 48% of the time of the single thread to complete execution. This is a greater than linear speed up which should not be possible, however, looking at the actual execution results this can be explained as follows: in the single thread run 13% of the total cycles were a result of the instruction fetch pipeline stalling to retrieve a full instruction packet. In the dual thread execution less than 6% of each threads execution cycles are due to this same effect. As a result of the parallel nature of the threads and each

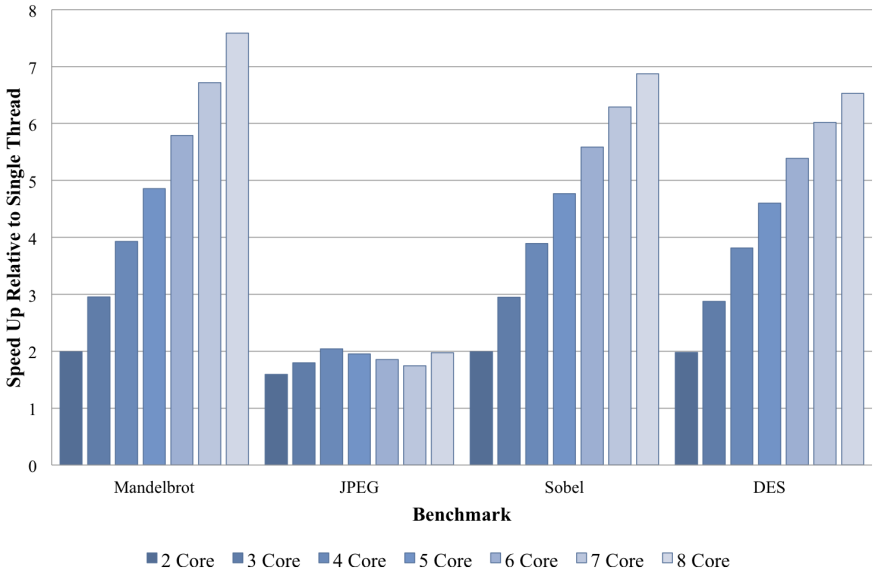


Figure 6.7: Benchmarks run on Leon3MP architecture simulating the *pthread_create* and *pthread_join* operation times of the Leon3 PThread library. Graph shows speed up relative to a single CPU when extra CPUs are available to perform computation.

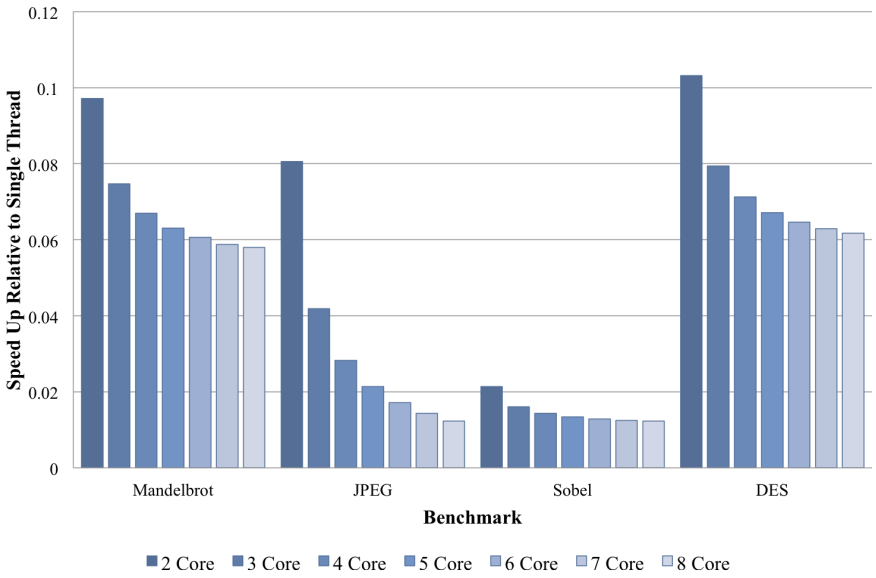


Figure 6.8: Benchmarks run on Leon3MP architecture with *pthread_create* and *pthread_join* times simulating that of the Leon3 PThread library. Graph shows speed up relative to a single CPU when extra CPUs are available to perform computation.

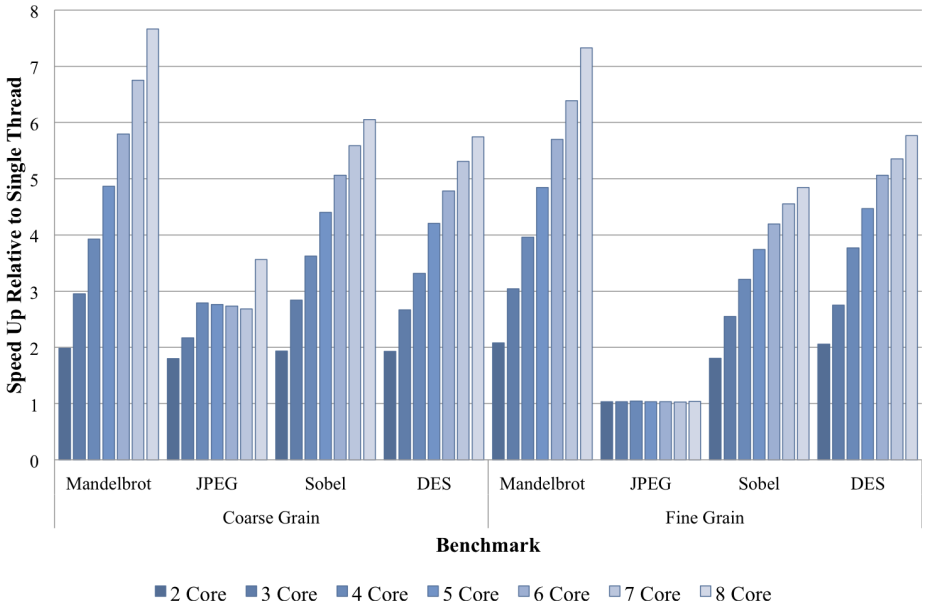


Figure 6.9: Benchmarks run on LE1 architecture showing speed up of multiple threads executing in parallel relative to a single thread of execution. Each bar represents the number of physical cores available for parallel execution, the number of PThread operations performed in each is described in section 6.4.1.

physical core having its own IRAM and instruction fetch unit all execution is performed in parallel resulting in the stalls being hidden as other cores are able to carry on executing. Another reason for the difference is down to the compilation and assembly stage, as each benchmark is different, the single thread code did not have any PThread calls in, resulting in the compiler being able to optimise the code differently. This in itself results in the instructions being packed differently which in turn can increase or decrease execution time.

6.4.2 CPUID

A selection of the benchmarks from the PThread implementation was modified to use another form of thread management available in both the LE1 VLIW CMP and Leon3MP architectures. The CPUID methodology enables thread parallelism with code being written explicitly to execute on each available core/CPU. In the PThread method a single "master" thread begins execution and creates and synchronises threads throughout the application. In the CPUID method all cores/CPUs begin execution simultaneously from the entry point of the application, with control flow being specified explicitly for each available core/CPU within the code being executed.

6.4.2.1 Benchmarks

The CPUID benchmarks use the same base code as the PThread benchmarks. All that is altered is the *main* function; this is changed to retrieve the executing cores/CPUs unique identifier. This unique identifier is then used to modify the control flow of the application to allow a single thread to perform initialisation and all others to simply compute their section of data. All CPUID benchmarks are parallelised at a coarse-grain, this is due to the lack of synchronisation operations as required in the fine-grained threading performed in the PThread benchmarks.

An example of the code used in the CPUID benchmarks is shown in Figure 6.10. Each core/CPU begins execution of the *main* function; each retrieves its own unique identifier and a single core/CPU is then used as a $MASTER$ which performs any initialisation tasks. While this is taking place all other cores/CPUs are waiting for the global variable *init* to be set before being able to perform their specified task. Once this variable has been set by the $MASTER$ core/CPU each calls the *cpuid_parallelism* function along with their own unique identifier. This function uses the unique identifier to set *status*, a global state table, used to store the status of all active cores/CPUS. This allows a basic form of synchronisation and can be used to check the status of all available cores/CPUs within the architecture. $SIZE$ is used to define the number of times the computation is to be performed; this is iterated over and this value is used along with the current thread $CPUID$ and total number of active threads to check whether the current core/CPU should perform computation. Finally the global state table used for synchronisation is modified to a $DONE$ status. This global status is used in these benchmarks to check computation is complete before calculating execution time and performing error checking.

The four benchmarks use the example code structure, shown in Figure 6.10, over different data sizes; the JPEG decoder is parallelised over a set of eight images which are decoded, so $SIZE$ in this case is set to 8 and the same pattern as shown in Table 6.3 is seen. For the Mandelbrot Set $SIZE$ is set to the height of the output image (160) which results in each thread computing single rows of the output image which are offset by the current $CPUID$. Similarly the Sobel Filter uses the height of the image to iterate through and split the computation across the available threads and finally the DES benchmark uses the number of 64bit messages to be encrypted as the quantum of parallelism.

6.4.2.2 Execution Results

The above benchmarks were executed on both the LE1 VLIW CMP and the Leon3MP architectures. In each benchmark a single core/CPU takes control of the system using a global state table to perform synchronisation. All other active cores/CPUs wait until

```
volatile int init = 0;

void main(void) {
        int id = getCPUID();

        if(id == MASTER) {
                /* Perform initialisation tasks */
                init = 1;
        }

        while(!init) { }

        cpuid_paralleism(id);
}

void cpuid_parallelism(unsigned int id) {
        int i;
        status[id] = BUSY;
        for(i=0;i<SIZE;++i) {
                if(!((i + id) % THREAD)) {
                        /* Perform Computation */
                }
        }
        status[id] = DONE;
}
```

Figure 6.10: Code example of using CPUID operation to define parallelism within an application.

this master has completed system initialisation. The master is tasked with setting up any input data which is required by the benchmark, setting the global state table for each other core/CPU then starting a timer. Each core/CPU then performs a selection of the overall computation using a unique identifier to specify what should be performed. Once complete each core/CPU sets its associated global state to a completed status. The master is also used to perform computation; once completed it is tasked with waiting for all other cores/CPUs to complete execution by polling the global state table. Once each is finished the master stops a timer and checks for correct execution by comparing against a known correct result.

This results in the time recorded only including the actual parallelised computation time.

Similarly to the PThread benchmarks the cycle counts extracted from each architecture configuration and benchmark are shown in Appendix J.3 and J.4 and the speed up gained by including extra threads/cores relative to a single thread performing execution is shown in Figure 6.11 and 6.12 below.

The results for both the LE1 and Leon3MP architectures are very closely matched to the results of the same benchmarks executed using the coarse-grained parallel PThread code. This is due to the overhead of the PThread operations being negligible in terms of the overall workload performed. As well as this the CPUID benchmarks include sections of code used for synchronisation to check that required computation has completed on all cores/CPUs. This is similar in theory to the *pthread_join* operation. Essentially, the use of the CPUID benchmarks does not have any advantage in these benchmarks as the overhead of using the PThread library operations to perform coarse-grained thread management does not greatly affect the overall execution time. Due to this it is far more useful to use the PThread library to manage parallelism as the library is highly portable. As an example, the CPUID benchmarks being executed on the LE1 and Leon3MP are not the same; each implements a different operation to return a unique identifier for each thread and so the benchmarks required modifications to work on each architecture. The extra code required to perform parallel tasks and keep cores/CPUs in sync has already been implemented and tested within the PThread library, as well at other threading libraries such as OpenMP and MPI. However, the LE1 CPUID operation is used in the next section by the behavioural synthesis tool to split the UML models over available cores. FalconML implements custom event queues to split computation across multiple cores and uses the CPUID operation to direct the control flow of each core to execute correct events in the queues based on their unique identifiers.

6.5 Unified Modelling Language

A selection of benchmarks was developed using both the PIUML model design notations proposed in this thesis and PSUML models which require the application developer to make direct modifications to the UML models based on the target architecture. The UML entry tool used is Modelio and the behavioural synthesis tool to convert the UML models into C code for the LE1 VLIW CMP is FalconML. This section is split into three sub sections:
A) Firstly methods of quantifying interactions with the UML design tool required for generating models is discussed and introduced.
B) The creation of benchmark UML models.
C) Finally the UML models are synthesised and executed and the results are discussed.

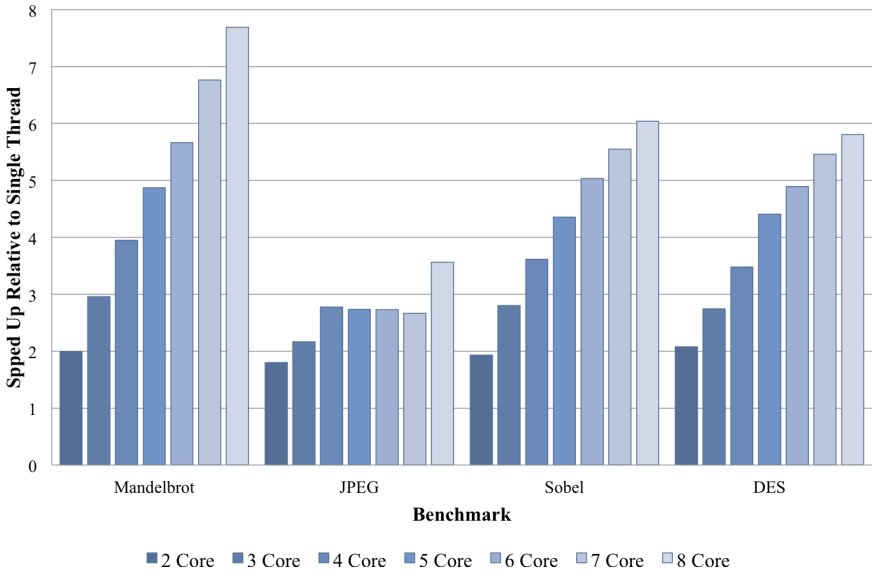


Figure 6.11: Benchmarks run on LE1 architecture showing speed up of multiple cores relative to a single core using CPUID instruction to split computation across available cores. Graph source data available in Appendix J.1

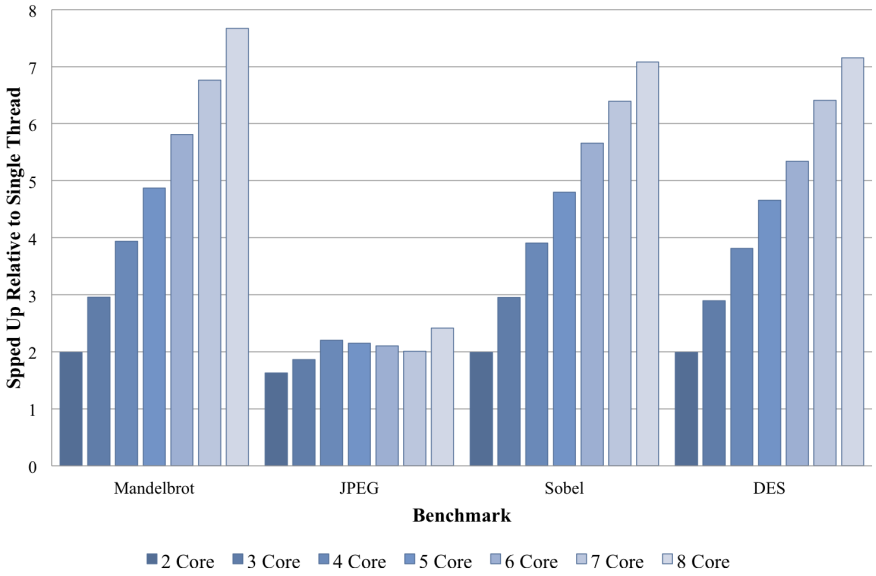


Figure 6.12: Benchmarks run on Leon3MP architecture showing speed up of multiple CPUs relative to a single CPU using CPUID instruction to split computation across available CPUs. Graph source data available in Appendix J.2

6.5.1 Quantifying UML Capture

The method of quantifying the work required by the application developer within the UML modelling tool logs the user interactions during the creation of the UML model and evaluates them in terms of events.

The initial creation of the base systems within the UML modelling tool is recorded, followed by the work required to modify the models directly to generate PSUML models as well as using the proposed UML notations in PIUML models. These hand crafted models (PSUML) are then executed and their performance compared with automatically generated statically mapped UML models based on the the PIUML models. The extra work performed by the application developer to create PSUML models will then be compared to the PIUML model creation.

The events captured will include:
A) Mouse Events - any mouse button interaction.
B) Keyboard Events - any time the keyboard is used.
(Keyboard interactions are registered as a single event between use of the mouse. This was decided not to factor in the length of variable names or action code required to generate a functional UML model.)

A custom program was created to capture required events from multiple input devices. Utilising library functionality from the X window system (X11) [107] and the XInput library [108] the program connects to a number of desired input devices through the X11 system and listens for the events specified above. A single log file is generated containing events from all devices. The data contained within this log file is then summarised to produce a total number of user interaction events required to generate the model. A custom program was required to be developed as there were no tools available which could log multiple input devices at the same time. The program code is included in Appendix H.

All UML model creation will be logged using this developed program. This will allow an investigation into the benefits of utilising the proposed UML design notations to create PIUML models compared to the creation of PSUML models. All UML models were produced by a single application developer after familiarising themselves with the Modelio GUI. This was performed to ensure that only user events related to the generation of the models were captured and removes any anomalies which may have been introduced by multiple users or becoming more familiar with the tool during model creation.

6.5.2 UML Creation

The benchmarks created in UML use the same behaviour as the previous PThread and CPUID benchmarks. The core behaviour of each benchmark was implemented within a

single UML class with input and output ports. This enabled the core behaviour to be instantiated multiple times within a single UML model and mapped to separate cores within the target architecture. Six base UML models were generated, two for each benchmark; this allowed the implementation of the RobinRobin and Split methods of parallelism, previously introduced in section 4.3.3.1. Two base models were required due to the differences in the behaviour and interface for both RoundRobin and Split methodologies.

The UML models generated for each benchmark include the behaviour of the application captured using classes, attributes, ports, operations, interfaces and state machines. Once completed the models were checked to ensure that behaviour was correct by running the behavioural synthesis tool on the unmapped model to produce a single core application. These base models were then replicated and modified to generate multiple PSUML models along with a single PIUML model. In order to make creation and execution simpler all models include a *Source* and *Sink* class; these classes generate input data as well as consume output data resulting in each model having a consistent input and output data stream. The user events required to generate the base models were captured and used as a reference when investigating the extra work required to manually map the model to specific architectures.

The base class diagrams of the Mandelbrot Set, Sobel Filter and DES benchmarks are shown in Figures 6.13, 6.14 and 6.15, respectively. Each shows the attributes and operations within the classes as well as the interfaces required to generate a valid model. The difference between the RoundRobin and Split methodologies is captured within the behaviour of the models and is not visible in these diagrams.

Each base model was then modified to include an architecture section; this architecture ranged from three to eight cores. The minimum value three was chosen due to the inclusion of *Source* and *Sink* classes which are instantiated and executed on their own cores, ensuring these extra objects do not affect the execution of the benchmark being investigated. In each case the architecture was generated within the UML model and *Source* and *Sink* objects were instantiated along with multiple instances of the benchmark classes (DES_alg, Mandelbrot, Sob). The number of instantiated benchmark classes is equal to the number of available cores minus two (taking into account the *Source* and *Sink* objects), which allows each object to execute on individual cores.

At this point one of the parallel methodologies was performed based on the design of the input UML model. Both the RoundRobin and Split methodologies were used to modify the application to fully utilise the available architecture. The RoundRobin methodology includes generating *Fork* and *Join* classes to perform execution; these modifications match those described in section 4.3.3.1. The Split methodology differs and no extra classes were generated; instead new ports were instantiated and modifications were made to the behaviour within the existing classes to be aware of the newly instantiated objects and ports.

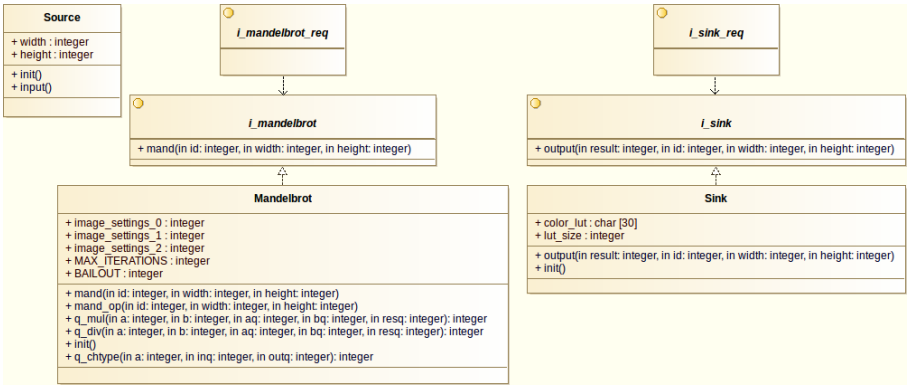


Figure 6.13: Class diagram showing the Mandelbrot Set captured in UML. *Mandelbrot* provides all internal operations and attributes to generate the outcome of a single pixel with the image based on *id* and the *width* and *height* of the image. *i_mandelbrot* shows the interface, realised by *Mandelbrot*, which provides a single operation (*mand*) to external classes. The *Source* and *Sink* classes produce and consume data within the UML model. In this example the *Sink* class includes a look up table to match the generate *Mandelbrot* value to a colour for image generation.

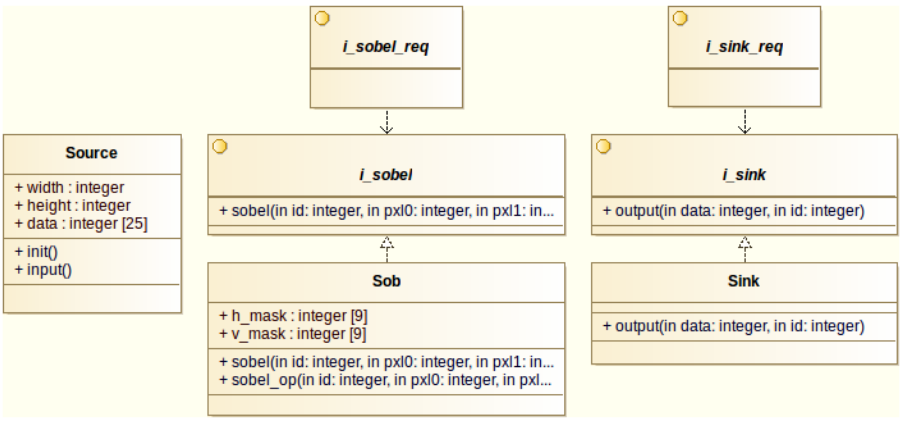


Figure 6.14: Class diagram showing the Sobel Filter captured in UML. *Sob* provides internal operations and attributes required to perform edge detection on a subset of pixel values passed to the operation. *i_sobel* shows the interface, realised by *Sob*, which provides a single operation (*sobel*) to external classes. The *Source* and *Sink* Classes produce and consume data within the UML model. This example performs edge detection using two 3 x 3 masks (vertical and horizontal) on a 5 x 5 image.

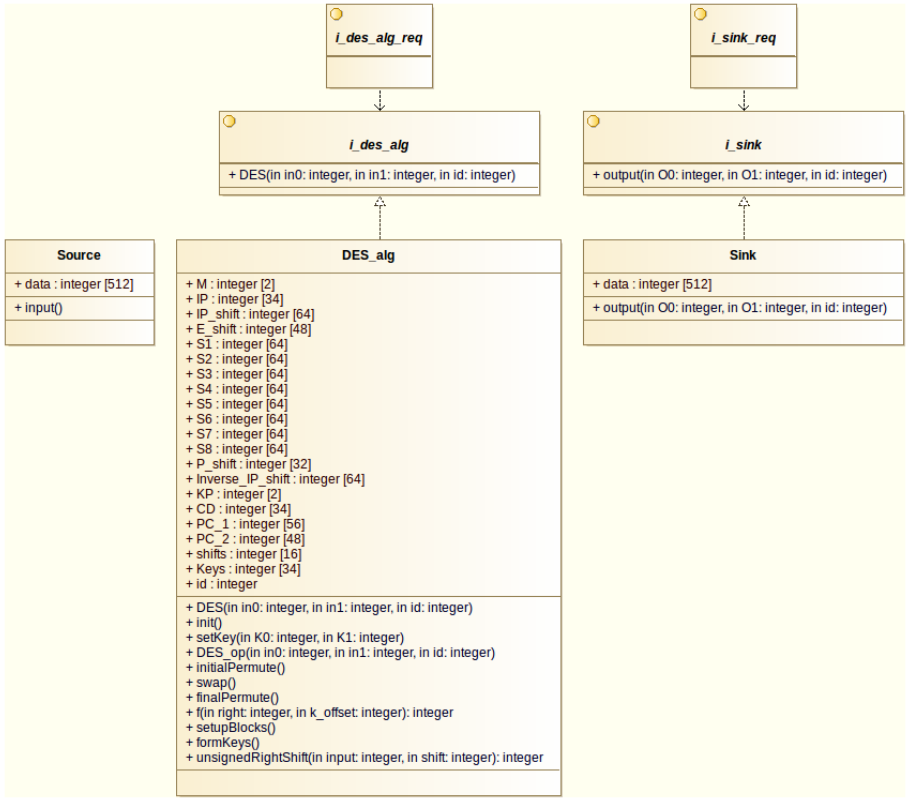


Figure 6.15: Class diagram showing the DES captured in UML. *DES_alg* provides all internal operations and attributes to perform encryption of two 32bit integers used to generate a 64bit plaintext to be encrypted. *i_des_alg* shows the interface, realised by *DES_alg*, which provides a single operation (*DES*) to external Classes. The *Source* and *Sink* classes produce and consume data within the UML model. This example encrypts 256 64bit messages.

Once these modifications had been made connectors were included between the available ports to allow data flow between objects within the model. Following this each instantiated object was statically mapped to a core within the architecture, resulting in a PSUML model which could be passed to the behavioural synthesis tool to generate C code for execution on the LE1 VLIW CMP.

Along with the PSUML models, a single PIUML model was generated. This was performed by generating a three core architecture along with *Source* and *Sink* objects and single instance of the benchmark class. A single core and the benchmark class were then modified to include a wildcard multiplicity as described in chapter 4. The *Source* and *Sink* objects were statically mapped to individual cores within the architecture (not including the core

marked with the wildcard multiplicity) and an adjustable mapping was generated between the application and architecture sections of the model.

This resulting PIUML model was then transformed through the tools and scripts introduced in chapter 4 to generate a statically mapped PSUML model to be passed to the behavioural synthesis tool. The proposed UML modelling methodology results in multiple possible static mappings which differ depending on the number of unmapped objects and cores available. The number of possible mappings is equal to:

$$Cores^{Objects} \tag{6.3}$$

In the benchmarks used this results in one mapping possibility for the three core architectures (3^1) and up to 32,768 possibilities for the eight core (8^5). As a result of this, diagrams of all generated static mapping permutations are produced to allow the application developer to visualise and choose which permutation to use. The execution runs in these examples use a single object mapped to each core. The four core DES model processed by the transformation scripts produces 17 images; Firstly the original mapping defined in the UML model is shown in Figure 6.16 where *source* and *fork* are statically mapped to core *i*, *sink* and *join* are statically mapped to core *i2* and a final adjustable mapping occurs between *TOP* and *CompResc*. The remaining images show all possible static mappings of the unmapped objects (*DES_alg_0* and *DES_alg_1*) onto all available cores (*i*, *i2*, *i1_0* and *i1_1*). Two possible static mappings are shown in Figure 6.17 and 6.18 where both *DES_alg* objects are statically mapped to individual cores. One of these images can then be referenced by the "Write Static Allocations" script, section 4.3.3.3.2, to generate a statically mapped PSUML model to be used by the behavioural synthesis tool.

The user events required to generate both PSUML and PIUML models were captured in the same manner as with the base UML model. These results are shown along with the events required to generate the base model in Figure 6.19 (All models were generated using the same methods in the same UML Model capture tool).

As can be seen, the addition of extra architectural elements and objects results in more user events required to generate the UML models. The RoundRobin models require a greater number of user events compared with the Split models; this is due to extra classes (*Fork* and *Join*) being created whereas the Split methodology required modifications only to the underlying behaviour of the model.

The number of user events required to include extra cores in the target architecture is similar across all of the benchmarks. This is a result of similar actions being required to make the modifications (adding objects, classes, ports and connectors and mapping the application elements to the architecture). As can be seen from the results, in some instances the number of user events required to modify the UML models to utilise the target architecture is greater

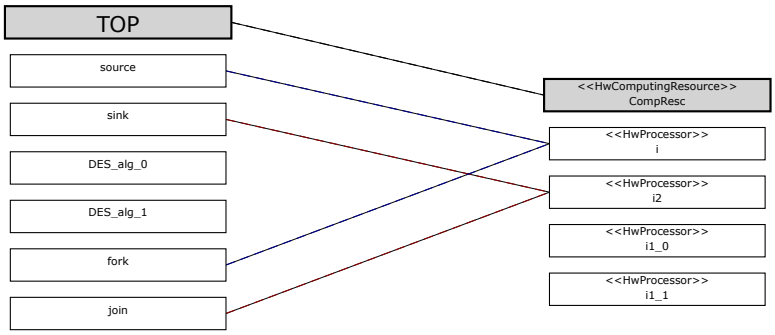


Figure 6.16: Four core DES adjustable mapping output. Shows the current mapping within the UML model which is used to generate the available static mappings.

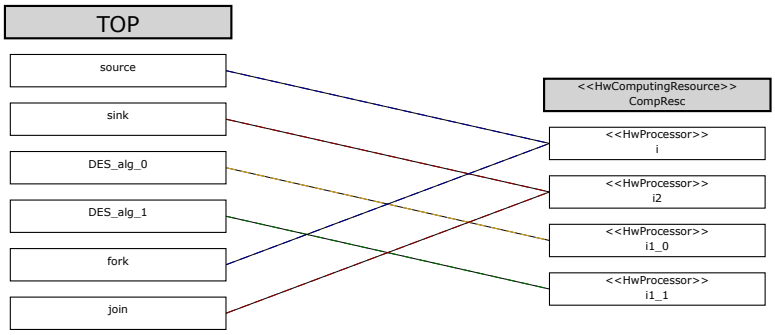


Figure 6.17: One Example of the desired output from the alloc.pl (see Appendix E) script showing generated objects *DES_alg_0* and *DES_alg_1* being statically mapped to individual cores.

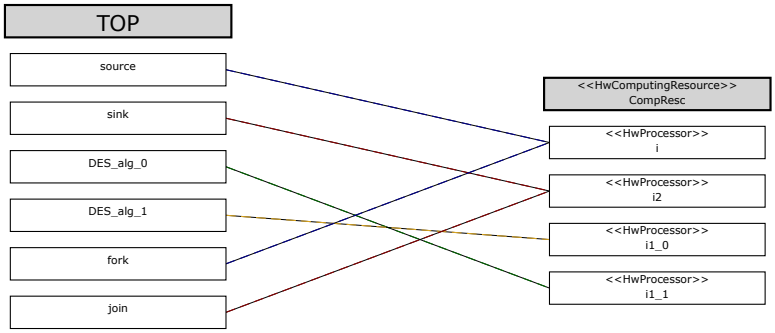


Figure 6.18: Alternative example of the desired output from the alloc.pl (see Appendix E) script showing generated objects *DES_alg_0* and *DES_alg_1* being statically mapped to individual cores, different from Figure 6.17 in that the objects are mapped to opposite cores.

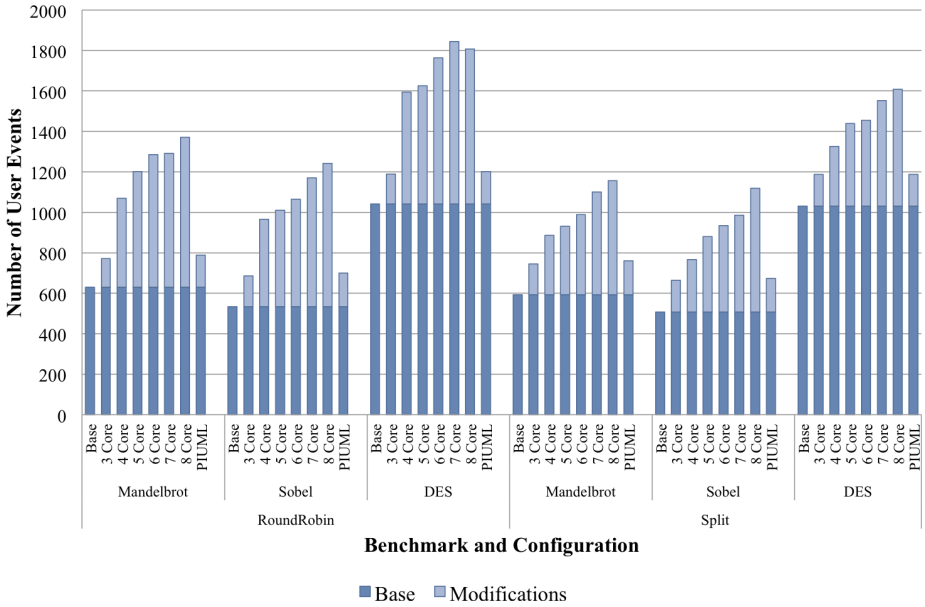


Figure 6.19: User events required to generate all UML designs for alternative architectures using both the RoundRobin and Split design method. Full data set is included in Appendix K

than the user events required to generate the base behaviour of the model. This is evident in both the Mandelbrot Set and Sobel Filter benchmarks. This shows that in the example applications used in this study that, even with a small number of processor cores, the application developer spends more time modelling the architecture and performing mapping than focussing on the behaviour of the application.

There is a large jump between the number of events required to generate the three and four core models for the RoundRobin benchmark. This is due to the three core model not requiring *Fork* and *Join* classes as there is only a single instantiation of the class performing the computation. The UML models containing four cores and above require these additional classes to perform the *fork* and *join* and so require extra effort by the application developer to include them within the model. This trend is not evident in the Split benchmarks due to nature of these implementations requiring only extra ports to be instantiated.

Trends which can be seen from the captured user events showed a higher variation of events required between the different models as more and more cores were added. This is due to the nature of the UML modelling tool; with more user events the probability of making a mistake is higher and so the deviation in the number of user events across the benchmarks is seen to increase. In the RoundRobin models it takes an average of 100 user events to include an extra core along with the instantiation of an extra object. The Split methodology takes

less, 70 user events, to perform the same task of including an extra core. This difference is due to the Split model not requiring extra classes to be generated as discussed previously.

The results obtained from generating the PIUML models using the proposed UML notations show the number of user events being slightly greater than that required to generate a statically mapped three core UML models. This is due to these being essentially the same model with the PIUML model requiring the extra work of tagging a single class and core with a wildcard notation. This implementation of the PIUML model allows the application developer to modify the base UML model with minimal user events and, once run through the transformation scripts introduced in chapter 4 along with an LE1 XML configuration file, the resulting UML model is automatically modified to fully utilise the underlying architecture. In the examples here this can save on average 580 and 420 user events, for RoundRobin and Split models, respectively, when comparing to generating the eight-core UML models. This is a worst case look at the figures, however, the single PIUML model can be used to generate all of the hand-crafted models presented here resulting in a lot of saved effort for the application developer. The next section shows the execution results for each of the generated models and investigates the execution times in relation to the number of user events required to generate them.

6.5.3 Execution Results

All of the UML models described in the previous section were executed on the LE1 VLIW CMP; the PIUML model utilising the proposed UML notation was used to produce six models ranging from three to eight cores to match those of the PSUML models and their execution cycle counts are shown in Figure 6.20. Results are also included in Tables L.1 and L.2 included in Appendix L.

The execution results are compared in three main forms. Firstly the RoundRobin and Split methodologies are compared and similarities made between these and the granularity of threading discussed previously in section 6.4.1. Then a comparison between the PSUML and PIUML models is performed. This comparison is then extended to take into account the user events required to generate the models.

The Split methodology displays a greater decrease in overall execution time; this is due to the input data being split across the available cores once and each available core performing an equal amount of computation. This is similar in nature to the coarse-grained threading in which a single thread creates a thread of computation for each available core and then synchronises once at the end.

Alternatively the RoundRobin method more closely matches fine-grained threading as creation and synchronisation of the "threads" on available cores through the *Fork* and *Join* objects occurs multiple times throughout execution. There is more computation required for

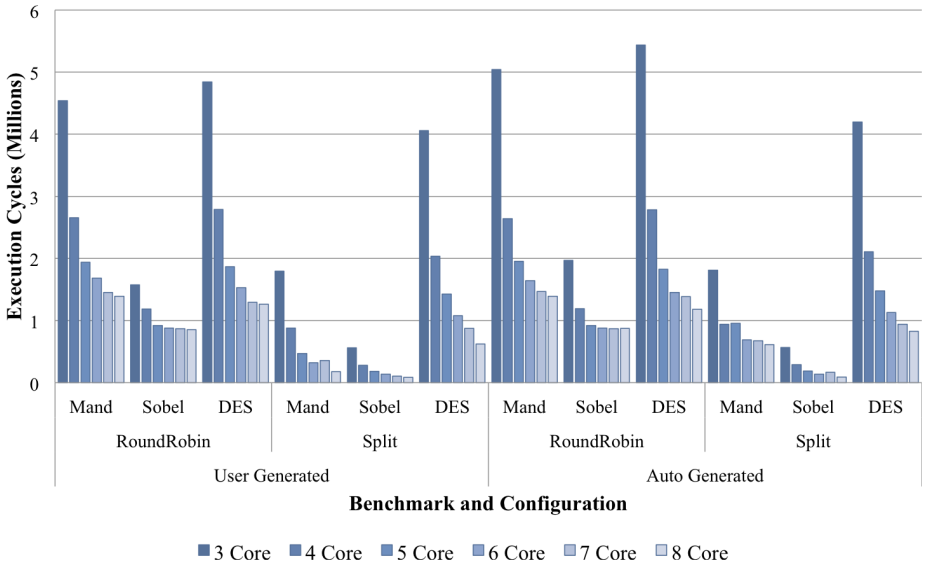


Figure 6.20: Execution cycles of UML Benchmarks on multicore LE1 up to eight cores. Both user modified RoundRobin and Split design methods shown alongside output using presented UML notations.

the creation and synchronisation of threads (previously referred to as thread management) in the RoundRobin method than in the Split method as it occurs more frequently. This results in a larger overhead in the former methodology. These two methods can be used depending on the type of application and computation required. For example if a large selection of data is available and ready for computation which can be readily parallelised the Split method is the best to use, however in a data flow application with data items in a stream, as long as they are truly independent of other data items, the RoundRobin method can be used to fully utilise the available architecture.

Another comparison point can be made between the execution times of the PSUML and PIUML models. Firstly, comparing RoundRobin execution results for both modelling approaches it can be seen that both sets of results are very similar. This is due to the statically mapped models generated from the PIUML model being very similar to the PSUML models, with the *Fork* and *Join* classes generated along with multiple instances of the other objects.

The fact that they are similar is not of any surprise, the one data point which stands out are the three core results; on average the statically mapped models generated from the PIUML model takes 15% longer to execute compared to the PSUML model. This is due to the transformation scripts including the *Fork* and *Join* classes into the model, resulting in the data having to pass through extra objects which are not required in the three core

case as only a single object (*Mandlebrot*, *Sob* or *DES_alg*) is instantiated and so data can be passed directly to this object.

Comparing the execution results between the two modelling approaches using the Split methodology displays a different trend. In these cases the auto generated models always perform worse than the hand generated models. This is the result of including extra classes to perform the *Fork* and *Join* of the data sets. In the PSUML models these are not required as the UML models are designed specifically for each available architecture resulting in optimal models being produced. In the statically mapped models generated from a PIUML model this is not the case as it was decided not to directly modify the structure and behaviour of classes within the model; this results in extra computation required to parallelise the computation across the architecture.

Comparing the PSUML and PIUML modelling approaches in terms of both user events and execution time results in a trade off between design time and execution time. In the RoundRobin examples the PIUML models would be a better approach to take. This is due to the user events required to generate the PIUML models being similar to those to generate a three core PSUML model with the execution time of this PIUML in all but the three core statically mapped model being similar to the PSUML models. As described previously the three core statically mapped model perform worse due to the inclusion of *Fork* and *Join* classes which are not required in this case. The inclusion of these classes could be easily removed from the PIUML to PSUML transformation scripts by checking the number of objects being instantiated and only including them if required. The PIUML modelling saves on average 378 user events per model created in this study while resulting in similar execution results as the hand generated PSUML models. As a side note, there is no limit to the number of cores which the tools can handle; this means that the same PIUML model could be used to generate a four core or a four hundred core statically mapped model with no extra work required by the application developer.

Comparing the Split models in the same way leads to a question of trade off; the statically mapped models generated from the PIUML models performed worse in terms of execution time than the PSUML models. This is a result of including extra classes to split the workload across available cores rather than directly modifying classes defined by the application developer. If timing was an issue the PIUML models could be used primarily as a method of testing the parallelism of an application on a target architecture. This method could be initially used to find where the limits of the system lie, in terms of the amount of exploitable parallelism within an application. A PSUML model could then be hand generated based on the outcome of the PIUML modelling. This would benefit in time not being wasted generating PSUML models which do not offer any benefit in terms of execution speed up.

6.6 Summary

This chapter presented investigations into the hardware PThread implementation as well as the PIUML model notation.

Initial results showed that the hardware PThread implementation in the LE1 VLIW CMP performs as well, if not better, than PThread libraries available on soft-core processors. The hardware PThread implementation was a subset of the full PThread library and supports only simple thread management operations. When compared with commercially available software PThread libraries, including the Xilinx PThread library on the MicroBlaze and the FSU PThread library on the Leon3, the LE1 hardware PThread implementation was found to be a factor of 25 and 3,800 times faster, respectively.

The MicroBlaze and Leon3 processors execute interleaved multi-threading and as a means of better comparison with the LE1 CMP a custom software PThread library which implemented the same subset as the LE1 was created for another CMP, the Leon3MP. With the Leon3MP clocked at 75MHz and the LE1 at 50MHz the hardware implementation was twice as fast as the software implementation.

This software implementation was modified to simulate the thread create and synchronise timings of the Leon3 and MicroBlaze processors based on initial micro-benchmarks, performed in section 6.3. This enabled comparisons into the granularity of threading achievable in a selection of benchmark applications.

Also presented is a selection of benchmarks written in C which were designed to utilise the available PThread operations to display the advantage of having a fast PThread library implementation. This fast implementation allowed threading to be performed at a fine granularity while still offering a decrease in overall execution time for all these benchmarks. The benefits of this fine-grained threading allows smaller sections of applications to be threaded, enabling explicit threading to be performed within large data-dependent applications at small data independent sections throughout the code.

The LE1 also supports a CPUID mode of parallelism, which uses a unique identifier for each core to modify the control flow of an application. This method was presented and shown to offer similar results as the PThread implementation in coarse-grained threaded applications. The main drawbacks of the CPUID method was the limitation in synchronisation methods and the need to modify code to work across alternative architectures as no standard CPUID operation is implemented. It was concluded that this method performed adequately for parallelising computation across available cores but it was suggested that using standard threading libraries would be more useful as they are supported by a larger set of architectures and systems, resulting in portability across alternative architectures.

The proposed UML notation, which allows a single, platform-independent UML model

to be used to generate multiple statically mapped UML models was then investigated and compared with UML models designed specifically for different architectures. Investigations included comparisons of both work required by the application developer and execution time between the PIUML and PSUML models. The use of three to eight core models was investigated along with two alternate methods of designing parallel UML models. These methods closely match the idea of fine and coarse-grained threading as investigated in the PThread benchmarks. The UML model creation process was quantified through the use of a program to log user events while generating the PSUML and PIUML models for each architecture. It was found that the translation from PIUML to PSUML performed differently when applied to the RoundRobin and Split methodologies. In the fine-grained method (RoundRobin method) the statically mapped UML model generated through the use of the PIUML design notation performed similarly to those which were designed for specific architectures. Alternatively, the coarse-grained method (Split method) resulted in worse performance due to the inclusion of extra classes into the model which were used to perform the creation and synchronisation of threads.

The next chapter concludes this thesis, discusses the research performed as a whole and suggests possible future work and extensions to the methodologies.

7

Conclusion

7.1 Chapter Objectives

This chapter concludes the research and development performed 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 hardware thread management along with a comparison with available threading libraries implemented for soft-core processors on FPGAs. The thread management presented in this thesis uses hardware tables and logic to create and synchronise threads within an application while reducing overheads when compared to a software based approach. The hardware PThread primitives are made available though custom instructions within the LE1 Very Long Instruction Word (VLIW) processor resulting in standard PThread code being executable with no modifications. It was shown that this method decreases overall execution time of thread management operations by a factor of between 25 and 3,800 when compared to the Leon3 and MicroBlaze soft-core processors synthesised on the same FPGA platform.

This decrease in overhead allowed the use of threading at a fine-grained level where small sections of an application can be performed in parallel to decrease overall execution time. This fine-grained threading, implemented using software threading libraries available on the Leon3 and MicroBlaze processors, resulted in an increase in execution time due to the PThread library overheads outweighing the work being performed within the created threads.

A second aspect of this research was the use of a Visual Programming Language (VPL) as an input to a tool chain targeting a configurable multicore architecture. However, targeting an architecture such as the LE1 using Unified Modelling Language (UML) required the

application developer to create models specifically for each architectural configuration. As a result of this a set of design notations and processes was devised to allow a single, platform-independent UML (PIUML) model to be automatically transformed to platform-specific UML (PSUML) models. This was shown, in section 6.5.2, to decrease system design time for large architectures while generating models which executed within 15% that of hand generated models designed for specific architectural configurations.

7.2 Summary of Thesis Objectives

The objectives as defined in chapter 1 were:

- Investigate parallelism methodologies.
- Implement parallel execution in VLIW CMP.

Parallel methodologies were firstly investigated with background research identified in chapter 2 followed by the implementation of a hardware threading mechanism described in chapter 4.

The initial research into parallelism methodologies resulted in the focus into Thread Level Parallelism (TLP). This was the result of a wealth of research in Instruction Level Parallelism (ILP) suggesting that theoretical maximum in this form of parallelism had already been achieved. It was decided to introduce TLP to an architecture which already exploited ILP to benefit from both forms of parallelism.

A hardware PThread implementation was designed to reduce thread management overheads compared to software libraries available in soft-core processors. The benefit of this is that parallelism is able to be extracted from within applications where it was not possible or difficult previously. For example, application developers attempting to parallelise large, data-dependent sections of code are able to utilise parallel threads within small, data-independent sections of the application to speed up execution. To incorporate ILP this hardware PThread implementation was designed for the LE1 VLIW CMP resulting in both implicit, from the ILP, and explicit, from the TLP, parallelism being exploited in the same system.

The performance evaluation of PThreads implemented in both hardware and alternative software libraries, including the Xilinx PThread and FSU PThread libraries, is shown in chapter 6 which shows the hardware threading mechanism outperforming these software libraries by a factor of 25 times.

- Design and Implement tool chain for unique VLIW CMP.

In relation to the previous objectives the LE1 VLIW CMP was targeted by the hardware PThread implementation. This resulted in the requirement of developing the tool chain for this processor, including assembly, libraries, simulation and configuration. The design and implementation of the LE1 tool chain is detailed in chapter 4.

- Investigate high level system design in UML.
- Develop UML modelling technique for statically customisable multicore platforms.

High level modelling in the form of object-oriented design and VPLs were investigated in chapter 2. Previous research suggested VPL reduced development time up to a factor of four times when compared with conventional text based development. UML is currently the de facto visual design language in both research and industry for software-intensive systems. As a result of this UML was chosen as the VPL to focus on, and previous design styles and methodologies were investigated.

The development of novel modelling techniques arose through targeting the statically customisable LE1 processor. When targeting different architectural configurations of the LE1 the application developer was required to directly modify both behaviour and structure within the UML model to match that specific architecture.

Due to this, novel UML modelling notations were created, including adjustable mapping and wildcard notations, to define a PIUML model which could then be refined to PSUML models. Processes to perform transformations between these PIUML and PSUML models were also generated. These novel design notations along with the transformation processes are introduced and described in chapter 4.

- Develop UML applications to utilise UML modelling techniques and tool chain.

Finally, this objective relates to both the parallelism and modelling objectives. This was met through the implementation of a selection of application in both C and UML as presented in chapter 6.

7.3 Contributions

chapter 3 presents the research areas which were targeted in this thesis. The key areas of work were:

- Investigation into hardware threads in the LE1 VLIW CMP.
- The capture of Platform-Independent UML models.

- Fully modifiable mappings for application mapping to statically customisable multicore platforms in UML.
- Intelligently threading applications in UML.
- Development of UML benchmarks.

These areas led to three main contributions to knowledge:
A) Hardware PThread implementation
B) High level design notation for multicore architectures
C) Method of quantifying UML design creation to perform investigations into design.
The Hardware PThread implementation allows multiple threads of execution to be managed on the LE1 VLIW CMP. This implementation enabled threading to be used at a finer grain than is possible through using two software PThread library implementations. This was due to less overhead, in terms of computation and time, in the hardware implementation when compared with the software PThread libraries of soft-core processors. As a result the management of threads could occur more frequently which enables threading within small loops to be performed while achieving an overall decrease in execution time.

Due to the LE1 VLIW CMP being designed to be highly configurable in terms of both VLIW elements, for example issue width and number of execution units, as well as the number of cores being instantiated, system design in UML initially posed a problem. This was due to the nature of the UML itself, where each time the underlying architecture was modified, the UML model required modification to fully utilise this new architecture. This was being handled in text based languages through the use of pre-processor macros to define constants and data sizes at compilation time. This idea was transferred to the UML modelling domain in the form of wildcard multiplicities and the modifiable mapping level. These allowed a single UML model to be modified to fully utilise a given architecture.

Finally, quantifying the creation of UML models played a key role in the conclusions made within the previous chapter. By logging the user interactions required to create UML models it was possible to perform comparisons between the amount of work required and the actual execution time of the UML models once synthesised to run on the multicore VLIW system. Previous research on using VPL over text based languages relies on a qualitative output from application developers to explain how they found the use of a VPL rather than the text based design. The inclusion of quantifiable data adds an extra layer of scientific rigour and allows both a simple comparison between alternative design methods, as used here, as well as in-depth studies into user interfaces and how user friendly they truly are.

7.4 Findings

This section introduces the finding generated as a result of the research performed in this thesis. Firstly the investigation and development of TLP is described. This is followed by an overview of the use of the LE1 VLIW and its impact on the overall research. Finally the use of UML as a high level design language for the multicore VLIW system is discusses.

7.4.1 Parallelism

Initial tests showed that the hardware PThread implementation in the LE1 VLIW CMP performs as well, if not better, than PThread libraries available for other soft-core processors. The hardware PThread implementation is only a subset of the full PThread library and supports only simple thread management operations. Due to this, a software version of this subset was created to make a more direct comparison with other software libraries. This custom software implementation was modified to match the timing results of thread creation and synchronisation operations identified from initial experiments in section 6.3. Using both the unmodified software implementation and two versions designed to simulate the thread creation and synchronisation time of the MicroBlaze and Leon3, the LE1 hardware PThread implementation was found to be a factor of 25 and 3,800 times faster than the single core IMT-based software libraries; it was also found to be twice as fast as the limited software implementation running on the Leon3MP at 75MHz versus the LE1 at 50MHz.

Also presented is a selection of benchmarks written in C designed for the hardware and software PThread implementations. The hardware implementation allowed threading to be used at a finer granularity while demonstrating a decrease in overall execution time for all C benchmarks. This ability to perform explicit thread parallelism at a finer granularity enables the application developer to extract a higher amount of parallelism within applications. This is a result of lower latencies associated with the hardware PThread implementation compared with the software libraries investigated.

7.4.2 Very Long Instruction Word Processor

The use of the LE1 VLIW CMP played a key role in all aspects of the research undertaken. It was the extensibility and configurability of the processor that led to the new UML design notation and the subsequent reduction in the amount of time required to generate UML models for that architecture.

The XML configuration file, used to define the machine description of the LE1 VLIW CMP, enabled the LE1 to be utilised within both the novel UML design processes and automated design space exploration, see chapter 5. It is used within the UML design notation

transformations to instantiate the correct number of cores within the UML model based on the target architecture and is used within the ENOSYS project to enable configuration by a design space explorer.

Finally, the ability to extend the LE1 instruction set architecture to include custom instructions enabled the development and implementation of the hardware PThread library subset, resulting in investigations being performed into the benefits of low latency thread creation and synchronisation operations.

7.4.3 High Level Design

The UML notations presented in this thesis attempt to reduce the amount of work required by the application developer when producing UML models targeting a statically configurable architecture. This is achieved through a process of developing a single PIUML model which is then able to be refined to produce PSUML models which fully utilise a specified architecture. The UML model design process was quantified through the logging of user events required to generate valid UML models for each architecture and using the proposed design notations. It was found that there was a substantial decrease in the amount of work required to generate PIUML models when compared to hand generated PSUML models. However, the resulting execution times differed depending on the type of parallelism being extracted, ranging between 1 and 1.15 times the execution cycles of the hand generated PSUML models.

7.5 Limitations of Research

Aspects of this research and development effort come with certain limitations. Firstly the UML benchmarks presented in chapter 6 were slightly limited in terms of computation and full usage of the UML specification. This was partly due to the learning curve of using both the UML standard and the tools required to generate valid UML models. The models were similar in terms of their structure and ease of performing computation in parallel; a single class implemented to perform certain computation with multiple instances of that class being connected to source and sink objects, the latter used to produce and consume data. A large UML application with only a subsection of the model being executed in parallel would have been ideal to test the usability of the proposed methods. However, there were no UML model benchmarks available for this purpose.

Another limitation came from the behavioural synthesis tool, FalconML. This required UML model exported in specific XMI structures which resulted in limiting the number of UML input tools which could be used. The generation of UML models was also limited to

a small subsection of the UML specification as supported by FalconML.

This resulted in the use of multiple simple UML models based on a similar structure which allowed the testing of the wildcard multiplicities and static and adjustable mapping notations. However, these UML benchmarks are not as close to the realistic usage of UML as investigated by the industrial partners in the ENOSYS project. The partners are also using a subset of the novel UML notations defined in chapter 4, those being the adjustable and static mapping. This then is used to perform design space exploration using the Jink tool to exhaustively explore their implementation space by mapping these models on the LE1 VLIW CMP, described in chapter 5

Another slight limitation was the unavailability of a PThread library on a soft-core CMP. The PThread operation times simulated for both the Leon3 and MicroBlaze, as presented in section 6.3, were taken from the processors executing IMT where each active thread was executing in a time interleaved nature. These micro-benchmarks were performed in a best case scenario where at most two threads were active, resulting in the least overhead possible for the tests.

To perform a more direct comparison with the LE1 VLIW CMP hardware PThreads a subset of the PThread library was implemented in software to execute on the Leon3MP CMP. This Leon3MP software PThread implementation was then used to simulate the PThread operation times extracted from the Leon3 and MicroBlaze IMT processors. This enabled an investigation into the granularity of explicit parallelism which can be extracted from applications in relation to the latencies of thread management operations.

Finally, the usage of a single UML modelling tool and a single behavioural synthesis tool implies that the results cannot truly be said to be universal. Although it is worth noting that the behavioural synthesis tool, FalconML, is extremely novel and there are no alternative tools offering similar functionality available at the time of writing. All models were produced using a single modelling tool, Modelio, by a single application developer. Although other UML modelling tools are available their use is not introduced in this thesis. This was performed to reduce the number of external variables in an attempt to produce repeatable, reliable results and conclusions with respect to the benefits of the novel UML modelling techniques introduced.

7.6 Further Research

As an extension to the previous section outlining the limitations of this research new research avenues could include the development of larger UML models taken through the tool chain and using the proposed UML notations. It is expected that with the inclusion of more cores and unmapped objects within a model the complexity of generating all possible static

mappings grows exponentially. This would result in a large amount of time required to generate all possible static mappings, which in turns leads to a large amount of time to process these possible static mapping through the entire tool chain and perform simulation or execution on the hardware system.

As described in section 6.1 the benchmarks are limited due to their statically compiled threads. An extension of the current work could investigate the use of load-balancing to schedule threads at run time rather than at compile time. This should achieve a more equal spread of computation across all available hardware resources and in theory reduce execution time.

Further possible research in the hardware PThreads could include the implementation of additional PThread primitives, such as mutual exclusions and conditional variables. The inclusion of these PThread constructs would allow more complex applications to be executed on the LE1 VLIW hardware. If implemented in a similar style, through the inclusion of hardware tables, finite-state machines and custom instructions, the overhead compared to software implementations could lead to speed improvements over more applications. The execution of more complex code with mutual exclusions and conditional variables would also allow the investigation into threading at a more coarse-grained level with data dependencies between threads.

8

Publications

The following are publications that have resulted from the work presented in this thesis. All publications have been reviewed and published in either conference proceedings or journals.

As mentioned, part of the research effort was used within the ENOSYS European FP7 project which resulted in internal documentation being produced to meet the project requirements. These documents are listed below along with their dissemination level and this authors direct contributions:

ENOSYS:
D2.5: The LE1 VLIW Processor and Tools - Public
Documentation regarding the LE1 Tool Collection and configurations.

[1] David Stevens, Nicky Glynn, Panagiotis Galiatsatos, Vassilios Chouliaras, and Dionysis Reisis. Evaluating the performance of a configurable, extensible VLIW processor in FFT execution. In *Electronics, Circuits, and Systems, 2009. ICECS 2009. 16th IEEE International Conference on*, pages 771–774, June 2009.

[2] David Stevens and Vassilios Chouliaras. LE1: A Parameterizable VLIW Chip-Multiprocessor with Hardware PThreads Support. In *IEEE Computer Society Annual Symposium on VLSI*, pages 122–126, July 2010.

[3] Vassilios Chouliaras, George Lentaris, Dionysis Reisis, and David Stevens. Customizing a VLIW Chip Multiprocessor for Motion Estimation Algorithms. In *Architecture of Computing Systems, 2011. ARCS 2011. 24th International Conference on*, February 2011.

[4] David Stevens, Vassilios Chouliaras, Vicente Azorin-Peris, Jia Zheng, Angelos Echiadis., and Sijung Hu. BioThreads: A Novel VLIW-Based Chip Multiprocessor for Accelerating Biomedical Image Processing Applications. *IEEE Transactions on Biomedical Circuits and Systems*, November 2011.

[5] Mark Milward, David Stevens, and Vassilios 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*, July 2012.

References

[1] Object Management Group. Unified Modeling Language Specification. `http://www.omg.org/spec/UML/`, June 2011.

[2] Michael J. Flynn. Some Computer Organizations and Their Effectiveness. *Computers, IEEE Transactions on*, C-21(9):948 –960, sept. 1972.

[3] B. Ramakrishna Rau and Joseph A. Fisher. Instruction-Level Parallel Processing: History, Overview and Perspective, 1992.

[4] R. M. Tomasulo. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development*, 11(1):25 –33, jan. 1967.

[5] 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.

[6] The MPI Forum. MPI: A message passing interface. In *Supercomputing '93. Proceedings*, pages 878 – 883, nov. 1993.

[7] Hewlet Packard. Hewlett Packard Labs. `http://www.hpl.hp.com/`, 2013.

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

[9] Binu Mathew. *The Computer Engineering Handbook*, chapter Very Large Instruction Word Architectures. CRC Press, December 2001.

[10] Intel. Intel Itanium Family Specifications. `http://www.intel.com/content/www/us/en/processors/itanium/itanium-processor-9000-sequence/Specifications.html`, 2013.

[11] J.-W. van de Waerdt, S. Vassiliadis, Sanjeev Das, S. Mirolo, C. Yen, B. Zhong, C. Basto, J.-P. van Itegem, Dinesh Amirtharaj, Kulbhushan Kalra, P. Rodriguez,

and H. van Antwerpen. The TM3270 media-processor. In *Microarchitecture, 2005. MICRO-38. Proceedings. 38th Annual IEEE/ACM International Symposium on*, pages 12 pp. –342, nov. 2005.

[12] N. Seshan. High VelociTI processing [Texas Instruments VLIW DSP architecture]. *Signal Processing Magazine, IEEE*, 15(2):86 –101, 117, mar 1998.

[13] Object Management Group. Object Management Group Homepage. `http://www.omg.org/`, 2013.

[14] Safouan Taha, Ansgar Radermacher, Sebastien Gerard, and Jean-Luc Dekeyser. An Open Framework for Detailed Hardware Modeling. In *International Symposium on Industrial Embedded Systems*, pages 118 – 125, 2007.

[15] Object Management Group. UML Profile For MARTE: Modeling And Analysis Of Real-Time Embedded Systems. `http://www.omg.org/spec/MARTE/`, June 2011.

[16] Object Management Group. MOF 2 XMI Mapping. `http://www.omg.org/spec/XMI/`, August 2011.

[17] Margaret Burnett, Adele Goldberg, and Ted Lewis. *Visual Object-Oriented Programming : Concepts and Environments*. Manning Publications Co., 1994.

[18] National Instruments. LabVIEW [Computer Program]. `http://www.ni.com/labview/`, 2013.

[19] S. Konrad, H. Goldsby, K. Lopez, and B.H.C. Cheng. Visualizing Requirements in UML Models. In *Requirements Engineering Visualization, 2006. REV '06. First International Workshop on*, page 1, sept. 2006.

[20] V. del Bianco, L. Lavazza, and M. Mauri. Model checking UML specifications of real time software. In *Engineering of Complex Computer Systems, 2002. Proceedings. Eighth IEEE International Conference on*, pages 203 –212, dec. 2002.

[21] A. Baruzzo and M. Comini. A Methodology for UML Models V&V. In *Software Testing, Verification, and Validation, 2008 1st International Conference on*, pages 513 –516, april 2008.

[22] R. Bendraou, J.-M. Jézéquel, M.-P. Gervais, and X. Blanc. A Comparison of Six UML-Based Languages for Software Process Modeling. *Software Engineering, IEEE Transactions on*, 36(5):662 –675, sept.-oct. 2010.

[23] Grady Booch, James Rumbaugh, and Ivar Jacobson. *Unified Modeling Language User Guide, The (2nd Edition) (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, 2005.

[24] Grant Edmund Marin and Dr. Wolfgang Muller. *UML for SoC Design*. Springer, 2005.

[25] Perdita Stevens and Rob Pooley. *Using UML : Software Engineering with Objects and Components*. Addison-Wesley, 1999.

[26] Bernd Oesterich. *Developing Software with UML : Object-Oriented Analysis and Design in Practice*. Pearson Education Limited, 2002.

[27] Safouan Taha, Ansgar Radermacher, Sebastien Gerard, and Jean-Luc Dekeyser. An Open Framework for Detailed Hardware Modeling. In *International Symposium on Industrial Embedded Systems*, pages 118 – 125, 2007.

[28] Elvinia Riccobene, Patrizia Scandurra, Sara Bocchio, and Alberto Rosti. A Model-driven Co-design Flow for Embedded Systems. In *FDL'06*, pages 345–351, 2006.

[29] Peter Green, Martyn Edwards, and Salah Essa. HaSoC - Towards a New Method for System-on-a-chip Development. *Design Automation for Embedded Systems*, 6(4):333–353, 2002.

[30] James Rumbaugh and Branislav Selic. *Using UML for Modelling Complex Real-Time Systems*. IBM, 2003.

[31] Abdoulaye Gamatié, Sébastien Le Beux, Éric Piel, Rabie Ben Atitallah, Anne Etien, Philippe Marquet, and Jean-Luc Dekeyser. A Model-Driven Design Framework for Massively Parallel Embedded Systems. *ACM Transactions in Embedded Computing Systems (TECS)*, 10(4), November 2011.

[32] Imran Rafiq Quadri, Abdoulaye Gamatié, Pierre Boulet, Samy Meftali, and Jean-Luc Dekeyser. Expressing embedded systems configurations at high abstraction levels with UML MARTE profile: Advantages, limitations and alternatives, 2012.

[33] P. Kukkala, J. Riihimaki, M. Hannikainen, T.D. Hamalainen, and K. Kronlof. UML 2.0 profile for embedded system design. In *Design, Automation and Test in Europe, 2005. Proceedings*, pages 710 – 715 Vol. 2, march 2005.

[34] Tero Kangas, Petri Kukkala, Heikki Orsila, Erno Salminen, Marko Hännikäinen, Timo D. Hämäläinen, Jouni Riihimäki, and Kimmo Kuusilinna. UML-based multiprocessor SoC design framework. *ACM Trans. Embed. Comput. Syst.*, 5:281–320, May 2006.

[35] Christina Dorotska, Bernd Steinbach, and Dominik Fröhlich. Synthesis of UML-Models for Reconfigurable Hardware. In *UML-SoC*, pages 24–29, 2005.

[36] Lun Li, Frank P. Coyle, and Mitchell A. Thornton. UML to SystemVerilog Synthesis for Embedded System Models with Support for Assertion Generation. In *ECSI Forum on Design Languages*, 2007.

[37] Dag Björklund and Johan Lilius. From UML Behavioral Descriptions to Efficient Synthesizable VHDL. In *20th IEEE NORCHIP Conference*, 2002.

[38] Mauro Prevostini, M. Lajolo, and A. S. Basu. UML Specifications Towards a Codesign Environment. In *FDL'04*, pages 313–325, 2004.

[39] Leandro Soares Indrusiak, Imran Quadri, Ian Gray, Neil Audsley, and Andrey Sadovykh. A MARTE Subset to Enable Application-Platform Co-simulation and Schedulability Analysis of NoC-based Embedded Systems. In *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2012 7th International Workshop on*, July 2012.

[40] A. Wendell Rodrigues, Frederic Guyomarch, and Jean-Luc Dekeyser. An MDE Approach for Automatic Code Generation from UML/MARTE to OpenCL. In *Computing in Science and Engineering*, 2012.

[41] Khronos OpenCL Working Group. The OpenCL Specification. `http://www.khronos.org/opencl/`, November 2011.

[42] Kai Hwang. *Advanced Computer Architecture: Parallelism, Scalability, Programmability.* McGraw-Hill, 1993.

[43] Bil Lewis and Daniel J. Berg. *PThreads Primer.* SunSoft Press, 1996.

[44] OpenMP Architecture Review Board. *OpenMP Application Program Interface.* OpenMP Architecture Review Board, May 2008.

[45] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[46] David W. Wall. Limits of instruction-level parallelism. *SIGARCH Comput. Archit. News*, 19(2):176–188, April 1991.

[47] Monica S. Lam and Robert P. Wilson. Limits of control flow on parallelism. *SIGARCH Comput. Archit. News*, 20(2):46–57, April 1992.

[48] P. Geoffrey Lowney, Stefan M. Freudenberger, Thomas J. Karzes, Woody Lichtenstein, Robert P. Nix, John S. O'Donnell, and John C. Ruttenberg. The multiflow trace scheduling compiler. *The Journal of Supercomputing*, 7(1-2):51–142, 1993.

[49] J.E. Smith and G.S. Sohi. The microarchitecture of superscalar processors. *Proceedings of the IEEE*, 83(12):1609 –1624, dec 1995.

[50] Emre Özer, Thomas M. Conte, and Saurabh Sharma. Weld: A Multithreading Technique Towards Latency-Tolerant VLIW Processors. In *Proceedings of the 8th International Conference on High Performance Computing*, HiPC '01, pages 192–203, London, UK, UK, 2001. Springer-Verlag.

[51] E. Ozer and T.M. Conte. High-performance and low-cost dual-thread VLIW processor using Weld architecture paradigm. *Parallel and Distributed Systems, IEEE Transactions on*, 16(12):1132 – 1142, dec. 2005.

[52] Pradeep K. Dubey, Kevin O'Brien, Kathryn M. O'Brien, and Charles Barton. Single-program speculative multithreading (SPSM) architecture: compiler-assisted fine-grained multithreading. In *Proceedings of the IFIP WG10.3 working conference on Parallel architectures and compilation techniques*, PACT '95, pages 109–121, Manchester, UK, UK, 1995. IFIP Working Group on Algol.

[53] Sanghoon Lee, D. Tiwari, Y. Solihin, and J. Tuck. HAQu: Hardware-accelerated queueing for fine-grained threading on a chip multiprocessor. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 99 –110, feb. 2011.

[54] K.D. Underwood, K.S. Hemmert, A. Rodrigues, R. Murphy, and R. Brightwell. A Hardware Acceleration Unit for MPI Queue Processing. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, page 96b, april 2005.

[55] Burton J. Smith. Architecture and applications of the HEP multiprocessor computer system. In *Advances in Laser Scanning Technology, SPIE Proceedings*, volume 298, page 241, 1981.

[56] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera computer system. *SIGARCH Comput. Archit. News*, 18(3b):1–6, June 1990.

[57] A. Agarwal, J. Kubiatowicz, D. Kranz, B.H. Lim, D. Yeung, G. D'Souza, and M. Parkin. Sparcle: an evolutionary processor design for large-scale multiprocessors. *Micro, IEEE*, 13(3):48 –61, june 1993.

[58] W. Grunewald and T. Ungerer. A multithreaded processor designed for distributed shared memory systems. In *Advances in Parallel and Distributed Computing, 1997. Proceedings*, pages 206 –213, mar 1997.

[59] E.A. Lee. The problem with threads. *Computer*, 39(5):33 – 42, may 2006.

[60] J. Diaz, C. Munoz-Caro, and A. Nino. A Survey of Parallel Programming Models and Tools in the Multi and Many-Core Era. *Parallel and Distributed Systems, IEEE Transactions on*, 23(8):1369 –1386, aug. 2012.

[61] Anant Agarwal and Markus Levy. The kill rule for multicore. In *Proceedings of the 44th annual Design Automation Conference*, DAC '07, pages 750–753, New York, NY, USA, 2007. ACM.

[62] Jenn-Yuan Tsai, Jian Huang, C. Amlo, D.J. Lilja, and Pen-Chung Yew. The superthreaded processor architecture. *Computers, IEEE Transactions on*, 48(9):881 – 902, sep 1999.

[63] Theo Ungerer, Borut Robič, and Jurij Šilc. A Survey of Processors with Explicit Multithreading. *ACM Comput. Surv.*, 35:29–63, March 2003.

[64] Sotirios G. Ziavras, Alexandros V. Gerbessiotis, and Rohan Bafna. Coprocessor design to support MPI primitives in configurable multiprocessors. *Integr. VLSI J.*, 40(3):235–252, April 2007.

[65] M. Gupta, F. Sanchez, and J. Llosa. Merge Logic for Clustered Multithreaded VLIW Processors. In *Digital System Design Architectures, Methods and Tools, 2007. DSD 2007. 10th Euromicro Conference on*, pages 353 –360, aug. 2007.

[66] Hewlet Packard Labs. VEX Tools [Computer Program]. `http://www.hpl.hp.com/downloads/vex/`, 2004.

[67] University of Michigan the CCCP Group. Trimaran: A Compiler and Simulator for Research on Embedded and EPIC Architectures [Computer Program]. `http://trimaran.org/docs/trimaran4_manual.pdf`, April 2007.

[68] Stanford University The SUIF Group. The SUIF 2 Compiler System [Computer Program]. `http://suif.stanford.edu/suif/suif2`, 1999.

[69] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization [Computer Program]. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002.

[70] ARM. ARM: Mali Graphics Hardware. `http://www.arm.com/products/multimedia/mali-graphics-hardware/`, 2013.

[71] Marco Garatti, Roberto Costa, Stefano Crespi-Reghizzi, and Erven Rohou. The Impact of Alias Analysis on VLIW Scheduling. In *Proceedings of the 4th International Symposium on High Performance Computing*, ISHPC '02, pages 93–105, London, UK, UK, 2002. Springer-Verlag.

[72] David Z. Maze. A Flexible Compilation Infrastructure for VLIW and SIMD Architectures, 2001.

[73] Tom Olson. Mali-400 MP: A Scalable GPU for Mobile Devices. `http://www.highperformancegraphics.org/previous/www_2010/media/Hot3D/HPG2010_Hot3D_ARM.pdf`, 2010.

[74] B. Ramakrishna Rau and Joseph A. Fisher. Instruction-level parallel processing: history, overview, and perspective. *J. Supercomput.*, 7(1-2):9–50, May 1993.

[75] S. Banerjee, H.R. Sheikh, L.K. John, B.L. Evans, and A.C. Bovik. VLIW DSP vs. superscalar implementation of a baseline 11.263 video encoder. In *Signals, Systems and Computers, 2000. Conference Record of the Thirty-Fourth Asilomar Conference on*, volume 2, pages 1665 –1669 vol.2, 29 2000-nov. 1 2000.

[76] D. Talla, L.K. John, V. Lapinskii, and B.L. Evans. Evaluating signal processing and multimedia applications on SIMD, VLIW and superscalar architectures. In *Computer Design, 2000. Proceedings. 2000 International Conference on*, pages 163 –172, 2000.

[77] R.P. Colwell, R.P. Nix, J.J. O'Donnell, D.B. Papworth, and P.K. Rodman. A VLIW architecture for a trace scheduling compiler. *Computers, IEEE Transactions on*, 37(8):967 –979, aug 1988.

[78] Shyh-Kwei Chen, W. Kent Fuchs, and W.-M.W. Hwu. An Analytical Approach to Scheduling Code for Superscalar and VLIW Architectures. In *Parallel Processing, 1994. ICPP 1994. International Conference on*, volume 1, pages 285 –292, aug. 1994.

[79] Soo-Mook Moon and kemal Ebcioglu. On Performance and Efficiency of VLIW and Superscalar. In *Parallel Processing, 1993. ICPP 1993. International Conference on*, volume 2, pages 283 –287, aug. 1993.

[80] P. Faraboschi, G. Brown, J.A. Fisher, G. Desoll, and F.M.O. Homewood. Lx: a technology platform for customizable VLIW embedded processing. In *Computer Architecture, 2000. Proceedings of the 27th International Symposium on*, pages 203 –213, june 2000.

[81] A.K. Jones, R. Hoare, D. Kusic, J. Stander, G. Mehta, and J. Fazekas. A VLIW Processor With Hardware Functions: Increasing Performance While Reducing Power. *Circuits and Systems II: Express Briefs, IEEE Transactions on*, 53(11):1250 –1254, nov. 2006.

[82] P. S. Paolucci, P. Kajfasz, P. Bonnot, B. Candaele, D. Maufroid, E. Pastorelli, A. Ricciardi, Y. Fusella, and E. Guarino. mAgic-FPU and MADE: A customizable VLIW core and the modular VLIW processor architecture description environment. *Computer Physics Communications*, 139:132–143, September 2001.

[83] Stephan Suijkerbuijk and Ben H. H. Juurlink. Implementing Hardware Multithreading in a VLIW Architecture. In S. Q. Zheng, editor, *IASTED PDCS*, pages 674–679. IASTED/ACTA Press, 2005.

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

[85] Axilica. FalconML [Computer Program]. `http://axilica.com`, 2013.

[86] Xilinx. ML605 Hardware User Guide. `http://www.xilinx.com/support/documentation/boards_and_kits/ug534.pdf`, June 2012.

[87] David Stevens and Vassilios Chouliaras. LE1: A parameterizable VLIW chip-multiprocessor with hardware PThreads support. In *VLSI (ISVLSI), 2010 IEEE Computer Society Annual Symposium on*, pages 122–126, July 2010.

[88] Robert P. Colwell, Robert P. Nix, John J. O'Donnell, David B. Papworth, and Paul K. Rodman. A VLIW Architecture For a Trace Scheduling Compiler. *IEEE Transactions On Computers*, 37(8), August 1988.

[89] B. Ramakrishna Rau, Vinod Kathail, and Shail Aditya. Machine-description driven compilers for EPIC and VLIW processors. Design Automation for Embedded Systems. Technical report, 1999.

[90] Vassilios A. Chouliaras, K Koutsomyti, T Jacobs, S Parr, David Mulvaney, and Robert Thomson. SystemC-defined SIMD instructions for high performance SoC architectures. In *Electronics, Circuits and Systems (ICECS), 2006 IEEE International Conference on*, 2006.

[91] Xilinx Inc. LogiCORE IP MicroBlaze Micro Controller System (v1.1) [Product Specification]. Online, April 2012.

[92] Xilinx. MicroBlaze Gnu Compiler Collection [Computer Program]. http://www.xilinx.com/tools/microblaze.htm, August 2012.

[93] David Stevens. Thesis Appendix [Code]. http://davestevens.github.com/appendix, April 2013.

[94] ENOSYS. ENOSYS White Paper. http://www.enosys-project.eu/downloads, November 2011.

[95] Mark Milward, David Stevens, and Vassilios 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*, July 2012.

[96] Mark Milward. Jink Design Space Explorer [Computer Program]. Unpublished, 2013.

[97] Softeam. Modelio[Computer Program]. http://softeam.com, 2013.

[98] University of Peloponnese. ACOT [Computer Program]. http://pelopas.uop.gr/UK/, 2013.

[99] Aeroflex Gaisler. LEON3 Multiprocessing CPU Core [Product Sheet]. Online, February 2010.

[100] Xilinx. Xilinx Inc. http://www.xilinx.com/, 2013.

[101] Aeroflex Gaisler. Aeroflex Gaisler. http://www.gaisler.com/, 2013.

[102] Florida State University. Florida state university's pthread (FSU-pthread) prototype library [Computer Program]. Packaged with Sparc-elf-gcc, 2000.

[103] Xilinx Inc. LogiCORE IP XPS Timer/Counter (v1.02a) [Product Specification]. Online, April 2010.

[104] Benoît B. Mandelbrot. Fractal aspects of the iteration of $z \to \lambda z(1-z)$ for complex λ and z. 357(1):249–259, December 1980. International conference on nonlinear dynamics.

[105] F. K. Musgrave and B. B. Mandelbrot. The art of fractal landscapes. *IBM Journal of Research and Development*, 35(4):535 –540, july 1991.

[106] Martin J. Fiedler. NanoJPEG: a compact JPEG decoder [Computer Program]. http://keyj.emphy.de/nanojpeg/, 2012.

[107] X.Org Foundation. The X Window System - X provides windowing on computer displays and manages keyboard, pointing device control functions and touchscreens [Computer Program]. http://www.x.org/, 1987.

[108] Philip Langdale. XInput - utility to configure and test X input devices [Computer Program]. http://www.x.org/archive/X11R7.5/doc/man/man1/xinput.1.html, 2008.

Appendix

This Appendix is also available online at `http://davestevens.github.com/appendix` **where all source code and other files are available to download or view in full.**

Appendix A

LE1 XML Configuration File

LE1 XML Configuration File.

XML file for configuring hardware and software sections of the LE1 tool collection.

```
<galaxy>
 <systems>1</systems>
 <type>homogeneous</type>
 <system>
  <contexts>2</contexts>
  <SCALARSYS_PRESENT>1</SCALARSYS_PRESENT>
  <PERIPH_PRESENT>0</PERIPH_PRESENT>
  <DARCH>DRAM_SHARED</DARCH>
  <DRAM_BLK_SIZE>1</DRAM_BLK_SIZE>
  <DRAM_SIZE>0x100</DRAM_SIZE>
  <STACK_SIZE>0xa</STACK_SIZE>
  <DRAM_BANKS>1</DRAM_BANKS>
  <context>
   <ISSUE_WIDTH_MAX>4</ISSUE_WIDTH_MAX>
   <ISA_PRSPCTV>VT32PP</ISA_PRSPCTV>
   <IARCH>IFE_SIMPLE_IRAM_PRIV</IARCH>
   <CLUST_TEMPL>1</CLUST_TEMPL>
   <HYPERCONTEXTS>1</HYPERCONTEXTS>
   <IFETCH_WIDTH>4</IFETCH_WIDTH>
   <IRAM_SIZE>0x100</IRAM_SIZE>

   <clusterTemplate>
    <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>4</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>4</IALUS>
      <IMULTS>2</IMULTS>
      <LSU_CHANNELS>1</LSU_CHANNELS>
      <BRUS>1</BRUS>
    </clusterTemplate>

    <hypercontext>
      <cluster>0_0</cluster>
    </hypercontext>
  </context>
 </system>
</galaxy>
```

Listing A.1: LE1 XML Machine Model configuration file.

Appendix B

LE1 API available within MicroBlaze

The set_cpu() function sets the control registers within the LE1 to match those defined in "current". "current" is a structure containing

```
typedef struct {
  int sys; /* system */
  int c;   /* context */
  int hc;  /* hypercontext */
  int cl;  /* cluster */
} currentT;
currentT current;

void
set_cpu (currentT current);
```

The ctrl_wait() function polls the CTRL_REG within the current LE1, as specified through the set_cpu() function. A do while loop polls the CTRL_REG and prints each time 1M loops have been performed. (NB: This figure does not give the number of LE1 cycles executed).

```
void
ctrl_wait (void);
```

The load_iram() function takes "inst_length" bytes as pointed to by "instructions" and populates the LE1 instruction RAM. "context_id" is used set the current LE1 being targeted

using the set_cpu() function. Instructions are written to the LE1, read back and checked for consistency. If read instruction differs from written instruction an error is printed and the program exits.

```
void
load_iram (char *instructions, int inst_length, int context_id);
```

The load_dram() function takes "data_length" bytes as pointed to by "data" and populates the LE1 data RAM. Data items are written to the LE1, read back and checked for consistency. If read data item differs from written instruction an error is printed and the program exits. If "MEMCHECK" is specified at compile time the LE1 data RAM not initialised is zeroed out, this is for means of performing a comparison after execution.

```
void
load_dram (char *data, int data_length);
```

The run_application() function initialises the stack pointer with in the current LE1 specified by "context_id" and writes to a control register to start the LE1 executing instructions from the instruction RAM. The stack pointer is calculated using: (("mem_top" - 0x100) - ("stack_size" * "context_id")). Where "mem_top" is the size of the data RAM within the LE1 and "stack_size" is the number of bytes allocated for each contexts stack. This function starts the LE1 context and then calls ctrl_wait(), then returns after the LE1 has completed execution.

```
void
run_application (int context_id, int mem_top, int stack_size);
```

The get_max_context() function performs a write and then read back to all LE1 instruction RAMs to compute the number of valid LE1 contexts in the current system.
Returns the number of contexts available within the current LE1 hardware configuration.

```
int
get_max_contexts (void);
```

The ctrl_wait_multicontext() function is similar to ctrl_wait() function except is loops from zero to "max_context" until all LE1 context complete execution.

```
void
ctrl_wait_multicontext (unsigned max_context);
```

The run_application_multicontext() function is similar to the run_application() function except it starts all available LE1 contexts as specified by "max_context" and then calls the ctrl_wait_multicontext() function. When all contexts have completed execution the function returns.

```
void
run_application_multicontext (int max_contexts, int mem_top, int stack_size);
```

Available if "MEMCHECK" is specified at compile time. This function compares the LE1 data RAM with an array of known correct data RAM which can be produced by the Instruction Set Simulator.
Returns 0 on completion with zero errors or -1 otherwise.

```
int
check_dram_after (void);
```

The dbg_wr() function simply writes "wrdata" to the address specified by "addr".

```
void
dbg_wr (int addr, int wrdata);
```

The dbg_rd() function simply reads from the address specified by "addr".
Return the value read from the LE1 at address "addr".

```
int
dbg_rd (int addr);
```

The busy_wait() function blocks until the busy register is clear. This is used after reads and writes to ensure data is correct.

```
void
busy_wait (void);
```

The write_special() function writes “data” to “special” registers within the LE1 space. (For example; the program counter).

```
void
write_special (int special_reg, int data);
```

The write_cr() function writes “data” to the control register specified by “ctrl_reg”.

```
void
write_cr (int ctrl_reg, int data);
```

The read_cr() function reads from control space registers specified by “ctrl_reg”. Return the value of the register specified by “ctrl_reg”.

```
int
read_cr (int ctrl_reg);
```

The grp_wr() function directly writes “wrdata” to the general purpose register specified by “reg”.

```
void
grp_wr (int reg , int wrdata);
```

Appendix C

LE1 Assembler

The LE1 Assembler used to translate the assembly output from VEX to match the LE1 Instruct Set Architecture. It is formed of a selection of scripts and library code to perform this translation and produce LE1 machine code from running a single script.

An diagram of LE1 Assembler scripts is shown in Figure C.1.

The code can be found online at `http://davestevens.github.com/appendix`

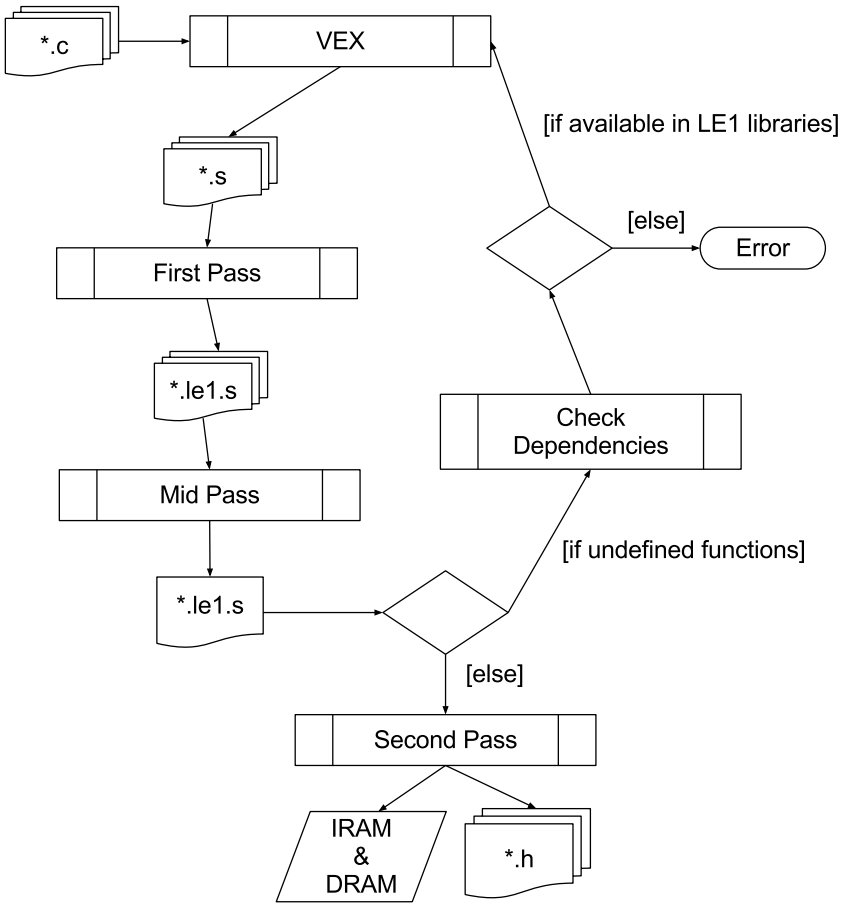


Figure C.1: LE1 Assembler scripts orchestration.

Appendix D

Insizzle

Insizzle is the LE1 VLIW CMP interpreted Instruction Set Simulator. Insizzle provides cycle accurate simulation of the LE1 hardware implementation.

There are two modes of simulation which match the different experiment executions performed throughout the thesis:
A) PThread mode: A single context begins execution, all other are able to have threads created on them.
B) CPUID mode: All contexts in the system begin execution from entry point of the application.

The argument list for Insizzle are shown in Table D.1 and the heuristics produced after successful execution are explained in Table D.2.

The code can be found online at `http://davestevens.github.com/appendix`

Table D.1: Insizzle arguments for LE1 Instruction Set Simulator.

Argument	**Description**	**Required**
model.xml	LE1 XML configuration file	True
-similarIRAM	In a multi-context system all instruction RAMS are filled with iram0.bin	False
-stack=%d	During execution there may be stack size warnings, it is possible to set the stack sizes at command line lever rather than editing the machine model	False
-printout	Displays all instructions being executed along with inputs and outputs (Produces a lot of data)	False

Table D.2: Description of heuristics from Insizzle Execution.

Name	**Description**
Start Time	Timestamp of when Insizzle began execution
End Time	Timestamp of when Insizzle completed execution
Total Time	Time in seconds taken for Insizzle to complete execution on host machine
Per Hypercontext	
cycleCount	Total number of cycles since 0 before HALT operation was executed
stallCount	Number of cycles where the hypercontext was stalled due to pipe refills, memory access, etc.
nopCount	Number of No-Ops executed
idleCount	In pthread mode the number of cycles waiting on other hypercontext (pthread_join)
bundleCount[ARRAY]	Total cycle count broken down into bundle counts
decodeStallCount	Number of cycles stalled due to instruction decoding stalls
branchTaken	Number of branches taken
branchNotTaken	Number of branches not taken
controlFlowChange	Number of times control flow was altered due to call, branch, goto
memoryAccessCount	Number of cycles stalled due to memory conflicts

Appendix E

Modelling Modification Implementations

Methods presented in chapter 4 used for transforming a Platform-Independent UML model to Platform-Specific UML models Implemented in Perl are explained here:

The code can be found online at `http://davestevens.github.com/appendix`

E.1 Modifiable Architecture

arch.pl
Script to modify the Architecture section of the UML model in XMI format using the wildcard multiplicity notation, *HwProcessor*s are found and replicated.
Requires an XMI file, LE1 XML file and output file name.
Generates an XMI file with modified architecture section to match that which is defined in the LE1 XML file.

E.2 Modifiable Application

appl_rr.pl
Script to modify the Application section of the UML model in XMI format using the wildcard multiplicity notation. Objects within the top level class are replicated along with the creation of surrounding structure to communicate with input and output objects. This method includes a fork object which passes subsequent calls to available replicated objects in a round robin manor and a join object which passes calls from multiple objects to a single

output port.
Requires an XMI file, name of top level class and output file name.
Generates an XMI file with extended application section.

appl_split.pl
Script to modify the Application section of the UML model in XMI format using the wildcard multiplicity notation. Objects within the top level class are replicated along with the creation of surrounding structure to communicate with input and output objects. This method includes a fork object which splits a data section over all available replicated objects and a fork object which waits for all forked jobs to complete before passing data to output port.
Requires an XMI file, name of top level class and output file name.
Generates an XMI file with extended application section.

E.3 Modifiable Allocations

alloc.pl
Script to calculate all possible static mappings based on the input UML model. Requires an XMI file, name of top level class, output file name and optionally a directory to generate images in. Generates an XML file containing permutations of all possible static mapping bases on the input UML model. Also generates an image associated with each mapping if required, this enables the viewing off generated static mappings.

E.4 Platform-Specific UML Model Creation

writeStatic.pl
Script to generate a statically mapped UML model, using the XML output from the alloc.pl script and permutation number to generate. It removes all mappings and UML dependencies and generates one of the static mappings defined in the XML file. Requires an XMI file, XML file and permutation to generate. Generates an XMI file containing the statically mapped platform-specific UML model specified by the permutation number in the XML file.

E.5 Miscellaneous

global.pl
Global functions to set MARTE tags dependent on the version used to export XMI, parse command line arguments and setup global variables.

Appendix F

Full Flow Example

UML to simulation on Insizzle, the LE1 Instruction Set Simulator. All files discussed can be found online at `http://davestevens.github.com/appendix`

F.1 UML Model

Creation of UML model within Modelio using the UML design notations defined in chapter 4. Example is based on a Sobel Filter UML model. The Class diagram showing the overall behaviour of the model is shown in Figure F.1 with the structure of the model shown in Figure F.2. The Class diagram display internal operations, attributes and ports of the classes along with interfaces and their realisations contained within the model. The Composite Structure diagram displays the structure of the system captured in the UML model, Figure F.2 displays three objects within a single top level class along with their ports and connections in the application section of the model. Also shown is the architecture section and the mappings between the application and architecture sections. It is in this diagram where the UML notations defined within this thesis are included with the "Sob" object and "i1" *HwProcessor* being tagged with a wildcard multiplicity as well as the static and adjustable mappings between the application and architecture sections of the model.

Classes within the UML model require state machines to define their overall behaviour, these are shown in Figure F.3(a) and F.3(b). The Source class state machine is shown in Figure F.3(a) where an initialisation operation is executed to put the class into a known state followed by an operation to start passing data to the Sobel Filter class. These are linked to transitions within the state machine with no triggers, resulting in them being executed once the class is instantiated as an object within the application. Figure F.3(b) shows the state

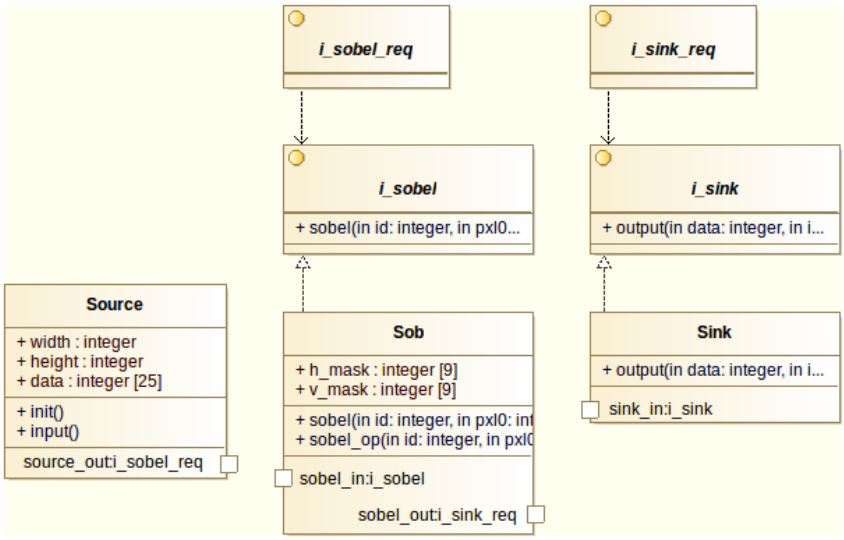


Figure F.1: Class diagram of Sobel Filer UML model

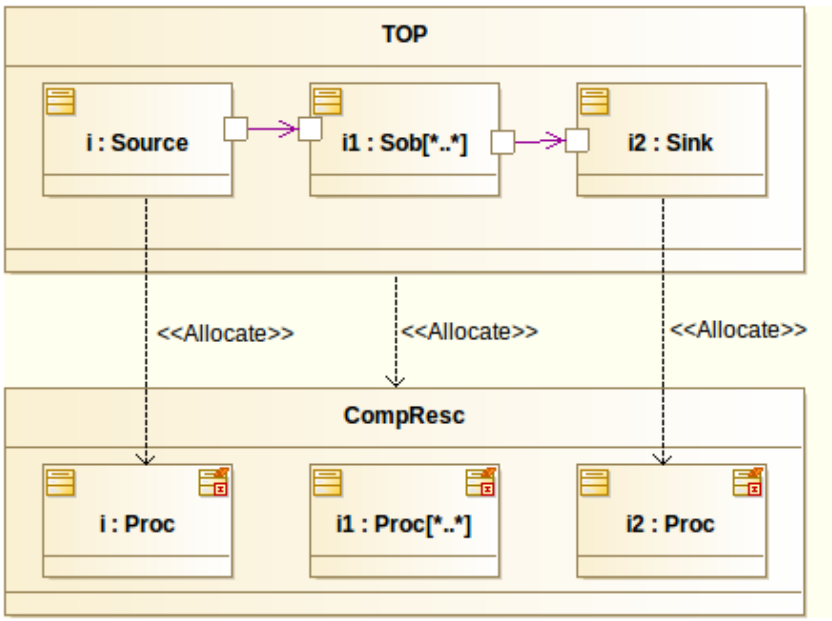
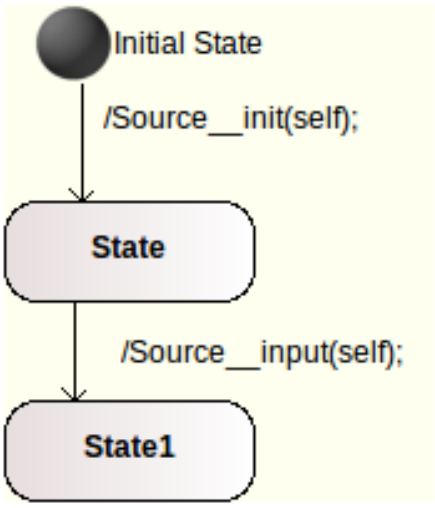
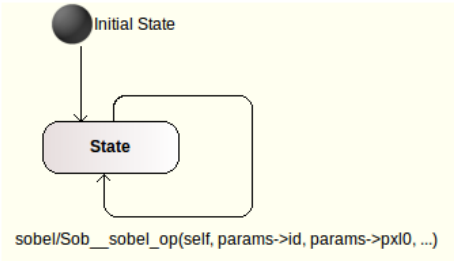


Figure F.2: Composite Structure of Sobel Filter UML model

machine associated with the Sobel Filter class. This differs from the Source state machine in that initially the class transitions to "State" and then waits for a trigger before taking



(a) State machine linked to the Source Class within the UML model.



(b) State machine linked to the Sobel Filter Class within the UML model.

Figure F.3: State machines defining behaviour of Classes within the UML model used for full tool chain example.

a transition back to the same state. This is used allow a continuous flow of information to be passed to the object instantiated from the Sobel Filter class. Each time the transition is triggered an internal operation is called with the passed data which is then computed within the object.

F.2 UML Notation

Running the exported XMI file through the scripts defined above.

As seen in Figure F.2 above the part within "TOP" named "i1" which is typed as "Sob" has a wildcard multiplicity, there is also an object typed as a "HwProcessor" within the HwComputingResource which is tagged with a wildcard multiplicity. This results in a UML model which can be modified to be executed on a base system with three or more cores. For ease of displaying all possibilities a four core LE1 base system was used, this results in 16 possible permutations of static mappings, these permutations are displayed in Table F.1.

To generate these permutations of the possible static mappings the following scripts are executed:

Firstly the architecture section is expanded to match that of "le1/4core.xml" which contains a 4 core LE1 system:

Table F.1: Possible permutations of statically mapped UML models based on execution on a 4 core LE1 base system.

Permutation	**Core**			
	i	**i2**	**i1_0**	**i1_1**
0	i, Sob_0, Sob_1, fork	i2, join		
1	i, Sob_0, fork	i2, join, Sob_1		
2	i, Sob_0, fork	i2, join	Sob_1	
3	i, Sob_0, fork	i2, join		Sob_1
4	i, Sob_1, fork	i2, join, Sob_0		
5	i, fork	i2, join, Sob_0, Sob_1		
6	i, fork	i2, join, Sob_0	Sob_1	
7	i, fork	i2, join, Sob_0		Sob_1
8	i, Sob_1, fork	i2, join	Sob_0	
9	i, fork	i2, join, Sob_1	Sob_0	
10	i, fork	i2, join	Sob_0, Sob_1	
11	i, fork	i2, join	Sob_0	Sob_1
12	i, Sob_1, fork	i2, join		Sob_0
13	i, fork	i2, join, Sob_1		Sob_0
14	i, fork	i2, join	Sob_1	Sob_0
15	i, fork	i2, join		Sob_0, Sob_1

```
perl arch.pl -i Sobel_roundrobin_wildcard.xmi -l le1/4core.xml -o
    Sobel_roundrobin_wildcard_arch.xmi -t TOP
```

Then the application is expanded using the RoundRobin method:

```
perl appl_rr.pl -i Sobel_roundrobin_wildcard_arch.xmi -o
    Sobel_roundrobin_wildcard_appl.xmi -t TOP
```

The allocations are then read and all possible static mappings are generated. “out-

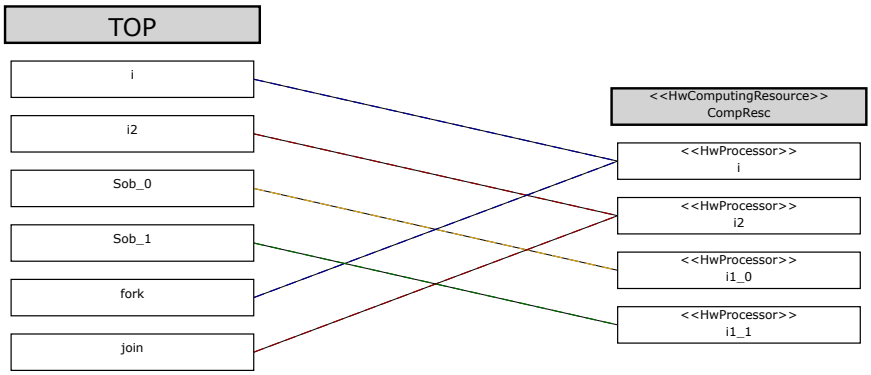


Figure F.4: Mapping example generated by the alloc script

put_images" is a directory which is populated with images showing all possible permutations.

```
perl alloc.pl -i Sobel_roundrobin_wildcard_appl.xmi -o
    Sobel_roundrobin_wildcard.xml -s output_images -t TOP
```

Finally, a statically mapped UML model is generated. Note in this example "11" is passed, resulting in the example shown in Figure F.4 being generated:

```
perl writeStatic.pl Sobel_roundrobin_wildcard_appl.xmi
    Sobel_roundrobin_wildcard.xml 11 > Sobel_roundrobin_wildcard_set.xmi
```

F.3 FalconML

The generated XMI model is then ran through FalconML in order to generate the multicore C code required to execute on the LE1 system:

```
falconml -modelio -target c -top TOP -multicore -use_memory_pool
    Sobel_roundrobin_wildcard_set.xmi falconml_output_directory
```

This produces a set of C files required to implement the UML model which is synthesised by FalconML.

F.4 LE1 Tool Collection

The generated C code is then passed to the LE1 Tool Collection which produces machine code for the LE1, the flags passed to the tool collection are shown below:

```
perl generate.pl falconml_output_directory -syscall -DNUM_EVENT_QUEUES=4 -
    pthread -cpuid -DNDEBUG -xmlMM=le1/4core.xml -DMEM_POOL_SIZE=100000
```

This generates a set of files required for both simulation and execution on a hardware instantiation of the LE1.

F.5 Insizzle

The compiled LE1 machine code can then be executed on either the Simulator or on the synthesised LE1 hardware. Shown here is the execution under simulation, run using:

```
./INSIZZLE machinemodel/model.xml -similarIRAM
```

The output from Insizzle displays the heuristics of each available core, in this case four, including the instruction types which were executed and the number of cycles spent stalling.

```
Insizzle (c1e194a)
PID: 17031
(send SIGUSR1 signal to produce state dump)
homogeneous system
Galaxy setup completed.
Please specify the location of the dram binary for system 0
> file: binaries/dram.bin
  Size of binaries/dram.bin is 0x845e4
  required size is: 67107840 bytes
LOADING BINARY
Please specify the location of the iram binary for system 0, context 0
> file: binaries/iram0.bin
  Size of binaries/iram0.bin is 0xecb8
  required size is: 262144 bytes
LOADING BINARY
Please specify the location of the iram binary for system 0, context 1
> file: binaries/iram0.bin
  Size of binaries/iram0.bin is 0xecb8
  required size is: 262144 bytes
LOADING BINARY
Please specify the location of the iram binary for system 0, context 2
```

```
> file: binaries/iram0.bin
  Size of binaries/iram0.bin is 0xecb8
  required size is: 262144 bytes
LOADING BINARY
Please specify the location of the iram binary for system 0, context 3
> file: binaries/iram0.bin
  Size of binaries/iram0.bin is 0xecb8
  required size is: 262144 bytes
LOADING BINARY
Using Stack Size / hypercontext of 256 KiB
start: 5005
done: 1192857
HALT operation received from [0][2][0] at cycle 1193186
HALT operation received from [0][0][0] at cycle 1193846
HALT operation received from [0][3][0] at cycle 1194340
HALT operation received from [0][1][0] at cycle 1194898
Start Time: 1355935756
End Time: 1355935758
Total Time: 2 (seconds)
performance statistics
  galaxy:          0
  system:          0
  context:         0
  hypercontext: 0
  Cycle breakdown
    cycleCount:        1193847
    stallCount:        429751
    nopCount:          235835
    idleCount:         0
      [0] = 429751
      [1] = 571430
      [2] = 81076
      [3] = 40586
      [4] = 10764
    decodeStallCount: 42547
    branchTaken:       5485
    branchNotTaken:    48699
    controlFlowChange:129068
    memoryAccessCount:31787
performance statistics
  galaxy:          0
  system:          0
  context:         1
  hypercontext: 0
  Cycle breakdown
    cycleCount:        1194899
    stallCount:        319013
```

```
    nopCount:           246914
    idleCount:          0
      [0] = 319013
      [1] = 491020
      [2] = 288829
      [3] = 19274
      [4] = 5449
    decodeStallCount: 119729
    branchTaken:        14002
    branchNotTaken:     5213
    controlFlowChange:66428
    memoryAccessCount:37891
performance statistics
  galaxy:           0
  system:           0
  context:          2
  hypercontext: 0
  Cycle breakdown
    cycleCount:         1193187
    stallCount:         332141
    nopCount:           254575
    idleCount:          0
      [0] = 332141
      [1] = 523382
      [2] = 184318
      [3] = 36710
      [4] = 29267
    decodeStallCount: 96878
    branchTaken:        10502
    branchNotTaken:     27922
    controlFlowChange:78421
    memoryAccessCount:46843
performance statistics
  galaxy:           0
  system:           0
  context:          3
  hypercontext: 0
  Cycle breakdown
    cycleCount:         1194341
    stallCount:         329992
    nopCount:           251704
    idleCount:          0
      [0] = 329992
      [1] = 519368
      [2] = 182639
      [3] = 36112
      [4] = 28347
```

```
    decodeStallCount: 95407
    branchTaken:      10448
    branchNotTaken:   28249
    controlFlowChange:78195
    memoryAccessCount:52970
filename: memoryDump_0.dat
```

Listing F.1: Insizzle simulation output

Appendix G

Leon3MP SMT Modifications

This section includes the Assembly and C code written to modify the Leon3MP system to make it able to execute a small section of the PThread library.

The code can be found online at `http://davestevens.github.com/appendix`

crt0.S
Assembly boot loader of Leon3MP system modified to set stack pointer registers for each CPU in relation to their CPUID as well as only formatting global memory if CPUID is 0.

leon3mp_pthread.h
Header file defining available PThread functions in the Leon3MP system.

leon3mp_pthread.c
Implementation of available PThread functions in the Leon3MP system. Also included are the sections of code used to stall the functions in order to match the execution times of the functions in other library implementations.

platform.h
Header file defining global structures and arrays for Leon3MP PThread platform.

platform.c
Implementation of platform initialisation and cleanup. CPU0 initialises all other available CPUs and then executes main body of code. CPU0 is seen as the master thread in this implementation.

Appendix H

X11 User Interaction Capture

Program code for capturing multiple input devices through the use of the X11 library.

The code can be found online at `http://davestevens.github.com/appendix`

capture.h
Header file including required X11 library files along and definitions of functions to capture devices.

capture.c
Implementation device capture, check specified devices and log key and button presses and releases.

main.c
Implementation to full system to specify input devices to log and then pull all event from X11 event queue and display information as specified.

Appendix I

PThread Execution Results

The cycle counts produced through execution of example applications on alternative architectures as presented in chapter 6 are included in full here.

Table I.1: Execution of Mandelbrot Set threading using coarse grain PThread operations on available architectures.

Platform	**Cores / CPUs**							
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
MicroBlaze	991532686	991579915	991610684	991639734	991684602	991727341	991783111	991817892
Leon3	1469380425	1484432700	1500021675	1502281500	1518018375	1552456800	1554512400	1562260200
Leon3MP	1468194900	737993925	496522875	373056375	301424550	252809400	217061850	191365950
Leon3MP (MicroBlaze)	1468194675	737995950	496529700	373061100	301430700	252818325	217071450	191375850
Leon3MP (Leon3)	1468194900	738009375	497271900	373790100	302517300	253574100	218529525	193535700
LE1	476779180	240094118	161549508	121351939	98032077	82282386	70631182	62231733

Table I.2: Execution of Mandelbrot Set threading using fine grain PThread operations on available architectures.

Platform	**Cores / CPUs**							
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
MicroBlaze	993694500	1127692136	1137219430	1142472937	1146155580	1149012033	1152451534	1153533971
Leon3	-	-	-	-	-	-	-	-
Leon3MP	1470098025	758702700	517788825	396089250	323202750	274188225	244403400	212340600
Leon3MP (MicroBlaze)	1470097875	884308350	686935950	587614650	526643625	485837100	462268350	434414475
Leon3MP (Leon3)	1470098025	15126905100	19679977050	21955768875	23320747800	24231221025	25030385025	25369155750
LE1	516154855	247990545	169700908	130393276	106514808	90550611	80816031	70432697

Table I.3: Coarse grain Mandelbrot Set execution speed up relative to execution on a single core/CPU.

Platform	**Cores / CPUs**						
	2	**3**	**4**	**5**	**6**	**7**	**8**
MicroBlaze	1.000	1.000	1.000	1.000	1.000	1.000	1.000
Leon3	0.990	0.979	0.978	0.968	0.946	0.945	0.941
Leon3MP	1.988	2.959	3.937	4.878	5.814	6.757	7.692
Leon3MP (MicroBlaze)	1.988	2.959	3.937	4.878	5.814	6.757	7.692
Leon3MP (Leon3)	1.988	2.950	3.922	4.854	5.780	6.711	7.576
LE1	1.984	2.950	3.922	4.854	5.780	6.757	7.634

Table I.4: Fine grain Mandelbrot Set execution speed up relative to execution on a single core/CPU.

Platform	**Cores / CPUs**						
	2	**3**	**4**	**5**	**6**	**7**	**8**
MicroBlaze	0.881	0.874	0.870	0.867	0.865	0.862	0.861
Leon3	-	-	-	-	-	-	-
Leon3MP	1.938	2.841	3.717	4.545	5.348	6.024	6.944
Leon3MP (MicroBlaze)	1.661	2.141	2.500	2.793	3.030	3.185	3.378
Leon3MP (Leon3)	0.097	0.075	0.067	0.063	0.061	0.059	0.058
LE1	2.083	3.040	3.953	4.854	5.714	6.369	7.353

Table I.5: Execution of JPEG Decode threading using coarse grain PThread operations on available architectures.

Platform	**Cores / CPUs**							
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Leon3MP	27261300	16758225	14366775	12392475	12513900	12821700	13427250	11237250
Leon3MP (MicroBlaze)	27264000	16781400	14517750	12405450	12553500	12898950	13501050	11310975
Leon3MP (Leon3)	27264375	17142375	15190725	13370400	13959150	14698050	15630225	13818150
LE1	30209314	16769381	13917679	10826805	10929481	11046915	11243771	8482569

Table I.6: Execution of JPEG Decode threading using fine grain PThread operations on available architectures.

Platform	**Cores / CPUs**							
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Leon3MP	3131325	2954775	2948925	2879025	2950725	2993250	2987175	2979375
Leon3MP (MicroBlaze)	3131250	3227475	3542325	3787275	4160700	4501200	4798725	5083350
Leon3MP (Leon3)	3131325	38860650	74803950	110685525	146694900	182665425	218602200	254518725
LE1	3450056	3335499	3345276	3297675	3331164	3339996	3350652	3314955

Table I.7: Coarse grain JPEG Decode execution speed up relative to execution on a single core/CPU.

Platform	**Cores / CPUs**						
	2	**3**	**4**	**5**	**6**	**7**	**8**
Leon3MP	1.626	1.898	2.198	2.179	2.128	2.028	2.427
Leon3MP (MicroBlaze)	1.623	1.880	2.198	2.174	2.114	2.020	2.410
Leon3MP (Leon3)	1.590	1.795	2.041	1.953	1.855	1.745	1.972
LE1	1.802	2.169	2.793	2.762	2.732	2.688	3.559

Table I.8: Fine grain JPEG Decode execution speed up relative to execution on a single core/CPU.

Platform	**Cores / CPUs**						
	2	**3**	**4**	**5**	**6**	**7**	**8**
Leon3MP	1.059	1.062	1.088	1.062	1.046	1.048	1.052
Leon3MP (MicroBlaze)	0.970	0.884	0.826	0.752	0.695	0.652	0.616
Leon3MP (Leon3)	0.081	0.042	0.028	0.021	0.017	0.014	0.012
LE1	1.034	1.031	1.046	1.035	1.033	1.030	1.041

Table I.9: Execution of Sobel Filter threading using coarse grain PThread operations on available architectures.

Platform	**Cores / CPUs**							
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Leon3MP	1215757950	611657400	412095075	311247525	253304775	214910175	190099725	171595875
Leon3MP (MicroBlaze)	1470910500	739912275	498852375	377528550	307923525	262305450	233432775	212422800
Leon3MP (Leon3)	1470910350	740272950	498902925	377814375	308647800	263350800	233908875	214051200
LE1	927618395	479936069	326716224	256128948	210709074	183354844	165937056	153249931

Table I.10: Execution of Sobel Filter threading using fine grain PThread operations on available architectures.

Platform	**Cores / CPUs**							
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Leon3MP	1234326300	669003225	475069275	378237300	318999600	279668775	253175325	236765325
Leon3MP (MicroBlaze)	1234324875	1172965050	1143620175	1136656425	1124633925	1120754625	1121373300	1111458000
Leon3MP (Leon3)	1234324800	57768775875	76722624375	86290815900	91956927225	96035451525	99051313500	100462736775
LE1	928530875	513599925	364332540	289351733	248155821	221338729	204067780	191785239

Table I.11: Coarse grain Sobel Filter execution speed up relative to execution on a single core/CPU.

Platform	**Cores / CPUs**						
	2	**3**	**4**	**5**	**6**	**7**	**8**
Leon3MP	1.988	2.950	3.906	4.808	5.650	6.410	7.092
Leon3MP (MicroBlaze)	1.988	2.950	3.891	4.785	5.618	6.289	6.944
Leon3MP (Leon3)	1.988	2.950	3.891	4.762	5.587	6.289	6.849
LE1	1.934	2.841	3.623	4.405	5.051	5.587	6.061

Table I.12: Fine grain Sobel Filter execution speed up relative to execution on a single core/CPU.

Platform	**Cores / CPUs**						
	2	**3**	**4**	**5**	**6**	**7**	**8**
Leon3MP	1.845	2.597	3.268	3.876	4.405	4.878	5.208
Leon3MP (MicroBlaze)	1.053	1.079	1.086	1.098	1.101	1.101	1.111
Leon3MP (Leon3)	0.021	0.016	0.014	0.013	0.013	0.012	0.012
LE1	1.808	2.551	3.205	3.745	4.202	4.545	4.831

Table I.13: Execution of DES threading using coarse grain PThread operations on available architectures.

Platform	**Cores / CPUs**							
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Leon3MP	168063600	84592125	57748875	42839025	36732825	30140250	26487975	23575575
Leon3MP (MicroBlaze)	168063300	84596175	57735600	42857775	35055750	29847825	25952925	23295900
Leon3MP (Leon3)	168062850	84970425	58454475	44066100	36519000	31197750	27936375	25735500
LE1	159629352	82779262	59900716	48102747	37932640	33379314	30071727	27787002

Table I.14: Execution of DES threading using fine grain PThread operations on available architectures.

Platform	**Cores / CPUs**							
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Leon3MP	167117400	85335150	60366450	46602525	39548325	33885375	30563250	27192525
Leon3MP (MicroBlaze)	167117775	98931075	76645275	64048050	57768375	52594050	49749675	46803375
Leon3MP (Leon3)	167117625	1619264550	2104054650	2344526325	2491062300	2587476300	2657341275	2707097100
LE1	159858703	77750224	58088911	42391540	35793100	31598587	29865978	27720625

Table I.15: Coarse grain DES execution speed up relative to execution on a single core/CPU.

Platform	**Cores / CPUs**						
	2	**3**	**4**	**5**	**6**	**7**	**8**
Leon3MP	1.988	2.907	3.922	4.566	5.587	6.329	7.143
Leon3MP (MicroBlaze)	1.988	2.907	3.922	4.785	5.618	6.494	7.194
Leon3MP (Leon3)	1.976	2.874	3.817	4.608	5.376	6.024	6.536
LE1	1.927	2.667	3.322	4.202	4.785	5.319	5.747

Table I.16: Fine grain DES execution speed up relative to execution on a single core/CPU.

Platform	**Cores / CPUs**						
	2	**3**	**4**	**5**	**6**	**7**	**8**
Leon3MP	1.957	2.770	3.584	4.219	4.926	5.464	6.135
Leon3MP (MicroBlaze)	1.689	2.179	2.611	2.890	3.175	3.356	3.571
Leon3MP (Leon3)	0.103	0.079	0.071	0.067	0.065	0.063	0.062
LE1	2.058	2.755	3.774	4.464	5.051	5.348	5.780

Appendix J

CPUID Execution Results

The cycle counts produced through execution of example applications on alternative architectures as presented in chapter 6 are included in full here.

Table J.1: Speed up of benchmarks on the LE1 using CPUID to perform parallelism, relative to execution on a single core.

Platform	**Cores**						
	2	**3**	**4**	**5**	**6**	**7**	**8**
Mandelbrot Set	1.996	2.959	3.953	4.878	5.650	6.757	7.692
JPEG Decode	1.802	2.165	2.778	2.732	2.725	2.667	3.559
Sobel Filter	1.931	2.801	3.610	4.348	5.025	5.556	6.024
DES	2.079	2.747	3.472	4.405	4.902	5.464	5.814

Table J.2: Speed up of benchmarks on the Leon3MP using CPUID to perform parallelism, relative to execution on a single CPU.

Platform	**CPUs**						
	2	**3**	**4**	**5**	**6**	**7**	**8**
Mandelbrot Set	1.988	2.959	3.937	4.878	5.814	6.757	7.692
JPEG Decode	1.631	1.866	2.203	2.151	2.105	2.008	2.415
Sobel Filter	1.988	2.950	3.906	4.808	5.650	6.410	7.092
DES	1.988	2.899	3.817	4.651	5.348	6.410	7.143

Table J.3: Cycle counts of all benchmarks executing on the LE1 using CPUID to perform parallelism within the applications.

Platform	**Cores**							
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Mandelbrot	476626084	238930539	161218988	120771547	97844959	84144576	70479738	61959510
JPEG Decode	30209399	16767078	13947885	10878791	11042612	11074709	11320455	8483213
Sobel Filter	928635366	481397691	331330070	256990751	213244923	184479216	167426393	153812096
DES	159629373	76786266	58128430	45911083	36222177	32625919	29244886	27503813

Table J.4: Cycle counts of all benchmarks executing on the Leon3MP using CPUID to perform parallelism within the applications.

Platform	**CPUs**							
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Mandelbrot	1468183725	737982900	496525050	373049100	301430025	252815400	217068450	191362350
JPEG Decode	27161700	16655400	14569425	12322875	12620475	12905550	13523400	11243025
Sobel Filter	1215660450	611568150	412035225	311183250	253327275	214915425	190203150	171625650
DES	166939050	84047700	57654000	43768275	35834100	31263975	26042325	23332950

Appendix K

UML Creation Results

User events logged while generating UML benchmarks for various architectural specifications. Both user events, as described in chapter 6, and time are included. Percentages are shown relative to the figures extracted for the creation of the base model and show the amount of extra work required to modify the UML model to fully utilise the underlying architecture. Although a similar trend is seen between the time and user events the reason for the different in percentage of time to generate the base model is due to the user events only logging a single keyboard event when inputting action code into an operation which, depending on the size of the operation, could take many seconds.

Table K.1: Statistics generated from generation of multiple Mandelbrot Set UML models using RoundRobin design, includes both User Events and Time Taken.

Configuration	**User Events**	**Time (Seconds)**	**User Events (%)**	**Time (%)**
Base	630	1905	100	100
3 Core	142	74	23	4
4 Core	440	521	70	27
5 Core	571	656	91	34
6 Core	655	668	104	35
7 Core	662	743	105	39
8 Core	740	826	117	43
Wildcard	159	97	25	5

Table K.2: Statistics generated from generation of multiple Mandelbrot Set UML models using Split design, includes both User Events and Time Taken.

Configuration	**User Events**	**Time (Seconds)**	**User Events (%)**	**Time (%)**
Base	593	1776	100	100
3 Core	152	63	26	4
4 Core	293	209	49	12
5 Core	339	274	57	15
6 Core	395	342	67	19
7 Core	507	414	85	23
8 Core	564	463	95	26
Wildcard	167	85	28	5

Table K.3: Statistics generated from generation of multiple Sobel Filter UML models using RoundRobin design, includes both User Events and Time Taken.

Configuration	**User Events**	**Time (Seconds)**	**User Events (%)**	**Time (%)**
Base	534	1146	100	100
3 Core	152	69	28	6
4 Core	432	538	81	47
5 Core	477	612	89	53
6 Core	531	698	99	61
7 Core	637	795	119	69
8 Core	708	860	133	75
Wildcard	166	96	31	8

Table K.4: Statistics generated from generation of multiple Sobel Filter UML models using Split design, includes both User Events and Time Taken.

Configuration	**User Events**	**Time (Seconds)**	**User Events (%)**	**Time (%)**
Base	508	1231	100	100
3 Core	156	58	31	5
4 Core	259	213	51	17
5 Core	372	289	73	23
6 Core	426	341	84	28
7 Core	477	395	94	32
8 Core	611	499	120	41
Wildcard	166	77	33	6

Table K.5: Statistics generated from generation of multiple DES UML models using RoundRobin design, includes both User Events and Time Taken.

Configuration	**User Events**	**Time (Seconds)**	**User Events (%)**	**Time (%)**
Base	1041	2882	100	100
3 Core	148	93	14	3
4 Core	553	711	53	25
5 Core	584	686	56	24
6 Core	722	761	69	26
7 Core	803	863	77	30
8 Core	766	889	74	31
Wildcard	161	123	15	4

Table K.6: Statistics generated from generation of multiple DES UML models using Split design, includes both User Events and Time Taken.

Configuration	**User Events**	**Time (Seconds)**	**User Events (%)**	**Time (%)**
Base	1030	2698	100	100
3 Core	158	65	15	2
4 Core	296	278	29	10
5 Core	409	320	40	12
6 Core	424	378	41	14
7 Core	522	441	51	16
8 Core	579	506	56	19
Wildcard	157	79	15	3

Appendix L

UML Execution Results

Table L.1: Cycle counts of UML models generated to match specific architectural templates executed on LE1 architecture.

Benchmark	**Type**	**Cores**					
		3	**4**	**5**	**6**	**7**	**8**
Mandelbrot Set	RoundRobin	4538352	2654619	1939050	1681681	1455082	1392839
Sobel Filter	RoundRobin	1577150	1188667	921733	879257	868846	853038
DES	RoundRobin	4843018	2788196	1866686	1529151	1294099	1260854
Mandelbrot Set	Spit	1794543	877295	467779	320965	357554	177271
Sobel Filter	Split	560801	280528	184188	134998	105662	85576
DES Encryption	Split	4060537	2035662	1426747	1078019	874280	625565

Table L.2: Cycle counts of UML models generated using wildcard UML notation and modified to fit alternative architectural template executed on LE1 architecture.

Benchmark	**Type**	**Cores**					
		3	**4**	**5**	**6**	**7**	**8**
Mandelbrot Set	RoundRobin	5040311	2638241	1952824	1643023	1469923	1389665
Sobel Filter	RoundRobin	1971750	1189263	919126	879713	869684	876498
DES	RoundRobin	5436725	2782548	1826698	1450682	1385417	1179274
Mandelbrot Set	Spit	1808716	941848	954158	689650	674658	615068
Sobel Filter	Split	569226	290232	188369	138354	166561	88928
DES Encryption	Split	4199028	2109589	1480685	1132517	938412	829370