THE UNIVERSITY *of* EDINBURGH

Automated Application-Specific Optimisation of Interconnects in Multi-Core Systems

Oscar Almer

Doctor of Philosophy
Institute of Computing Systems Architecture
School of Informatics
University of Edinburgh
2012

Abstract

In embedded computer systems there are often tasks, implemented as stand-alone devices, that are both application-specific and compute intensive. A recurring problem in this area is to design these application-specific embedded systems as close to the power and efficiency envelope as possible. Work has been done on optimizing single-core systems and memory organisation, but current methods for achieving system design goals are proving limited as the system capabilities and system size increase in the multi- and many-core era. To address this problem, this thesis investigates machine learning approaches to managing the design space presented in the interconnect design of embedded multi-core systems. The design space presented is large due to the system scale and level of interconnectivity, and also feature inter-dependant parameters, further complicating analysis. The results presented in this thesis demonstrate that machine learning approaches, particularly *wk*NN and random forest, work well in handling the complexity of the design space. The benefits of this approach are in automation, saving time and effort in the system design phase as well as energy and execution time in the finished system.

Acknowledgements

Acknowledging Nigel Topham for his support and expertise in arranging experiments, research, finance, and engineering.

Acknowledging Björn Franke for his support in experimental setup, analysis, and preparation of text.

Acknowledgement also goes to Igor Böhm and Richard Vincent Bennett, and my other office partners, without whom I would have been even more lost.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

Chapter 4 is based on a paper [4] published at *ARCS2011* authored by myself, Nigel Topham and Björn Franke.

Chapter 5 is based on a paper [5] published at *NoCArc '11* authored by myself, Miles Gould, Nigel Topham and Björn Franke.

In addition, the papers [2, 3] are also authored by, among others, myself, but these papers are not an intergral part of this thesis. Chapter 3 contains some paragraphs originally written by Nigel Topham or Richard Vincent Bennett for an unpublished paper regarding the EnCore which have been adapted for inclusion.

(*Oscar Almer*)

Table of Contents

List of Figures

Listings

List of Abbreviations

ASIC ..... Application-Specific Integrated Circuit
AXI ..... Advanced eXtensible Interface
BRAM ..... Block RAM
BVCI ..... Basic Virtual Component Interface
CPU ..... Central Processing Unit
DDR ..... Double Data Rate
FIFO ..... First In, First Out
IC ..... Integrated Circuit
I/O ..... Input / Output
FPGA ..... Field-Programmable Gate Array
ISE ..... Instruction Set Extension
JTAG ..... Joint Test Action Group
NOC ..... Network-on-Chip
NRE ..... Non-Recoverable Engineering
RAM ..... Random Access Memory
SOC ..... System-on-Chip
SVM ..... Support Vector Machine
UART ..... Universal Asynchronous Receiver / Transmitter
VLSI ..... Very Large Scale Integration

1 Introduction

This chapter presents a brief background and motivation for the thesis, and serves as a guide to its contents.

1.1 Background

It is a feature of modern society that we take small, powerful computing devices for granted. Some of these devices are deeply embedded, in that their function is internal and is a black box, whereas others are user accessible. A given, modern, consumer product may contain several of each kind; for example, a mobile phone handset may contain deeply embedded display controllers and GPS receivers as well as user accessible applications processors.

Whether deeply embedded or user accessible, computing devices are essential in modern commercial, social and private life; their performance and features are thus important for the commercial success of a great many products. Manufacture of these devices is through standard methods and is a well-known and -tuned process. The manufacturing process affects to some extent how the devices perform in terms of temperature, energy consumption, and application performance, but this is within an error bound of a standard device. What we need to look at to improve these devices, then, is how they are designed.

The design of a device starts with the function the finished device is to perform, which then defines the internal structure and elements needed to fulfill this function. Secondly, requirements in terms of timing and functionality are identified so that the desired function is accomplished, and the device is then engineered to fulfill those requirements. These devices are termed System-on-Chip (SOC), because they are in general stand-alone silicon chips comprising a complete albeit small computing system. Engineering of a SOC requires consideration of the sub-blocks of the device and how they are connected, as well as of the software (usually firmware) that the device is expected to run.

Sub-elements in such a SOC commonly include processors, input and output interfaces, memory blocks, and certain fixed-function hardware blocks. These elements are usually pre-existing, with some specific to the design at hand, and are interconnected in the design to form the complete SOC. The requirements of the finished SOC have to

be met by a combination of the sub-elements, and importantly, by the interconnection method.

Some large SOC designs like to regularize the interconnect as a network of components, usually referring to the interconnect as a network-on-chip (NOC). Whether the connections between blocks in a SOC are thought of as a NOC or an interconnect is irrelevant from a larger perspective; both terms will be used in this thesis.

A classic approach to designing the interconnect of an SOC is to over-provision in bandwidth, power, *et cetera*, so that the final design will be more power hungry and larger than necessary but will definitely meet its requirements. Alternatively, one could spend a lot of Non-Recoverable Engineering (NRE) resource on a single SOC to optimize the chip as a whole, including the interconnect. Both of these approaches incur direct costs to the customer, one in terms of running costs (higher power consumption) and one in terms of up-front costs (more expensive SOC). It is therefor clear that a cheap, fast, method of optimizing the interconnect in SOC designs will save money for both manufacturers, and indirectly, customers.

In most cases the scope for choices in how to design the interconnect is large or very large, depending to some degree on the number of sub-elements in the system, but also on the data-flows required and bandwidths necessary. The number of different ways in which to construct the interconnection network for a given SOC is easily tens if not hundreds of thousands; but not all of those ways will fulfill the design goals. Some designs will be viable for performance, functionality, or design requirements reasons, whereas others will not. Finding these designs then becomes the task of finding even a viable design among the hundreds of thousands of possible designs. Completely enumerating and testing these design spaces are prohibitively expensive, as testing a single design may take many hours of compute time, leading to optimistic enumeration times in the millions of hours CPU-time, which is not conducive to fast and cheap evaluation.

There are thus several problems of scale with the design space of interconnect configurations. Firstly, only with large effort can the design choices that lead to a non-working SOC be found, so that they can be eliminated from further consideration. Secondly, much effort needs to be expended in finding those designs that are optimal in some regard, such as power consumption or throughput. Thirdly, there is a need to repeat the process, spending the same effort again, if there is a change in requirements, or even if there is a change in the firmware, as the optimal point may have shifted. And fourthly, the effort used in problems one through three is prohibitively expensive in terms of both engineering and compute time.

As a contrast, a novel method using limited design-space exploration and machine

learning over these design spaces is presented. This method does have an initial, up-front cost of having to test some of the designs in the space, but the cost can be a minuscule fraction of the cost of a complete enumeration. Even with such a small part of the design space explored, machine learning methods trained on the resultant data are able to predict the performance of designs not yet implemented, and to do so in a fraction of the time a full design test would take. An iterative approach may be taken, where the predicted best points are tested and the results fed back to the machine learning system, improving its predictions and quickly finding optimal configurations, while only fully implementing a fraction of a percent of the full design space. In addition, should the design require changes to the software, the machine learning methods are able to predict a new optimal point in the already explored design space with a single run of the new software on an already implemented design. Iterating through the design space as above may then be used to find even better points, if desired.

This thesis presents this novel design-space and machine learning approach in detail, and validates the approach through experiments on actual synthesized hardware designs.

1.2 Motivation

In small geometries, such as 32 and 28 nm, the number of transistors available in even small silicon devices (10 mm^2) is large enough for tens of processor cores. The ARM A7 is cited to use .45 mm^2 in 28 nm [9], meaning 20 of these cores can be integrated onto a 10 mm^2 die; in addition, this core includes advanced and area-hungry features such as floating point and branch prediction. Simpler and more efficient cores may use even less area per core, increasing the amount of available cores in a given chip size.

These multicore SOC designs, due to their multicore nature, vary widely in energy and runtime performance depending on scheduling, cache contents, and other applications present. Optimising the interconnect and system in such an environment is a complex problem; even more so if the variables of cache sizes, I/O latency, and clock frequencies are also available, as they are during SOC design time.

Figure 1.1 shows the distribution of energy and runtime performance of a single fixed application run on 71 different SOC designs from Chapter 5. These 71 designs are taken from a SOC design space of 510,000 total possible systems, and represent a random sample from that design space. Evaluating the full design space would take a very long time, and therefore a randomly chosen sample is used here. The

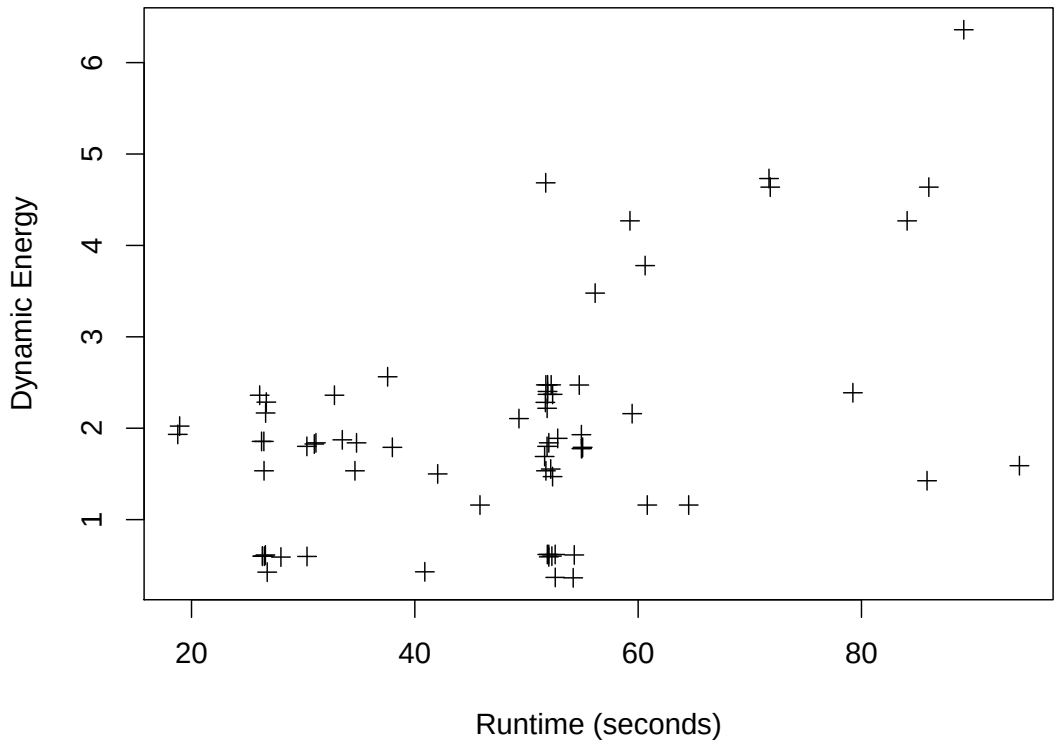


Figure 1.1: Spread of performance of a single application over 71 different but related SOC systems. Data from the dataset presented in Chapter 5.

confidence interval of this sample is ±5% and and the confidence level is 97.5%. In this sample, the spread of runtime is between 20 and 120 seconds; the dynamic energy measurements similarly vary by a factor of 12. This spread shows that by selecting the right SOC and, crucially, interconnect design up to a factor of 5 in runtime and up to a factor of 12 in energy can be saved. Very few of the SOC designs are located on the Pareto curve, indicating that most designs in this space are inefficient in energy consumption, runtime, or both.

Because of these potential savings, and the expertise needed to evaluate the design tradeoffs, this thesis applies machine learning methods to the field of SOC interconnect design to automate the design selection process.

1.3 Contributions

This thesis presents novel techniques that address the problem of NOC optimization and NOC specialization. These contributions are as follows.

1. By using complete or limited design space exploration over an interconnect design space an understanding of the performance trade-offs in a given implementation can be gained.

   (a) Showing that a NOC implementation can be varied in ways that generate potentially very large design spaces.
   (b) Showing that the design space exploration does not need to be exhaustive to find good designs.
   (c) Showing that good points are generally close by in some dimensions of the design space.

2. Showing that by using machine learning on our design space data, it is possible to predict points that are optimal in some regard.
   (a) By evaluating several machine-learning and statistical methods, suitable methods for the field can be found.
   (b) Showing that accurate predictions of optimal points in the design space for previously unseen applications can be achieved.
   (c) Showing that accurate predictions the energy and runtime performance of a known application on a new design can be achieved.
   (d) Showing that accurate predictions the energy and runtime performance of an unknown application on a known design can be achieved.
   (e) Showing that in the presence of design system faults, the level of functionality of a new design can be predicted.

3. Showing that these methods can be used to cut down on search time for new optimal designs in the design space.
   (a) showing that the presented method is scalable to large design spaces and sparse exploration of these design spaces.

4. Showing that using real synthesisable hardware allows for deeper evaluation than using simulated results.
   (a) Showing that by using real programs, real hardware designs and synthesis flow the approach is suitable for the field.
   (b) Showing that by using real programs, real hardware designs and synthesis flow the accuracy of the metrics can be ensured.

5. This is the first time machine learning methods have been applied to the design space of synthesisable SOC designs.

1.4 Thesis structure

Chapter 2 discusses theoretical background to the contributions presented in this thesis, covering a wide range of topics that are used throughout this thesis. This chapter is focused on the necessary technical background to understand the rest of the thesis.

Chapter 3 discusses the specific experimental setups used in this thesis, including hardware systems engineering and tool flow specifics. This also includes details on the design space variables and configurations used in chapters 4 and 5, as well as detailed discussions on the machine learning methods employed.

Chapter 4 covers initial experiments in machine learning over the interconnect space, targeted at the small-scale embedded systems space. A small design space of about 250 distinct designs is discussed and exhaustively enumerated to find the optimal points in this space for various applications. This chapter also develops and use a flavour of *wk*NN to predict the best design for new applications based in this design space. This chapter is based on a paper published at ARCS 2011 [4] authored by myself, Nigel Topham and Björn Franke.

Chapter 5 covers further experiments in machine learning over the interconnect space, with a view towards large-scale many-core architectures. The use of SVM and *wk*NN to predict the best tested design for a larger and much more capable design space is investigated. It is found that *wk*NN is in almost all cases useful for these types of problems, and that using machine learning in this space can save time designing application-specific systems. This chapter is based on a paper published at the NocArc Workshop 2011 [5], authored by myself, Miles Gould, Nigel Topham and Björn Franke.

Chapter 6 further looks at the data from chapter 5 and attempt to find a good method for efficiently predicting the performance of entirely unknown programs and designs. This chapter focuses on evaluating machine learning methods and how accurate they are on this type of data. It is found that different approaches work for predicting area, runtime, and power of new design points, and that all methods take less than a few seconds for predictions, showing significant improvements over brute-force enumeration.

Chapter 7 Summarises the work and ideas presented and concludes the thesis.

1.5 Summary

This introduction has presented the field and specified the problem this thesis aim to address, *viz.* a novel design-space / machine learning approach to the problem of SOC design. I have further reinforced the need for such a method through a concrete example of a deeply embedded system.

In addition, the particular contributions that this thesis makes to the larger body of knowledge have been identified. And finally, an outline of the further parts of the thesis have been presented.

2 Background

This chapter contains technical background material considered relevant to the rest of this thesis. The focus here is on giving an understanding of the background problems and their solutions, and how these may affect the larger system assembled in Chapter 3. To meet this goal, this chapter surveys the literature, describes various relevant experimental methods, and presents brief discussions on relevant issues.

This chapter is ordered as follows. Section 2.1 covers circuit design, implementation fabrics, background information on interconnects and their properties, a summary of embedded processors, and an overview of application and processor specialisation. Section 2.2 covers the mathematical and literary background of the machine learning methods used in this thesis. Section 2.3 summarizes this chapter.

2.1 Embedded systems

Embedded systems cover the area of computing that are broadly defined as 'systems designed to perform a dedicated function', i.e. systems such as mobile phones, set top boxes, industrial automation, engine control, and a multitude of other applications. All these tasks benefit from the increased capabilities offered by having programmable control, but only rarely is the programmability exposed. This is slowly changing as the capacity and capability of embedded systems is increasing.

This section discusses the background to the technology and circuit issues encountered in Chapter 3, in particular synchronicity in sections 2.1.1.1 and 2.1.1.2, FPGA technology in section 2.1.1.3 and ASIC implementation technology in section 2.1.1.4. Further, background information on Interconnects and NOCs is presented in section 2.1.2, covering the features and properties of various NOC types (section 2.1.2.1) as well as the background of the particular AXI interconnect used (section 2.1.2.2), followed by a discussion of previous attempts at automating the memory interface synthesis (section 2.1.2.3). Section 2.1.3 discusses the processing elements common in embedded systems (section 2.1.3.1), and linking back to the ASIC background, presents expected development trends for future embedded systems (section 2.1.3.2); further a brief discussion on embedded applications is presented (section 2.1.3.3) as well as background to the specialisation of processors and applications in embedded systems and how this is complementary as well as similar to the approach presented in this thesis (section

2.1.3.4). Finally, section 2.1.4 summarizes the embedded system background.

2.1.1 Technology and circuits

Circuit technology, and the ability to quickly and easily design large circuits, is central to the problems and techniques presented in this thesis. As such, a brief background look at how we design large circuits and the problems of timing and asynchronicity is warranted; these issues are covered in sections 2.1.1.1 and 2.1.1.2, respectively.

Very Large Scale Integration (VLSI) is the name used for integrated circuit designs larger than, approximately, a thousand transistors. This name is now on the way out and has largely been supplanted with application-specific integrated circuit (ASIC), which has many of the same connotations of large, custom integrated circuit. This thesis will use both terms where appropriate. Field-Programmable Gate Array (FPGA) technology, by contrast, still has the same name as when it was invented, and is covered in section 2.1.1.3. Current ASIC technology and its constraints is covered briefly in section 2.1.1.4

2.1.1.1 Synchronous circuit design

High-level synchronous circuit design is made simple by the use of hardware design languages, such as VHDL [63] and Verilog [64]. These languages derive their utility from being able to describe the behaviour of a circuit rather than the components. This behavioral description is an abstraction level higher than the circuit description.

As these languages describe synchronous circuits on the level of functionality and behaviour rather than circuit detail, they are faster and simpler to work with than direct low-level circuit details, which may contain too much detail for quick analysis and understanding. This also allows synthesis tools to optimize the desired logic, i.e. to do the low-level circuit optimizations, automatically, and to tailor it for the logic fabric in question. Combined, these features allow for increasing the productivity of human designers combined with automatic logic optimizations, allowing the conceptualizing of larger and more capable designs. The designer must still understand the specifics of the design in order to be able to make trade-offs for fabric, time, clock cycles, and other design parameters, but this is made easier.

Thus the problem of large-scale synchronous design of electronic circuits is addressed by abstractions and high-level specification, enabling the type of design that the rest of this thesis will develop. The main implementation work in this thesis was carried out using Verilog and with synthesisability in mind. No particular synthesis tool-chain was targeted in the Verilog code itself to allow for both FPGA and ASIC synthesis paths.

2.1.1.2 Multiple clock domains

Synchronous circuits have issues when dealing with unsynchronised inputs. These issues are rooted in the fact that there is no guarantee that the input will not occur in any particular relation tot he synchronising clock.

Such inputs may be generated by truly external devices, such as I/O devices receiving new data from a remote source, or by synchronous systems operating in another clock domain. Clock domain here refers to a set of synchronous elements which are clocked by the same clock; these elements are in the same clock domain. If two sets of internally synchronous circuits are not clocked by the same clock or a division of the same clock, they are effectively asynchronous with respect to each other.

Transferring data between two internally synchronous clock domains is an issue due to the hazard of changing the inputs to an element very close to the clock transition. This may lead to unpredictable behaviour of the receiving element, generally causing a failure in the system. This is referred to as Metastability [46, 71], and is a problem wherever differently clocked synchronous systems intersect. Due to the problems with distributing a clock in VLSI systems, it is possible that different parts of a single silicon die are effectively operating asynchronously, meaning that this is a real problem in VLSI design.

One may even design a SOC to have different clocks for different purposes; i.e core clock, bus clocks, etc. Every one of these clock domains is effectively asynchronous to each other, unless controlled precisely, and in every case care needs to be taken when data crosses such a clock boundary.

In general the metastability problem is solved by a synchroniser circuit [34] that resynchronises a asynchronous signal to a local clock, generally by dedicating a number of flip-flops (and clock cycles) to overcome the problem. The usual number of flip-flops to delay by is two, but this may be increased if it is found that the particular process and implementation chosen still has an unacceptably high chance of entering metastability. In addition, asynchronous FIFO [26] blocks may be used to cross these boundaries with different latencies.

2.1.1.3 FPGA technology

A FPGA (field programmable gate array) device consists of a number of specific-function circuits, linked by a large network of wires. By programming the connectivity of these wires, the function of the entire device may be changed, so that the same device can be used for different purposes without re-fabrication. The programming information for the connectivity is generally kept in an off-chip storage system, which is read im-

mediately on power-on to set up the FPGA correctly, implying that a FPGA device is not ready for operation immediately after applying power.

Originally, FPGA devices consisted of simple gates connected by configurable wiring, but over time as technology and manufacturing ability has increased, the functionality in a FPGA device has likewise increased. In a modern FPGA, such as the Virtex-6 family [75], there are numerous specialized blocks such as memory blocks as well as logic function blocks. Due to the size and complexity of modern FPGA devices, it is possible to support several embedded processors and even full-scale consumer processors [72].

This thesis uses FPGA devices from Xilinx, Inc., from the Virtex-5 and-6 families, specifically, devices with model numbers XC5VFX70T and XC6VLX240T. The first is a Virtex-5 device from the 'embedded processor' line, signifying that it has a PowerPC core in addition to reconfigurable logic; this core was never used. This particular Virtex-5 part also has 11,200 reconfigurable slices and 148 block RAMs, as well as some hard multiplication blocks and I/O resources [74]. The second device is a Virtex-6 device with 37,680 available slices in addition to 416 block RAM instances and the requisite I/O blocks [75].

The basic unit of measurement in Xilinx FPGA devices, the slices, each consist of four independent units, which in turn consist of a 6-input look-up table that connect to two flip-flops. The look-up tables can also be configured as two-output 5-input tables, justifying the second flip-flop each; and a slice thus has 8 flip-flop outputs with up to 24 independent inputs. The function of the look-up tables is fully programmable, and forms the basic unit of reconfigurability. In this way the logic function of any given slice is fully programmable, leading to fully customizable devices. The main drawback of FPGA technology over ASIC is the reduced speed of operation; due to the inability to optimize the routing and logic, a FPGA implementing a function will have a lower maximum operating frequency (f_{max}) compared to a ASIC. Because of this, FPGA devices are used where performance is less of a concern than fast reprogrammability, such as for testing, evaluation, and short time-to-market applications.

Designing circuits specifically for FPGA implementation is done in some cases, when it is known that the functionality needed will only ever be instantiated in a FPGA. Such circuits usually take advantage of the flip-flop-rich environment of the FPGA, increasing the clock rate accordingly, but also on a design-level lengthening the pipelines and possibly introducing longer delay times. In contrast, structures designed solely for ASIC implementation tend to fare badly in straight FPGA synthesis, as they are optimized for a different flip-flop to gate ratio.

2.1.1.4 ASIC technology

Application-specific integrated circuits (ASIC) is a generic term for integrated circuits that are made to fulfill one purpose, as opposed to gate arrays and other more generic devices. These are at the very other end of the spectrum from a FPGA, in that they are wired and set up specifically for a single purpose. As such, they can be faster and more capable than a FPGA, as there is no need to pay for the generic structures so prevalent in a FPGA. The drawback is that manufacturing time and testing is a much longer, more involved, and expensive process, and thus is not generally undertaken unless there is a market for the resultant device.

ASIC technology is the method primarily chosen for implementing embedded processors and systems for customer use. As such, the definition of ASIC is slipping somewhat, as some systems are quite generic and let the software specialize the chip, saving manufacturing costs. The term ASIC has thus come to mean generic instantiation of a circuit as a silicon device, for all but the most mainstream ICs.

The manufacturing technology for integrated circuits has been improving steadily over the past decades, and has reached the point where transistor gates are tens of atoms across. The 45nm technology node is discussed here as an example; the measurement refers to the half pitch of an average DRAM cell. Within this technology node there is some variability between processes, but according to [54] the physical gate length of Intel's 45nm process is 35 nm. The lattice spacing of pure silicon is 0.5430710 nm, fractionally more for doped silicon [56], and as a result the gate in this process node is approximately, on average, 64 atoms wide. At these levels, and with future technology nodes at 22nm and 16nm the limits of practically manufacturable circuits is approaching. It may also be noted that the patterning passes in the lithography of manufacturing integrated circuits has been using ultraviolet light, with a wavelength that goes down to 193nm, meaning that there are significant detail resolution issues. These effects, and others, contribute to the state of the art in ASIC manufacture; that the devices manufactured has a large spread of performance, energy consumption, and in some cases may fail early. This is referred to as process variation [15], and is a simple result of pushing the envelope of what can physically be manufactured. Process variation also affects the interconnect wiring of any manufactured devices, and thus also any NOC type structures on may wish to build [39]. The impact of the variability in the manufacturing process manifests itself as performance variation between wafers, within wafers, and even within single chip dies. To manage this problem in a competent manner, therefore, we need not only device-manufacturability awareness, but also design-level awareness of these issues and their impact.

Processor core sizing and capability is one issue that may need reevaluation due to process variability [55]. This is discussed in section 2.1.3.2, but also summarized here. If a processor core is larger than the area in which variability can, in some sense, be regarded as constant, the entire core is forced to operate on the worst-case corner, resulting in a lower f_{max} and higher voltage requirements. Using smaller, independently clocked and managed, cores may be a better solution as it may allow better utilization of the area, letting some parts run faster than others to gain back some performance, or, potentially, benefit from a particularly favourable variation.

Thus, designing systems for ASIC implementation is getting more complex as technology nodes have progressed, and it may be necessary to make architecture-level changes to manage the impact of process variability. In addition, dynamic management of the processing environment, allowing individual deviation in manufactured devices, is something that should be considered. This thesis takes a large-scale architecturally independent approach to system design, using processing cores clocked asynchronously to the interconnect and memory, and this could be used to implement the kind of system that is expected to be resilient to process variability.

2.1.1.5 Technology and circuits summary

This section has discussed some of the background of the technology used in ASIC and FPGA design. In essence, one of the major problems in large-scale digital design is the clock generation and distribution, which together with upcoming issues in ASIC development will serve as a bottleneck for performance in terms of clock frequency. FPGA technology serves as a intermediary before committing to ASIC manufacture as well as a system building tool for non-performance-critical applications. This thesis uses mainly FPGA technology in the context of test systems, as a substitute to ASIC manufacture, to save time and effort. As such, an awareness of ASIC design parameters is required for the test systems presented.

2.1.2 Network on chip and interconnects

The field of Networks on Chip (NOC) is relatively new, but has roots in the work that has been done in supercomputer interconnects as well as some aspects of VLSI design. There are some differences between the possibilities presented in on-chip wiring and the supercomputer schemes [68]. These differences manifest through the different constraints of the on-chip network versus the inter-chip network; on-chip networks can contain a much larger number of wires, but is constrained in the routing of wires due to the Manhattan routing used in IC manufacture. In addition, the amount of buffer memory is sharply limited on-chip, and finally, the routers compete with the

cores and I/O devices for on-die area, and thus need to be as small as possible while still being efficient. These differences has led to the common inception of a NOC as a square mesh network, with other topologies having a minority role [14, 60], although other topologies have been investigated.

A basic treatment of interconnects for supercomputers can be found in [37], a more thorough treatment is found in [28]. On-chip interconnects and network on chip architectures are largely treated as synonymous, and it is generally accepted that an on-chip interconnect is synonymous with routing packets between nodes. Much effort has gone into NOC architectures, topologies, and routers [6, 10, 29, 44, 45].

Strictly speaking, networks on chip is a catch-all term for any on-chip systems interconnect that utilizes non-static routing, whereas the term interconnect can be used for systems with static routing. This taxonomy is not fixed, as some levels of a system interconnect may be statically routed while other levels may not be; as such, there is a large grey zone between a pure NOC and an interconnect. In addition, there is a propensity in the literature to refer to all interconnects as NOC. It could be argued that a NOC cannot be built without it also being an interconnect.

Nevertheless this section discusses topologies and bandwidth of interconnects together with a note on their general usability. Much of this section is inherited from supercomputer networks, but adapted due to the different constraints on a SOC.

2.1.2.1 Topology

Topology in an interconnect defines its basic routing strategy as well as, ultimately, its cross-section bandwidth. Various topologies have been suggested in this area [53], including full crossbar, reduced crossbar and butterfly networks [42, 61], as well as extending the crossbar in the other direction, to a hypercube [27]; rings have also been investigated [41].

Much of the work on differing topology has been done with supercomputers, where point-to-point links can be done diagonally a lot easier than on-chip. Other differences, such as the normalised length of the interconnect and power requirements, also differs from-on chip implementations. Primarily, however, supercomputer interconnects are perceived as easier to reroute and experiment with than static on-chip structures, leading to a relative stagnation in on-chip topology.

The general thrust of NOC research has agreed that mesh networks are the best solution due to power consumption, latency, and simple implementation [14]. A lot of the literature in the NOC area concerns regarding the optimal routing strategy and switch architecture assuming a square mesh network.

The original problem that interconnects are addressing occurred when trying to

connect telephone subscribers to any other telephone subscriber in parallel fashion, so that there could be a large number of subscribers on the phone at once. Thus the crossbar solution appeared, which is simply the amalgamation of a large number of switches, located so that any incoming wire can connect to any outgoing wire, in a grid arrangement. This, naturally, takes $m \times n$ switches, and the number of cross-points, and thus switches, grows very quickly. Various theoretical developments [24] lowered rather unfortunate upper bound on the number of cross points in the crossbar, and these derivatives are now referred to as Clos networks or, in some configurations, a Beneš network. These same developments were later adapted to packet-switched networks. Eventually, these networks came to be referred to as fat-tree networks, as the topology can be thought of as a tree with several root nodes linking the same child nodes [50]. The term butterfly network also came to be used as a generic catch-all for this kind of network, due to the likeness of a single switch in the topology diagram. [51] discusses a fat-tree or butterfly topology, and concludes from their limited experiments that it is a feasible topology, which has trouble competing with mesh networks on smaller devices but scales better. Dragonfly is a particular implementation of a butterfly network for supercomputer use, using specialized router ASICs [43]. Omega networks [37] are also related to this family of topologies, and it can clearly be seen that these forms are all variations of the same concept, and is generally referred to as a butterfly network. .

Bandwidth is largely dependent on the topology, and where the bandwidth is measured. For on-chip mesh networks, the node-to-node bandwidth has been the main concern. This is not a good proxy for system performance, as nodes are generally processors, and system performance is largely dependent on the processor to memory bandwidth (the memory wall). Even so, the interconnect bandwidth does have an impact on system performance. The topology affects this bandwidth, but so does the clock speed; both these affect the energy consumption. In addition, for on-chip networks, the area of interconnect (the number of transistors required) also affects the energy requirements and performance. This area is rarely touched upon in literature as it is complex tradeoff. The work that does exist in this area is limited to the networks themselves and not system behaviour or else are limited to analytical methods [11, 32].

2.1.2.2 AXI protocol

The AXI protocol [7] is a protocol intended for SOC devices, that is, single-silicon devices with integrated processors and I/O elements. The specification outlines the logical connection between a master and a slave device, and how they exchange data,

Channel	Purpose	Direction
AW	Write address and property signalling	master to slave
W	Write data	master to slave
B	Write completion signalling	slave to master
AR	Read address and property signalling	master to slave
R	Read data and completion signalling	slave to master

Table 2.1: Summary of the AXI channel properties.

including the arbitration signals. An AXI master, or slave, connection is divided in a number of channels, which are individually arbitrated. The properties of the AXI channels are described briefly in table 2.1. Each channel, furthermore, is unidirectional; the AW, AR and W channels go from an AXI master to an AXI slave, whereas the R and B channels go from an AXI slave to an AXI master.

Transfers on any given channel are therefore independent of transfers on any other channel, though it is the case that transfers have conditional dependencies on each other; a transfer on AR is followed by transfers on R, and similarly for AW, W, B. The AXI specification also have a provision for request numbering, in that each transfer has an associated ID number. The ID number links a number of transfers into a single request, so that data returned on the R channel in response to a AR transfer will have the same ID as the original AR transfer. In this way, the protocol can handle overlapping requests, by assigning each request a unique ID number.

The arbitration for each channel is identical, using a two-wire synchronous protocol. The signals used are referred to as READY and VALID. In each channel, the READY signal is the only signal that is driven in the reverse direction, as its purpose is to regulate the transfer of data. Data is considered to have been transferred over a channel only when both the READY and VALID signal are both high, and both a master and a slave can therefore force any channel to wait indefinitely.

The AXI protocol does not specify a topology or connection structure, but leaves the physical implementation of the actual links open for design, even if it is subtly indicated that a point-to-point multiplexed interconnect may be suitable. In this way it is not locked down to a topology that may be sub-optimal for a given application. The implemented interconnect, discussed in Section 3.2.2, consists of variable topologies and network structures designed to carry AXI channels and packets in accordance with the AXI specification.

The AXI protocol is intended for high-speed, on-chip connections, and especially for high-performance SOC systems. As such, it has support for multiple independent

transactions, prioritising transactions, and cache policy on requests that are not used in simpler systems. Due to these features, it takes more resources and wires than may be necessary for a simpler interconnect protocol, but it is expected that AXI is more flexible and adaptable. In addition, while the protocol supports these features, it is not a requirement that they be utilised.

2.1.2.3 Interconnect specialisation

Interconnect partitioning and optimisation has been attempted before by analytical methods coupled with limited design-space exploration [47, 52, 70]. These methods use high-level architectural or application information and attempt to produce a NOC that can support the application. Crucially, they depend on abstract application and NOC models that deliberately do not capture the full complexity in these systems. This allows the models and evaluations to be significantly faster than a full evaluation would be while still allowing the resultant design to be optimised for the application.

The abstract application models used in these studies are generally on the form of communication graphs, specifying communicating entities and bandwidths necessary. This model is good for dividing a larger task into small, communicating tasks and find the task requirements, but is not ideal as a model of side-effect prone application code. These graphs are lacking information about the actual behaviour of the application code; notably, operating system calls and system management events are not taken into account; neither code size, which may affect the cache behaviours and latencies in the final system.

Additionally, the NOC models evaluated are of a similar level of abstraction; generally, they are analytical models with abstracted properties. Thus, there is generally a disparity between the model properties and the properties of a NOC implemented in a given technology. The disparity between abstract NOC models and specific technology implementations may well be significant [40], especially in the face of V_{th} aware synthesis tools. These models have the effect of smoothing the design space, allowing for faster evaluation, but not capturing the full complexity.

These error factors are both effect of abstracting complexity to make the problem of optimising a SOC tractable in limited time. Due to the simplifying assumptions the design space for such a SOC becomes much simpler as it is not reflecting some of the tradeoffs present in the system. Thus these methods are suitable for guiding initial design efforts on a new system, when detailed software and/or hardware is missing, but are lacking when evaluating some of the more detailed aspects. There is a sliding scale of model accuracy versus evaluation time; these methods generally utilize lower-accuracy models to enable evaluation of the entire design space but by doing so risk

simplifying the design space so that the results are not representative of the eventual implemented system.

Analytical partitioning of the interconnect based on knowledge of the application is possible in limited application and design sizes; but quickly reaches a feasibility limit as the applications and systems grow in size. Larger applications, which may not be completed, running on multi-core systems are too complex to clearly analyse with this method without investing a large amount of NRE. In addition, due to the nature of the method, it requires knowledgeable and experienced engineers to perform the optimizations.

2.1.2.4 Interconnect generation

Methods to find promising NOC design with a view to automatically generate the NOC has also been investigated [13, 49]. The emphasis for these methods is not the design space exploration, but the mapping of an application to a interconnect structure. A limited design-space exploration may implicitly be part of these methods, but more likely it is using analytical methods to find a solution for a given application [23].

Similarly to the interconnection specialisation methods discussed, these synthesis efforts use software models and average bandwidth demands, leading to inaccuracies. As these methods use better NOC structure models, it would be expected that they generate more reliable results, but the detailed aspects of the software and hardware is still not taken into account. In addition, these methods are likely taking longer than high-level design space exploration techniques.

These methods themselves are also used in manual design-space exploration exercises, where a design space is instantiated using such automated analytical methods. The result these exercises could potentially be subjected to machine-learning techniques to further refine and improve the performance in finding good architectures.

2.1.2.5 Network on chip and interconnect summary

This section has discussed aspects of interconnect and NOC research that are important for the results in this thesis. In particular, some of the tradeoffs in NOC design have been presented, such as bandwidth and topology versus clock speed, area and energy. This section also introduced the background for the AXI interconnect, which will be used extensively in the experimental sections.

2.1.3 Processor and applications

Processors are the workhorses of embedded systems. Depending on the system they either provide the oversight or the majority of the work to be performed, and are thus on the critical path of system performance. In embedded systems, the processor is generally designed in as another component alongside task specific accelerators and I/O devices. Processor performance and capabilities are chosen from a set of available processor IP instead of being designed from scratch.

2.1.3.1 Processors

For embedded systems, a large range of processor cores area available. These start at the very small, slow, such as 8-bit microcontrollers, and go all the range up to applications processors with all the features (if not quite the performance) of mainstream server processors. For the purpose of this thesis, the interest is on the mid to upper half of this range, as smaller microcontrollers or processors would not be used for computational reasons. In this range a number of mainstream processors are available; notably, almost all ARM processors (except the most capable) are in this range. In addition, the locally developed EnCore is solidly in this category, as are some Digital Signal Processors (DSPs).

The ARM range of processors come in two flavours; the M series of deeply embedded microcontrollers and the A series of application controllers [8]. The latter is used in many widely known devices, such as the iPhone, tablets, and others; the former are used as system components in a large number of application-specific digital systems.

For this thesis the EnCore microprocessor was used. This processor is a derivation from the ARC 700 series of processors [66] and is comparable in performance to a ARM A series core [67]. The EnCore processor occupies only 0.096 mm^2 and consumes as little 0.145 mW/MHz in a 90 nm process. The EnCore can be fitted with with instruction set extensions (ISEs), in addition to being highly configurable.

2.1.3.2 Processor core sizes

As a result of process variation, chip foundries will manufacture batches of chips with large variations in power, timing, and reliability [16, 69]. These variations are inherent in the manufacturing of deep-submicron circuits, and cannot be eliminated. Such variations effectively drive up the cost of manufactured SOC designs, as failed chip manufacture must be factored into the bottom line. As technology node progression continues, the process variation in manufacturing will become more pronounced [15]. Hence, process variation will potentially be the next important design constraint for

SOC systems. What is needed are scalable, high-level designs targeted specifically at mitigating the costs of process variations in deep-submicron technology nodes while delivering increased compute performance within a given power budget.

An equivalent area of silicon to an Itanium processor [59] could contain a few thousand complete and independent cores, leading to vastly higher levels of parallelism and overall execution rates. The problems faced by this design paradigm are those of power consumption, process variation and off-chip bandwidth, and novel solutions in the on-chip management and interconnect are necessary to address these issues. For example, the diagram in Figure 2.1 shows the number of working cores as result of induced transistor faults for two processor designs with different complexity (50k and 500k transistors per core). This graph is based on a probabilistic model spreading random faults over the chip area, and assuming that a fault is severe enough to make a core fail. For the larger design the number of working cores quickly drops as the number of statistical transistor faults increases, whereas the design based on smaller cores is more resilient against these faults. This model does not account for process variation, which would cause the working cores to also exhibit execution rate variations. It has been conclusively shown that the effect of process variation is also best mitigated through the use of small, independent and asynchronous, on-die structures [38]. In addition, these smaller cores dissipate less power due to decreased speculation and less complex in-order execution, and thus provide greater power savings, compared to a single, more complex, core.

2.1.3.3 Applications

The application in an embedded system is largely static; the computer system has a single task to perform, often as a chain in a larger system. The tasks may be things such as GPS coordinate calculation from radio signals, USB to serial translation, and many others. These applications are commonly specially written for the task, to run on specific processors. The system is therefore specialized for the task, and the task for the system. This is in contrast to general-purpose computers, that run a multitude of software, and which are generally cheaper than specially-built systems. Because of this, the software design can influence the hardware design in the system, something that is not possible in general-purpose computing.

In addition, a large part of the more compute intensive embedded systems are digital signal processing applications, running such algorithms as fast Fourier transforms and digital filters. These applications are generally run on specific processors engineered from the ground up specifically for the task.

Applications may also assume there exists certain hardware implementations of

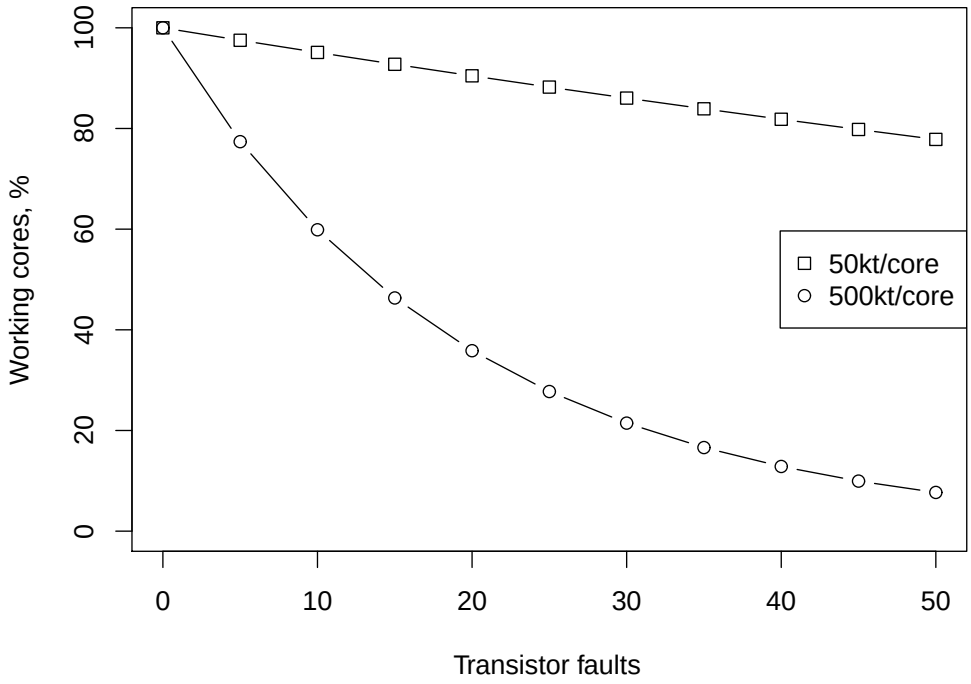


Figure 2.1: Analytical probabilistic model of number of cores without faults for cores of size 50k transistors and 500k transistors for a 10 million transistor die.

compute-intensive tasks, such as Viterbi decoding, that would be prohibitively expensive in software. The system as a whole is then specialized, as the application will not function appropriately without these accelerators.

Because of this, there is a tradeoff between software and hardware in terms of efficiency and power, leading to the area of hardware-software co-design [20]. This thesis does not attempt to address this tradeoff, but focuses on specialising the hardware system given an application.

2.1.3.4 Processor and application specialisation

The main way of specializing processors to applications is through instruction set extensions (ISEs). These are simply extra instructions inserted into the baseline instruction set that favour certain applications. By building a processor supporting these extensions, the processor perform better on these applications. An amount of work has gone into ISEs [12, 33], and, specifically, how to automatically generate ISEs from applications. ISEs may be used in the EnCore, and there is infrastructure to automatically generate ISEs from an application.

Encryption applications such as AES have historically seen the largest benefit from ISE use, as there are large contiguous areas of data-flow involving predominantly simple bit-wise operators. Many bit-wise operations can be performed in series within

a single clock cycle, and moreover those operations which fall in parallel can also be evaluated in parallel.

The specialisation of processors with ISEs is one way of specialising an embedded system. This is orthogonal to the interconnect specialisation presented in this thesis, in that both can be performed at once to specialise a system further. If this is desired, the ISE specialisation should be performed first, as it may impact the interconnect data flow.

2.1.3.5 Processor and applications summary

This thesis will use the EnCore processor core for its performance and area features, and also for the ease of FPGA synthesis. This processor is presented in more detail in the experimental setup, section 3.2.4.

The main concern of this thesis is specialising the system to the application chosen, not the processors. The processors can be specialised toward this goal, and there is a a body of work covering this aspect as seen in Section 2.1.3.4; consequently, and because this approach is orthogonal to specialising the interconnect, this work has not been duplicated. There is significantly less work on specialising the interconnect to the application.

2.1.4 Embedded systems summary

This section has covered some of the important aspects and features of embedded systems that are of importance for the rest of this thesis. There are other parts of embedded computing and electronics that are not represented in this section, as they are less relevant.

This section covered the required background reading in circuit design, implementation fabrics, and NOC implementation tradeoffs and details. These concepts will be used in the experimental setup to construct a design space that represent a realistic, synthesisable SOC system with a parameterisable NOC. In addition, this section also covered the higher-level tradeoffs in processors and applications, as well as processor extensions. These concepts are not widely used in this thesis, but are complementary to the efforts presented herein.

2.2 Machine learning

This section discusses the mathematical background of machine-learning techniques used in chapters 4, 5 and 6. This discussion is based on standard machine-learning textbooks [17, 73], and is summarized in subsections here for completeness.

The problem, and general usage area, of machine learning is to use past data to inform current choices. To do so, we use a generic algorithm which is tuned by the application of training data to arrive at the known correct decisions when fed the training data. If the training data is comprehensive, this effectively tunes the algorithm to perform the choices desired automatically.

The machine learning used in this thesis falls into two categories; classification and regression. Both methods use a set of input data consisting of *feature vectors* mapping to a *response* value. A feature vector generally consists of a set of measurements or values that define a data point, whereas the response value is the classification or value at that point. The difference between methods is that in classification the response value is one of a limited set of values, aka classes; whereas in regression the possible responses is generally in the form of a unbounded natural number. The number of responses for a classifier is commonly only two, and it is understood that more classes can be achieved by chaining such binary classifiers. Classification can further be thought of as binary regression, picking one of two values instead of producing a output figure.

Chapters 4 and 5 use classifiers, whereas chapter 6 uses both regression and classification. Some of the methods described in this section are 'natural' classifiers, whereas others are regression methods. As a classifier cannot be directly used for regression, there are some limit to the applicability of classification methods. In reverse, the mapping from regression to classification is an extra step that needs to be tuned by the past data.

A given algorithm is trained to map from a certain set of feature vectors to output values by minimizing the error between predictions and response values over the entire learning data set. Exactly how this is accomplished differs from method to method, and will be discussed in the following sections. Having established the learned algorithm, we then wish to know its accuracy in handling feature vectors which have not been used for training. The evaluation of the accuracy and predictive power of the algorithm is reported using such non-trained feature vectors.

As a final metric, the time taken to train and execute the algorithm has to be included in an analysis, as too long or too complicated algorithms may defeat the purpose of the machine learning; this may allow simpler empirical methods to prevail.

For implementations of these methods, R [57] and in particular the packages related machine-learning was used to a large extent, except for in chapter 4 which uses specially-written code.

Section 2.2.1 covers using linear regression in a machine-learning way. Section 2.2.2 covers *wk*NN, followed by a description of SVM in section 2.2.3. The decision tree

method is discussed in section 2.2.4, and its extension, Random Forest, in section 2.2.5. Gaussian Processes are summarised in section 2.2.6. Finally, section 2.2.7 summarizes the machine learning methods discussed.

2.2.1 Linear regression

Linear regression is commonly used for estimating curves from data to show a trend or otherwise derive a general relationship between two sets of values.

Given $q \in \mathbb{N}$ and vector values $\mathbf{P}_q = (p_{q0}, p_{q1}, \ldots, p_{qn})$, each mapping to a value through a function $f(\mathbf{P}_q) = r_q$. We seek a vector $\mathbf{X} = (x_0, x_1, \ldots x_n)$ such that the difference between r_q and $e_q = \mathbf{P}_q\mathbf{X} = p_{q0}x_0 + p_{q1}x_1 + \ldots + p_{qn}x_n$ is as small as possible for all q. The vector $\mathbf{X}$ then contains the weights for approximating the function $f()$. It follows that we require at least n linearly independent vectors to guarantee a unique solution for $\mathbf{X} = (x_0, x_1, \ldots, x_n)$. This is equivalent to finding the n-dimensional surface that passes as close to all the points in the training set as possible. $\mathbf{P}_q$ are our feature vectors, with r_q the response values. The requirements of having at least n linearly independent feature vectors is not generally a problem as machine learning tends to be used on large sets of data, implicitly satisfying this criterion. To use this regression method for machine learning, we observe that we can use the method to encapsulate a numerical feature vector mapping to a response value. We then derive the linear function minimizing the error for the training data, and use that function to classify or predict for new data. Specialisation of this machine learning method is then equivalent to finding a good fit to the given learning data.

In rare cases, with no noise in the system and measurements, it may be possible to obtain $\mathbf{X}$ uniquely and specifically, but in most machine-learning datasets an approximate solution is the best that can be found. To obtain such a solution one would write the vector equation system as a matrix, and use numerical approximation methods to minimise the error. This step is well studied and implemented in the baseline R library as the `lm()` method call.

In general, given more input vectors than dimensions, and with noisy feature vectors, linear regression tends to over-map and overspecialize the function. The result of this is that the learning step achieves large conformance with the input feature vector set, but when tried on other, non-learning data, the results are less than optimal.

2.2.2 Nearest neighbour

The paper [36] describes the different varieties of *wk*NN we use in detail.

The weighted-*k*-Nearest-Neighbour (*wk*NN) method is based on the observation that in linear systems (eg, without discontinuities) the response values of similar, or

close, feature vectors are similar. This can then be used to derive the response value for a new feature vector by interpolating (averaging) between known (learning) data points. This method should cope well with relatively smooth vector fields, but have problems in the presence of discontinuities.

We only use the k closest neighbours, as we otherwise would always get the average r of the entire learning set. To cope better with discontinuities, we also scale the impact of each neighbour with the distance between the new point and the learning points, effectively weighting each point.

The 'distance' metric is one of the parameters to this method. In general, we use Euclidean distance ($d = \sqrt{(a_0 - b_0)^2 + (a_1 - b_1)^2 + \ldots + (a_n - b_n)^2}$), but this can be generalized using Minkowski distance ($d = (\sum_{l=0}^{n} |a_l - b_l|^p)^{1/p}$). Using the Minkowski distance allows us an additional parameter, p, that can be used to specialize the machine learning function. Using $p = 2$ renders the Euclidean distance, $p = 1$ is equivalent to Manhattan distance and $p = \infty$ is generally referred to as Chebyshev distance, also written as $d = max_l(|a_l - b_l|)$.

The 'weighting' method is another parameter that can be tuned to the dataset. This parameter determines the form on which the distance to the n closest points is transformed into a weight for the value of that point; this can also be referred to as a kernel function. The *wk*NN implementation used recognises the weighting methods 'rectangular', 'triangular', 'epanechnikov', 'biweight', 'triweight', 'cos', 'inv' and 'gaussian'.

It is thus clear how we would use this method; by simply storing the learning data set and, when presented with a new feature vector, compute the closest points using the distance metric, select the k closest, retrieve their response values, and average them using weights derived from the distances computed. We thus obtain a value that should be close to the value, assuming the learning set has good coverage of the search space and there are no unrepresented discontinuities.

There is no good method to find the k nearest neighbours to an arbitrary point other than by computing all the distances involved; some divide and conquer approaches may be attempted, but they also tend to be of limited use in high dimensions. For obvious reasons, computing the distances also become significantly more expensive with large data sets; each dimension adds an extra multiplications equal to the number of feature vectors, and each extra feature vectors adds multiplications equal to the number of dimensions. Nearest neighbour is a conceptually simple method, but suffers from computational problems in high dimensions or with large data sets.

The method described in detail in chapter 4 is a form of Euclidean distance *wk*NN, with specifically named parameters.

2.2.3 Support vector machines

Support vector machines (SVM) [25] are based on the same n-dimensional model of the feature vectors as the nearest neighbour algorithm. It differs in that its an eager algorithm, transforming the learning data into a condensed version for prediction. For classification, support vector machines try to generate a hyperplane that cleanly separates the output classes; this is usually infeasible in n-dimensional space, however, and therefore the data is transformed into a higher-dimension form. In the larger-dimension space, it is then hoped, separating the outcomes may be easier as the opportunity for linearly separating the components increase.

Specifically, the method determines a maximum-margin hyperplane by deriving a set of support vectors from the training data; these vectors combine to form the normal of the hyperplane in such a way that the hyperplane is placed with a maximum margin to the learning data. As such, the method extracts linearly independent features from the training data. In addition, and to be able to deal with non-linear feature spaces, a kernel function that transform the original feature vectors to a higher-dimensional form is used; by selecting the kernel function appropriately SVM can be adapted to the data set.

The parameters for a practical use of SVM comprises the chosen kernel method, any parameters to the kernel method, and a variable C that effectively trades off accuracy for noise immunity. The SVM implementation used in this thesis (from the `e1071` package for R, which relies on LIBSVM [22]) have methods for linear, polynomial, radial and sigmoid kernels. The linear kernel takes no extra parameters, while the other kernels takes a parameter γ, used to tailor the function to the data set. In addition, this implementation operates in both regression and classification mode. Finding the best set of variables for a given data set is also automated with exhaustive grid-walking, given a range of C, γ parameter inputs.

2.2.4 Decision trees

Decision trees are another instance of an eager algorithm, in that it transforms the learning data internally to a more compact and useful format. It does this through deriving a decision tree that can be used to represent all the cases in the training data; effectively, generating a set of yes/no statements about the attributes that lead to a classification. When performing predictions, the method tests the prediction feature vector according to the binary choices in this tree to arrive at the classification.

Building the decision tree from the source data is then the main task of the algorithm. In general, this is accomplished by creating a root node associated with one

input variable, and splitting the input data into two sets depending on the value of the variable. Each of those sets are then split on a different variable, and so on, recursively, until all variables have been used. The leaf nodes then indicate the classifications. Using every variable in only one split per branch guarantees that the tree is finite, and that the algorithm terminates.

Deriving this initial decision tree is performed differently, depending on whether the initial variables are continuous or discrete; for the former, we need to find a value for splitting, for the latter, we need to find which cases are significantly different. As the measurements in this thesis are mostly concerned with regression and not classification, and the input variables tend to be continuous, there is more use for the continuous case. In general, the problem of choosing a variable on which to split is based on an entropy measurement, so that each split maximises the entropy of each chosen subset, whereas determining the value to split on is done through exhaustive search. If the possible entropy gains are outweighed by not splitting a node, however, the algorithm would keep the node as-is and not split it.

Decision trees are prone to overfitting, and to address this problem, they may be pruned after having been built. Pruning the tree is based on comparing the error rates at each node in the tree and discarding nodes in which the error at that node is lower than the sum error of nodes beneath it. As such, one would need a pruning data set or other method of generating the prediction errors at each node, which may not always be practical.

The R library `rpart` was used as the reference implementation for decision trees used for regression, and this library is in turn based on [18].

2.2.5 Random forest

The Random Forest implementation in R is based on [19] and as such encapsulates the ideas presented therein.

Random Forest is a term for a collection of decision trees, each generated from a randomly generated subset of the training data. The method generates a large number of trees by uniformly selecting, with replacement, a subset of feature vectors from the input data set, and building decision trees without pruning from these subsets. Each such generated decision tree are then given a vote (in case of classification) or generates a continuous variable which is averaged (in case of regression).

There are some external variables to this method, namely how many trees to build, how many features to select between at each node split and how deeply to grow the trees. The first two of these can be set at run time, and in general should be somewhat correlated with on how many feature vectors is in the training set and how many

features there are. The default values for these in the R implementation is to grow 500 trees, to use $\frac{1}{3}$ of the features per split, and to stop splitting nodes when 5 features are left.

The Random Forest method do not suffer from overfitting in the same way as plain decision trees, due to the law of large numbers, indeed, it is quite resilient to noise in the data. As such, no extra steps are necessary to correct for overfitting. In addition, it is possible to extract approximate importance scores for the features from the method, giving an indication of the relative importance of certain features.

The paper [21] discusses the performance of machine-learning methods in high dimensions, and suggests that Random Forest is preferable for these applications.

2.2.6 Gaussian process

Gaussian Processes are a complex methodology. An in-depth description of the method is available in textbooks [58].

Gaussian Processes are related to SVM as both are kernel methods, and take a kernel parameter. They both use the kernel trick to avoid explicitly transforming the parameter space into a high-dimensional space by reformulating the operations necessary in high-dimensional space as vector (dot-product) operations in the origination space. Instead of deriving the support vectors for a hyperplane in high-dimensional space, it models the distribution as a set of gaussian functions. This inherently provides some smoothing and noise-insensitivity, as well as implicitly performing feature weighting and selection. As with SVM, some parameters can be given to match the algorithm to the data.

Gaussian Processes take, in general, a kernel function as its main specialisation parameter as well as an initial noise variance. The kernel function, in turn, may take some parameters (hyper-parameters), depending on the exact type of kernel. Commonly these parameters are selected based on the type of data, but this as this thesis deals with a large body of data and evaluations, the process is automated to select the best parameters from a range instead.

The complexity is wrapped into pre-written third party packages for R. The `kernlab` package was used in this work as it contains Gaussian process regression methods conforming to the generic R regression interface. The chosen implementation package contains pre-existing functions for a number of kernels, which can easily be substituted in.

In practice, only a few of the kernels available were used as it was found that most of the available kernel methods produced significantly worse results than the default. However, the methods `rbfdot`, `laplacedot` and `besseldot` were found to have a low

cross-validation error and were thus used. Similarly the initial variance was varied from 10^{-3} to 10^{2} in order of magnitude increments. For training, the predictor with the lowest cross-validation error from this range of parameters was chosen, achieving specialisation of the predictor to the dataset in addition to the inherent feature selection in Gaussian Processes.

2.2.7 Machine learning summary

This section presented a selection of machine learning methods and algorithms, so that the reader may have an idea of the complexities and intricacies of the methods. Each of these methods have been a subject of research in their own right, as parts of a drive to find machine learning methods that perform well. As such there are large bodies of literature on variations and evaluations of these methods, but as this thesis intend to use the methods as black boxes for prediction, no further references are made. As stated in the section introduction, this thesis will generally use pre-written and -optimised code from the R statistics software as the practical implementations. In many cases, this package reference or use the original implementations of these machine learning methods, and as such, the quality and accuracy of the code is not an issue.

2.3 Background summary

This chapter has presented and discussed the diverse background for this thesis.

The background primarily focused on the manufacture, design, and implementation of embedded systems, as these are the main topics of this thesis. This included the basics of modern VLSI design languages and implementation methods, introducing Verilog as the implementation language for the experimental system. A brief background about metastability and asynchronicity is warranted, as some of the design space variability in the experiments will come from exploiting the clock domain variability. This continued with an overview of FPGA versus ASIC implementation methods, highlighting the differences and similarities as well as discussing the specific uses of FPGA devices in this thesis as well as how ASIC fabric variability is expected to influence system architecture.

The next section discussed NOC and interconnect topics. Crucially, a summary of the AXI interconnect protocol was covered here, as it will be used extensively in the experiments. The discussion of interconnect topology will form part of the design space in the setup, and thus the background to the topology is shown here.

The processors the experimental system is going to use are also presented, together

with a discussion on processor sizes, linked back to the ASIC fabric variability previously discussed. The applications generally executed in a embedded system are also touched upon.

Finally, the machine learning methods that will be used to evaluate the design space have been presented. These are in general off-the-shelf code, and thus is the only place they are discussed in any detail.

3 Experimental setup

This chapter covers the experimental setup required for performing the experiments covered in chapters 4, 5 and 6.

The experimental setup is complex and detailed, as any real SOC is. The complexity is necessary due to the aims of the thesis; to show the successful use of machine learning in the SOC design space. A simplified or simulatable model of a SOC would not suffice for this goal, as such a simplification would by necessity omit low-level details, and it is these low level details that can give rise to systematic effects on power and throughput. Small details in the implementation, such as the buffer depths in interconnect switches, can have a large effect on overall performance. The exact behaviour of the interconnect when moderating access to shared memory resources has an impact on the interconnect latency, and thus on the system throughput. The exact system performance depends on a multitude of such small details, which aggregate to systemic effects. As such, to generate accurate data for the machine learning method, and by extension, show accurate machine-learning results, it is necessary to build a detailed, synthesisable, model of a SOC that execute complete programs, with all the issues this entails. Thus, this chapter presents the parts and instances of a multi-core SOC necessary to show the derivation of the design space parameters used in this thesis.

Implementation to a FPGA fabric allows for faster simulation results than a software solution would [3]. The overhead to pay is in the form of longer synthesis and preparation time; in addition the more complete system has a larger possibility of bugs and other untoward effects, simply because it is a fully instantiated system with no simplifications. However, the synthesis time and bugs would also be present in a real SOC, and are an unavoidable consequence of the complexity and implementation methods chosen.

The rest of this chapter is arranged as follows. Section 3.1 presents an overview of the various experimental systems, their differences and commonalities. Section 3.2 discusses the detailed SOC implementation details, down to gate-level in some cases. Section 3.3 will detail the hardware counters and other methods employed to extract accurate data from the test setup. Section 3.4 will briefly summarise the synthesis chain and options used for the FPGA targets. Section 3.6 will detail how the extracted data was used in a machine learning setup to arrive at the experimental

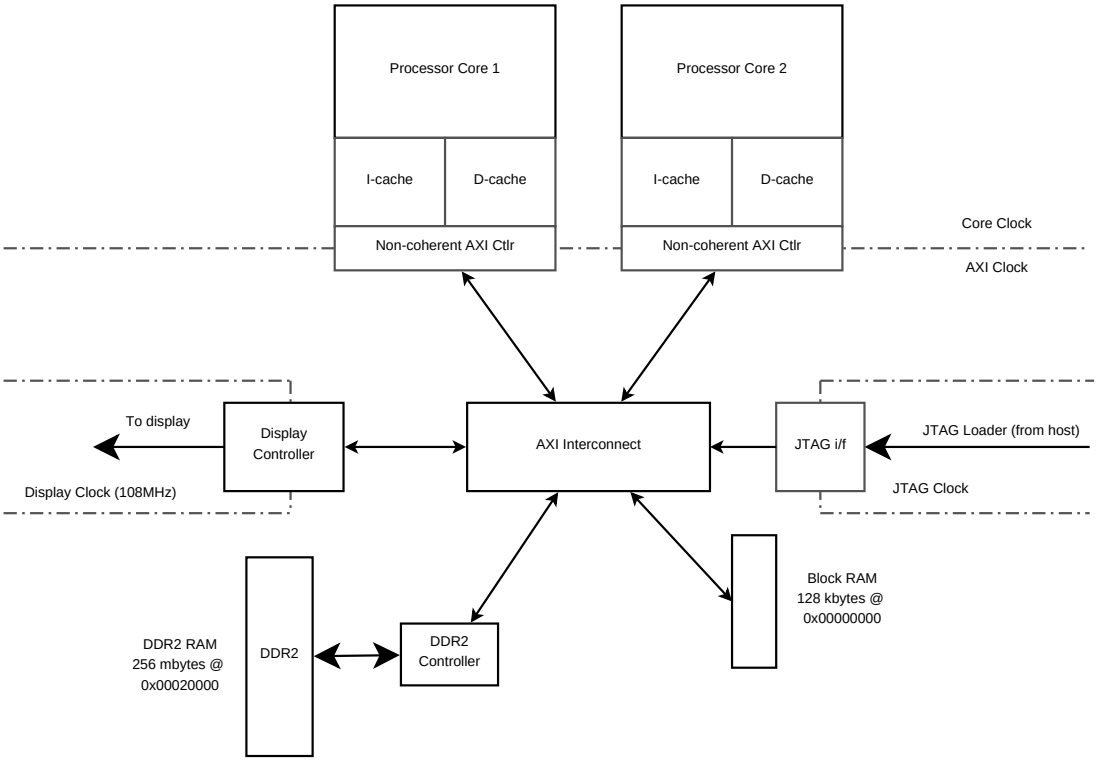


Figure 3.1: Schematic overview of the ML507 system. Memory latency depends on clock relationships and AXI channel width, which are parameters of the design space.

results. Finally, section 3.7 summarises and concludes this chapter.

3.1 Overview

The experimental systems created came in several main types, depending on the components used and the FPGA board targeted. This section gives a brief overview of these flavours, and also discusses who created which parts of these systems.

The main physical parts of the experimental SOC is the EnCore processor core, the interconnect, memory devices, and I/O devices. These devices were all implemented in synthesisable Verilog. Together they form the complete hardware of the SOC, enabling software to run.

Figures 3.1 and 3.2 shows a schematic overview of the two SOC systems implemented. The memory hierarchy in these systems are relatively flat, with only the core-attached L1 caches between he cores and the RAM. In addition, all caches are incoherent with regards to each other, and there is no hardware implemented to resolve cache coherency issues. The RAM access times in these systems are dependant on the clock relationships, topology depth and AXI widths selected.

The software is divided into libmetal and applications, separating common library

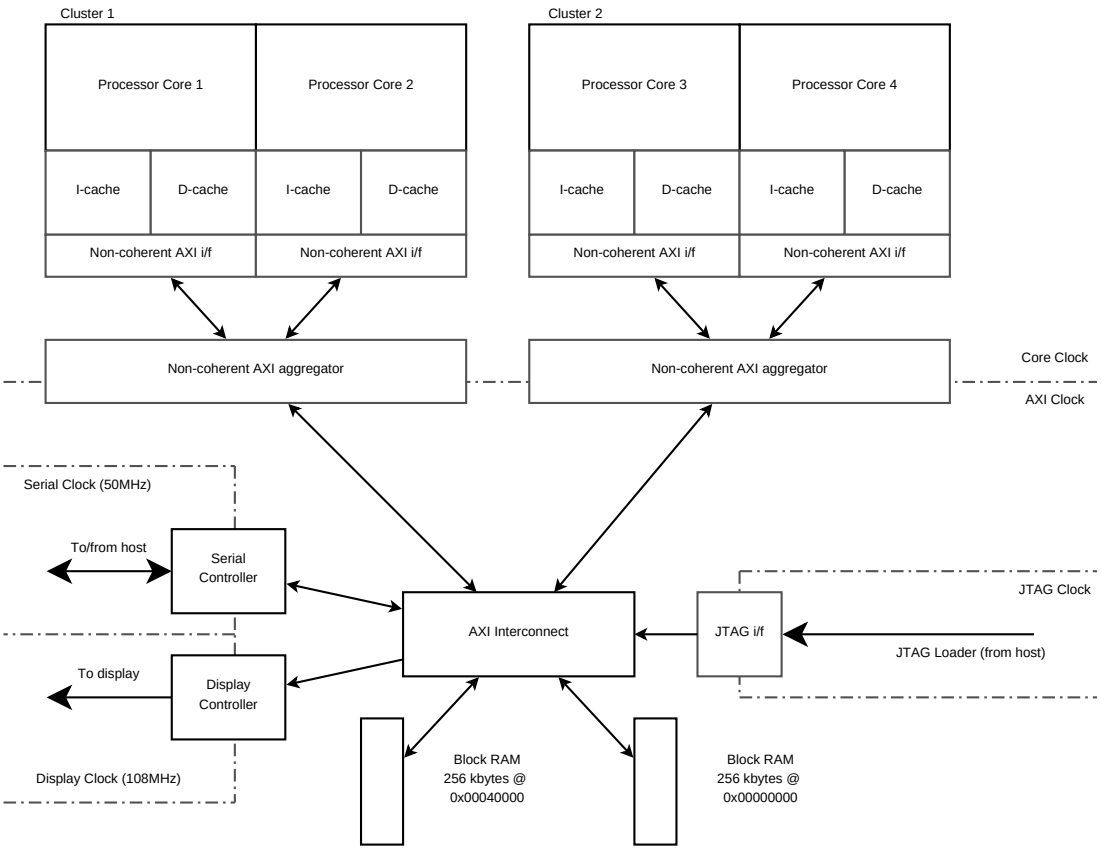


Figure 3.2: Schematic overview of the ML605 system. Memory latency depends on clock relationships, a parameter of the design space. AXI width is not a parameter in this system. This schematic shows two clusters with two cores each and two BRAM blocks; these are variable as parameters to the SOC design space.

code and application-specific elements. Further, synthesis software and FPGA devices are used to instantiate and test the SOC designs. Finally, some hardware counters and control software are used to extract the experimental data for use in a machine learning flow. These parts are all discussed in detail in the rest of the chapter, but this section presents how they fit together to form the system as a whole. A total of two different experimental system generators were assembled from these components, in addition to a test bed for the EnCore processor development.

The architecture of these experimental SOC systems is, compared to existing available SOC systems, differing in two main regards. Firstly, the number of I/O devices in the experimental systems is much lower than those in a commercial design, and secondly, the chosen designs are cache-incoherent. Both these deviations from commercial systems are justified, in that implementing either would be a large undertaking, and their presence would not impact the performance of non-communicating applications on the system. As such, the SOC designs used in this thesis are close to commercial systems in terms of device size, capabilities, and performance, and are therefore suitable for experiments in SOC and NOC design.

3.1.1 Attribution

FPGA devices and boards were all standard commercial products bought from Xilinx, inc, who also supplied the synthesis software. The EnCore processor was primarily built by Nigel Topham and Xinhao Qu, with myself providing physical test harnesses and bug reports. The interconnect system, memory devices and data capture systems were designed and built from the ground up by myself. Libmetal was similarly created by myself, with bug fixes, support and some device-specific code provided by Xinhao Qu, Christopher Thompson, and Gordon Parke. The applications executing on top of libmetal were either standard benchmarks or written entirely by myself.

3.1.2 ML605 system setup

The ML605 is a standard design test board for the Virtex-6 family of FPGA devices, and comes with a wide variety of I/O devices to support many possible uses. Figure 3.3 shows the experimental setup using the ML605 board, display device and I/O cabling. Of note in Figure 3.3 is the display device, here connected via A-DVI to a 24" wide screen monitor. The display output looks somewhat stretched due to the wide screen nature of the monitor; the output DVI signal is not wide screen resolution. The soft JTAG port (the loose, multicoloured leads) is used to start, stop and write to the system memory. Figure 3.4 shows a schematic overview of the ML605 SOC as a whole, including processor cores, interconnect, BRAM blocks, and I/O devices.

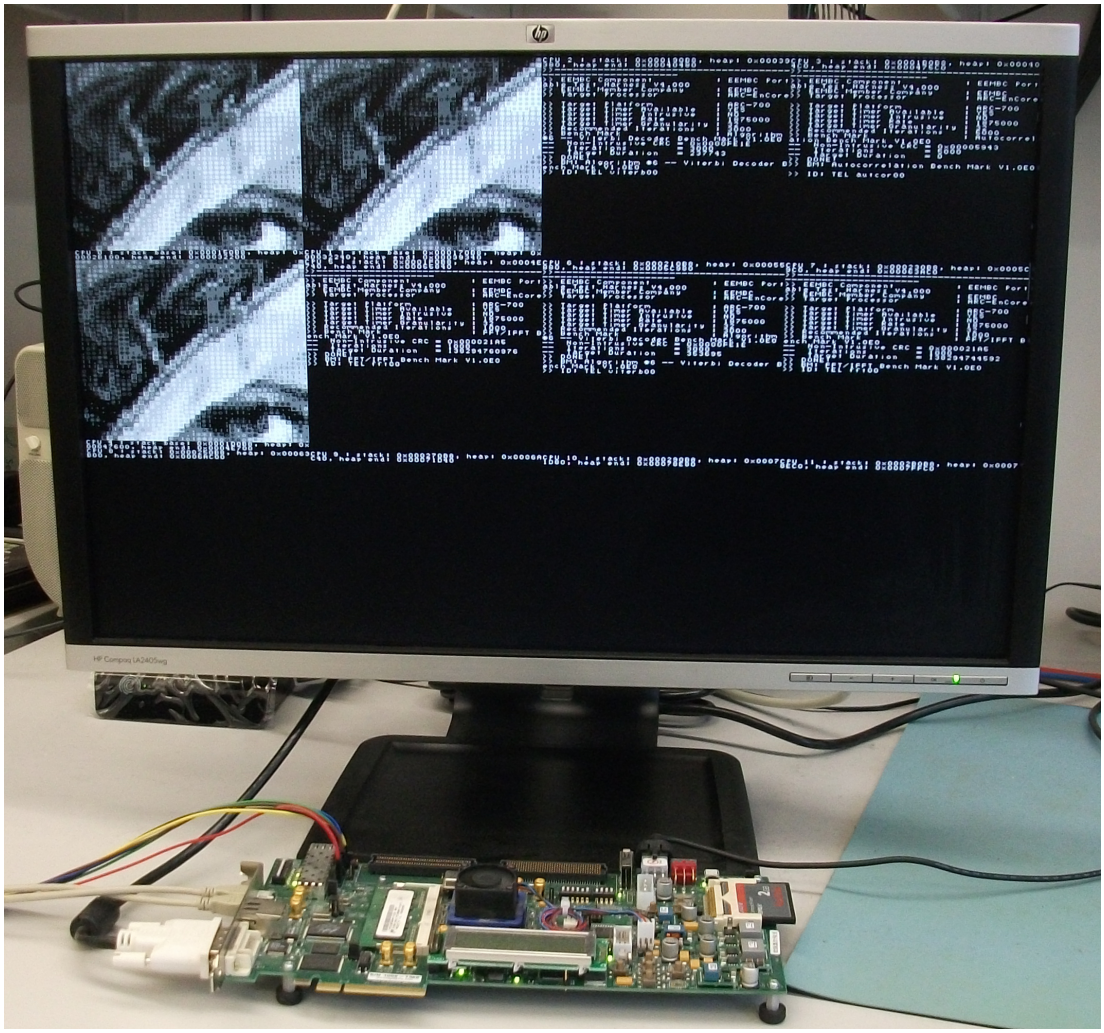


Figure 3.3: Experimental system setup: FPGA board and display device. Display shows a 12-core design having executed an 8-benchmark workload.

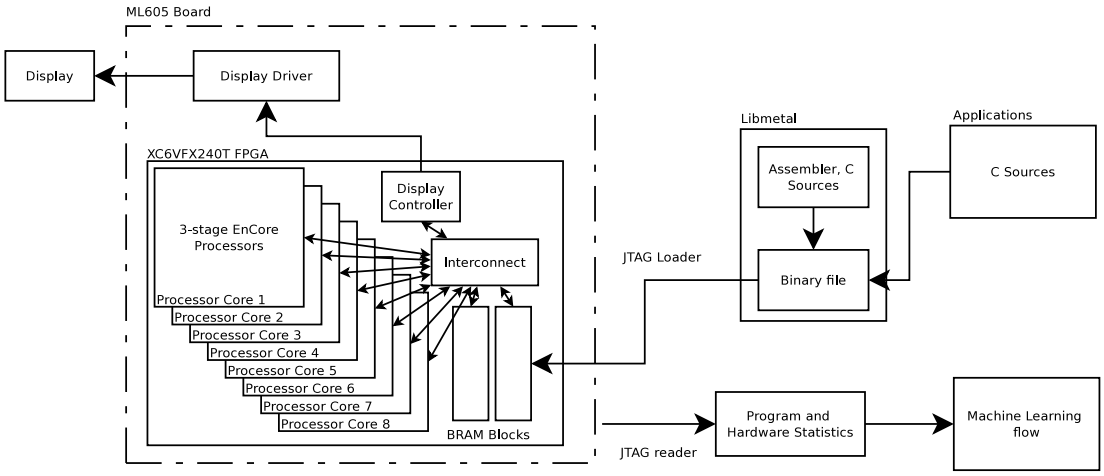


Figure 3.4: Virtex-6 ML605 system with major components indicated. System shows an 8-core system for illustration purposes.

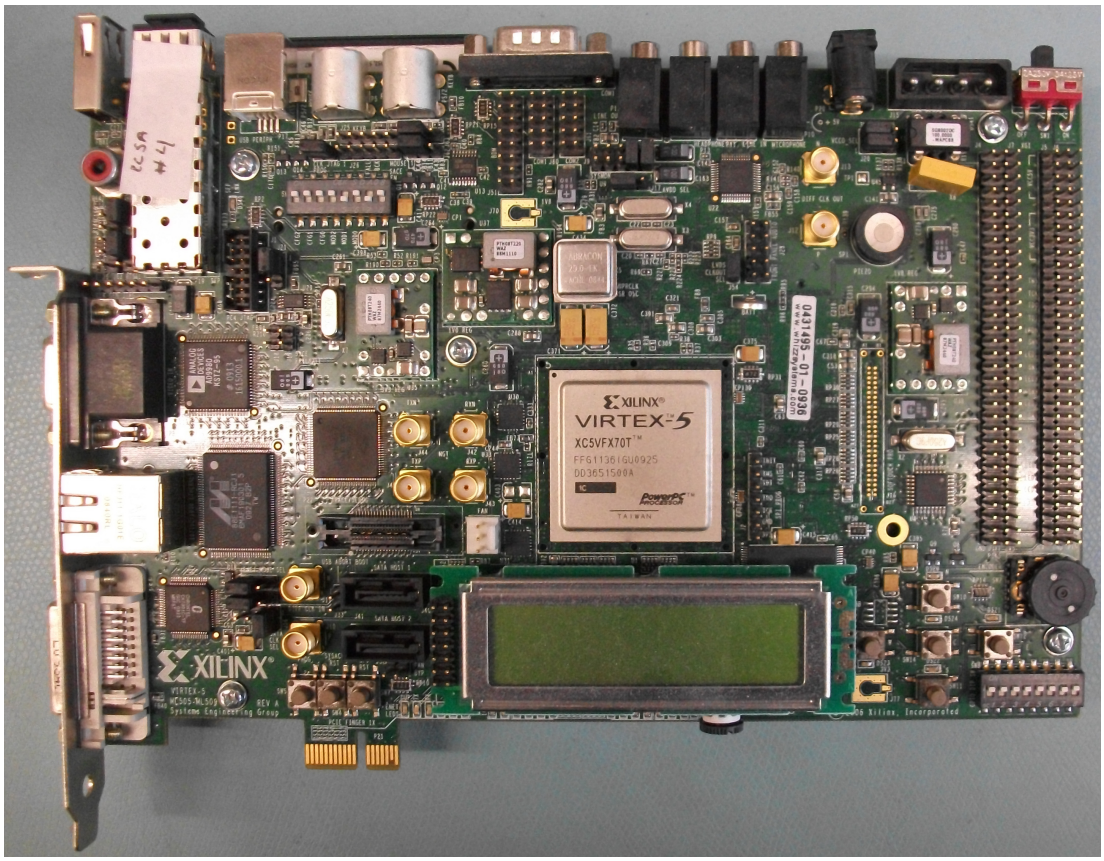


Figure 3.5: Bare Virtex-5 ML507 board, used for experiments. Shown here not connected to the experimental setup. Most of the connectivity from this board is unused in experiments.

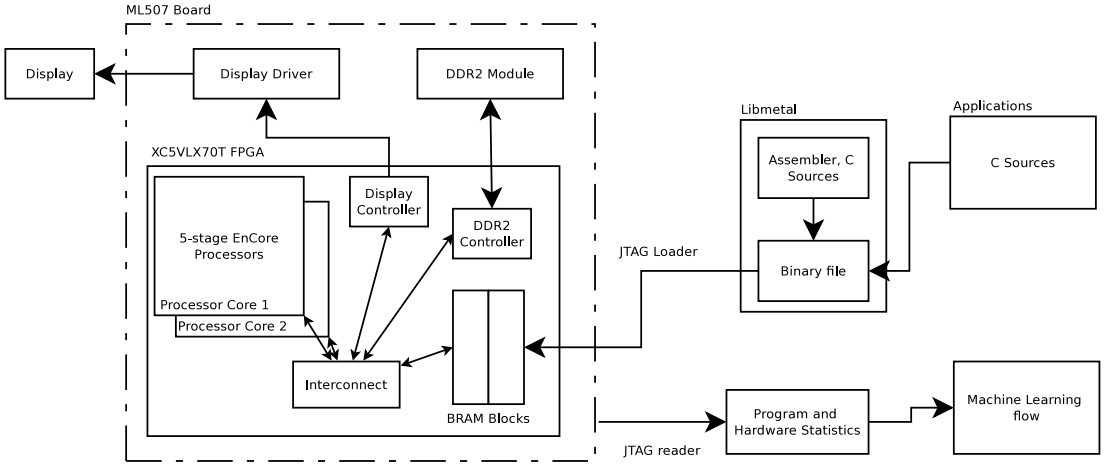


Figure 3.6: Virtex-5 ML507 system with major components indicated.

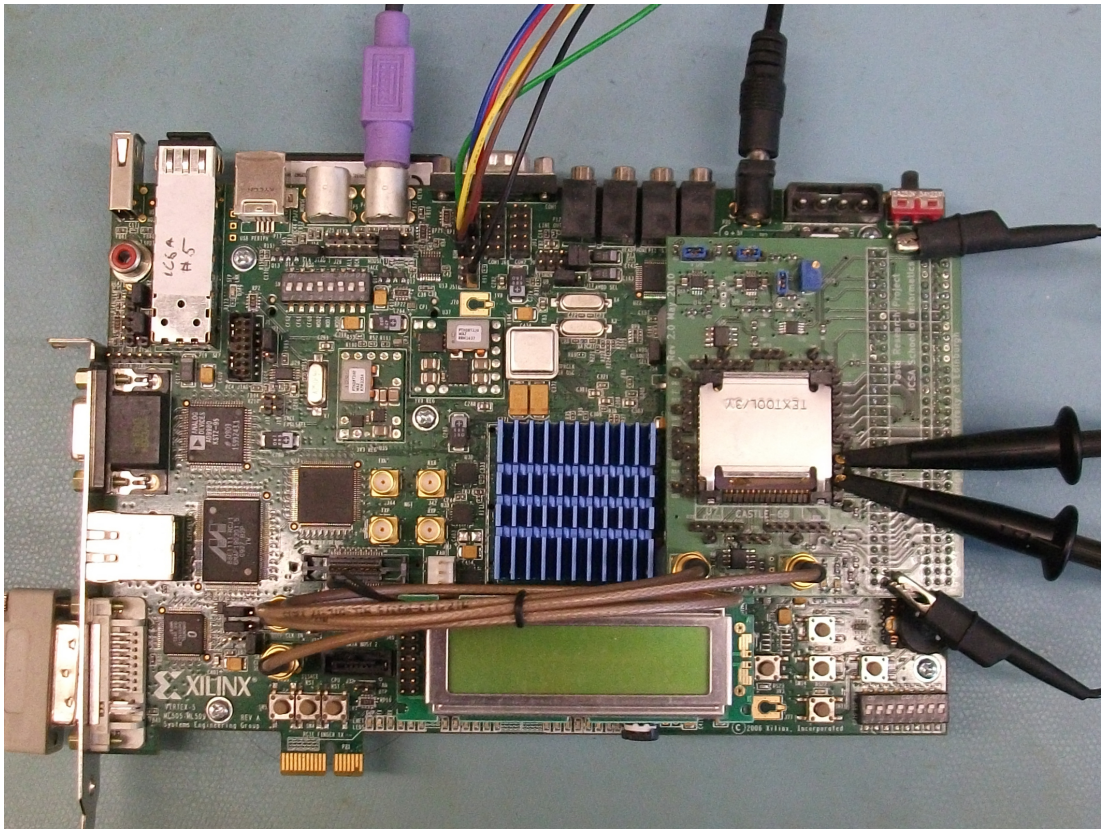


Figure 3.7: Virtex-5 ML507 board with added mezzanine board holding a 90nm EnCore.

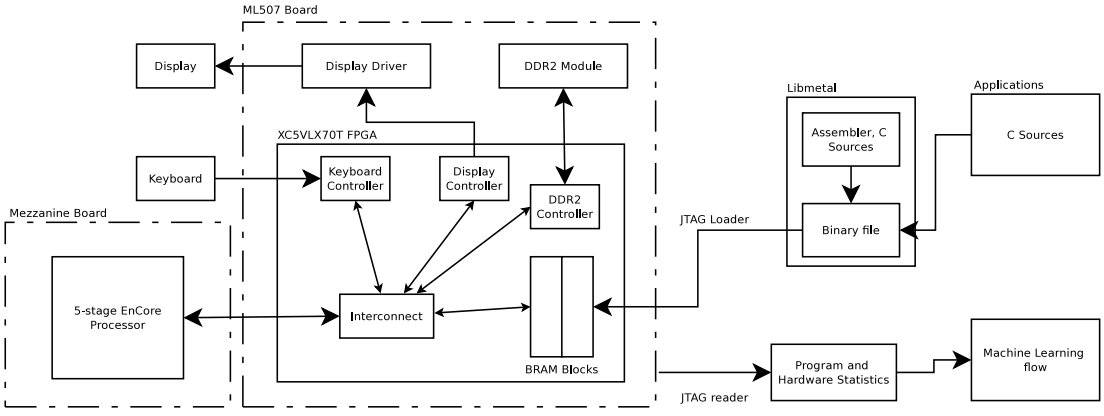


Figure 3.8: Virtex-5 ML507 system with mezzanine board and major components indicated.

The ML605 SOC implementation comprised of up to 12 3-stage EnCore processing cores, grouped in up to 4 clusters, connected to an asynchronous interconnect fabric using butterfly topology. Memory consisted of 512 kBytes of the on-chip BRAM, clocked at the interconnect clock, and broken in up to 8 logical banks. The I/O devices were a screen device and a serial port.

The memory and interconnect can be clocked up to 125 MHz, and the processors manage up to 75 MHz, as the core clock is distinct from the interconnect clock, any combination of frequencies is acceptable, and this forms an important part of the design space. Other parameters are the clustering of EnCore processors, amount of buffering in the interconnect, the topology of the interconnect, number of cores, and number of BRAM blocks. In total, the design space for this system is 510,000 distinct designs. This system, and design space parameters, is used in chapters 5 and 6, and is the main focus of this thesis.

3.1.3 ML507 system setup

Figure 3.5 shows a bare ML507 board that was used for early experiments. The ML507 board contains one XC5VLX70T FPGA device, which, it was found, was capable of supporting up to two five-stage EnCore processors. The parts of this system and their relationships are shown in figure 3.6. This system was used for the experiments in chapter 4.

This system contains up to two EnCore processors but has a DDR2 controller in addition to an early version of the BRAM controllers which do not support the locking mechanisms in AXI. The AXI interconnect logic, JTAG programming port and display driver this system is shared with the ML605 version. The interconnect in this also allowed for varying the width of the data paths, an ability that was removed to enable a better BRAM controller in the ML605. Another major difference between this system and the ML605 is that this version uses the 5-stage EnCore pipeline.

The design space parameters in this system are the core and interconnect clocks, the width and buffering of the interconnect channels, interconnect switch behaviour, and topology. Overall, due to larger cores and a smaller FPGA, the number of designs for this system is much smaller than for the ML605 system, with only 280 designs in all. These are exhaustively enumerated and tested in chapter 4.

3.1.4 EnCore silicon test platform

Figure 3.7 shows the ML507 fitted with an add-on mezzanine card for hosting an ASIC version of the EnCore processor for validation purposes. The taped out version of the EnCore processor was found to be functioning at up to 600 MHz, with the two-way

memory bus connection to the FPGA running at up to 37 MHz (1184 Mbit/s). The components of this system are largely shared with the previous, and thus the taped-out core had access to the DDR2 RAM as well as the I/O devices. Among other things, this setup was used for experiments with real-time operating systems and speed tests of the EnCore processor. This design was not used for any experiments presented in this thesis, but is included as a development platform for the EnCore processor, to show that this core is physically realizable and thus a useful part of a complex, realistic SOC.

3.1.5 Overview summary

This section has presented the major parts of the system and how they fit together. Further information about clock boundary transitions and low level circuitry can be found in section 3.2.1. Details on the AXI interconnect implementation and switch fabric is in section 3.2.2, and the topology variable is discussed in 3.2.3. The EnCore processors are discussed in section 3.2.4, and RAM interfaces in section 3.2.5; in addition details about BRAM interfaces and some I/O devices can be found in Appendix A. The JTAG circuitry, loading and extracting of data is covered in section 3.3. The synthesis flow for the SOC is covered in section 3.4, which finishes the hardware side of the components. For the software parts of the system, libmetal and applications as well as the software compilation tool-flow is discussed in section 3.5. Finally, the machine learning flow used is presented in section 3.6.

3.2 Experimental system details

This section will detail the low-level structural and logical blocks of the experimental SOC. The system was implemented in synthesisable Verilog, and evaluated using behavioural simulation and primarily FPGA implementation.

This section start with the very low level logic blocks involved in forming the interconnect design space in section 3.2.1. Section 3.2.2 will then show how these blocks are integrated into the AXI interconnect fabric, and how the low-level circuit behaviour is extracted into design space variables. Section 3.2.3 discuss the topology variations chosen for the interconnect, and how the design space variables for this are derived. Section 3.2.4 then takes a step to the side and looks at the EnCore processors used, how they are validated, and how they connect to the AXI fabric. Sections 3.2.5 then deal with the memory devices also connected to the fabric. In addition, some I/O devices are covered in Appendix A.

3.2.1 Logic blocks

This section discusses some of the various logic blocks that were necessary for the test setup. The emphasis is on the circuit behaviour of the low-level clock-domain crossing circuitry. These logic blocks were necessary for the function of the system as a whole, and their design and implementation affect the system performance. These blocks are therefore discussed in reasonable detail.

3.2.1.1 Two-phase resynchronising circuit

The two-phase resynchronising circuit is based on the four-phase resynchroniser [34]. The changes implemented from the four-phase synchroniser do not affect the stability of the synchroniser, and the two-phase version is discussed as a valid alternative.

The logic for the synchroniser is shown in figure 3.9. This was implemented as two separate Verilog modules, one for each clock domain, for ease of embedding in other design modules. The data bus width, shown in figure 3.9 as a single flip-flop, was supplied as a parameter to the modules. Actual data bus width is of no concern to the operation of the circuit so the data width is ignored in the following description.

The synchroniser samples the input data on the clock edge when valid is asserted and there is no operation already in progress. This data is then held in the internal `Tx_data` as the valid signal crosses the clock boundary; this will take up to 3 `CLK_B` cycles. The data is then sampled in the `Rx_data` buffer and `Valid_out` asserted for a single `CLK_B` cycle, during which it is assumed that further logic in the `CLK_B` domain sample the data. A ready signal is also sent back over the clock boundary, which releases the Tx part to accept new data no later than 3 `CLK_A` cycles later.

This resynchronising circuit uses two flip-flops to guard against metastability when crossing the clock boundary. As discussed in 2.1.1.2, two flip-flops may not always be enough, depending on the manufacturing process, and a third flip-flop may be required. This will slow the resynchroniser down further, increasing the worst-case time to 4 `CLK_B` cycles + 4 `CLK_A` cycles.

It is possible to expand this circuit to fully responsive external signalling, by deriving ready signals to complement the already present valid signals. It can clearly be seen that the external `Ready` signal in the Tx module is already present in the circuit but is not brought out to the interface. The `Ready` signal to the Rx module would additionally require an extra and-gate, making the Rx logic a perfect mirror of the Tx logic, but this is also a trivial change.

In general, while this circuit is correct, the worst-case delay of 3 `CLK_A` + 3 `CLK_B` cycles means it is not suitable for high-bandwidth applications. It is cheaper in terms of

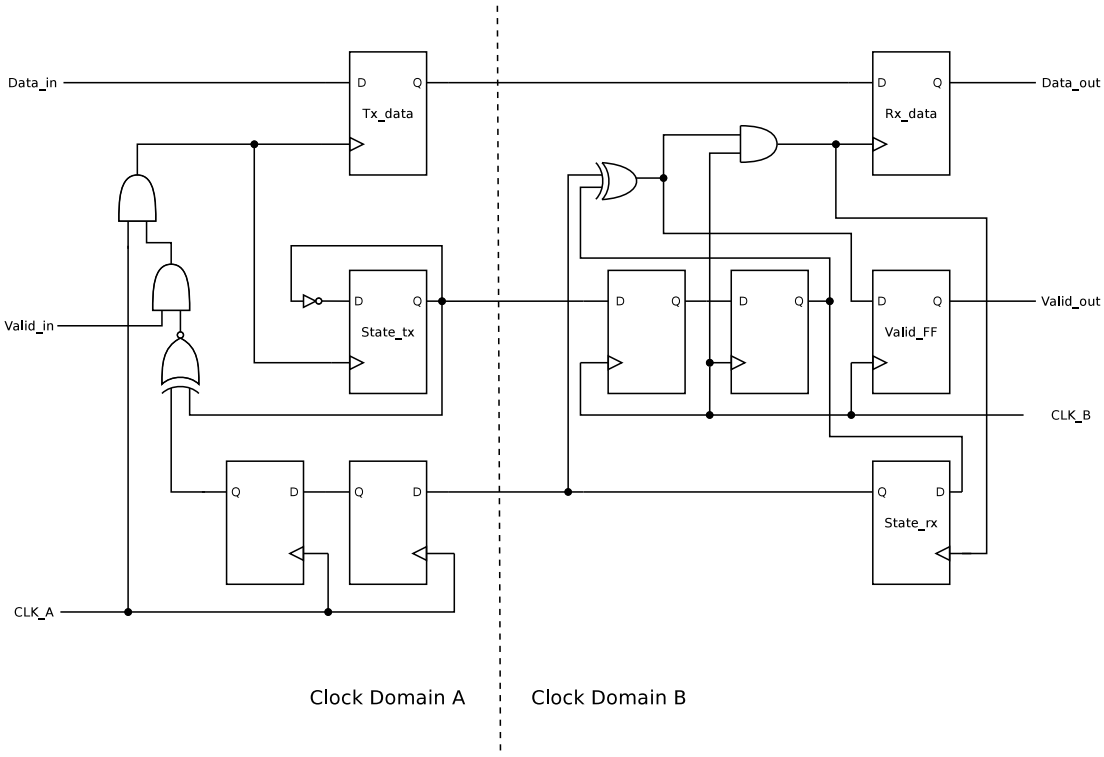


Figure 3.9: Two-phase resynchroniser. Uses valid signalling; there is also an option for fully responsive signalling using valid and ready. Based on the four-phase resynchroniser from [34].

area and complexity than the Asynchronous FIFO (Section 3.2.1.2) and is therefore suitable where complexity constraints are paramount to bandwidth, such as for auxiliary low-impact monitoring circuits.

3.2.1.2 Asynchronous FIFO

An asynchronous FIFO is the tool of choice when crossing a clock domain boundary and requiring higher bandwidth then can be delivered by the basic resynchroniser. The asynchronous FIFO is basically a two-port asynchronous RAM with some logic around it to reconcile the read and write signalling and to avoid collisions. Usually, the RAM is addressed with Gray-coded address generator, so as to minimise state transition problems. This type of device is described in [26] and its design is now well understood, but will be summarised here as the design of this element allow an increase in the system design parameters. The basic circuit diagram of a asynchronous FIFO is shown in figure 3.10.

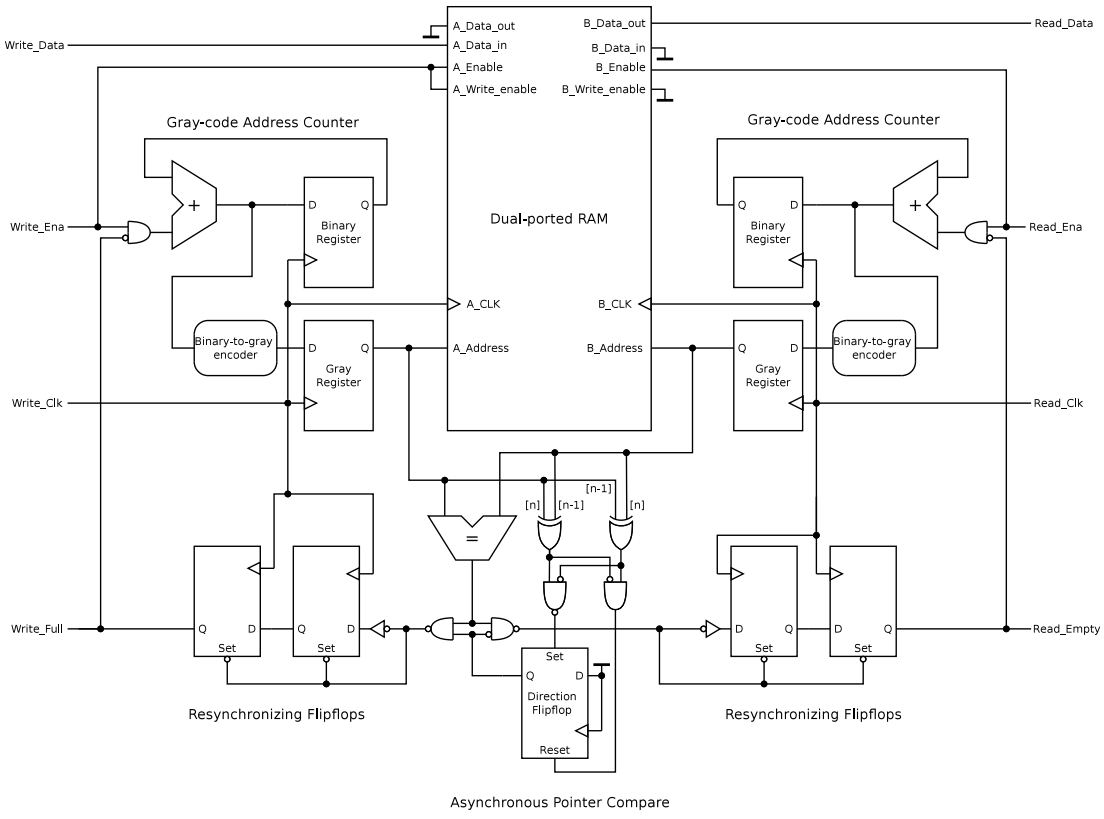


Figure 3.10: Asynchronous FIFO circuit diagram with Gray-code address generation and asynchronous pointer compare. Adapted from [26], according to [1] this is the approximate architecture used for the hard FIFO blocks in the Virtex-5 Family.

The Asynchronous FIFO shown in Figure 3.10 is centred around the true dual-

ported RAM block that can be written in one clock domain and read in another. As such, it can accept data at a higher rate than the previous resynchroniser, as it buffers the data with single-cycles delays on one side and read the data out at single cycle delays on the other. For obvious reasons, the total bandwidth across the resynchroniser is still limited by the slowest clock domain.

The addresses accessed in the dual-ported RAM is incremented for each write or read operation, and are usually referred to as the head (write) and tail (read) pointers. To enable this type of resynchroniser to work, a comparison of the head and tail pointers across the clock domain boundary, so that determination of when the FIFO is full (and thus cannot accept any more data) and when it is empty (and thus cannot output any more data). Both these operations require a comparison of the head pointer to the tail pointer, and additionally, as equal can mean both full and empty, some historical information about whether the FIFO were going full or going empty. Theoretically, their pointers can be exchanged using a resynchroniser similar to the two-phase version shown above, and do the comparisons in each clock domain separately. It was discovered [26] that the actual comparison - which is fully combinatorial - could be done asynchronously, and the results (full and empty) are the only signals which needs resynchronising. Figure 3.10 shows this type of control circuitry, and according to [1] this is the kind of control circuitry used in the built-in FIFO of the Virtex-5 FPGA family. This saves having to implement and validate this highly asynchronous design ourselves, and provides a level of guarantee that it will perform correctly as a resynchroniser.

This asynchronous FIFO is available as a macro block in the Virtex-5 and Virtex-6 family of devices, and these instantiations were used directly in the experimental SOC design. The data width of the AXI channels is such that occasionally two or three of these hard FIFO blocks in parallel were needed, and accordingly a wrapping Verilog module was developed that takes a width and a depth parameter. The depth sets the number of elements that can be stored in the RAM, and consequently sets the amount of slack available in case of widely varying clock rates or burst data. In general, the depth of the Asynchronous FIFO blocks were set as a parameter of the maximum amount of pending traffic across the FIFO.

The Asynchronous FIFO blocks were used to cross the EnCore - interconnect clock boundary. The use of this type of resynchroniser enabled these two clock domains to be independently controlled, letting us set different clock speeds in each domain. This led directly to a multiplication of the number of possible designs, while retaining the high-bandwidth nature of the CPU- memory interface.

3.2.1.3 Synchronous FIFO

This section will discuss the synchronous FIFO used for buffering in the AXI fabric.

In addition to the asynchronous FIFO necessary when crossing clock boundaries, the system also require a synchronous FIFO for generic buffering. The asynchronous FIFO could be re-used for this purpose, but as it is optimised for asynchronous operation and therefore sometimes has extra wait states, it would be preferable to use a synchronous FIFO. Figure 3.11 presents the circuit diagram of the chosen solution.

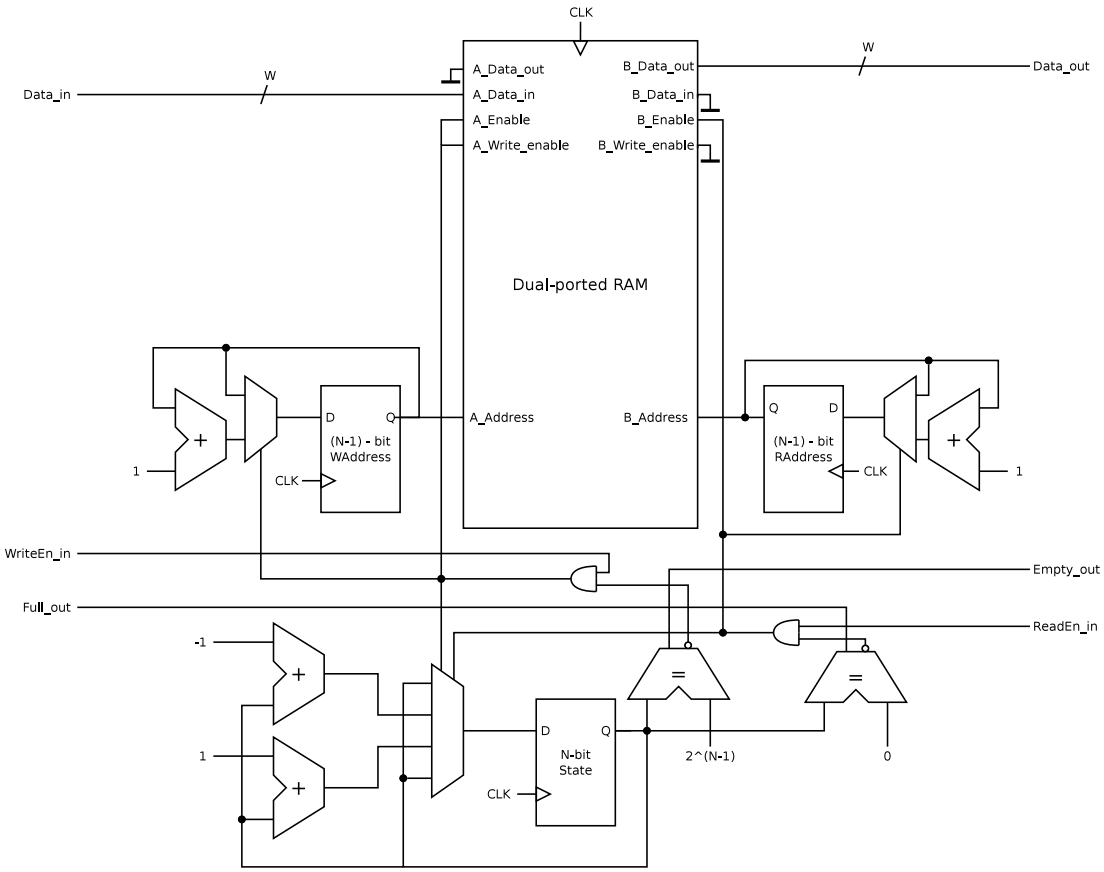


Figure 3.11: Synchronous FIFO circuit diagram with separate state and address registers. The RAM is assumed to have 2^N lines of W bits.

The synchronous FIFO was implemented as a behavioural Verilog module with the depth and width given as parameters. The parameter N in Figure 3.11 is specified as the bit-width of the address registers, and impact the size of the RAM as well as the register banks. The width parameter is labelled W in the figure, and only impact the RAM size and the `Data_in` and `Data_out` vector widths. The synchronous FIFO shown uses $3n - 2$ flipflops in addition to a RAM block of 2^N entries with W bits per entry.

The FIFO works by keeping three data values around, the read address, write address, and the state, which is encoded as the number of elements currently in the FIFO.

These can clearly be seen as registers in Figure 3.11. Control signals for reading and writing are derived from the state register, so that it is not possible to write to a full FIFO or read from an empty FIFO. The update the read and write addresses are controlled by these control signals.

It is possible to generate the state signal dynamically, from the read and write pointers, but this requires more complex combinatorial logic to account for the wrapping at the end of the pointer range. It was felt that keeping the count separate was an easier solution, despite using more flip-flops. In addition, keeping the count separate enables easy derivation of the full and empty signals as well as, as an extension and not shown, the half-full / half-empty signals.

The depth of the dual-ported RAM forms one of the system parameters of the AXI interconnect. It was found that in the Virtex-6 fabric, depths of up to 8 elements were automatically implemented using slice flip-flops as RAM storage, whereas at depth 16 or above the tools maps it to a BRAM instead. It was further found that in large designs this may lead to a shortage of BRAM blocks, possibly leading to synthesis failure. Thus there is a distinct trade off where buffers that are too large lead to large resource use, but buffers that are too small may impact system performance.

3.2.1.4 Logic blocks summary

This section has presented a number of low-level logic blocks of importance for the interconnect architecture. Importantly, it has focused on clock-crossing and buffering techniques and circuits, as these are some of the most influential and important circuits in the interconnect. If the clock crossings are not correct, then the system as a whole does not function; similarly, if the buffering is sub-optimal, the system will exhibit lower than necessary performance.

The next section will discuss these considerations in the context of the AXI interconnect network, and present how the presented blocks fit into this hierarchy.

3.2.2 AXI implementation

The origin and uses of the AXI protocol were discussed in some depth in Section 2.1.2.2. This section will cover the Verilog implementation of the protocol, and the resulting interconnect, in some detail.

An AXI fabric, as referred to from now on, consists of a means of connecting a number of AXI master interfaces to a number of AXI slave interfaces. The devices connected to these interfaces are usually processor cores, DMA engines, or other systems logic, if masters, and I/O, RAM and ROM blocks if slaves.

Taking a high-level approach and treating the interconnection fabric as a 'black box', the exact connection points of the various system devices does not matter. This is a false simplification, as such a fabric is a generalization of the ideal properties desired, and exhibits none of the tradeoffs necessarily present in a interconnection fabric. In a real system, traffic flow between devices affect the system due to contention, as only a limited amount of data can be transferred per unit time, per link or in the fabric as a whole. Which master and slave is connected where is therefore as relevant to system performance as what the internal topology of the fabric is. In addition, how the switching is performed is yet another parameter that affects system throughput in a non-obvious manner, as it depends on both locality of devices and topology.

The AXI fabric consists logically of a number of AXI *switches*, themselves comprised of AXI *channel switches*, arranged in some *topology*.

For the ML507 system, the AXI width parameter is also present. The width of data transferred over an AXI interface can vary; the physical width of the interface likewise. AXI, in itself, has a logical provision for transporting data in different sizes based on powers of two, from 1 up to 128 bytes wide. The physical size of the R channel alone can thus be varied between 17 up to 1033 wires, according to $2^{(w+3)} + 9$ for the R channel, $w \in \{0,...,7\}$; a similar range applies to the W channel. This has an impact on the number of wires and the amount of on-chip wire routing that is needed for any design. The exact point where a width is 'too large' or 'too small' is hard to quantify.

The rest of this section discusses the exact arrangements of these elements in the experimental system.

3.2.2.1 2x2 AXI switch variations

An AXI switch, as referred to here, implements two AXI master interfaces and two AXI slave interfaces. It arbitrates transfers from the master interfaces to the slave interfaces (AW, AR, and W channels) as well as transfers from the slave interfaces to the master interfaces (R and B channels). In effect, a single AXI switch consists of five AXI-channel switches, as each AXI channel is arbitrated separately. They are loosely linked, as the switches for the W, R and B channels rely on previously transfers on the AW and AR channels. Apart from this, an AXI switch can be logically thought of as a 2x2 crossbar.

It is, of course, possible to create 3x3, 4x4, or even non-square crossbars in a similar manner, but these will suffer more from wire length delays and arbitration complexities (leading to validation issues) than the 2x2. In addition, four 2x2 crossbar can be used to implement a 4x4 crossbar equivalent, which can easily be used in a 3x3 configuration. A 2x2 crossbar can therefore be used as a generic element to build larger switch fabrics.

The implementation of a 2x2 AXI channel switch in behavioral logic, then, is what interests us. There are several design choices to be made in implementing this structure, not least in how the buffering is handled; buffering is, as seen previously, essential to the AXI system.

There are three main areas where different design choices result in different behaviour; namely in the input/control stage, the post-switch buffering stage, and translating back to AXI from the buffer. These areas will be discussed in greater detail below.

3.2.2.2 AXI to control logic stage

This stage is responsible for, in any given clock cycle, 1) inspecting the ID and/or ADDR fields of the two transfers 2) based on this, determining which output ports to send the transfer to 3) checking if there is a conflict, and, if so, generating hold-off signals (de-asserting READY) and 4) driving the output ports with the correct values. It is clear that this could, especially in case of a wide and / or long input link, take quite a long time; longest path here is from the source register (in another switch or device), through the wires to this switch, through the routing look-up logic, through the arbitration logic, and back to the source device. It is also important to note, here, that the depth of the routing look-up logic may well have a relation to the AXI fabric topology, as discussed later.

The path through all this logic may very well turn out to be the critical path, seeing as it is passing through the actual transfer link twice. This path would then be setting the f_{max} of the entire AXI fabric and it is therefore a prime candidate for pipelining. The problem occurs when pipelining this signal with a register, inserting a register on the incoming transfer is also needed, otherwise it may have acknowledged a transfer not actually registered anywhere, simply because the de-assert of READY comes one cycle too late. Therefore a register for each wire in the incoming transfer must be inserted, if even a single register is inserted in the critical path. As shown earlier, depending on AXI width, this can be between 11 and 130 registers. There is therefore a trade-off between clock cycle length and area / power consumption, when making the choice of how to pipeline this stage. This choice is influenced heavily by two other factors; firstly, the width of the AXI fabric, and secondly, the delay of the logic on the far side of the AXI links. The former is a parameter influenced by the desired throughput of the system, while the latter is something that is influenced by the buffer to AXI logic in the other switches, as well as the corresponding logic in the system devices themselves.

Another way to simplify the problem is to take advantage of the topology and statically pre-route transfers so that no lookups are needed in the fabric itself. This is greatly assisted by using a non-redundant topology, so that there is only one right path

through the fabric for any one destination. Pre-routing simplifies away the routing lookup, allowing the critical path to be shortened further.

In essence, the optimal pipeline depth of this stage is influenced by - and directly influences - the important design parameters of throughput per unit time and throughput per unit power. The optimal choice is certainly not immediately obvious.

3.2.2.3 Buffering stages

As in the control logic stage, the main question in the buffering stage is the amount of area / power to spend versus the throughput of the system as a whole. The buffering stage can be implemented as a FIFO using flip-flops, a FIFO using a dedicated RAM, or just as a single register bank. It is even possible to introduce no buffering at all, and build an entire AXI fabric in a combinatorial manner.

As a high-level block, the buffering is standardised, but on a lower level the exact layout and, in particular, depth of the buffer will influence the performance of the final system, in specifically with regards to the throughput per unit power measurement. The width of the fabric will clearly also have an influence on this, as this directly affect the width of some of the channels.

The synchronous FIFO used for buffering are described in some detail in Section 3.2.1.3. The depth parameter of these FIFOs is brought out to the top-level configuration, and is a system parameter. A single AXI channel switch has two of these FIFO blocks; a full AXI switch thus has 10. The depth parameter is therefore important to control the size and area of the AXI interconnect, and the optimal value of this parameter is not known.

3.2.2.4 Buffer to AXI signalling circuit

This section discusses the interface between a generic FIFO buffer and the AXI `VALID` / `READY` signalling.

Asynchronous FIFO buffers are designs engineered to transfer data between two different clock domains. A discussion of the asynchronous FIFO buffer can be found in section 3.2.1.2. Synchronous FIFO buffers are by contrast limited to one clock domain only, and are used where different circuits may have the same average data generation and consumption rates, but bursty behaviour needs to be smoothed out. The design of synchronous FIFO buffers can be found in 3.2.1.3.

The interface to a FIFO buffer is relatively standard, but is not directly compatible to the AXI interface, creating a problem as there is a significant need to insert FIFO buffers into AXI channels in several places. A FIFO output has three signals associated with it; `output Empty`, `output Data`, and `input Read_Enable`. `Data` has valid data one clock

Listing 3.1: Initial FIFO to AXI attempt

```
always @ (posedge ACLK or negedge ARESETn)
begin
  if(!ARESETn)
  begin
    VALID <= 1'b0;
  end
  else
  begin
    VALID <= ~Empty & READY;
  end
end
assign Read_En = READY;
```

cycle after Read_Enable is asserted, provided Empty is not High. An AXI channel has two enabling signals, VALID and READY, which both have to be high in the same clock cycle to transfer data [7]. VALID is driven by the transmitter, and READY is driven by the receiver. A naive solution to connecting those two interfaces is presented in listing 3.1.

An advantage to this solution is that it uses a single flip-flop to arbitrate the interface but unfortunately it also has some problems. The main problem is that is is susceptible to data loss if READY goes low in the clock cycle when VALID goes high. This type of interface could work, if READY cannot go low at any time or if READY only goes low in response to the VALID wire going high, i.e. with one clock cycle delay. This is not something that can be relied on in an environment with different AXI devices aggregating together, such as in out SOC design, so a better solution must be sought. Listing 3.2 presents an improved solution, which does work in all cases. It is based on a three-state machine regulating the Read_Enable and VALID signals.

The solution presented in Listing 3.2 uses two state flip-flops, and decodes their state to drive the VALID and Read_Enable signals. The increase in hardware resource is justified in that this solution does not suffer from problems with dropped transactions as the previous example. There is still another problem with this design, namely that it under-performs in time; there is at most one transaction every two clock cycles, whereas the AXI protocol allows for one transaction every cycle. Effectively, using this construct halves the available bandwidth on the AXI channel. Depending on the application, this may fulfil the requirements, but as the application here may well be bandwidth dependant, a better solution must be sought.

The problem in this second listing stems from the fact that there is a delay between the issuing of the Read_Enable and the data being available; during that time period, the READY signal may disappear, and it is thus not possible to issue two consecutive Read_Enable signals as this could lead to data loss. The solution is to speculate that the READY signal will remain high, and add hardware to recover when this is not the case. This enables us to issue a Read_Enable every cycle, as long as READY permits. Listing 3.3

Listing 3.2: Second FIFO to AXI attempt

```
always @ (posedge ACLK or negedge ARESETn)
begin
  if(!ARESETn)
  begin
    state <= `IDLE;
  end
  else
  begin
    case(state)
    `IDLE:  state <= ~Empty ? `TX; `IDLE;
    `ENA:   state <= `TX;
    `TX:    state <= READY ? (~Empty ? `ENA : `IDLE): `TX;
    endcase
  end
end

always @(state)
begin
  VALID = 1'b0;
  case(state)
  `IDLE:  VALID = 1'b0;
  `ENA:   VALID = 1'b0;
  `TX:    VALID = 1'b1;
  default:VALID = 1'b0;
  endcase
  Read_Enable = 1'b0;
  case(state)
  `IDLE:  Read_Enable = 1'b0;
  `ENA:   Read_Enable = 1'b1;
  `TX:    Read_Enable = 1'b0;
  default:Read_Enable = 1'b0;
  endcase
end
```

shows this in Verilog.

Here, an extra state flip-flop has been inserted as well as a number of flip-flops, equal to the data width, to hold the data when recovering from an unexpected drop of READY. The exact number of additional flip-flops depends on the width of the FIFO and AXI channel, which is in general between 10 and 100 bits wide. The extra flip-flops required for this is justified in that this method has no forced bandwidth penalty; as long as there is data available the data rate is one transfer per clock cycle.

For clarification, the state machine of this solution is shown in figure 3.12. The TX1 through TX3 chain is the one followed in case of a mistaken early enable being asserted, with the xtra flipflop register being used in state TX2.

The extra multiplexer introduced to arbitrate between the backup flip-flops and the FIFO output does have an effect on the maximum length of the interconnect, which may lead to a slightly worse timing solution for the interconnect as a whole. This, together with the increased cost in flip-flops, has to be weighed against the 100 percent increase in bandwidth. Thus, the introduction of this structural improvement is only warranted if the bandwidth demands are such that the previous circuit cannot meet them.

These two latter constructs are both capable of arbitrating between a FIFO interface

Listing 3.3: Final FIFO to AXI signalling

```
always @(posedge ACLK or negedge ARESETn)
begin
  if(!ARESETn)
  begin
    state <= `INIT;
    xtra <= 0;
  end
  else
  begin

    case(state)
    `INIT:    state <= (Empty) ? `INIT : `RD;
    `RD:      state <= READY ? `TX1 : `TN;
    `TN:      state <= (Empty & READY) ? `INIT : (READY ? `RD : `TN);
    `TX1:     state <= (Empty & READY) ? `INIT : (READY ? `TX1 : `TX2);
    `TX2:     state <= (Empty & READY) ? `INIT : (READY ? `TX3 : `TX2);
    `TX3:     state <= (Empty & READY) ? `INIT : (READY ? `RD : `TX3);
    default:  state <= `INIT;
    endcase

    case(state)
    `TX1: xtra <=  fifo1_dout;
    default: xtra <= xtra;
    endcase

  end
end

always @(state)
begin

  VALID = 1'b0;
  case(state)
  `TX1: VALID = 1'b1;
  `TX2: VALID = 1'b1;
  `TX3: VALID = 1'b1;
  `TN:  VALID = 1'b1;
  default: VALID = 1'b0;
  endcase

  Read_Enable = 1'b0;
  case(state)
  `RD: Read_Enable = 1'b1;
  `TX1: Read_Enable = 1'b1;
  default: Read_Enable = 1'b0;
  endcase

end

assign Data = (state == `TX2) ? xtra : fifo1_dout;
```

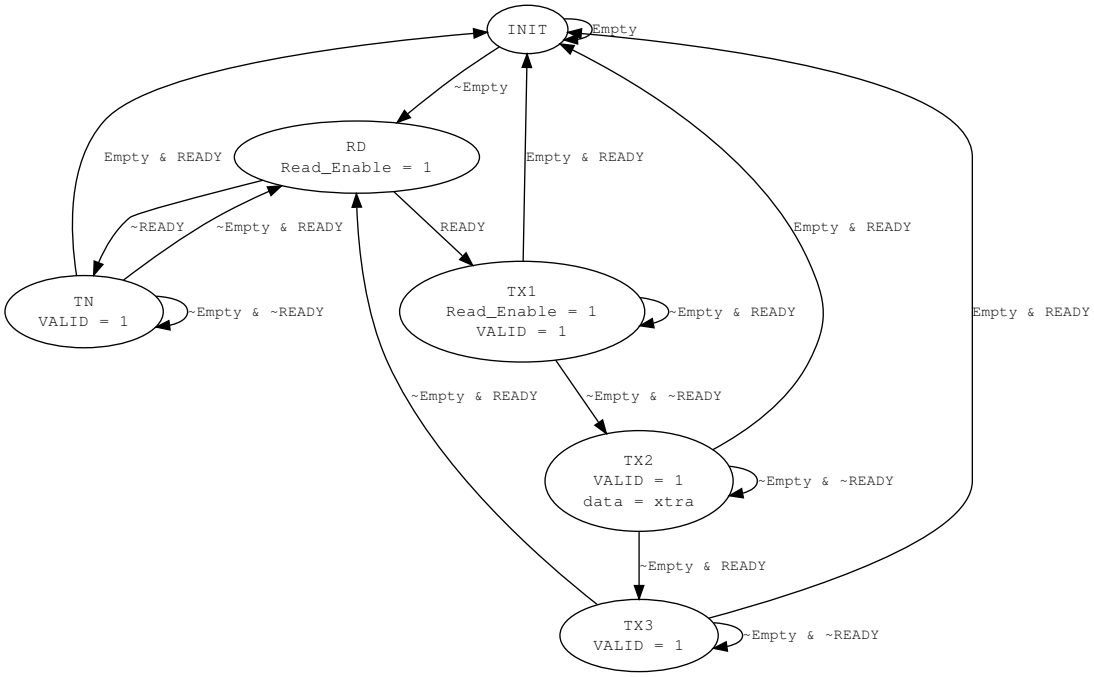


Figure 3.12: State machine for the FIFO to AXI layer

and a AXI interface correctly, but at different area and power, as well as throughput, points. Therefor, It cannot be said with certainty, without lengthy investigation, which one is best suited for a given application. It is, again, a multi-dependant problem that affects the whole system.

3.2.2.5 AXI extensions

AXI as a protocol does not handle certain types of traffic; it is intended solely as a memory access network. As such it has no provisions for CPU to CPU messages for coherency updates or interrupt signalling. Adding those to the protocol requires some additions to the baseline AXI protocol.

Interrupt signalling can be reasonably easily added, as it is in general a unidirectional channel from a memory / I/O device to a CPU core. The only problem is allocating the interrupt to a suitable core, depending on available resources. This can also be managed in a distributed manner, if each core publishes information of whether it it able to take an interrupt or not. There is thus one extra forward AXI channel with the interruptable information, and one reverse channel with interrupt requests. These can be routed along the same wire paths as other AXI channels, but require somewhat different routing logic inside the switches. This is not an insurmountable problem; all that is needed is that interrupt request messages be forwarded to the upstream switch that last indicated it was able to take an interrupt. If nothing is available to

take the interrupt, then the interrupt request will wait until something is. This extension was implemented in the ML605 system as it was required for multi-core interrupt handling.

Core to core traffic is harder to retrofit in than interrupt signalling. There are no already existing core to core communication links, and in addition, to make coherency work, extra logic is required on the core level to implement said coherency. No attempt to implement coherency using AXI was made.

3.2.2.6 AXI implementation summary

This section has presented the implementation of the 2x2 AXI channel switch, and in particular, has presented the design trade-off between two types of buffer to AXI translators. Additionally, the AXI extension implemented in the ML605 system has been presented.

The switches and routing methods developed herein were tested extensively in simulation and FPGA synthesis. Many issues relating to concurrent operation in switches, dropped transactions, and related problems have been found and resolved during the development. The AXI interconnect presented is therefore a reasonable mature and tested design, suitable for the experiments carried out in this thesis.

The exact composition of a AXI channel switch is, as discussed above, a complex problem with many inter-dependencies. The number of possible switches, as given by the possible choices for the constituent parts of the switches, is between 10 and 100, not including choice of AXI fabric widths, and depending on the permissible elements in a given design. Finding the optimal combination for a homogeneous fabric (where every switch is the same) is therefore a problem of searching through the switches and measuring the performance. This is further influenced by the selection of fabric topology, as well as fabric width, as discussed below. This problem is exacerbated if it is desired to find the optimal switch for every position in a heterogeneous fabric. In addition, in both the heterogeneous and homogeneous case, the optimal design is influenced by the traffic patterns and throughput demanded by the application running on the system.

3.2.3 Network-on-Chip topology

The topology of the AXI fabric is relevant to the overall throughput of the system; 2x2 AXI switches can be arranged in several different ways that allow any-to-any connections, but which have different characteristics in terms of total bandwidth, switches used, and total latency. For a given system, with n AXI masters and m AXI slaves, there are a large number of different topologies that would satisfy the any-to-any criteria.

The most obvious is the tree-based approach, where all masters are switched down using a tree into a single 2x2 switch, which is then expanded, again using a tree, up to the number of slaves necessary. This is shown in figure 3.13, which is taken from a automatically generated working circuit implementation. Note that this figure also shows the routing "table" for each exit to a switch, which is implemented in logic in side the switch itself. The tree structure is clearly visible in this example, and it is clear the that bandwidth limit here is in the switch labelled "00".

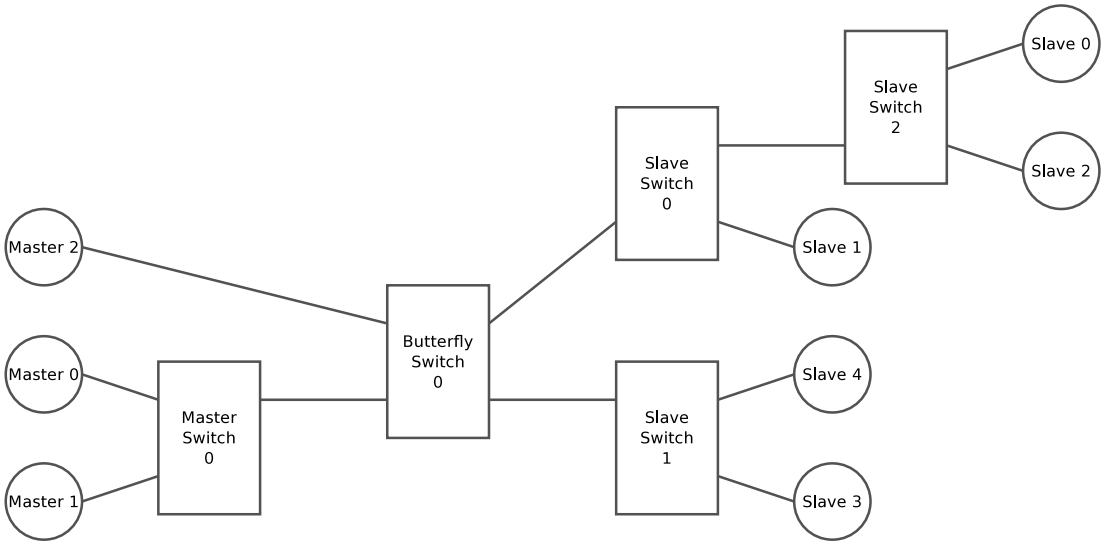


Figure 3.13: Singular fabric, demonstrating $s = 0$ and automatically generated input/output tree switches. 3 masters, 5 slaves. Boxes are 2x2 AXI switches, circles are system devices.

Apart from the tree approach, there is another way of arranging the 2x2 AXI switches, based on a perfect shuffle. this topology is generally referred to as a butterfly, due to the butterfly-like figure created in crossing wires in this fashion. This layout has better properties in terms of total available bandwidth, but may use more switches, and importantly, crosses wires more often than the tree based approach; this may lead to extra area needed for this implementation. In addition, the size of the perfect shuffle net is an issue, the question is whether this should be based on the number of masters or number of slaves, or on the smaller or larger of the two. To this end, two implementations were made for this; one that uses the maximum of the n and m, and one that uses the minimum. These are shown in figure 3.15 and 3.14, respectively. From these it can clearly be seen that using the maximum of the two incurs a penalty in terms of switches but have a greater throughput available and may also have simpler routing "tables", whereas the minimal is very close to the tree based approach in terms of switches but has more complicated wiring. This is the approach used in the ML507 systems, which used the 'maximum' and 'minimum' configurations.

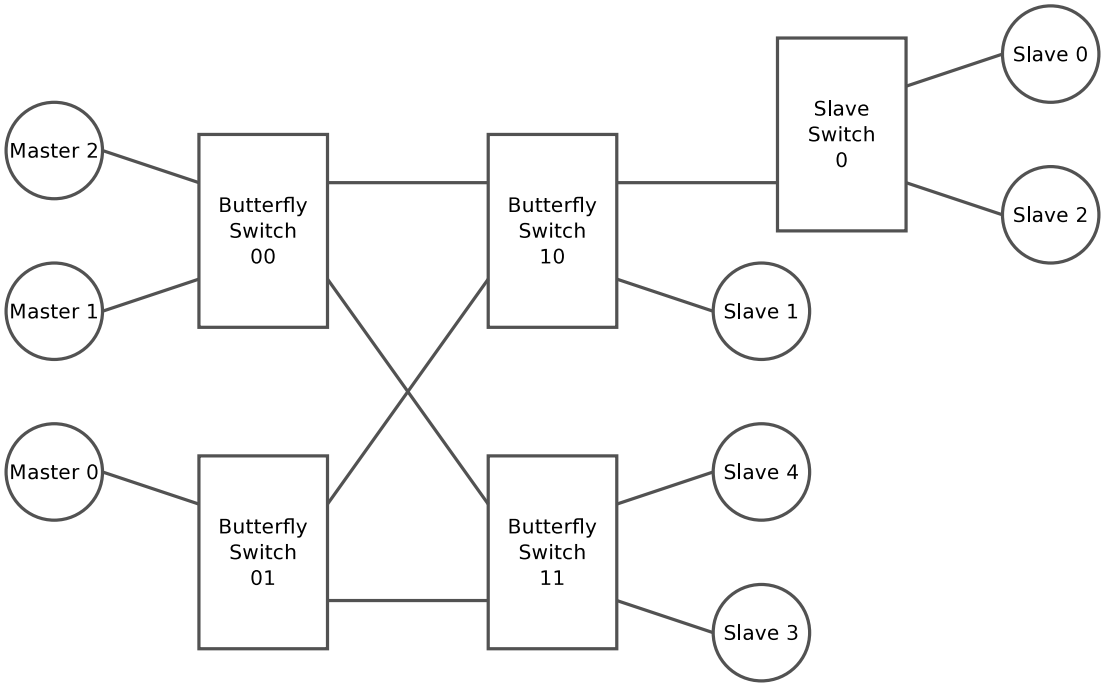


Figure 3.14: Minimal fabric demonstrating $s = 1$. 3 masters, 5 slaves. Boxes are 2x2 AXI switches, circles are system devices.

It can clearly be seen that picking the correct fabric, even between these three options, is not easy. There are also more options; in the above, the size of the shuffle part of the fabric was selected using a fairly simple function of n and m. In reality the only limitation is that the size (that is to say, the height of the fabric) is a power of two; i.e 2^s. s can then be chosen from the range $0 \leq s \leq \max(m,n), s \in \mathbb{N}$, and any such choice of s will lead to a working solution with different properties of latency, bandwidth, power consumption, etc. The original case, that of two root-to-root tree structures, can then be thought of as the case $s = 0$, and so, the topology of the network is determined solely on the basis of s, and the number of ways to choose this number increases as the number of devices in a system goes up.

Blocking is an important design parameter of interconnect networks, and refers to the capability of handling several traffic flows at once without interference. The AXI switches, as designed, are non-blocking, i.e, as long as the two input traffic flows are routed to different output ports, one flow does not hinder the other. If both flows are designated to go to the same output port, however, one will be forced to wait.

By extension, the butterfly networks built from these switches are non-blocking. Any input or output tree networks, however, are by their topological arrangement blocking.

In summary, the topology of the AXI fabric is reduced to the single variable s. The

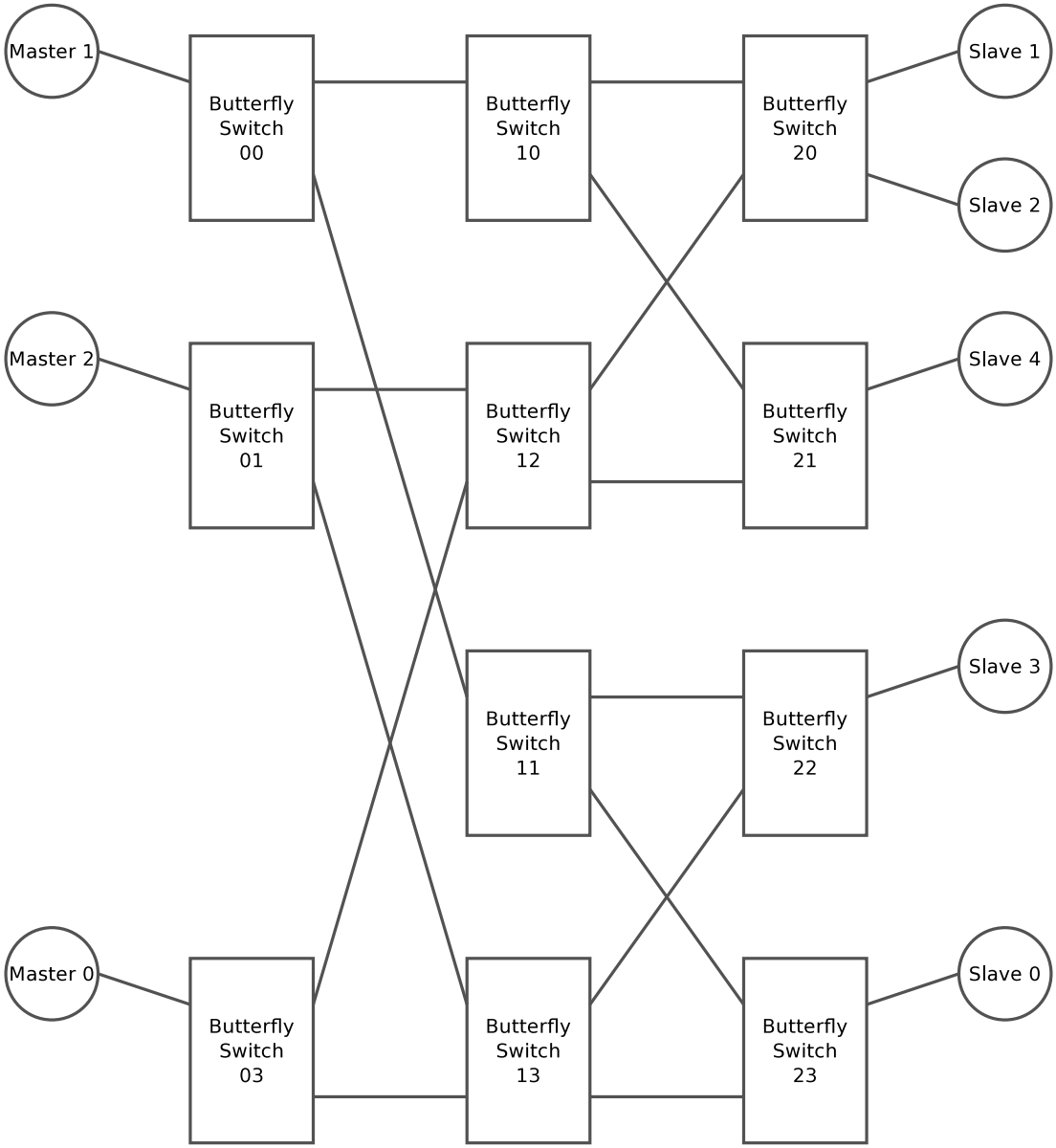


Figure 3.15: Maximal fabric, demonstrating $s = 2$ and unused switch removal. 3 masters, 5 slaves. Boxes are 2x2 AXI switches, circles are system devices.

figures 3.15 and 3.14 then represent the cases $s = 1$ and $s = 2$, showing the versatility of this parameter.

3.2.4 Processors

This section introduces the basic properties of the EnCore processor, and discusses the rationale behind its design and the design process and its various incarnations. In addition, it covers how it it is connected to the AXI fabric, and the optional clustering of cores. Parts of this section are based on an unpublished paper discussing the implementation and power efficiency of the EnCore processor, and as such, some paragraphs in this section have originally been written by Nigel Topham and Richard Vincent Bennett. The EnCore processor was developed at the University of Edinburgh by Nigel Topham, assisted by Xinhao Qu and myself. In addition, a number of people have been involved with the development of the cycle-accurate ArcSim simulator [3], supporting the same ISA as the hardware.

The single-core EnCore processor is configurable across a wide variety of options. Firstly, additional instructions can be selected either from a pre-defined set of extension options or using a dynamically-reconfigurable on-chip fabric. Secondly the caches can be configured in terms of capacity, associativity and block size. Thirdly, the system interfaces can be configured, which is taken advantage of when incorporating this core into a multi-core system. The processor supports a RISC instruction set, accessing a configurable big- or little-endian 32-bit address space. In general, the addresses above 0xFF000000 are reserved for I/O devices. For reasons of energy efficiency and die size, the current implementations are single-issue with in-order execution. This core also supports a comprehensive feature set, including timers, interrupts, exceptions and traps, and also support host-based debugging via a JTAG link. This JTAG link is, for this work, connected to the soft JTAG chain described in section 3.3. The instruction set includes all basic operators one would expect up to integer multiplication and division, and can be extended with floating-point instructions as required.

For this thesis two different core implementations were used. The initial work was undertaken with the 5-stage pipeline; for the later work and as development of the core progressed, the 3-stage version was used instead. Both versions of the core exist as simulatable models in the ArcSim simulator and synthesisable Verilog code, which have been extensively tested, both as a part of this thesis and in other settings. Since both versions of the core are compatible in terms of ISA and interfaces, there were no major differences in results due to the switch from 5-stage to 3-stage. Ideally, exploring the trade-off between a 5-stage and 3-stage pipeline using the methods in this thesis

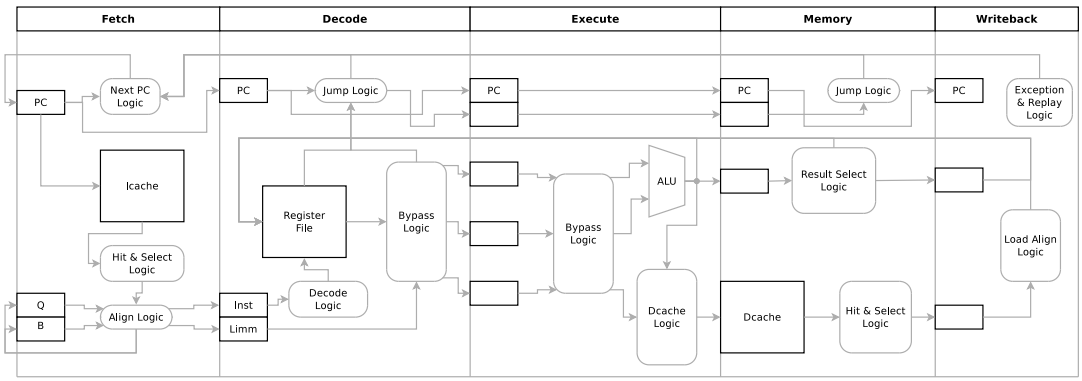


Figure 3.16: Overview of pipeline in the 5-stage version of the processor.

would be beneficial, and by extension also other heterogeneous configurations, but this was not feasible due to time constraints.

The EnCore 5-stage pipeline is outlined in figure 3.16, and should be reasonably familiar from most RISC designs.

3.2.4.1 IO properties

From the point of attaching this core to the interconnect, a few properties are worth noting. First of all, as the core has its own internal cache controller, it is capable of issuing multi-cycle operations to read and write cache lines. This is the only way multi-cycle operations can be issued, and these are not generally under the direct control of software. It is possible to execute special cache-ignore loads and store instructions, but as these are 32-bit, they do not become multi-cycle operations in a 32-bit wide or larger AXI interconnect. Secondly, while the core is waiting on a cache refill or a cache-ignore operation, its in-order nature means it is effectively in a wait state. While this lasts, the clock gated nature of the design (see section 3.2.4.4) ensures that most of the transistors in the core are not switching, and it is effectively in a low-power state. This also means that a single core can only have up to two outstanding requests at once; one for the dcache and one for the icache. In most cases a core will only have one outstanding request. The multi-request nature of AXI is therefore ill suited to mapping directly to one of these cores, and this is addressed in section 3.2.4.6. In addition, as each core waits on outstanding requests, it is up to the interconnect and I/O systems answering that request to release the core back to execution. If the outstanding request is an operation that cannot be completed for some time, such as an I/O write to a full device, the core in question will be idle until it does complete. It could be possible to interrupt the core to switch to another task in these cases, but this has not been implemented in the operating environment (see section 3.5) as it has a number of potential problems

and is not necessary for system operation. Thirdly, the 5-stage version of the EnCore does not currently export the lock request / release information from the atomic operations to the interconnect, and as such, hardware locks are not implemented for this pipeline. The 3-stage version does export these signals properly, and hardware locking is supported all the way to the memory controllers with this configuration. In addition, it should be noted that the core does not support coherency between the instruction and data caches, and that the instruction cache never writes back to RAM. This latter property somewhat simplified some of the implementation details, but the lack of coherency also means that the core cannot reliably handle self-modifying code.

3.2.4.2 Extensions

For processing data-centric applications, ISEs may be used in the EnCore. The mechanism by which these are supported exists in both the pipeline and the instruction set itself. The instruction word allows for up to 256 extension instructions, each of which may represent a large area of application data-flow. The extension interface currently relies on vectors and permutation of same in both the encoding and the hardware pipeline, so as to allow the maximum number of operands in the relatively constricted 32-bit instruction word. Up to three input and two output vectors each containing four 32-bit operands may be specified, along with permutations. The hardware component for such extensions is generally limited only by the physical constraints placed on the silicon, such as die area, energy consumption, and critical path length. ISEs with multiple pipeline stages are supported by the EnCore processor, alleviating the critical path constraint and allowing further acceleration to be achieved through temporal parallelism. ISEs can also actually reduce energy and even power consumption, as parts of the core are clock-gated and need not remain active during ISE execution.

For this thesis, no extensions were generated or used. Extensions could be used to de-homogenise the experimental system, and thus using EnCore processors with extensions presents a subtly different problem than the one addressed in this experimental setup.

3.2.4.3 Development

Creating a new low-power processor is a sizable undertaking, requiring coordinated research and development of tools, microarchitecture and system architecture, and a significant effort in functional verification. A high-speed functional simulation model was developed, linked to hardware simulation for co-simulation and verification, an experimental compiler, an FPGA prototyping platform, and a sequence of silicon devices implemented in 130nm and 90nm standard cell CMOS processes. The depth and

breadth of infrastructure and experimental systems constructed during this process is unusual for an academic research project, and as such is worth explaining in more detail.

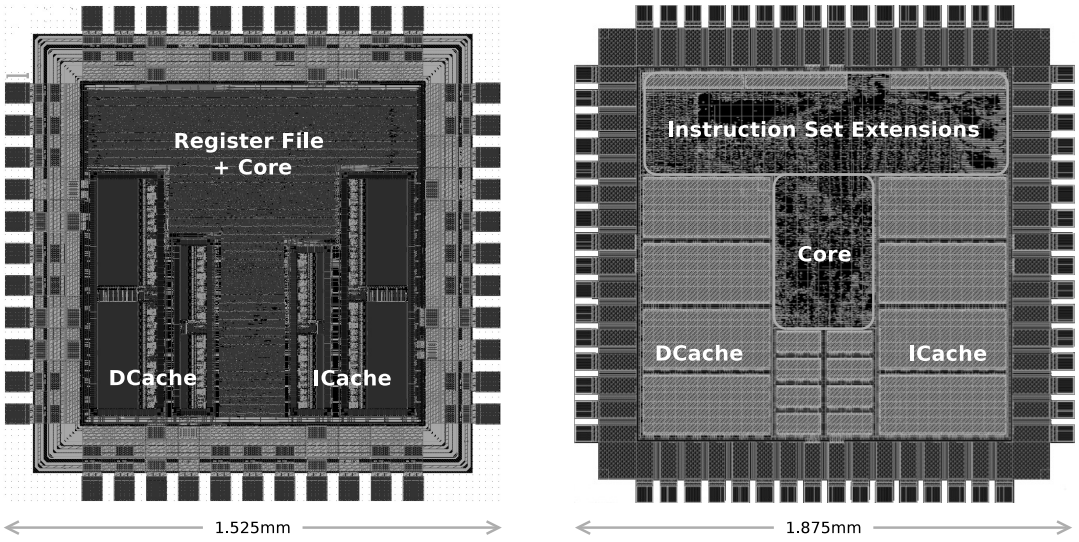
3.2.4.3.1 Simulation A critical element of any realistic processor development is the creation of a *Golden Reference Model* against which hardware can be verified. In this case it took the form of an instruction set simulator (ISS), which was also fast enough to run applications in real-time, boot a Linux kernel, and serve as a target during compiler development. In order to support microarchitectural design-space exploration, the simulator also included a cycle-approximate performance model capable of modelling cycle-level behaviour to within $\pm$ 5%.

This was extended with a Verilog PLI interface to allow a hardware simulation of the processor to step through the same program on both hardware and software, verifying that the state transitions made by both models were identical. This is a powerful paradigm for functional verification, as each model checks the other, and the probability of common-mode errors is very low.

A random test program generator was constructed, and from this a large number of random but perfectly-formed test programs were generated. These gave full coverage of the instruction set and allowed all aspects of the processor to be verified against the simulated model. This multi-modal simulator later became a major project in its own right, and is now in commercial use.

3.2.4.3.2 FPGA Design In order to reduce the inherent risk in the silicon fabrication process, the processor and its associated system-on-chip architecture was first run in an FPGA platform. This later became an extremely important route for the development of multi-core prototypes, which is discussed towards the end of this section.

Various I/O devices, controllers and memories had to be added to enable actual code execution from main RAM. Initial tests synthesised the 5-stage pipeline for operation in a Xilinx Spartan-3 device, and successfully ran it at a clock speed of 25 MHz. Subsequently Virtex-5 and Virtex-6 devices were targeted, in which the clock speed could be increased to beyond 50 MHz, as well as enabling multi-core instantiations due to the larger fabrics available. The logic of the complete system was divided into two parts; that which would be fabricated in silicon, and that which would remain in the FPGA. these two parts were initially mapped to two separate Xilinx ML507 FPGA boards, connected via a set of signals identical to the I/Os of the planned silicon devices. This allowed testing of all chip interfaces prior to fabrication. A daughter board for the ML507 was designed, with a socket for the test chip, allowing us to simply re-



(a) (b)

Figure 3.17: 3.17(a) GDS image of the 130nm test chip with functional areas highlighted. 3.17(b) GDS image of the 90nm test chip with functional areas highlighted.

move one of the ML507 boards and replace it with the daughter board. This assembly can be seen in Figure 3.7.

3.2.4.3.3 130nm Test Chip The first test chip was fabricated using a 130nm UMC process, on a Europractice multi-project wafer. This first tape-out of the EnCore was configured with 8 KB instruction and 8 KB data cache, both direct-mapped, together with the 5-stage version of the core. An in-house RTL-to-GDS silicon implementation flow was used. This was based on Synopsys front-end synthesis and back-end layout tools, and Mentor Graphics Calibre physical verification tools. The complete flow took approximately 8 hours of compute time on a quad-core Xeon server.

Physical dimensions of this implementation was 2.25 mm^2, 0.366 mm^2 of which were taken up by the core itself; the rest was dedicated to caches, pad ring, and a small number of placement keep-outs designed to aid routing. As the shared wafer used was intended for mixed-mode RF devices rather than logic, the top-layer of this 8-layer process was designed for inductor implementation rather than signal routing. The design is therefore restricted to 7 metal routing layers.

Static timing analysis indicated the ability to run this core at about 250 MHz, 1.1V in the worst case. In practice it was found that all devices performed correctly up to 375 MHz, at 25 Celsius and nominal 1.2V. The devices were packaged in a 7x7mm 48-pin plastic QFN package.

3.2.4.3.4 90nm Test Chip Having successfully completed a 130 nm test chip, moving to a 90 nm process and a slightly larger die was a logical step. The primary motivation was to incorporate a dynamically-reconfigurable instruction set extension fabric to experiment with novel mechanisms of trading more die area for lower energy consumption. As well as the process shrink, the die area increased to 3.33 mm^2, and reduced the baseline core area to 0.096 mm^2. The 90nm chips incorporated a core with separate 32 KB 4-way set-associative caches for data and instructions. This proved to consume more power, and despite higher performance may be too much for a single core. The design tools predicted this core would perform at 350 MHz in the worst case, and in the laboratory it was found possible to run the manufactured cores reliably at 600 MHz, under typical temperature and voltage conditions.

3.2.4.4 Power efficiency

Power efficiency is of paramount importance for future many-core systems; inefficiencies within a single core will be replicated many times, and must therefore be avoided. The goal was to maximise the computation bandwidth per mW of power, through the use of power-conscious design and power-optimised physical implementation.

One of the most important factors in dynamic power optimisation in synchronous logic is *clock gating*. The processor is written in a style that permits synthesis tools to infer gated clocks automatically. When this is enabled during synthesis, 99% of all flip-flops were driven by a gated clock. In addition, it is also important to minimise the number of spurious signal transitions in each block of combinatorial logic. In practice this can be achieved by designing the combination logic in ways that prevent signal transitions from propagating into unused portions of combinatorial logic. Great care was also taken to maximise the resource sharing between different aspects of functionality, where those aspects were never active simultaneously. This helps to minimise die area, and hence leakage power.

3.2.4.5 AXI translation layer

As mentioned in section 3.2.4.1, the EnCore in-order pipeline is not optimal for the multi-transaction nature of AXI. Initial system builds, in particular those involving the Virtex-5 boards and 5-stage pipelines, nevertheless attached one EnCore as an AXI master device. This was done through a translator attaching to the existing IBUS and DBUS structures of the 5-stage pipeline. This translator kept some state, and translated with a clock cycle delay, the two incoming BVCI [35] streams into a customised AXI master interface. In addition, the translator was running at the core clock frequency, but issued the AXI requests into a set of asynchronous FIFO that crossed the boundary

to the interconnect clock domain. In this way, it was possible to keep the core clock and the interconnect clock separate.

3.2.4.6 Clustering

The BVCI to AXI translation was eventually found to be wasteful and unnecessary, and as work progressed on the 3-stage EnCore pipeline, a better alternative was needed.

Thus an interface for this core version was developed that instead of connecting to the BVCI buses, connected to the proprietary internal data and instruction cache buses. As it turned out, these internal buses were more similar to AXI signalling than the BVCI buses, and thus less logic was required for this immediate interface.

In addition, the interface did not translate these internal buses directly to multi-channel AXI, but instead onto a shared-wire concatenated AXI version. This is different from normal AXI in that it does not use separate wires for each channel, but multiplexes the 5 AXI channels onto two sets of unidirectional wires, one for traffic from the core (AW, W and AR) and one to the core (R and B). An extra three wires were inserted into these concatenated channel wires to distinguish between the 'real' AXI channels for any transfer. The signalling on these channels are otherwise identical to the AXI specification, using fully responsive ready / valid signals and carrying the full width of the AXI channels. This concatenated AXI interface is optimised for a low-traffic master, as such a master does not tend to overlap transactions. Thus each core was fitted with an interface converted from internal logic to concatenated AXI. As the internal protocol and AXI was already quite similar, this was found to be almost entirely combinatorial with a zero-cycle forwarding cost in most cases; a single buffering set of flip-flops were required for rare cases when the core internals expected a value to be kept on the data lines. As each core can only issue up to two separate transactions at once, one for data and one for instructions, these were numbered as 0 and 1 on a single AXI ID bit line, indicating which transaction belongs to which. This concatenated AXI interface was then brought into a clustering module, capable of attaching to up to 8 such concatenated AXI interfaces and breaking them out into a single fully capable AXI master interface. This module, too, is fully combinatorial, and is in effect a 16-cross-3 (downstream channel) and a 16-cross-2 (upstream channel) crossbar. It can thus handle a transaction on each of the master AXI channels at once, and forward them to different concatenated AXI interfaces, as dictated. This module also fills in the extra 3 bits of the AXI ID field, so that each ID corresponds to a particular processors' data or instruction buses. Finally, the generated AXI are fed into asynchronous FIFO buffers, as with the simpler translator, meaning that core and interconnect clocks can be separated.

Combining of EnCore processors into common-interface blocks in the described manner is referred to as clustering, as it gathers the cores into clusters with common clocks and top-level interfaces. These clusters are then connected to the AXI interconnect, as there should be enough traffic from the cluster to warrant the increased capacity of full AXI. Additionally, a cluster is a larger unit than a single core, and while a single EnCore as part of a SOC is not large enough to warrant its own voltage/frequency island, the cluster may well be; this is thus a convenient grouping to enable these technologies in the EnCore based system. This was not attempted in the experimental work, and remains to be done.

3.2.4.7 EnCore summary

EnCore is now, mostly due to the undertaken development work, a tested, mature and capable processor, with some unique properties in the area of power and size for its capabilities. In this work it has been used as a generic processing unit due to these properties, as fitting up to 12 of these cores onto a single FPGA is possible, as well as clocking them at up to 75 MHz. This far outstrips any gate-level simulation methods that may be conceived, and is only rivalled by real-silicon implementations and high-level instruction set simulation, such as ArcSim in JIT mode [3]. These cores have additionally been taped out and been shown to work in real silicon, and they are thus shown to be functionally correct. These cores can be connected to an AXI interconnect framework, either as stand-alone or as clustered. Also, particular properties of the cores in terms of the the interconnect performance and design have been shown.

3.2.5 RAM controllers

A RAM controller connects an AXI slave interface to an actual memory device, such as a on-die SRAM or off-die memory chip. In effect, the controller translates the AXI requests to memory operations and manages the memory device appropriately, including resolving atomic requests and other concurrent traffic.

The RAM controllers used come in two types; one that connects a specifically instantiated proprietary Xilinx DDR2 controller, which supports the DDR2 memory module on the ML507 board, and one that only uses on-die BRAM. It would be possible to develop a DDR controller to interface with the Xilinx DDR3 controller for the ML605 board, but as this is a DDR3 part with an increased latency, it is not a straight re-use of the same Verilog module but rather a re-write. Moreover, it was not found to be necessary as the internal BRAMs in the X6VLX240T is enough for 512 kBytes of RAM.

This section will describe these two types of RAM controller in detail as the RAM latency is a system concern. It should be noted that the main design-space parameter

relating to the RAM controllers is their number; from 1 to 8 controllers were used, effectively breaking the memory into as many banks. This is expected to have an impact on the system performance directly and indirectly.

3.2.5.1 DDR2 controller

The DDR2 controller interfaces to a (off-chip) DDR2 RAM. These RAMs are in general built on dynamic technology, and thus need refreshing and managing with certain time delays. A DDR controller manages these timing constraints and drives the control wires appropriately.

For these designs a proprietary DDR2 controller from Xilinx was used, fitted with an AXI translation layer. This translation layer is the most important part of the design; the DDR2 controller can be taken off-the-shelf. The translation layer makes great use of synchronous FIFO to integrate and derive data streams, and in the following they have been abstracted as generic blocks. Section 3.2.1.3 has more details on the synchronous FIFO setup used.

The AXI translator consisted of three parts. One module connecting to the AXI AW, W and B channels, generating address and data / strobe pairs to be written to RAM. One module connecting to the AR channel, generating addresses to read, and one module using the previous two together with the Xilinx controller to arbitrate requests and format them appropriately, delay issued requests until they are responded to, and format the R channel responses.

The translator required several management processes to first arbitrate read and write accesses, and secondly to reconcile the different read/write lengths. A write cache was not implemented as this was thought to make the module structure and logic more complex.

The AXI translator used here does not handle atomic operations and other complex requirements signalled in the AXI protocol. These features were not required for the base system, and were not exposed by the 5-stage processor pipeline that was used with the Virtex-5 FPGA and the ML507 systems.

3.2.5.2 Block RAM interfaces

Block RAM is the native memory resource in the Xilinx FPGA families, and is provided as hard blocks on the die. A device was developed that interfaces the AXI slave interface to these blocks, fully supporting locks and concurrent accesses. This device has no bearing on the design space, however, and thus further information about this device can be found in the appendix, section A.4.

3.2.6 Experimental system debugging

No complex system is without imperfections, or rare cases where the behaviour of the system is unexpected. Much effort has gone into correcting such imperfections in the experimental system described, coupled with testing both in-FPGA and in behavioral and gate-level simulators. Many problems have been found and corrected in these efforts, and many configurations have been shown to perform appropriately as a result. Due to the additional complexity of the configurability required, it is however likely that there are undiscovered problems that manifest in some configurations and not others. These problems may prevent an otherwise viable design from functioning appropriately for not readily apparent reasons, and it may take days of investigation to find one such imperfection, due to the complexity and potential size of the system. Thus there are bound to be undiscovered, or uninvestigated, problems with the experimental system, which may manifest as complete or partial failure to complete benchmarks. This does not invalidate these efforts, as it is known that there are configurations that do perform flawlessly.

3.2.7 Experimental system summary

This section has presented some of the important low-level design tradeoffs in the experimental system, as well as the architecture of the processors and memory interfaces. Some of these details, such as FIFO depth, AXI width, and interconnect topology, are brought out into the top-level system configuration, becoming system parameters and eventually design space parameters. Other details, such as the exact FIFO arrangements and clock domain crossing details, are included for completeness and to ensure that these topics are accounted for, and, ultimately, to show the level of confidence possible regarding the reliability of the system. In addition, extra information about some of the I/O devices is included in Appendix A, as it does not have immediate bearing on the design space.

The topics discussed in this section will not be covered elsewhere except as parts of the entire experimental system.

3.3 Data extraction

As mentioned previously (Sections 3.2 and 3.2.4), the experimental system contains a soft JTAG [62] chain. This JTAG chain has three purposes, namely to provide a initial programming path, to provide a method of extracting experimental data, and for debugging the EnCore processors. JTAG in itself is a method for serially accessing

pin data for the purpose of testing the board connectivity between devices on circuit boards. A single board needs only one JTAG chain, as JTAG controllers can be chained, and the protocol includes methods for detecting and enumerating devices. JTAG implies an extra 4-pin header on a device, but due to its serial and stateful nature enables many test scenarios. This functionality was used not to test the physical pins but to provide the required functionality of master-driven communications with the instantiated experimental system. As a master device for the soft JTAG chain a series of scripts were used, which communicate with the parallel port on the host machine, as well as a parallel-to-JTAG cable. By using scripts it was possible to easily accommodate the variety of devices present on the custom JTAG chain, but at the cost of slow access.

JTAG, in itself, is another clock domain in the experimental system, but as its interactions are all slow with respect to the internal system clocks, the two-phase resynchroniser (Section 3.2.1.1) can be used to arbitrate these crossings.

3.3.1 Programming

Initially, before the faster UART method from section A.2.2 was built, programs were loaded under the direct control of the JTAG master. This involves a specific JTAG-to-AXI translator, which works as a generic JTAG device and AXI master. By writing a 32-bit address and 32-bit data field into the JTAG device, these would be automatically inserted into the interconnect as a AXI write transaction. Repeating this procedure for every address in the program effectively transfers the program to memory. This device acts as a single JTAG device in the JTAG chain, with a single 64-bit writable register.

This device also has registers for starting and stopping the cores globally, so that the host can halt the cores while programming the memory and then reset them to start the experimental system when programming is finished.

3.3.2 Experimental counters

To extract the operational data at the end of the program's run, counters that tallied the important transactions within the experimental system were needed. These counters were attached to some internal registers in the EnCore, such as the instruction retire signal, halted signal, pipeline stall signal, etc, so that they could tally the values in Table 3.1 from inside each EnCore.

The counters attached to the EnCore control wires were 48 bits long, so that they could capture sufficient amounts of data. It was found that 32 bits was not enough, as a 32-bit counter will overflow after 57 seconds at 75 MHz. The 48-bit counters do not overflow until 3.753×10^6 seconds, or 6.2 weeks, providing ample measurement margin.

Measurement	Explanation
not-halted	Total number of clock cycles in which the core is not halted
commits	Total number of clock cycles an instruction was committed
stalls	Total number of clock cycles in which the core is stalling
I/O-cycles	Total number of clock cycles spent waiting on I/O channels
I/O-operations	Total number of I/O operations performed by thus core

Table 3.1: Measurements collected by the counters in each EnCore in the experimental system. Each counter in the EnCore is 48-bit, and thus these counters can capture up to 3.753×10^6 seconds worth of data at 75 MHz core clock.

Two counters were attached to each instantiated AXI channel switch to monitor the two output paths, giving us the total number of transactions on every channel path in the interconnect. These counters were relatively small, 32 or 48 bits only, and the logic surrounding them very simple. As a FPGA is generally logic and not flipflop-constrained, they were not a major drain on FPGA resources.

All counters were built to transfer their values to the JTAG clock domain through a two-phase resynchroniser. In the JTAG domain, they were all connected to form a single shift-register. As the number and placements of the counters varied from system instantiation to system instantiation, the contents of this shift register also varied. To assist in decoding this data stream, the system configuration step was enhanced to also output a file describing the contents of this shift register. In a larger system with several cores and a complex interconnect, this shift register could be up to 1 kilo-bit. This shift register was then connected to a JTAG controller, so the shift register could become part of the JTAG chain as necessary. Through this it was possible to read out system data after the system had finished execution, without perturbing the system state. In addition, the long runs of 0 due to the length of the core counters helped verify the alignment of the shifted out data, increasing the confidence in the data extraction process.

The host detects that a program has finished executing by periodically reading out the counter values to see if they are still incrementing. If they have changed since the last read out, the program is still running, but if they are static it has halted. The toolflow then saves the halted values as the final values for a given program.

3.3.3 Energy model

This section describes the energy models used for this thesis. The energy model depend on the extracted counter values, using these as indicators on how much energy

is spent. The underlying assumption for each model is that the number of switching events in the interconnect is linearly related to the energy spent, and thus these models use switching events as a proxy for energy. The energy model is not intended to present a absolute value for the energy spent in the interconnect, but to exhibit the difference in energy in the interconnect between different applications. The power model used do not try and account for power elsewhere but in the interconnect, and also does not integrate static and dynamic power but treats them as separate entities, as it is not possible to fix the ratio between these quantities.

Static power is, for a given technology node and manufacturing run, linear in regards to the design area. The FPGA utilization ratio is therefor taken as a proxy for the static power. Optimising for, and predicting, area is thus also influencing the static power. The FPGA utilization ratio used in this thesis includes all cores and I/O devices in the system, and thus is a system-wide metric. It was found that the interconnect area in the dataset from Chapter 5 and 6 was on average 30% of the FPGA utilization, and consequently, the interconnect would be responsible for 30% of the static energy for these designs.

Dynamic energy is related to the number of switching events that happen in the circuit under operation, and is therefore influenced by the application. For a given application, the dynamic power spent in I/O and processors is assumed to be constant, as the program will need to perform the same operations regardless of the interconnect. The energy spent in the interconnect is then taken to be related to the number of switching events on the wires in the interconnect.

The data extraction counters were used to find the number of switching events for each connection in the interconnect. The considered events were those at the output of a FIFO, so that switches without an output buffer did not contribute to the number of switching events. Each switching event was multiplied by the number of wires in the AXI channel being switched, so that we obtained a total number of switched wires metric. By the assumption that switching events are linearly related to the energy spent, this value is taken as a proxy for the dynamic energy spent.

In addition, for the ML507 system in Chapter 4, the number of clock switching events, i.e, the cycle count, is added in to the total number of switching events. This is included to approximately model the effect of the interconnect clock distribution networks on the energy consumption. Including the clock events has the effect that longer-running benchmarks consume more dynamic power, assuming constant clock frequency. The equation for the total number of switching events in this model is given in Equation 3.1, in which W is the width of the interconnect design parameter and M_{fsb} is the frequency multiplier separating the core and interconnect clock domains. The

X_{evt} variables refer to recorded switching events on each channel aggregated over the entire interconnect fabric.

$$\begin{aligned} E = & AR_{evt} \times 60 + \\ & AW_{evt} \times 60 + \\ & B_{evt} \times 12 + \\ & R_{evt} \times (13 + W) + \\ & W_{evt} \times \left(11 + 1\frac{1}{8} \times W\right) + \\ & \max(cyc_0, cyc_1) \times M_{fsb} \end{aligned} \tag{3.1}$$

Adding the clock period was not done in the ML605 system in Chapters 5 and 6, and there is no influence from the clock on the energy consumption in these experiments. Total energy in these chapters refers to only the dynamic energy from the switching within the interconnect, taking the buffering of the interconnect into account, but not including static energy or clock tree energy. The equation used in these experiments is similar to the one in Equation 3.1, but without the dependancy on W, as W was fixed to 32 in this setup, and omitting the last line.

The presented proxy measurements are not in themselves power measurements, but are considered to contain the effective ranking and comparative relations present in the SOC systems studied. Real power figures would require full synthesis, extraction and gate-level simulations, something which is very time consuming, especially for full applications. It is implicitly assumed in this model that the wire lengths and drive strengths of flip-flops are uniform across all configurations, which will tend to slightly underestimate the dynamic energy consumed by the larger configurations. This is offset by drive-strength and threshold voltage aware synthesis tools and is thus a justified assumption. For these reasons, these proxy measurements are presented as power measurements in this thesis.

3.3.4 Debugging ports

In addition to the system control and data extraction JTAG controller, each EnCore also has its own JTAG controller attaching to an internal debugging bus. This enables complex debugging operations such as halting, resuming, injecting instructions, and inspecting the full microarchitectural state. These controllers were also attached to the JTAG chain by the configuration scripts, so that the debugging operations would be available if necessary. The debugging protocol for the EnCore is not, naturally, part of the JTAG standard, but is a proprietary protocol on top of JTAG. In practice, the

debugging ports were rarely used due to their complexity and the lack of a suitable multi-core-enabled debugger. This did mean that a JTAG scan could tell how many EnCore processors were present in a given uploaded system and should it be necessary interact with them.

3.3.5 Data extraction summary

The programming and data extraction from the experimental system relies in large parts on the JTAG protocol, and on a a number of soft controllers in the instantiated systems. These are included in synthesis by the system configuration scripts, which also configure and connect them together in a manner appropriate to the parameters given. The host system also uses this interface to start and stop experiments, as well as detecting when a program has finished.

3.4 Synthesis Toolflow

Experimental system synthesis consisted of two general steps; configuration and compilation. The configuration step takes the generic source for the entire system and configures it to the experimental system setup desired, guided by an input description file. The compilation step then takes this configured source and generates a bitfile that can be used to program the FPGA, or, if the design does not meet constraints, report an error.

The compilation step described is tailored towards FPGA implementation, but this should not be taken as in indication that this is the only compilation available. Given the appropriate tools and libraries, a configured system could also be compiled to an ASIC or other technology fabric, but this is beyond the scope of this thesis.

3.4.1 Configuration

The experimental system exists as generic system files coupled with a set of configuration scripts. These scripts specialise the system files, and in some cases, generate source files entirely, to meet the system specification. The configuration scripts are driven by top-level Makefiles that configure the sub-modules hierarchically.

In addition to the interconnect and I/O modules, the configuration steps also instantiate and configure the EnCore project files, as a subsidiary project. This project has its own configuration system that was invoked to generate the EnCore source.

In general, the Verilog modules in the experimental system exist as generic Verilog

with macro statements, using the VPP* system. By specifying the macro variables used in these files, the configuration system generates plain, customised Verilog files. Much of the module-sizing configurability is enabled this way.

For larger, structural setup, this is not enough, as it requires more customisability. The interconnect topology and top-level system modules in particular requires more customisability than can be provided through this macro system. These files are instead generated wholly custom by the configuration scripts, and are thus able to be wildly customizable, as the design-space generation demands. In addition, the scripts generate header files that then influence the macro expansion described previously.

This customization process is guided by a single input files that describe the important parameters of the desired system, such as number of processor cores, the amount of memory banks, the interconnect topology, the memory map, I/O devices used, preloaded files, and other parameters. In essence, this file contains the parameters of the design space that the customised system will be part of. These configuration files are 1 to 2 kBytes each, and are really just short human-readable descriptions for the desired system. To enumerate the design space, then, a number of input files need to be generated, one for each way the system can possibly be configured. In the final 3-stage pipeline Virtex-6 system, the number of possible input files was 510,000.

Configuring a system takes a few minutes and a few hundred MB of disk space, and as such it is possible to enumerate all possible input files but not configure all possible systems. In addition to the source files, the configuration system also generates some files needed by the subsequent compilation tool flow.

3.4.2 Compilation

Compilation consists of a relatively standard Xilinx tool flow, driven by customised Makefiles. As two different FPGAs were used for these experiments, a toolchain change similarly necessary. The ML507 experiments used Xilinx tools version 12.2, but as these did not support the Virtex-6 FPGA, forcing an upgrade to version 13.1 for the ML605 system. The toolflow however stayed the same, as only minor modifications to the arguments and license options were required.

The compilation flow is shown in figure 3.18. This is largely a standard flow for the Xilinx toolchain, but with the options of generating both a simulation model and the timing closure information added. The simulation model includes timing information, and thus gives a good reference for simulating what is actually in the FPGA, as opposed to what the pre-compilation simulation may show. This simulation also

*Verilog Pre-Processor: a simple macro-expansion language mirroring the macro facility in C. Part of the Verilog standard, but implemented as a stand-alone step in our toolchain, with extra features.

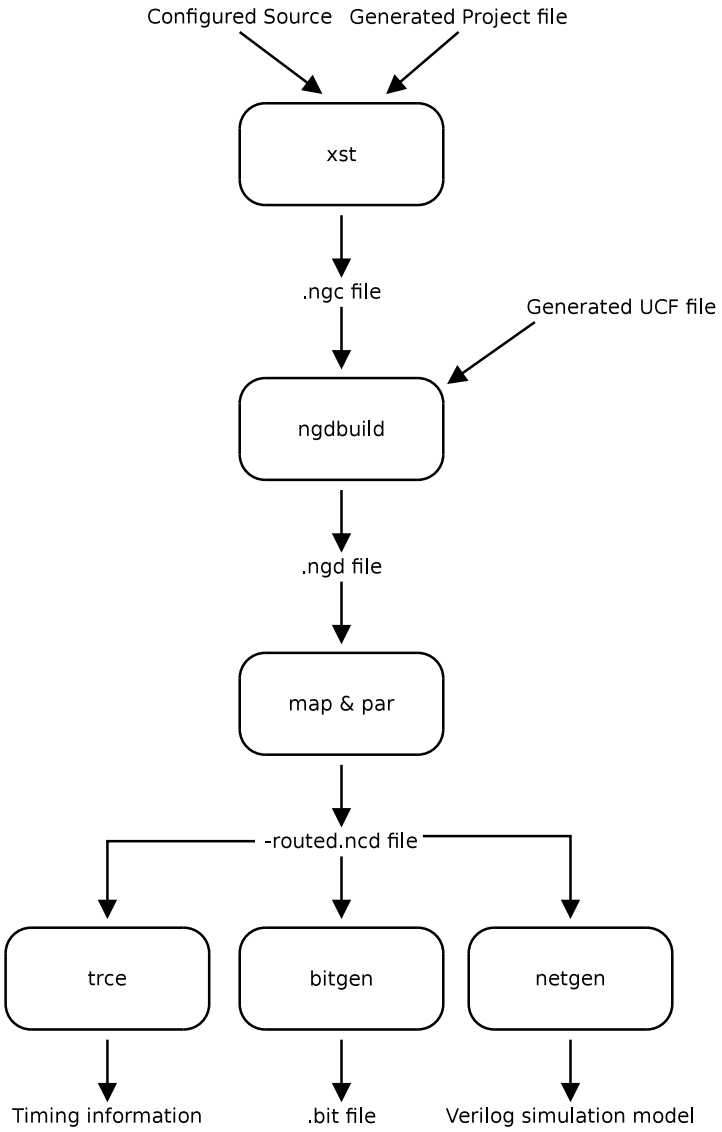


Figure 3.18: Compilation flow for hardware system. Input is a configured system, output is a bitfile for programming the appropriate FPGA.

includes timing delays and other imperfections, and was occasionally useful for debugging I/O problems. Most logic issues were better served with a faster and more easily interpreted behavioral simulation.

The timing information gives simple information about what paths are the longest, and if the design fails timing. This is largely used to tweak marginal designs, and as such was of limited use in the experiments which are pass or fail. The information in this file could be used to improve the synthesis results, but this process is not thought to be automatable.

3.4.3 Synthesis summary

This section has presented the configuration and compilation flows used to synthesise the experimental system described previously. The compilation flow is relatively standard, but the configuration flow is highly specialised to the system at hand. In particular, the configuration flow performs some complex logic connection and also includes the EnCore Verilog projects as subsidiaries.

3.5 Software Setup

The software for this experimental system consisted of two parts; an underlying utility library (libmetal) and the benchmarks and workloads necessary for the experiments. Libmetal was created specifically to allow us to run compiled code on 'the bare metal' of the processor, and essentially act as a program launcher and setup utility. The workloads themselves range in size but are constrained by the abilities from libmetal.

For compiling any benchmarks, as well as linking and assembling the results, the GCC toolchain version 4.2.1 were used, compiled for supporting the Arc architecture. As EnCore is ARC-700 compatible, this produces correct code without having to reimplement an assembler / linker / compiler.

Figure 3.19 shows the compilation flow necessary for libmetal and any applications linking to it. Of note is the separate compilation and linking steps, which are required for passing all necessary options and libraries to the linker. In addition, some applications were compiled first to assembler code and then to object code, for debugging purposes.

3.5.1 Libmetal

Libmetal started as a collection of intro routines and register setup code, written in assembler, that were required to make programs run comprehensively on the EnCore

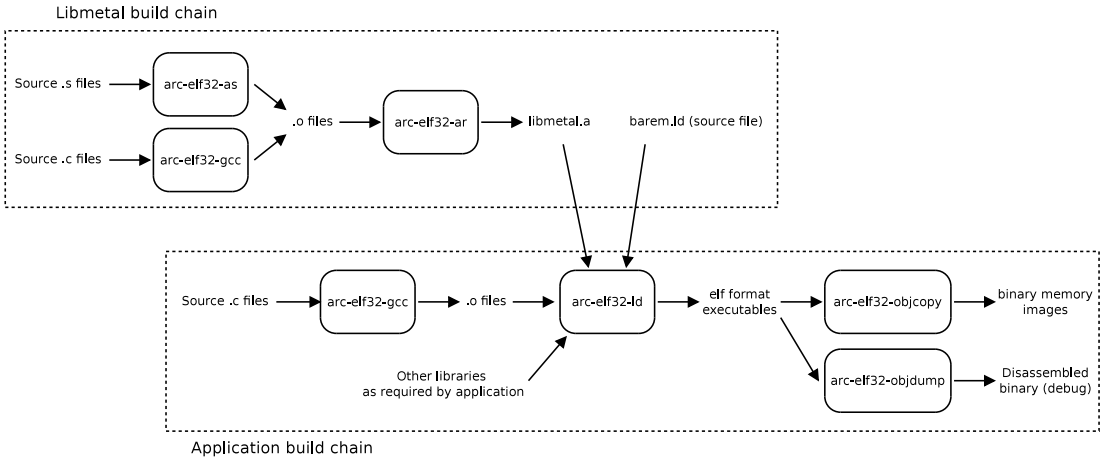


Figure 3.19: Compilation flow for libmetal and application code, targeted to the experimental system. Rounded boxes represent program invocations with the program name enclosed. Final output is a flat-memory binary image, ready to be uploaded.

processor. The processor requires certain structures (the interrupt vector table) in certain places in memory (address 0x00000000) and these files saw to that need. This included some linker scripts for ld that ensured these sections were placed at the right addresses.

In addition, for running code on the bare metal, it was not possible to use the elf-format files that were the default output from the toolchain. As the initial program has no loader, but is supposed to load itself, it was necessary to use objcopy as a pre-loader to generate a straight binary image, as it was to appear in the memory of the system. This image could then be loaded over to the experimental system, or included in a FPGA synthesis as a default memory image. In addition, because the complete memory image was needed, all binaries had to be compiled statically, as there is no mechanism for dynamic loading.

After a while, certain utility functions (such as number conversion, string formatting, and printf) were needed by the application programs. These functions would, in normal software, generally come through linking in a libc-equivalent library with the application. The easiest way to include this functionality in the case of the static bare-metal binaries was to include them directly in libmetal, and break the libc / operating system division present in nearly all computing systems. It is possible to do this because any code is executing as if it is in kernel mode, and there is no security to start with; in addition, it is necessary to do this because there is no operating system to run on top of.

Libmetal then became a hybrid of startup code and libc implementation, with some operating system functionality added. As stated in other sections, libmetal gained

some networking code to handle the serial I/O capability, and printing routines to output text to the screen. The capability to display images as character-mapped dithered entities was also included, described in Section A.1.

The major hurdle of libmetal was introducing multi-core support. The EnCore processor is not, as stated before, cache coherent even between its L1 caches, and the interconnect does not include coherency traffic. Thus the unusual situation of a multi-core system with no cache coherence occurs. To handle this situation, libmetal was coded in a way that relied on low-level cache-ignore instructions whenever critical shared data was handled. There were two problems with this, guaranteeing mutual exclusion would work before, as well as, allocating memory from a shared pool. These problems occur about 60 instructions into any multi-core program using libmetal, namely when assigning the stack pointers to the cores.

Multi-core support in libmetal works on the basis of a single memory location or auxiliary register that is unique to each core. This CPU ID forms the basis of differentiation between malloc areas, CPU state structures, and stack pointers. The libmetal startup code is executed by every core, but takes this auxiliary register into account, allowing each core to set up and boot itself into a useful state.

The 5-stage EnCore does not export the atomic operations available in the ARC-700 instruction set, and thus did not have native exclusion capabilities. To address this, an implementation of the Bakery Algorithm [48] was included that used a reserved region of memory for mutual exclusion. The main drawback of this implementation was that creating new locks was non-trivial, as they require reserved regions of memory in the binary image.

With the implementation of the 3-stage pipeline, the mutual exclusion problem found a better solution with the instruction set atomic instructions, but as before, these still required a separate non-cached area of memory. With the atomic instructions it is possible to pack a number of locks into a single cache line, so the problem is much less pronounced.

Malloc is implemented as a metadata tree attached to the operative area to be allocated. Essentially, the first few bytes of a memory area, together with the known length of the area, can be made to act as metadata bytes, indicating whether the following area is either free, allocated, or split into two sub-areas. The problem of allocating then become recursive, as does the free calls. On startup, fixed memory pools (of known size, given from a startup memory sizing routine) are initialised with the first few bytes of metadata, and left until subsequent malloc / free calls. These calls then use the size data together with the pointers to the pools to allocate data.

The libmetal malloc implementation is suitable for a single core implementation,

as the metadata is located in the same cacheable memory as the data itself. This does not work for a shared-pool allocator, globally locked or not, as the caching of metadata means it would not be available to other cores executing malloc, and thus would result in double allocation or non-reallocation of released areas. To solve this problem, each core was designated a malloc area by splitting the total pool into equal-size chunks and giving each core a single chunk that it can allocate from. Even so, the boundaries between chunks must be on cache line boundaries, as the same problem would otherwise reoccur.

Some thought has gone into alternate incoherency-aware many-core malloc implementations, but none have yet have been fully implemented in libmetal. The base problem of multi-core incoherent malloc, apart from making sure the metadata is never cached, is in the ownership and cacheability of an area of memory. If core n_1 have allocated an area a, do some work in the area, and then call free on a, there may well still be modified copies of the data in a in the cache of n_1, and, if the area has been used for moving data between cores, possibly also in the caches of other cores $n_2, n_3, \ldots$. This cached data may not write back until some time later, possibly after the memory area has been reallocated, generating essentially random memory corruption. To solve this problem a method of clearing any pending write-backs from all cores to a region of memory is needed. The hardware to accomplish this is so far only present in the debugging extensions to the 3-stage EnCore pipeline, but the hardware only manages the single-core case. It is likely that to solve this problem in the many-core case, data transport between incoherent cores should only be done through calls that are known to invalidate and write back caches appropriately.

Other extensions to libmetal that would be useful are in the areas of scheduling and process management. Currently libmetal has a simple static non-preemptive scheduling method, where tasks are divided to cores depending on how many cores are present and no other considerations. A better solution would be a global scheduler allocating tasks as cores free up, or even a preemptive scheduler. Adding this would transform libmetal from a system monitor type library into an operating system.

In summary, libmetal is a system monitor startup and execution platform with a lot of utility functions, functioning both as a basic hardware abstraction layer and limited libc in one. To enable programs, of any variety, to run on the experimental system, they need to be statically linked against libmetal using the linker scripts, which produces a suitable binary.

3.5.2 Workloads

Statically compiled workloads composed of up to 9 benchmarks chosen from a set of 7 were used. The benchmarks used to build the workloads were VITERBI, FFT, FBITAL, CONVEN and AUTCOR benchmarks from EEMBC-1, the COREMARK benchmark [30] and a specially composed I/O-heavy application.

CoreMark is an industry-standard embedded benchmark, provided by Embedded Microprocessor Benchmark Consortium. CoreMark is comprised of small and easy to understand ANSI C code with a realistic mixture of read/write operations, integer operations, and control operations. Unlike EEMBC's primary benchmark suites, CoreMark is not based on any real application, but the workload is actually comprised of several commonly used algorithms that include matrix manipulation (to allow for the use of MAC and common math operations), linked list manipulation (to exercise the common use of pointers), state machine operation (common use of data dependent branches), and Cyclic Redundancy Check (CRC is a very common function used in embedded systems) [30].

The EEMBC-1 suite of benchmarks, again provided by the Embedded Microprocessor Benchmark Consortium, are larger and more capable benchmarks, each targeted as a particular task. As such, they are larger and harder to get to work on a small embedded system such as the experimental system presented herein. Only a few of the EEMBC-1 benchmarks reliably complete correctly, and therefore those benchmarks were used for the experiments.

As the standard benchmarks are all focused on the processor core performance, a benchmark that used the I/O system was necessary, providing a different point in the application space. To fulfill this need an image display workload was created. This workload simply drew a picture to the display device using the libmetal display routines and display hardware discussed in the previous section and A.1, respectively. This application took an image and scrolled a window over it, updating the screen image with each pixel offset. The number of iterations were determined by the size of the display window, and was approximately set to one million pixels drawn, to account for differences in display window sizes between different experimental systems. This display benchmark can be clearly seen on cores 0, 1 and 4 in Figure 3.3, with the window showing a detail of the standard greyscale 'lenna' picture, used since 1973 in computer image and -graphics research.

Scheduling was handled through libmetal, and as such, each core got a number of benchmarks depending on the number of cores and order of the benchmarks. As there are many possible workloads that can be constructed from combinations of the chosen benchmarks, 35 unscheduled and 47 scheduled workloads were randomly selected

from the total set. It was expected that the scheduled workloads would provide a somewhat more even design space, so that machine learning would perform better, but little evidence for this was found in the results. As such, the results for the ordered and random order benchmarks have been aggregated, and the set is treated as having 72 workloads, consisting of up to 9 benchmarks each.

A limitation of the workload building and scheduling method is that it became possible for different runs of the same benchmark to interfere with each other, due to the shared global variables in that benchmark. Different benchmarks have different variables, so do not interfere, but multiple runs of the same benchmark has the possibility of self-interfering, depending on the particular scheduling and cache behaviours of the rest of the workload. This is a systematic issue with the build system, that can be avoided if each copy of a benchmark within a workload is given its own set of global variables.

3.5.3 Software summary

Software consisted of a special-purpose operating system replacement layer, named libmetal, and a number of workloads running top of it. These were compiled together into complete self-hosted binary images, suitable for upload and execution in the FPGA-based experimental system.

3.6 Machine learning flows

The final step in the setup was the machine learning to find the best experimental systems. For this it was needed to fairly evaluate the pros and cons of various machine-learning methods from the literature (See section 2.2) to find the best match to the application domain at hand.

The use of machine learning methods developed as the work progressed, and thus different methods were used in chapters 4, 5 and 6.

In chapter 4 a variant of *wk*NN is used, with weighing based on approximate distance to the closest n neighbours, using euclidean distances in the space. The predictor is used as a classifier, and is based on the `AI::Categorizer` Perl library. More details on this method is provided in chapter 4.

Chapter 5 uses *wk*NN, SVM and random, again in classification modes, to try and find the best design within an explored set of points of a larger search space. This chapter uses implementations from the R statistics software suite [57] to perform the classifications more efficiently than the previous chapter. More details on these methods can be found in chapter 5.

Chapter 6 again uses the R methods, and the same search space as chapter 5, but this time uses the regression mode of the predictors to find predicted values for new designs. The chapter evaluates a large amount of machine learning methods against this dataset to find the best method. The goal is to find a suitable predictor for predicting which is the best design in the full search space rather than in the limited, explored, dataset. More details on this methodology can be found in chapter 6.

In general, the machine learning method for all chapters followed the flow shown in figure 3.20.

This flow was used in all experiments, with varied parameters, i.e. the split ratio, the machine learning method, the prediction mode and the metrics used. In some cases, the steps from splitting the dataset to measuring accuracy was repeated several times, due to an uneven split, this was done to minimise the bias impact of a random split. The metrics reported are either straight prediction accuracy (with a standard deviation) or accuracy of a given metric, such as energy performance. Straight prediction accuracy is easily understood as it gives a true/false of how good the predictor is, and is reported as an average percentage, i.e. 72% of predictions were correct. Accuracy of a metric is less easily understood but gives a better measurement, as even if the prediction is slightly wrong, it may predict a design or outcome that is very close to the correct value. This metric captures and quantifies this behaviour and thus gives a truer picture of the actual performance of the method.

3.7 Setup summary

This chapter has described the setup of the experimental system and the flows, tools, and software chains for assembling and using this system. There are two major types of system presented, one based on the Virtex-5 with a 5-stage EnCore pipeline, and one based on the Virtex-6 using a 3-stage pipeline.

In essence, the experimental system is a source-customizable generic Verilog model of a butterfly-type interconnect, linking EnCore processors to I/O devices. These parts, and the system itself, have parameters such as the exact interconnect topology, the FIFO depths, the number pf processors, the number of memory channels, and others. These parameters form a design space over which the optimal system is to be found. The additionally presented machine learning methods helps to accomplish this, and the results of these experiments are presented in the following chapters.

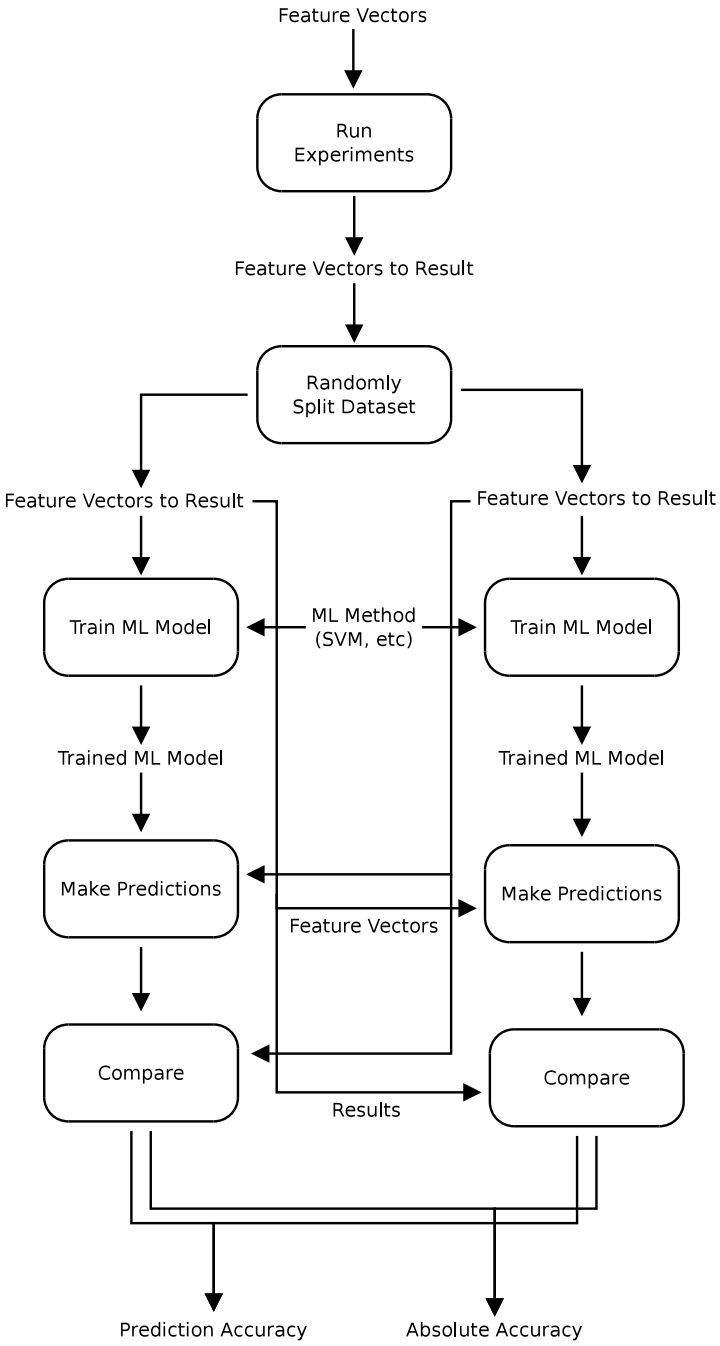


Figure 3.20: Generic Machine-learning flow used in all experiments. The split ratio varied, and the accuracy metric was chosen between straight accuracy and an optimality metric.

4 Predicting for a small design space

This chapter is based on a paper Co-written by Oscar Almer, Nigel Topham and Björn Franke [4]. The majority of the work was carried out by the author, but recognition goes to my co-authors for their assistance in preparing, executing, and writing for the contained experiments.

This chapter is organised as follows. Section 4.1 covers the experimental setup of the particular experiments in this chapter, and is an extension of Chapter 3. Section 4.2 covers the evaluation methodology specific to this chapter, including describing the specific variant of *wk*NN used. Section 4.3 presents the results of the experiments, and section 4.4 summarises and concludes this chapter.

4.1 Experimental system architecture

This chapter presents initial results of applying the machine learning algorithms, in this case *wk*NN, to the design space of SOC interconnect design. Accomplishing this involves performing experiments using a reconfigurable, parameterisable, interconnect design. The configurability of the network allows us to explore a range of designs from those with low-cost and limited bandwidth, to those with high bandwidth and hence higher cost. The networks are also characterised in terms of their dynamic energy, allowing us to optimise independently for performance and energy, as well as for combined energy-delay (ED) product.

The architecture of our experimental system consists of two processors, three blocks of memory, a JTAG interface, and a character-based graphical display device. More details about this setup can be found in Chapter 3. The processors and memories are connected by a statically-configurable NOC fabric, with network links implemented using the AXI interface. The experimental system is fully synthesisable and all results reported in this chapter were obtained from implementations running on FPGAs, from which real data was extracted through hardware instrumentation.

4.1.1 NoC architecture

Master devices in this SOC are either processor cores with an AXI translation layer or a JTAG programming port, used to load the benchmarks into the system. Our system

Parameter	Description	Range	Configurations
Width	Width of the AXI data bus	8 - 128 bits	5
Complexity	Complexity of the fabric; determines the topology	max, min	2
FIFO Depth	Depth of FIFO used in registered AXI switches	2, 4, 8, or 16 lines	4
Clock Multiplier	Ratio of AXI clock to CPU clock	0.1, 0.2, 0.5, 1, 2, 5, 10	7
		Total	280

Table 4.1: Parameters defining configurability of the interconnect.

utilises two processors, each running an independent task from a selected workload. As these tasks do not share data, memory coherency is not a concern.

Table 4.1 shows the four parameters through which the NOC interconnect can be configured in our experimental architecture. These are AXI bus width, the complexity of inter-connectivity, the depth of FIFO buffers in each switch node, and the clock ratio between AXI channels and CPU.

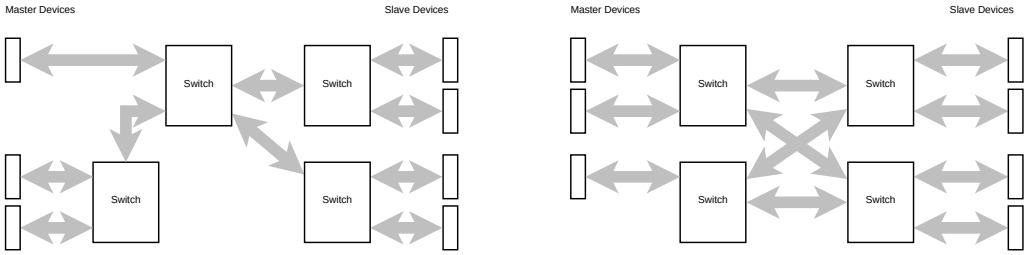
The switches of the interconnect are conceptually 2x2 crossbars with registered outputs, arranged into a optimised tree structure, each of which is responsible for five uni-directional AXI channels connecting a master to a slave device. These channels are routed independently, allowing each AXI channel to be transported independently through the switch.

Each channel uses independent two-way handshake for transferring data, allowing the use of FIFO buffers for both master and slave interfaces; the *depth* of these buffers is one of the NOC configuration parameters.

The number of AXI packets sent on each channel depends on the channel width and the total traffic volume generated by each master. Channel width is therefore another parameter of the network, and can be varied in powers of two from 8 to 128 bits. The AXI address and control signals remain unchanged by this parameter, and present a fixed overhead regardless of channel width.

As the switches are buffered, the clock ratio between the core and the network can be varied, and therefore a *clock multiplier* configuration parameter is included. In our experimental system the CPU clock was artificially slowed down to 16.6 MHz to enable a large range of clock multipliers while still being implementable in an FPGA fabric.

The *complexity* parameter determines the throughput of the central switching fab-



(a) Single-tree complexity (b) Minimum-width complexity

Figure 4.1: Illustration of the *complexity* parameter.

ric. Our NOC topology takes the form of a tree structure [50, 51], with optionally increasing bandwidth near the root. In this study two topologies are used, as shown in Figures 4.1(a) and 4.1(b), each with three master and four slave devices. In the simplest network, shown in Figure 4.1(a), the minimum number of AXI links is used to connect the devices in a tree-structured topology. In the more complex network, shown in Figure 4.1(b), more links are provided between the internal nodes of the network. This provides a greater bisection bandwidth, but at greater cost and higher energy consumption. The precise topology is not critical to this study, provided each configuration presents a different trade-off between cost, performance and energy consumption.

Our experimental hardware system is implemented as parameterisable Verilog, synthesised to a Xilinx Virtex-5 XC5VFX70T FPGA device. This is one of the smaller devices in that family, allowing at most two processors to coexist in each configuration. The Xilinx ISE synthesis tools were used to synthesise a total of 280 configurations defined by enumerating all combinations of design parameters. Configuring and synthesising a single design point takes approximately two hours on a Intel Core2 processor, but accurate simulation of the benchmark traces would take several days, as execution times for the benchmarks were in excess of 30 seconds of real time. Clearly, using gate-level software simulation is not a viable approach to assessing optimal conditions, whereas an FPGA implementation can achieve the same result in a fraction of the time.

4.2 Methodology

This section firstly describes the machine learning model developed to predict the best NOC architectures. This is followed with an explanation of our experimental methods and benchmark applications.

Measurement	Explanation
$e1$	Total number of IOPS/s performed by CPU 0
$e2$	Total number of IOPS/s performed by CPU 1
br	Total number of reads/s when any core is active
bw	Total number of writes/s when any core is active

Table 4.2: Measurements used by the machine learning algorithm, derived from the hardware counters of the experimental system. IOPS refer to I/O Operations, *i.e.* a single memory read or write, including operations managing memory mapped I/O devices.

4.2.1 Extracting performance data

Hardware counters were inserted into the AXI fabric and processor cores so that the performance of the system could be evaluated. These counters were attached to a separate scan chain, accessible through the JTAG interface, allowing transparent data collection that would not itself perturb the system under test.

Each switch collects data about each AXI channel independently, which is then aggregated to form overall switching data for the network. This switching information is independent of the topology, and depends only on the task assignments and task behaviour. Each processor reports the number of clock cycles it has been running, the number of cycles during which it is inactive while waiting for network data, and the number of completed I/O transactions, as summarised in Table 4.2. This data is used to estimate energy consumption based on switching activity, and to measure overall performance, in terms of instructions/cycle for each processor. The energy model used in these experiments is presented in Section 3.3.3.

All possible combinations of design parameters were explored and these designs synthesised for a FPGA fabric, enabling us to run entire benchmark programs without the performance penalty incurred in software simulation. Using the FPGA implementation of each design, it was possible to measure the relative performance, design complexity, and energy efficiency, of each design configuration.

4.2.2 Machine learning algorithm

The machine learning algorithm used is based on the conjecture that if a new workload has similar characteristics to a previously seen workload, then the known optimum configuration of the NOC for that workload is also likely to be near-optimal for the new workload. The workload is characterised in terms of three measurable quantities; the number of network transactions issued by each processor, the total Read

bandwidth, and the total Write bandwidth. In our limited set of configurations there are just two processors, resulting in a total of four measured features; an I/O rate for each CPU, plus the total Read and Write bandwidths. Together these characterise the network activity of the workload for a specific assignment of workload task sets to each processor. A discussion how to handle the possibility of alternative assignments of those task sets is discussed in 4.2.2.1.

The method developed here is a variation of wkNN, which is discussed in section 2.2.2.

This optimisation problem operates over fixed set of training workloads $W = \{w_1, w_2, \ldots, w_m\}$ and a fixed set of configurations $S = \{s_1, s_2, \ldots, s_n\}$. For each NOC configuration $s \in S$ each workload $w \in W$ is executed on the FPGA. From this measurements the I/O rates for the two processors, $e1$ and $e2$, and also the total read-/write bandwidths, br and bw is obtained. For each workload w, these measurements are combined with the test configuration s to define a point $\mathbf{x}$ in the Euclidean space $E_w \in \mathbb{E}^5$.

$$\mathbf{x} = (x_0, x_1, x_2, x_3, x_4) = (s, e1, e2, br, bw) \tag{4.1}$$

Denote X_w to be the set of all experimental points $\mathbf{x}$ obtained for workload w. The training space therefore contains the set of sets $T = \{X_w : w \in W\}$. As training progresses, each point $\mathbf{x}$, derived from a single experiment, is annotated in the training space with the pair (r, p), representing the overall execution rate and power consumption of workload w running on configuration s.

When the experimental data has been collected, the prediction model is constructed by performing a linear search of all points $\mathbf{x} \in X_w$, for all workloads $w \in W$. This searches independently for the optimum configuration for each workload w, with respect to speed and power. For each w the sr_w and sp_w are found, each representing respectively the configuration with the optimum annotated value of r and p experienced by w for any $s \in S$ during training. This requires at most $\mid W \mid \times \mid S \mid$ search steps, each involving simple table look-ups and comparisons.

The tuple associated with each point $\mathbf{x} \in X_w$ is then augmented with sr_w, and sp_w, creating a mapping from each pair (s, w) to the optimal configuration for workload w, with respect to each of the metrics of interest. The training phase of the learning algorithm can therefore be viewed as a process of populating a Euclidean space with points representing the execution rate and power consumption for each training workload on each configuration. These points are then post-processed to link each point to the optimum configuration for the workload associated with that point.

When a new and previously unseen workload w' is encountered, it will be run once on some arbitrary configuration $s' \in S$, in order to characterise its NOC requirements.

The new workload characterisation yields point $\mathbf{y} \in \mathbb{E}^5$, thus:

$$\mathbf{y} = (y_0, y_1, y_2, y_3, y_4) = (s', e1, e2, br, bw) \tag{4.2}$$

Our goal is now to find point $\mathbf{x}'$, within T, that has the maximum similarity in its NOC usage characteristics, $(e1, e2, br, bw)$.

Assuming that each characteristic is equally important, then this is equivalent to finding a *predictor* $\mathbf{x}'$ that is the minimum Euclidean distance from point $\mathbf{y}$, irrespective of configuration and training workload. To measure Euclidean distance irrespective of configuration and workload, consider all points in T and ignore dimension 0, which represents the test configuration. The Euclidean distance between point $\mathbf{x}$ and $\mathbf{y}$, irrespective of configuration and workload, is given by $d(\mathbf{x}, \mathbf{y})$:

$$d(\mathbf{x}, \mathbf{y}) = \left(\sum_{i=1}^{4} (x_i - y_i)^2 \right)^{\frac{1}{2}} \tag{4.3}$$

Hence, $\mathbf{x}'$ is found thus:

$$\mathbf{x}' = \min_{w \in W} \left(\min_{\mathbf{x}_i \in X_w} d(\mathbf{x}_i, \mathbf{y}) \right) \tag{4.4}$$

From point $\mathbf{x}'$ it is possible to directly obtain the associated sr_w, and sp_w, and use these as the predicted optimum configurations for workload w', when optimising respectively for speed and power. The predictor $\mathbf{x}'$ will always be found in a training workload w'' that is different from the new workload w', as w' is never included in the training set. By finding a workload w'' that has the greatest similarity to w' in terms of its NOC characteristics, it is conjectured that the optimum configuration for w'' will also be near-optimal for w'. If this conjecture is valid, then $\mathbf{x}'$ will be a good candidate configuration for workload w'.

In general, the choice of test configuration s' will have some affect on the accuracy of the resulting prediction. When workload w' is run on configurations s_1 and s_2 it will produce distinct result vectors $\mathbf{y}_1$ and $\mathbf{y}_2$. The optimum configurations predicted by the nearest points to $\mathbf{y}_1$ and $\mathbf{y}_2$ need not be identical, and in many cases will not be. However, if the model is robust then the performance differential between the two predictions will be small.

4.2.2.1 Modelling the effect of task mapping

In order to explore the impact of different assignments of the tasks within a given workload to the available processors, the prediction phase considers all possible assignments. In our limited workload, mapped to just two processors, it is feasible to consider all such assignments.

The characterisation of a new and unseen workload is given by $\mathbf{y}$, where:

$$\mathbf{y} = (y_0, y_1, y_2, y_3, y_4) = (s', e1, e2, br, bw) \tag{4.5}$$

In the simplest case of two processors, the affect of swapping the static assignment of tasks to each processor can be modelled by exchanging $e0$ and $e1$ in $\mathbf{y}$. In the more general case of k processors, and k task sets to assign statically to those processors, there may be as many as $k!$ permutations of task assignments. For moderate sizes of system, *e.g.* $k \leq 8$, an exhaustive search of all permutations is feasible, as the cost of each search is relatively small. However, for systems with significantly more than 8 processors a different approach would be required. This is an area for future work.

This approach uses Euclidean distance to predict the best configuration for the new workload, based on previously seen workloads on all possible NOC configurations, irrespective of the process to processor mapping. For homogeneous multicore systems this involves repeated searches for the nearest point $\mathbf{x}$ to any $\mathbf{y}_j \in Y$, where Y is the set of points obtained from all permutations of $(e1, \ldots, ek)$. For heterogeneous systems, the number of possible mappings of tasks to processors will be much reduced by the task assignment constraints.

When all permissible permutations of workload assignment have been examined, the closest result will indicate both the optimum NOC configuration and the optimum task assignment for the new workload.

4.2.3 Benchmarks

The workloads used in this study comprised six applications: four EEMBC V1.0 benchmarks (*autocor00*, *fbital00*, *fft00* and *conven00*), the EEMBC *CoreMark* benchmark [30], and an *ImageDisplay* benchmark generated by ourselves for this study. Each workload consists of two applications from these six, one for each of the two CPUs, and for this experiment five workloads were analysed. Several compute-intensive benchmarks and an I/O intensive benchmark were used to achieve contrast between the workloads.

All benchmarks other than *ImageDisplay* are largely compute-bound, and suffer few cache misses. Hence a I/O bound benchmark was developed which animates visual images stored in the memory mapped display driver. This display is located in non-cacheable memory, accessed through non-cacheable Load and Store instructions. *ImageDisplay* uses data sets much larger than the CPUs 8KB 2-way set-associative level-1 data cache, resulting in significant NOC traffic. This is quite unlike the network traffic generated by the CPU intensive applications.

Table 4.3 gives a list of the benchmark combinations used.

	w_0 (CM)	w_1 (CO)	w_2 (IM)	w_3 (EE1)	w_4 (EE2)
Task 1	CoreMark	CoreMark	ImageDisplay	autocor00	fbital00
Task 2	CoreMark	ImageDisplay	ImageDisplay	fft00	conven00

Table 4.3: Composition of applications within the five workloads used in the evaluation.

The tasks comprising each workload run in parallel, one on each core of the system, and therefore compete through the interconnect for access to memory. Each experimental run starts from a clean hardware reset, and all cycle counts include the time to start-up and shutdown the application. However, this accounts for less than 1% of the execution time for any of the benchmarks presented. There are no interrupts or task switches in the inner loops of the benchmark. For the CoreMark benchmark the score (iterations / MHz) was computed and recorded for each core of each design in addition to all other data recorded.

4.2.3.1 Selection of training data

The empirical data from four workloads was used to train our machine learning algorithm, allowing it to then predict the most appropriate combination of design parameters for the remaining other benchmark workload. This was repeated five times, training each time on a different combination of workloads in order to predict the NOC configuration for each workload using distinct training and test data. Therefore, when predicting the network configuration for workload w_i the model is trained on all workloads $w_j \in W, j \neq i$.

Our goal was to determine whether the predictions from this model would match the empirically determined best configurations for those benchmark combinations when given only one data point. Although training process is time-consuming, if the resulting predictions are accurate then the model-driven search procedure will be extremely efficient in finding good NOC designs.

4.3 Results

Tables 4.4, 4.5 and 4.6 show the results of predicting the optimal configuration for each workload, with respect to dynamic energy, run time, and energy-delay product respectively. The final column reports the average performance across all experiments. The first row indicates the optimal NOC configuration for each workload. All optimal configurations are reported, and in some cases there are two or three with optimal

	Predicted Workload					
	CM	CO	IM	EE1	EE2	Average
Optimal Design	s_{253}, s_{254}	s_{254}	s_{140}, s_{142}, s_{143}	s_{196}, s_{198}, s_{199}	s_{140}, s_{142}, s_{143}	
Predicted Designs and prediction ratio	s_{254} (20.5%), s_{198} (79.5%)	s_{142} (4.6%), s_{253} (90.7%), s_{198} (4.6%)	s_{254}	s_{142}	s_{198}	
Performance of predicted config as absolute %age	100%, 106.8%	115.6%, 100%, 104.3%	164.1%	103.5%	100.1%	114.43%
Performance of predicted config as %age of optimal	100%, 99.48%	99.20%, 100%, 99.77%	74.47%	99.88%	99.99%	94.75%
%age of designs worse than prediction	99.23%, 98.07%	96.53%, 99.23%, 98.07%	60.62%	94.59%	97.30%	89.95%

Table 4.4: Summary of the prediction results for **Dynamic Energy**.

	Predicted Workload					
	CM	CO	IM	EE1	EE2	Average
Optimal design	s_{261}, s_{262}	s_{261}, s_{262}	s_{261}, s_{262}	s_{261}, s_{262}	s_{148}, s_{149}, s_{150}, s_{151}	
Predicted designs	s_{261}	s_{261}	s_{261}	s_{149}	s_{261}	
Performance of predicted config as absolute %age	100%	100%	100%	100.2%	100%	100%
Performance of predicted design as %age of optimal	100%	100%	100%	99.43%	99.98%	99.9%
%age of designs worse than prediction	99.23%	99.23%	99.23%	91.12%	97.68%	97.29%

Table 4.5: Summary of the prediction results for **Execution Time**.

	Predicted Workload					
	CM	CO	IM	EE1	EE2	Average
Optimal design	s_{253}, s_{254}	s_{254},	s_{156} - s_{159}	s_{196}, s_{198}, s_{199}	s_{140}, s_{142}, s_{143}	
Predicted designs	s_{254} (20.5%), s_{198} (79.5%)	s_{158} (4.6%), s_{254} (90.7%), s_{198} (4.6%)	s_{254}	s_{142}	s_{198}	
Performance of predicted config as absolute %age	100%, 109%	388.4%, 100%, 106.5%	580.3%	107.1%	100.1%	213.6%
Performance of predicted design as %age of optimal	100%, 99.23%	83.68%, 100%, 99.63%	61.32%	99.73%	99.99%	91.9%
%age of designs worse than prediction	99.23%, 97.29%	42.08%, 99.61%, 98.07%	11.52%	97.30%	97.30%	80.1%

Table 4.6: Summary of the prediction results for **Energy-Delay Product**.

Design	Complexity	FIFO	FSB Multiplier	AXI Width
s_{262}	min	4	10	128
s_{261}	min	2	10	128
s_{254}	min	4	0.1	128
s_{253}	min	2	0.1	128
s_{199}	min	8	0.1	64
s_{198}	min	4	0.1	64
s_{196}	min	2	0.1	64
s_{159}	min	8	2	32
s_{158}	min	4	2	32
s_{157}	min	2	2	32
s_{156}	min	16	2	32
s_{151}	min	8	10	32
s_{150}	min	4	10	32
s_{149}	min	2	10	32
s_{148}	min	16	10	32
s_{143}	min	8	0.1	32
s_{142}	min	4	0.1	32
s_{140}	min	16	0.1	32

Table 4.7: Parameters of designs appearing in Tables 4.4, 4.5 and 4.6.

performance. The second row indicates the configurations predicted by the machine learning model. The test workload was run on all reference configurations to see how sensitive the model was to the reference configuration chosen for the prediction run.

The optimal configurations from all of Tables 4.4, 4.5 and 4.6, and their configuration parameters, are shown in Table 4.7. It can clearly be seen that while the execution time metrics clearly favour some variations on fast and wide interconnect (s_{261} and s_{262}), dynamic energy instead favours slightly different designs with, in general, much lower clock ratios. In addition, the combined energy-delay metric favours mainly the same designs as in the dynamic energy case, except for the I/O heavy IM benchmark, where a fsb multiplier of 2 and a smaller width was the best choice.

It was found that the predicted configurations are, in some cases, dependent on the reference configuration, but this does this affect the predicted performance significantly. The appearance of this feature is an indication that the machine learning algorithm is functioning correctly. All predictions are listed in the table, together with the percentage of reference configurations that led to each prediction. The third row indicates the performance of the predicted configuration as a percentage of the optimum configuration. Not surprisingly, it was found that the optimal configuration is different for each workload, and for each optimization goal. By applying our machine learning algorithm to the design space, and using any four of the workload combinations it is almost always possible to predict the best possible design point for the fifth benchmark. In most cases, the difference between the predicted and optimal values is small, indicating that our machine learning algorithm manages to predict a near-optimal NOC configuration with a high certainty. For all workloads except IM the predicted configuration is either the same as the optimum configuration or performs almost as well. The average performance of predicted configurations is within 9% of the performance achieved by the optimum configuration for energy-delay product, significantly better for run time and energy when considered separately. The fourth row indicates the closeness of the predicted configuration's performance to the optimum configuration, expressed as a percentage of the difference between the best and worst configurations. For the IM workload, when predicting for optimal EDP, it can be seen that the predicted configuration is more than 60% of the way from the worst configuration to the best configuration. This is also seen in the graph presented in Figure 4.3.(c). The IM workload is the least predictable of all, and this is due to the widely differing characteristics of the *ImageDisplay* application compared with the compute-bound benchmarks. This suggests that a certain minimum degree of similarity must exist between the training workloads and the predicted workload in order for the learned model to work accurately.

Overall, a single run of a new workload on a randomly chosen reference design is normally enough to arrive within a few percent of the result given by any other reference design, showing that our method is agnostic to the choice of reference configuration.

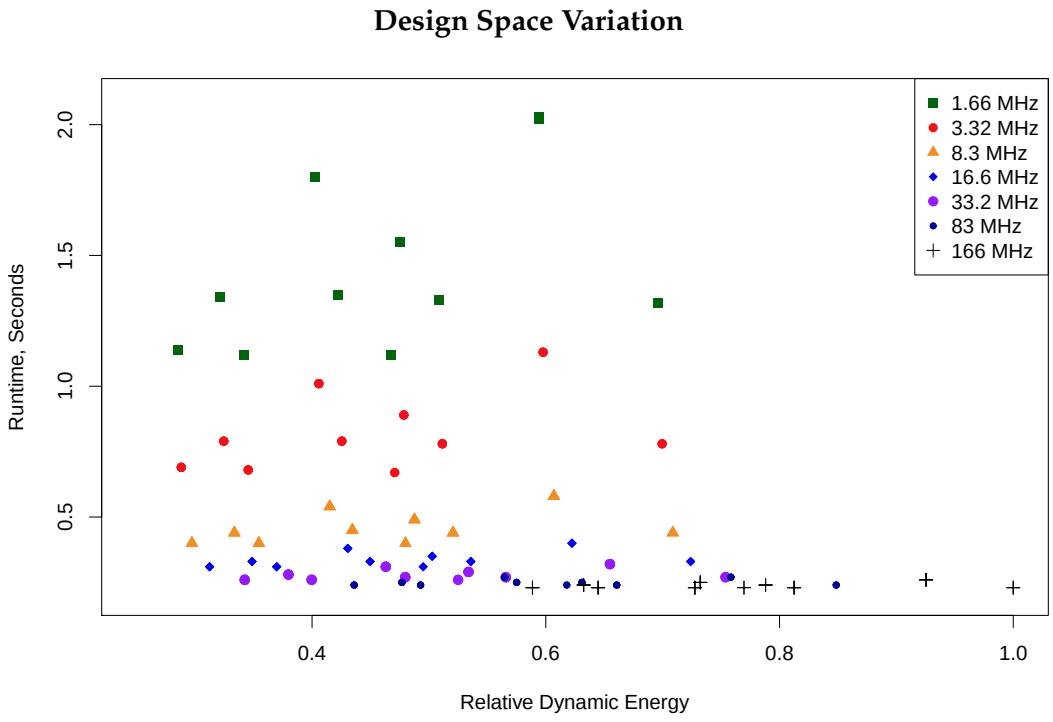
It is also interesting to examine the variation that exists within the design space for different workloads. This is shown for the *ImageDisplay* and *CoreMark* applications in Figures 4.2(a) and 4.2(b). These are scattergraphs showing the relationship between dynamic energy on the horizontal axis with run time on the vertical axis. Configurations are colored according to the configured network clock period. These results show a 1313% variation in dynamic energy for *CoreMark*, and the time taken for the inner loop of *CoreMark* varied by 75%. Conversely, the dynamic energy for *ImageDisplay* varied by only 251%, significantly lower than the variation in *CoreMark*, due to its greater reliance on interconnect speed.

In general, despite the size of the design space explored in this chapter, there remains significant variation in the performance metrics of interest to designers, and the space is quite complex.

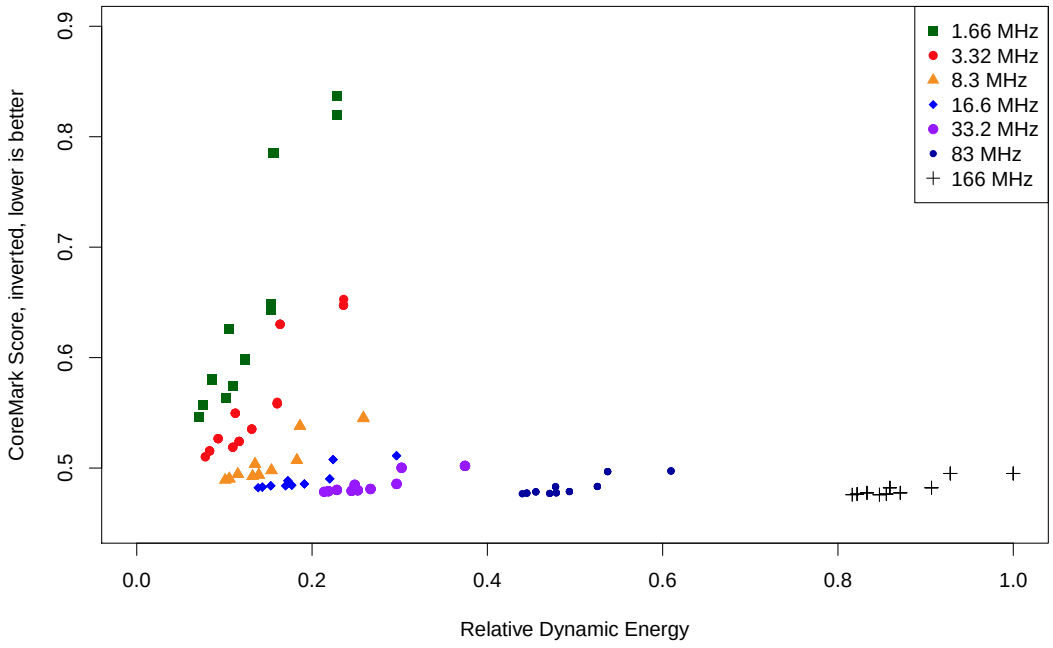
A single run of a new workload on an existing reference design takes up to a minute of real time, whereas exhaustively finding the optimal design would take 280 times longer, assuming that all designs are retained after being synthesized. The nearest neighbour algorithm takes at most a few seconds to perform the prediction, resulting in a speedup of 280 times over exhaustive search.

Figure 4.3 shows the predictions as vertical lines superimposed on the performance distributions of the workloads. These figures are a graphical presentation of the data in Tables 4.4, 4.5 and 4.6. It is very clear from the differing performance distributions that the workloads presented are varied, with CM and CO being relatively similar due to both containing *CoreMark*, which dominates the measurements. The IM workload is quite clearly differentiated from the others due to its I/O bound nature where all others are CPU bound.

It is found that predicting the IM workload is harder than predicting any other workload, especially when optimising for EDP. This is entirely due to the training data not covering the entire search space. To predict this type of application, workloads would need to contain a range of other I/O bound applications. There is no other perceived problem with predicting for this type of application as well as for CPU-bound applications, given the appropriate training data.

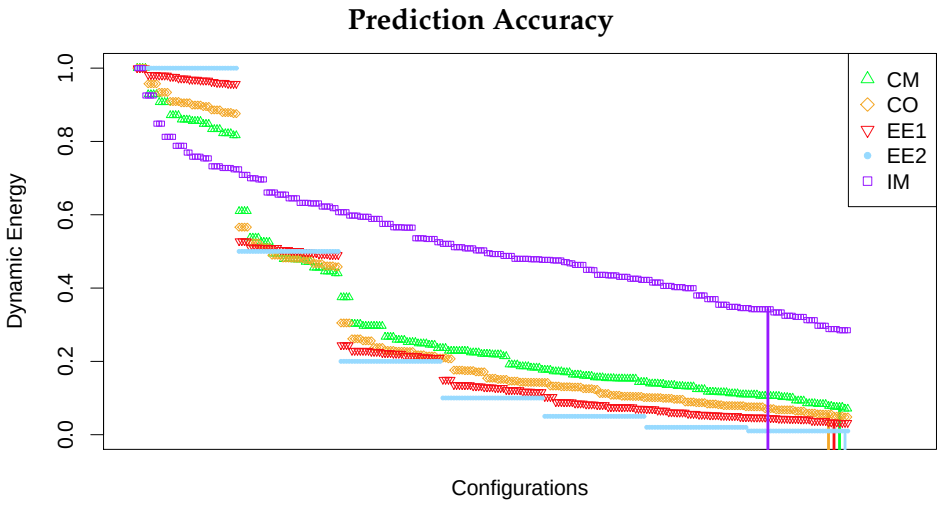


(a) Image Display

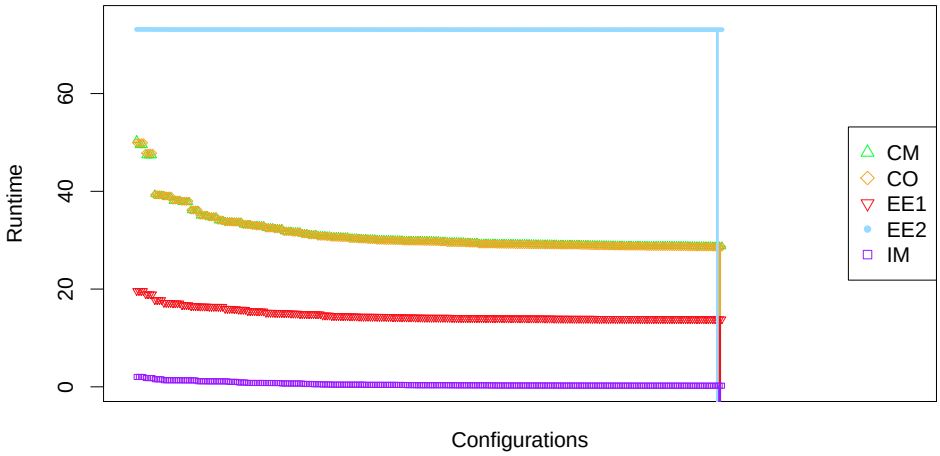


(b) CoreMark

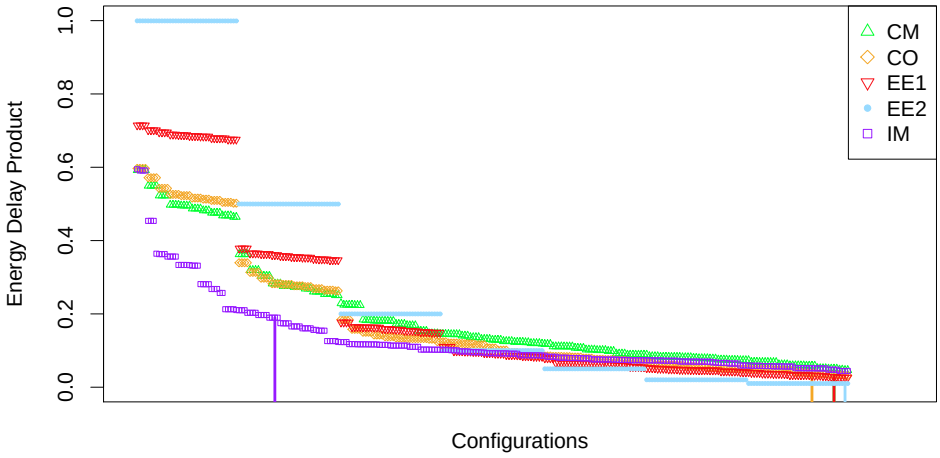
Figure 4.2: Dynamic energy vs Runtime, showing interconnect frequency distribution. Lower is better on both scales; the large range of system performance created by the varying design parameters is clearly visible. Also note the difference in design parameter impact on the I/O heavy workload (Image display) and the compute-bound workload (CoreMark).



(a) Dynamic Energy



(b) Runtime



(c) Energy-Delay Product

Figure 4.3: Graphs showing the distribution of results over design points, with predicted points marked with vertical lines. It is clear that predicting the IM workload from the CPU intensive workloads is less accurate, especially when optimising for EDP. Predicting the CPU intensive workloads is highly accurate.

4.4 Summary and conclusions

This chapter has presented the results of a design-space exploration of the interconnect space of a compact, synthesised, SOC system. Despite the small size of the experimental system the NOC design trade-offs are significant and non-intuitive. The results show that the interconnect architecture design space is large and complex, and shows wide variations in performance and energy efficiency. The results support the view that the optimal design point of an NOC is dependent on both the design parameters and the application workload. It is shown, for example, that different applications have different optimal NOC configurations.

Furthermore, this chapter has demonstrated that modelling the characteristics of the NOC design space is tractable using a machine learning algorithm. The results show that when a machine learning model is trained with workloads that are similar to a new and as-yet-unseen workload, then the model can predict a near-optimal network configuration for the new workload. For such workloads the predicted configuration are, on average, within 0.1%, 6% and 9% of the optimum, for performance, energy and energy-delay product respectively.

By using a modest sized set of workloads this chapter has been able to highlight the shortcomings of attempting to predict an optimum network configuration for a workload with widely differing characteristics from the training set. Future work will involve using more benchmark combinations to establish a larger set of application behaviours, as well as using more configuration options for the NOC and SOC system as a whole, increasing the search space dramatically.

The goal of this chapter was to determine the extent to which NOC configurations could be predicted by a model trained using data from previously seen benchmark workloads profiled across a broad spectrum of reference configurations. The results indicate this approach has potential, although further work is needed to extend this to a wider scope. Such work has been carried out in chapter 5 where a larger design space is explored.

5 Predictions in a large-scale system

This chapter is based on a paper authored by Oscar Almer, Miles Gould, Nigel Topham and Björn Franke, presented at NocArc '11 [5]. This chapter discusses the generation of a significantly larger design space than Chapter 4, and also presents the results of several experiments in applying machine learning to the explored parts of this design space.

The experiments in Chapter 4 showed that machine learning was a usable and efficient technique on a limited embedded NOC design space. Due to the limitations of the system in 5, it was desired to find if the method would scale to a larger system and a larger design space. In addition, it was desired to address some inefficiencies in the previous experimental setup, such as the EnCore to interconnect connection.

To address these points an amount of extra circuit and system design was undertaken to produce a larger design space with more cores. This chapter details the generation of this design space, as well as generation of a larger set of test applications. These components are used to derive a large set of runtime and energy data taken from this design space.

Experiments were carried out to find if it was still possible to efficiently apply machine learning to this design space using this data, as well as finding what level of performance could be expected from this machine learning technique. In addition, support vector machines (SVM) were introduced as a new machine learning method to evaluate its performance alongside that of *wk*NN.

This chapter has the following structure. Sections 5.1 covers the specific setup used to generate the design space data. Section 5.2 covers the actual generation of the said data. Section 5.3 presents and discuses the results of experiments using this data in detail. Section 5.4 summarizes the chapter.

5.1 Design Space

This section covers how the design space was constructed, largely by contrasting with the previous design space. The details of the system parameters and size are important for understanding the further experiments.

Much of the detailed setup of the hardware system is covered in chapter 3, and will not be repeated here.

5.1.1 SOC design parameters

The design space in chapter 4 was found to be too limiting, and importantly, it had been exhaustively enumerated and examined. It was believed that while the limited system that generated that design space was valuable and real, a larger system and design space was necessary to properly test whether machine learning can be applied to specializing NOC designs.

To generate such a design space, a larger FPGA device is necessary; the size restrictions in Chapter 4 were largely due to the amount of reconfigurable logic available in the XC5VLX70T device. For generating this larger design space, a larger XC6VFX240T device was used. It was empirically found during the development process that this FPGA can house up to 12 3-stage EnCore processors; in addition, it has a significantly larger amount of block RAM to use for main memory.

These constraints informed the meta-system used for generating the design space. In addition, engineering changes in the EnCore to AXI interface, as detailed in Section 3.2.4.6, enabled the use of several cores per AXI master port. These changes were only applied to the 3-stage EnCore pipeline, which was to be used for this meta-system due to it being further along in development at that point.

A part of the design space size is the number of cores. The enablement of more than one core per AXI master exacerbated this, as a design with the same number of cores arranged in different clustering configurations cannot be considered equivalent. The design space was specified to have between 1 and 16 cores, with up to 4 clusters of up to 4 cores per cluster. It can easily be seen that the amount of different configurations possible from this is already large.

In addition, the AXI clock and core clocks are distinct, and an arrangement where each could be independently set was adopted; due to the nature of the FPGA clock generator arrangements they could be set independently in increments of 25 MHz. The core clock was set a maximum of 125 MHz and the AXI clock could maximally be set to 150 MHz, as it was known from development that the AXI could be clocked higher than the cores.

Regarding the AXI interconnect itself, the depth of the switch buffers was set to either 2 or 16 entries, as it was known from chapter 4 that there was no major difference between a depth 2, 4 or 8, but that a depth of 16 caused synthesis into a BRAM block instead of flip-flops. An option for unregistered fabric was also introduced, which effectively has a buffer of depth 0. The AXI width was not varied in this design space, as the locking memory controller required an AXI width of 32 to work correctly, and this was also the default width of the EnCore interface.

The number of memory blocks was varied between 1 and 8, but the total amount

of system RAM was always 512 kBytes, meaning that as the number of controllers went up, the amount of RAM controlled by each went down. The interconnect generation program handled the details connecting this variable number of RAM blocks, as well as the varying number of clusters, to the specified switches. The last and possibly most important parameter to this design space is the topology of the interconnect, which was given as 1, 2, or 3, indicating switch networks of height 2, 4 and 8 respectively. For an explanation of this parameter, see Section 3.2.3.

Parameter	Value Range	Configurations
Cores in Cluster 0	1-4	4
Cores in Cluster 1	0-4	5
Cores in Cluster 2	0-4	5
Cores in Cluster 3	0-4	5
Core clock	(1 to 4)× 25 MHz	4
Switch type	registered, unregistered	2
AXI clock using registered switches	(1 to 6)×25 MHz	6
AXI clock using unregistered switches	(1 to 5)×25 MHz	5
FIFO Depth using registered switches	2, 16	2
FIFO Depth using unregistered switches	0	1
Topology s parameter	1, 2, 3	3
RAM blocks	1, 2, 4, 8	4
		510,000

Table 5.1: Summary of system parameters

The parameters of this experimental system are listed in table 5.1. In total, the parameters of this design space combine to 510,000 possible designs. As the design space is over-specified in some ways, there are designs in this space that cannot be synthesised correctly to a XC6VFX240T device due to the device limitations. This was intentional, and aimed at applying machine learning to find synthesisable designs in addition to experiments on synthesised designs.

5.1.2 Application programs

The applications that were run on the generated designs were statically compiled workloads composed of up to 7 benchmarks. The benchmarks used to build the workloads were VITERBI, FFT, FBITAL, CONVEN and AUTCOR benchmarks from EEMBC-1, the COREMARK benchmark[30] and a specially composed I/O-heavy application. The

Benchmark	FBITAL	COREMARK	CONVENC	VITERBI	FFT	AUTCOR	I/O
Instructions	930m	690m	410m	547m	53m	3.5m	180m
Cache Bypass	23k	23k	23k	23k	23k	23k	10m
I$ misses	610	2m	600k	264	570	557	13k
D$ misses	20k	540	150	140k	30k	66	1.4m
I$ hit ratio	100%	99.7%	100%	100%	100%	99.9%	99.9%
D$ hit ratio	99.9%	100%	100%	99.9%	99.8%	99.9%	96%
Total operations	44k	2m	620k	163k	53k	24k	11.5m

Table 5.2: Approximate cache hit rates and non-cached operations for workloads obtained through simulation. Libmetal and startup overheads are included, but task-switching induced misses are not. Cache bypass refer to cache-ignore instructions which are not tallied in the data cache hit ratio. Total operations includes these instructions.

benchmark properties are shown in Table 5.2, which lists the number of instructions and cache hit rates generated when running each benchmark in isolation. The combined workloads generated from these benchmarks were compiled together with libmetal, discussed in section 3.5 that handles the required system calls and functions to enable correct functionality of our benchmarks. The libmetal code also handled start up and processor identification, dynamic processor enumeration and memory sizing, enabling the same binary image to run on systems with different numbers of processors. This was a critical requirement to ensure equality between the test systems as they can vary significantly in processor count.

Scheduling was handled on a task by task basis, so that each processor in a system got a static number of tasks to complete. No dynamic or preemptive scheduler was present in libmetal, each core preformed each task assigned to completion before starting the next task; no processors executed tasks not initially assigned to it. The scheduling for any given workload consisted of allocating a number of benchmarks per available core in a round robin manner, so that, in a three-core system, core 0 would be assigned workloads 0, 3, 6 and so on. The benchmark ordering in the workload thus has an effect on the effective scheduling, but the scheduling was static for any given number of cores. Each workload binary was thus a self-hosting statically-scheduled image that was ready to run on any of the designs in the design space.

We generated two sets of workloads, one which had the benchmarks ordered randomly and which had a general ordering with the longer-running tasks first. These ordered workloads were introduced to make sure longer tasks finished first. As there are many possible workloads that can be constructed from combinations of the chosen benchmarks, 35 unordered and 47 ordered workloads were randomly constructed. It was expected that the ordered workloads will provide a somewhat more even design

space, so that machine learning would perform better, but little evidence for this was found in the results. In addition, some of the ordered workloads failed on designs that completed all unordered workloads and were considered fully functional; these workload - design combinations were not used in the data set.

After workload completion the inserted design counters are read out using a separate non-invasive scan chain, discussed in section 3.3. We thus obtain information on the performance of a particular hardware design for a particular program. By using these counters the necessary measurements, such as average cycles per I/O request and average transactions per interconnect switch are extracted. Table 5.3 summarize the counters inserted into our designs. From these counters the values required, that cannot be measured directly, can be derived.

Counters	Description	Size
5 x number of cores	Counters for I/O cycles, I/O operations, committed instructions, etc.	48 bits
2 x number of Interconnect Switches	Traffic counters, counting AW channel traffic.	32 bits
2 x number of Interconnect Switches	Traffic counters, counting W channel traffic.	32 bits
2 x number of Interconnect Switches	Traffic counters, counting B channel traffic.	32 bits
2 x number of Interconnect Switches	Traffic counters, counting AR channel traffic.	32 bits
2 x number of Interconnect Switches	Traffic counters, counting R channel traffic.	32 bits

Table 5.3: Summary of inserted counters

5.2 Dataset generation

This section discusses the generation and use of the design space dataset. A flow diagram of this can be found in Figure 5.1. This diagram is intended as a complement to the machine learning evaluation diagram in Figure 3.20 in chapter 3 on page 81, as it details both he machine learning process and the dataset generation process.

Using the full range of parameters of the design space generation results in 510,000 possible different designs. Some of these designs are far larger and more complex than necessary, or otherwise violate device constraints. Synthesis for our chosen FPGA

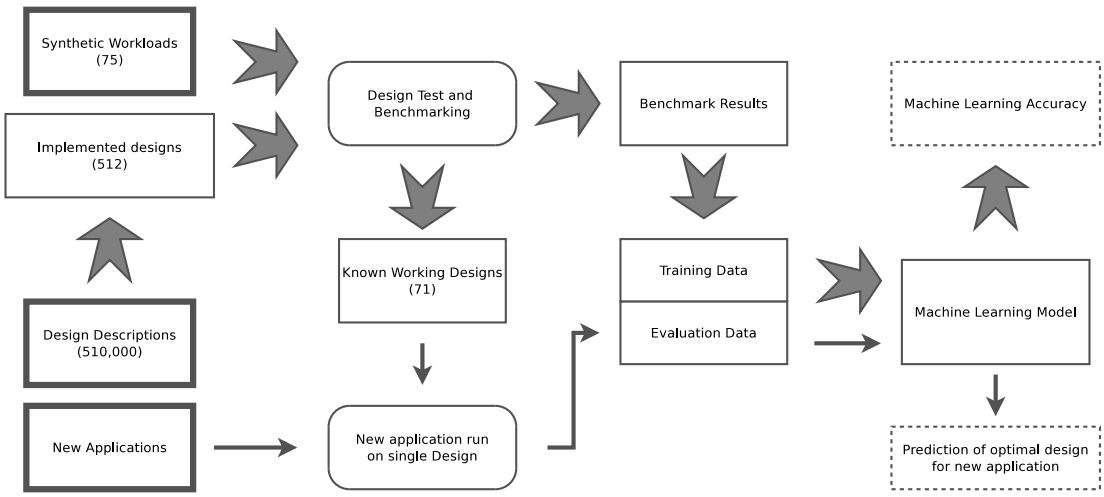


Figure 5.1: Flow diagram. Large arrows are training and evaluation flow, thin arrows the steps for a new application. Thick boxes are inputs, dashed boxes outputs.

device is used as the design constraint. Two steps of synthesis failure were recorded; firstly, if the design tools managed to map the design in the available FPGA logic, creating a size constraint, and secondly if the design met the timing specification.

It would be conceptually simple to substitute a set silicon synthesis flow instead of the FPGA synthesis path, which would then implicitly use silicon manufacture constraints instead. This silicon synthesis process could use the same steps of attempting to fit the design into a set area for a given process, and further finding if the implemented circuit is within the timing constraints of the design. Using such a step can give different, and possibly more accurate, area and timing information compared to the FPGA synthesis step used here; but the FPGA synthesis step still needs to be performed to evaluate dynamic runtime and energy metrics of the design. Thus, while additional data could be gained in a silicon synthesis step, it was found that the equivalent FPGA process generated usable data.

As the SOC design space is very large, the experimental design space was generated by randomly selecting 512 designs, covering 0.1% of the design space. This was almost twice as many as used in chapter 4, and as these designs were larger in terms of the amount of logic incorporated, synthesis took on average a longer time. A single design synthesis consumed approximately 4 CPU-hours and up to 6GB of RAM; the latter property prevented a massively parallel implementation phase due to the relative scarcity of RAM compared to processor cores. In total, running 5 synthesis processes in parallel, the entire synthesis process alone consumed approximately two weeks of real time.

By selecting out experimental designs randomly from the large design space it

was ensured that the sample is indicative of the design space as a whole. Randomly choosing 512 designs from 510,000 translates into a confidence of 97.5% that the mean of the measurements taken from this sample are within ±5% of the mean of the entire design space. It is therefore possible to treat the 512 selected designs as indicative of the entire space.

Of the 512 randomly chosen designs, 459 fitted into the FPGA size constraint; 278 also met the timing constraints.

Due to expected hardware and software issues, not all designs that met timing constraints performed flawlessly in executing benchmarks. Because of infrastructure limitations (Sections 3.2.6 and 3.5), and the very large amount of designs and software combinations, it was not practical to debug all problems, and thus designs were removed from consideration. Some designs completed a subset of workloads, but these designs were not used as not all benchmarks were completed, and it was assumed that a flaw in that design was at fault. Due to this conservative selection of designs, 71 designs were used for the performance predictions.

The data gathered from the synthesised counters for each program was used to derive some specific measurements. Total runtime is set as the time, in seconds, it takes for an entire workload to complete. Due to the above time limitations, this will never be more than 180 seconds. The dynamic energy is extracted using the energy model described in section 3.3.3.

Due to the FPGA fabric used and the variability between designs, it is not possible to attach a direct energy value (in Joules) to the energy measurements. Thus direct change in dynamic energy with changing hardware design is measured, rather than measuring total dynamic energy. The obtained measurement is only relevant when compared across designs.

The dataset thus contain a large number of application - designs runs, each with a attached runtime and energy measurement. The design parameters discussed in Section 5.1 are normalised and fitted into the feature vector directly; the workload composition and ordering of tasks is also used.

5.3 Results

In this section empirical results and findings of internal experiments on the dataset are discussed. These experiments aim to first demonstrate that machine learning can be used over this expanded dataset, and secondly to show the performance of the predictions.

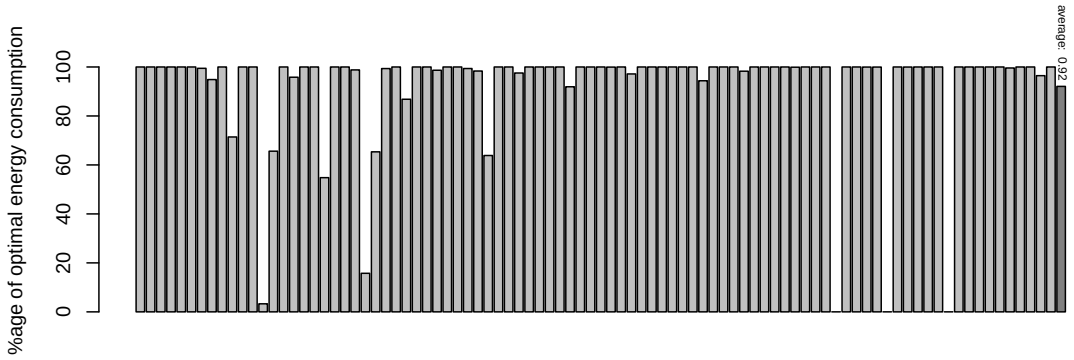


Figure 5.2: Performance of predicted design as a percentage of performance of best known design, optimising for energy.

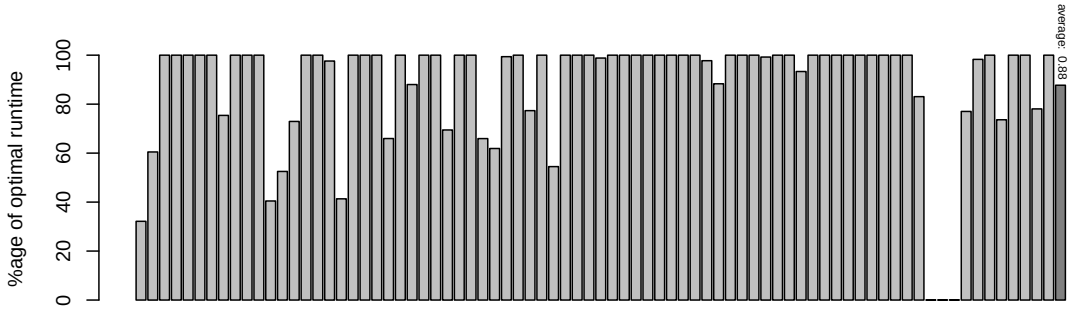


Figure 5.3: Performance of predicted design as a percentage of performance of best known design, optimising for runtime.

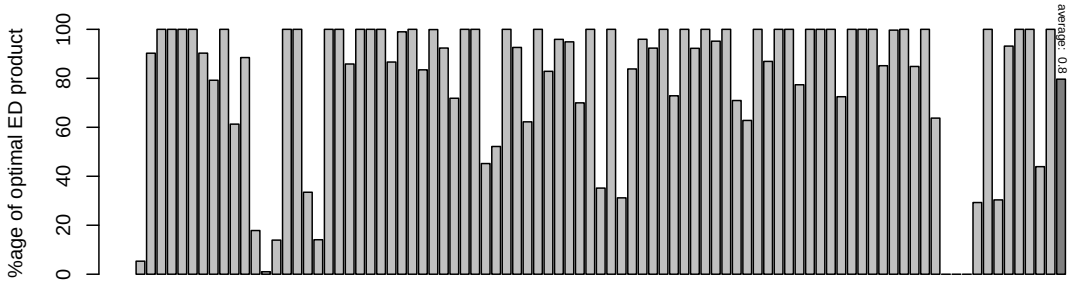


Figure 5.4: Performance of predicted design as a percentage of performance of best known design, optimising for ED product.

5.3.1 Predicting optimal designs

This experiment, in common with the evaluations in Chapter 4, uses classifiers to predict the best design.

For each of the 82 applications, a classifying *wk*NN predictor was trained on set of all other applications, run on a single sample design, mapping to best designs. Thus each predictor had a training set of 81 different applications and the best designs for those application, and was trying to predict the best design for the 82nd application. This is also known as leave-one-out cross-validation; but is used here as the evaluation step. The number of neighbours used in this prediction is 5, and a triangular distance weighting scheme was used. No attempt to train *wk*NN further was made. The feature vectors is made up of selected application metrics, such as number of interconnect operations and latency, derived from the design counters.

Once trained the method was applied to the application under investigation, producing a classification of which of the known designs would be the best one. The relative performance of the predicted design against the best design was obtained from this, and plotted in Figures 5.2 to 5.4.

Despite using a classifier, the absolute performance comparison is presented as the evaluation metric. In some cases a relative performance of 0% compared to the optimal application was recorded; these were the cases where the classifier could not predict a design with high enough confidence. This absolute performance metric is averaged across the applications to generate an average prediction performance.

Figure 5.2 shows that the average performance when finding the best design for energy performance is 92%. Figure 5.3 gives that the average performance in finding the best design for runtime performance is 88%, while Figure 5.4 shows the average for predicting the best design for ED performance is 80%.

The obvious gaps in Figure 5.4 are explained by exploring what program features led to useful predictions; the applications with low predictions typically consisted of a small number of benchmarks which typically did not stress the interconnect greatly. The resultant low measurements were not enough to distinguish which design would be the optimal.

This section has shown that machine learning in the form of *wk*NN can be used to effectively map new, unseen programs to optimal hardware designs. The classifications presented took fractions of a second to perform, which compared to the multi-hour undertaking of design synthesis and evaluation, is very cheap indeed.

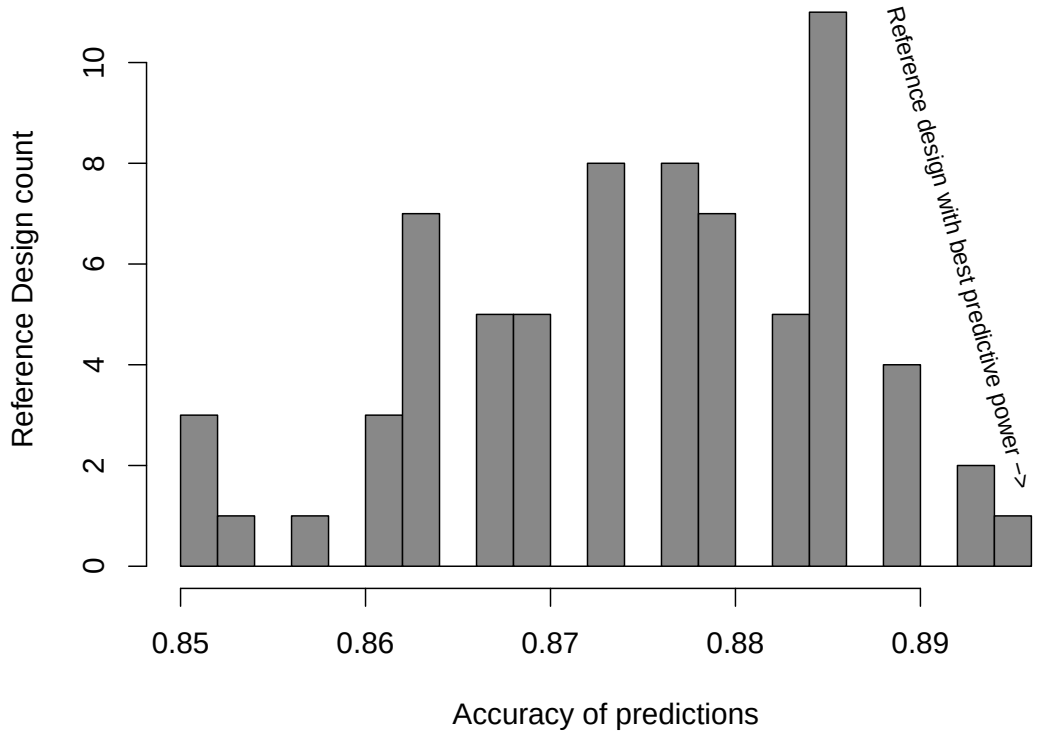


Figure 5.5: Spread of k-NN classifier accuracies for different reference designs. Right is better.

5.3.2 Predictive power

A question remaining from the previous section is which design should be used to run new applications on, as each classifier is trained only on applications from a single design. Application runs from different designs cannot be mixed in this evaluation, as the feature vectors consist of counter values comparable only within the same design.

A number of experiments similar to the one in section 5.3.1 were carried out, one for each of the 71 working designs, and the average prediction performance plotted in Figure 5.5. This histogram shows that only one test hardware design scored 92% average performance in finding new designs for new applications, but all designs scored above 85%. This was the SOC design that was used in the previous experiments.

The difficulty in predicting the best design is illustrated in Figure 5.6. This graph shows the distributions of number applications considering a design optimal against the designs, represented by their design identification numbers. Runtime, energy and ED product is plotted independently here. From this graph it is clear that finding the best design for an application is dependant both on the optimisation goal and the application itself. For example, even though design s267380 is good for energy-optimisation for a majority of applications, it is not the best for any other optimisation goal, or indeed, all applications.

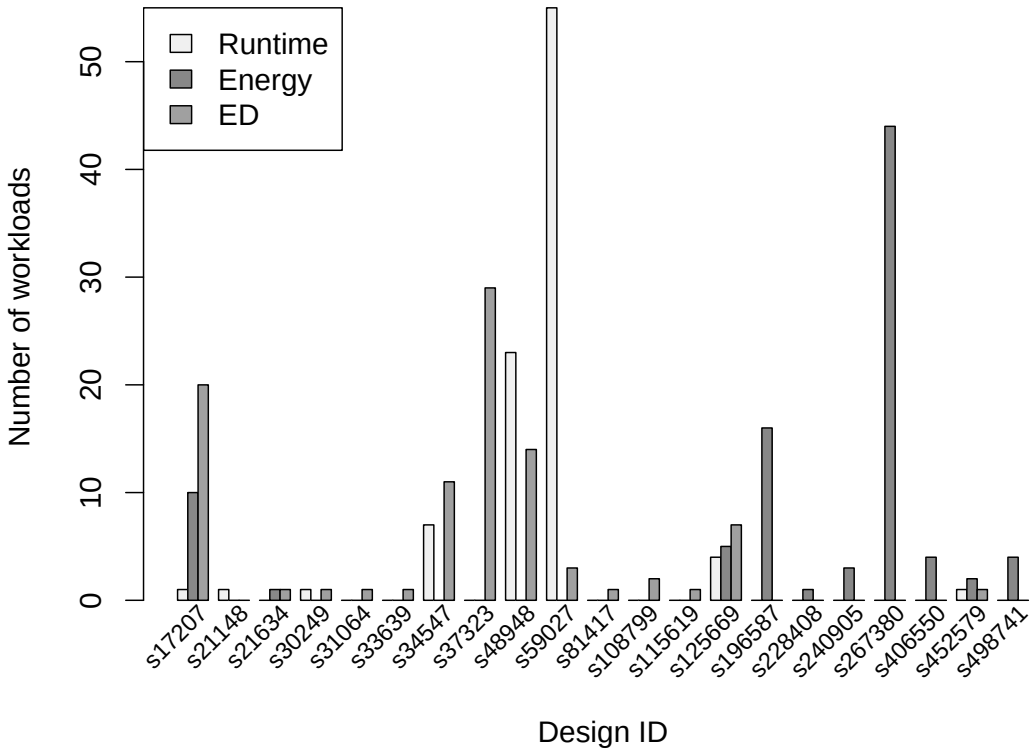


Figure 5.6: Distribution of best designs from workloads, for Runtime, Energy and Energy-Delay product. Ordered by design ID and thus in approximate order of overall complexity.

Table 5.4 shows the design parameters for the designs in Figure 5.6 that have more than 10 applications for which it is optimal. This table reinforces the diversity of optimality that is already shown in Table 5.6, and shows that for a given application, the optimal design is not obvious. For instance, both designs s196587 and s267380 are optimal in ED product for a large number of designs, but they are not overtly similar; the only thing they do have in common is core clock and switch type - the number of cores in these designs is radically different. In addition, designs s48948 and s59027 are the best for runtime for many applications, and are reasonably similar; but unintuitively, both have very slow core clocks. This shows the non-intuitiveness of the design space presented and how selecting the optimal design for one application does not by necessity mean it is best, or even close to the best, for another application.

5.3.3 Predicting working designs

For predicting which designs work, the set of synthesised designs was used by randomly selecting 12 designs for validation; the other 500 were used for training. We ran 1000 such training / validation cycles to average out any problems with the randomisation.

Design	Cores	Core clock	AXI clock	Switch type	s	RAM
s17207	8 (4, 0, 3, 1)	75 MHz	25 MHz	Unregistered	2	1
s34547	4 (4, 0, 0, 0)	50 MHz	25 MHz	Unregistered	2	4
s37323	10 (2, 3, 2, 3)	50 MHz	50 MHz	Unregistered	3	4
s48948	11 (4, 4, 3, 0)	25 MHz	50 MHz	Unregistered	3	1
s59027	9 (2, 0, 3, 4)	25 MHz	25 MHz	Unregistered	3	1
s196587	3 (2, 0, 0, 1)	75 MHz	125 MHz	Registered	2	1
s267380	10 (2, 2, 4, 2)	75 MHz	50 MHz	Registered	1	4

Table 5.4: Design parameters for any design with more than 10 optimal applications from Figure 5.6. s parameter refers to topology, and RAM is the number of separate logical RAM blocks used.

As a baseline, a guessing predictor, that predicts randomly based solely on the relative partitioning of the training data set, was used. This predictor managed an accuracy of 83% over 1000 trials, with 3.2% false positives and 13.7% false negatives. This is indicative of the extreme slant of the search space: trying to find the small portion of working designs among many non-working designs is a challenging task.

Using baseline 7-NN to predict if a given design will synthesise correctly achieves an accuracy of 85.6%, with 0.6% false positives and 13.7% false negatives. This is an improvement on the baseline, particularly in terms of false positives. Using this predictor has the advantage that it will very rarely clear a design that will not work.

A linear SVM algorithm, without dataset specialisation however manages an overall accuracy of 87.8%, with 5.3% false positives and 7% false negatives. Specialising this predictor could render larger gains, but was not attempted. This is 27% fewer mispredictions compared to our baseline, but at the cost of a higher false positive rate.

5.4 Conclusions

This chapter has shown that the method of machine learning, exemplified mainly though the *wk*NN method, can address the problem of automated application-specific SOC design in a time-efficient manner. In addition, the SVM algorithm is suited for predicting working designs from gross system configuration information, including determining suitability and conformity to design constraints. This will be expanded on in Chapter 6, where predicting different levels of functionality will be investigated.

*wk*NN is better at classifying the optimal design for a previously unseen workload. A *wk*NN classifier achieve on average up to 92% of the optimal application perfor-

mance across our 82 applications. Which classifier is the best one is another theme that will be revisited in chapter 6.

The classifiers, once trained, take less than a fraction of a second to classify a new set of features, enabling large savings of time in optimising designs for new applications. Using this method of finding optimal application-specific SOC designs is very efficient for a new application on a design space that has been previously explored even lightly, but the presented methods are unable to look outside the explored area.

6 Predicting metrics of new designs

This chapter deals with evaluating machine learning methods for predicting performance of new designs and programs. Crucially, it deals with predicting performance from architectural and high-level program features, so these methods can be used to evaluate the entire design space. The accuracies of machine-learning methods for predicting the level of functionality, the system area, software run-time and energy performance of designs that are not in the training set are presented. These accuracies are representative of the entire design space, and thus these predictors can be used to find new designs that perform optimally in the design space.

This chapter uses the same design space as Chapter 5; see Sections 5.1 and 5.2 for further details on how this design space is constructed.

This chapter is divided as follows. Section 6.1 covers methods for predicting which designs are faulty or not worth synthesizing from the architectural parameters. Section 6.2 covers methods for predicting the metrics, such as energy consumption, of new designs and programs. Section 6.3 summarises the results and conclusions from this chapter.

6.1 Predicting non-functioning designs

For a new SOC design in the design space, some base questions about its viability need to be answered before it can be evaluated as a potentially optimal design. For instance, it is necessary to know if it will conform to synthesis constraints, (i.e if it is too large or too complex); to predict if it will hit timing, and finally some indication of whether it will actually run the software, or if a bug may prevent it from doing so, would be beneficial. As synthesis and testing take a good few hours, even for a single new design, these outcomes needs to be predicted from pre-synthesis architectural knowledge. This section will show that machine learning can be used to predict these outcomes with less complexity than full synthesis. There are several different methods of machine learning (see Section 2.2). The aim was to evaluate the accuracy of a few common machine learning methods in predicting these outcomes.

6.1.1 Evaluation methodology

It is necessary to predict three different metrics, which are fulfilled or not, in the base dataset. From the data set of 512 synthesised designs, 459 met synthesis constraints. Of those, 278 hit timing and were used to derive the data set.

As the sample of 512 designs is a representative random sample from the full design space, it is expected that the fractions are similar across the full design space. It would thus be beneficial to predict for functionality, as predicting that a design will not hit timing alone will save us 50% of experimental synthesis runs.

The goal is to find what prediction method may give good results when predicting these functionality levels. To evaluate prediction methods, three data sets were generated.

- One data set with all 512 designs, to use for predicting which ones will synthesise correctly.
- One data set with all 459 designs that finished synthesis, for predicting which will hit timing.

Predicting that a design will meet timing should be enough to warrant further testing for performance, this is done in Sections 6.2.3 to 6.2.8.

As a baseline for prediction, random datasets of the same length and distributions as the original data sets were generated. These were used as the predictions. These random predictions conform to the known distribution of the training set, but are not correlated to the features of the instance to be predicted. Each one is therefore a lower bound for predictive performance. In the results section all machine learning methods are presented with both their accuracy figures as well as a comparison to the lower bound and their improvement upon it.

The prediction methodology follows the flow presented in Section 3.6, summarised here for completeness. The numbers presented are obtained through two-fold cross-prediction, where the predictors are trained on a randomly selected half of the original data sets; the trained predictor then predicts the other half, and vice-versa. The actual prediction score is the combined score of the two half-data predictions. Where predictors have internal feature selection or tuning-by-search functionality, and do their own internal cross-validation to find tuning variables, these are also only given half the data set for tuning.

The machine learning methods that were assessed in this experiment were Linear Regression, SVM, Random Forest and *wk*NN. As this is a classification problem between two classes, Linear regression was modified to round to the nearest integer and then saturate to either 0 or 1. The other methods have 'native' classification

modes, which generate classification predictions with no further input. For νSVM, the tuning options available were utilised to automatically find optimal γ, C parameters within $\gamma \in \{10^{-6} \ldots 10^{-1}\}$ and $C \in \{2^1 \ldots 2^5\}$ but keep the radial kernel function. Random Forest inherently performs feature weighting and selection, and no extra options or tuning parameters were given. For wkNN the training method automatically selects the $k \leq 10$ and weighting function, but Euclidean distance was used throughout. These parameters essentially allow the machine-learning methods to fit better to their source data in some regard. The full customisability of these methods were not used as this would make training unnecessarily long and can over-fit to the training data, making predictions worse.

No feature selection was performed on the source data. For Random Forest it is unnecessary, as it does its own internal feature selection; Linear regression similarly discards features that does not help in the curve fit. wkNN and SVM may benefit from feature selection, but these are instead tuned through adapting their input variables to the data set, achieving a similar effect of tuning the algorithm to the data instead of the data to the algorithm.

6.1.2 Accuracy in predicting non-functional designs

Tables 6.1 and 6.2 present the performance obtained with these machine learning methods.

Method	Correct	False pos.	False neg.	Accuracy	Improvement
Empirical Random	420	49	43	82%	0%
Linear Regression	469	43	0	92%	53%
SVM	476	20	16	93%	61%
Random Forest	477	33	2	93%	62%
wkNN	463	40	9	90%	47%

Table 6.1: Machine learning methods to predict conformance to synthesis constraints. Correct is number of correct predictions. Improvement is prediction improvement over the Empirical random guesser.

It can clearly be seen that the performance of predicting for synthesis success and timing is high, sufficiently so that these methods can be used across the design space. In aggregate, the above training and predictions took 104 CPU-seconds on a 2.4GHz Intel® Core™ 2 Duo. Most of this time was spent training and evaluating potential machine-learning coefficient combinations; the actual predictions took a small fraction of this time. It is therefore clear that running these predictions is significantly cheaper

Method	Correct	False pos.	False neg.	Accuracy	Improvement
Empirical Random	237	109	113	52%	0%
Linear Regression	367	50	42	80%	59%
SVM	379	43	37	83%	64%
Random Forest	379	60	20	83%	64%
*wk*NN	363	62	34	79%	57%

Table 6.2: Machine learning methods to predict timing performance. Correct is number of correct predictions. Improvement is prediction improvement over the Empirical random guesser.

than synthesizing and evaluating the full design, enabling savings in design time and selecting which design to synthesise.

6.2 Predicting metrics of new designs

Once a promising new design has been found, and it is fairly certain it will hit timing, we need to find what performance it is likely to achieve in order to evaluate whether the effort in implementing it is warranted. For this we need three essential measurements; the area of the new design, which is linearly related to the static energy dissipation; the dynamic energy consumption, and the run-time of the programs. Together, these three metrics together would give us a good idea of the performance of a new design.

While it is important to find the metrics of new designs, it is occasionally required to find the metrics for a new program on a design that is present in the training set. This is essentially an orthogonal problem; one way is to predict metrics of new designs for programs in the data set, the other way is to predict metrics of new programs on designs in the training set. In addition, these axes may need to be combined, so as to predict the metrics of program not in the training set on a design not in the training set; this would intuitively have a lower prediction accuracy than the two previous ways.

This section will discuss the results of finding a method for predicting these metrics. Section 6.2.1 discusses the evaluation methodology of these experiments. Section 6.2.2 then discusses the results of predicting the area of previously unseen designs. Section 6.2.3 covers the results of predicting for run-time on unseen designs with seen programs, and Section 6.2.4 similarly predicts for energy under the same conditions. Section 6.2.5 then covers predicting the run-time for new programs running on known designs, and its sibling, predicting energy for new programs on known designs is cov-

ered in Section 6.2.6. Sections 6.2.7 and 6.2.8 then cover prediction of run-time and energy respectively, for new programs running on unseen designs. Finally, Section 6.2.9 summarises these experiments.

6.2.1 Evaluation methodology

For evaluating run-time, area and energy metrics, we need to use predictors that generate continuous predictions instead of classifications, i.e. regression methods. To this end several machine learning methods were evaluated in regression mode to find which methods may give the best predictions.

For this evaluation the data set from Chapter 5 was used, which was obtained as detailed in Section 5.2. Only the 71 designs that were declared fully functional were used, each running a set of up to 82 workloads, for a total of 4264 complete feature vectors. Not all workloads were run on all designs leading to a discrepancy in the number of complete feature vectors. Predicting the area did not use the program-specific data, as the program performance has no impact on the area, and was limited to 73 feature vectors.

Feature selection was not used in preparing this data set, as all approaches that were tried (Random Forest and linear recursive feature selection, filtering with Random Forest or linear weights) led to worse predictions. In addition, Random Forest and Gaussian Processes inherently contain their own internal feature selection filtering, and do not require an external feature selection step. Gaussian Processes were specialised by evaluating the predictors for the kernel methods `rbfdot`, `laplacedot` and `besseldot` and variances from 10^{-3} to 10^{2} in order of magnitude steps, and picking the predictor with the smallest cross-validation error. For SVM the predictor was specialised by finding the optimal predictor for $\gamma \in \{10^{-6} \ldots 10^{-1}\}$ and $C \in \{2^{1} \ldots 2^{5}\}$, and automatically selecting the best predictor through internal cross-validation runs on the training set. This customization of SVM and Gaussian Processes is very expensive in terms of computation, but improves the results by tuning the algorithm to the data. Decision trees did not receive any treatment for specialization. For *wk*NN the predictor was specialised by automatically selecting $k \leq 10$ and also leaving the choice of weighting function open. Absolute distance was used in all cases. Specialising *wk*NN is also expensive in terms of compute time, but as *wk*NN is a much simpler method than SVM or Gaussian Processes the impact on overall run-time is not as large.

In all cases, the method followed the flow presented in Section 3.6, and the machine learning methods are presented in Section 2.2. All data presented is predicting for the non-training data with a trained predictor. For each prediction case, an empirical random method is also shown as a baseline for prediction performance; this is

essentially a random guesser, picking in each case a random results from the training set as the prediction. This is supplied as a worst-case predictor and a baseline for comparisons between predictors. It is in general a better baseline than a flat 100% error, as such an error rate would indicate a predictor that always predicts the worst possible outcome*.

For all following sections, saying that data is 'known' or 'seen' means that it is included in the training set. For example, Section 6.2.6 predicts the energy of new programs on known system implementations, meaning that all system implementations used in the test set are also in the training set.

The design-space data is subjected to a random two-way split in Sections 6.2.2 through 6.2.6, with one half used for training, one half for evaluation, and the results being aggregated. Sections 6.2.8 and 6.2.7, by contrast, utilise data split into four random parts, so that there are two pairs that do not share either design or application data. These are then used for four training and validation passes, and the results are aggregated before presentation. Because these sections consequently use smaller training sets and try to perform more complex predictions, intuitively they should be less accurate.

The figures for the rest of this chapter are, for reasons of layout, moved to the end of the chapter; page references are inserted where necessary. The figures remain ordered according to the sections in this chapter. Each figure is prefaced with a blank page with a figure heading to facilitate browsing and increase the separation of figures, facilitating reading this section.

The experiments in the following sections are presented as graphs, showing a) a plot of the predicted values versus the actual values, such that a perfect prediction is on the line of gradient 1 passing through the origin, and b) a histogram of the deviations in the result from this perfect prediction, with the mean (blue, value on the left) and standard error (red, value on the right) drawn in. This shows the distribution of the predictions, and, by inference, the distribution of the original dataset, as well as showing the distribution of the errors and allowing an easy comparison between the various methods. In addition, the graphs are labelled with the time taken for the method in question to complete on the data set provided, giving an indication of the run-time. These times are all measured running the appropriate R implementations of the machine learning methods (as discussed in Section 2.2) on a 2.4GHz Intel® Core™ 2 Duo with 4GB of RAM, which is otherwise unoccupied except for these prediction tasks. The measured time includes the full dataset training and prediction time for both branches of the cross-validation methodology in aggregate. The time taken

*Such a predictor would be quite valuable, as being 100% wrong generally means that a 100% correct predictor can always be derived by inverting the answer appropriately.

would vary somewhat between different runs, and the times measured are thus presented as approximate and are intended to give an indication of the typical run-time rather than an exact measurement. For the same reason, the runtime is only presented to two significant figures.

The standard error measurement is taken as the optimisation target, which is used to evaluate the performance of the machine learning methods. The standard error may be small even though the error range and maximum error is large, and this indicates that the method may mispredict by a large amount very rarely even if it is very accurate most of the time. These potentially very large mispredictions would be unfortunate in an applied setting. However these mispredictions are rare and, again in an applied setting, easily corrected through the feedback loop likely to be employed for other reasons, they do not pose a problem to the method. The use of the standard error metric as the evaluation function for these methods is therefore appropriate.

A table at the end of every section summarises the methods, times, and achieved performance in absolute and relative terms. While this table condenses the information from the graphs, it is not a substitute; differences in the error distribution are clearly visible in the graphs but may not be adequately represented in the standard error measurement.

6.2.2 Accuracy in predicting area for new designs

This section presents the prediction results for the area of non-synthesised designs from architectural parameters. The graphs are split into error and deviation graphs. The first graph for each method plots the actual value against the predicted value, and thus shows actual prediction distribution. The second graph plots a histogram of error magnitudes and indicates the mean and standard deviation of the errors.

Figures 6.1(a) and 6.1(b) on page 129 show the baseline measurements, where the prediction is taken randomly from an identical distribution to the input data. The standard deviation of the error between the predictor and the actual measurements is used as the metric of goodness of the predictor; in the case of the random guesser, this comes to 22.47% of the area. Linear regression, or other directly linear methods, are expected to do well on this, as the area is largely linearly dependent on the components, with strong correlation to the number of cores and memory banks.

Gaussian Processes and plain Decision Trees were also tried on this dataset, but the specific implementations of these methods did not allow for the scarcity of data in this dataset, and did not generate results.

From Figure 6.1 on pages 129 to 130 it is clear that the best method for predicting area, with the limits on the data set, is linear regression in Figures 6.1(e) and 6.1(f) with

a standard deviation of 5.1% of the total area. Linear regression, especially on this size of data set, is also the fastest method. SVM also performs well, but potentially suffers from not enough training data.

Method	Training time	Prediction time	Std. Error	Improvement
Empirical Random	0.6	0.00016	22.47%	0%
*wk*NN	0.14	0.046	13.95%	37.92%
Linear Regression	0.018	0.0097	5.148%	77.09%
Random Forest	0.22	0.014	8.656 %	61.48%
SVM	0.028	0.014	7.168%	68.10%

Table 6.3: Summary table of the accuracy scores of predicting area for new designs. Time is recorded at 2 significant figures, improvement is improvement over the Empirical Random baseline.

Table 6.3 shows the summary values for the methods for predicting the area. Linear regression, Random Forest and SVM all managed more than 50% improvement over the random predictor, whereas the *wk*NN performed poorly on this data set. Overall, linear regression was the most accurate as well as the fastest method for predicting area.

6.2.3 Accuracy in predicting total run-time for new designs

This section presents the results of predicting the run-time of a known program on a new, non-synthesised design. The baseline is presented in Figure 6.2(a), and its error distribution is shown in Figure 6.2(b), both on page 132. As before, the standard deviation of the prediction error is used as the evaluation metric, and the baseline here is 22.3%. The results of the other machine learning methods are shown in Figures 6.2(c) to 6.2(n) on pages 132 to 134.

These results were harder to anticipate compared to the area; it was difficult to see any pattern in the run-times from the design parameters other than higher clock frequencies that in general would perform better. The machine learning methods did however in general manage to tease out significant relationships; in particular Random Forest (Figures 6.2(i) and 6.2(j) on page 133) managed to predict the relationship very well. This is even more surprising, as Decision Trees (Figures 6.2(g) and 6.2(h) on page 133) fared so badly; the tree built by this algorithm suffered heavily from the noisy nature of the input data and only used a couple of parameters before the entropy limit halted further growth of the tree. This is directly evident in the quantised nature of Figure 6.2(g).

Method	Training time	Prediction time	Std. Error	Improvement
Empirical Random	0.0024	0.00018	22.3%	0%
*wk*NN	11	8.3	6.756%	69.7%
Linear Regression	0.052	0.019	7.8%	65.02%
Decision Tree	0.52	0.024	7.19%	67.76 %
Random Forest	45	0.46	2.937 %	86.83%
Gaussian Process	1960	1.4	3.92%	82.42%
SVM	700	1.1	3.442%	84.56%

Table 6.4: Summary table of the accuracy scores of predicting run-time for known programs on new designs. Improvement is improvement over the Empirical Random baseline.

Random Forest was clearly the best predictor for this scenario, but SVM and Gaussian Processes (Figures 6.2(k) and 6.2(m) on pages 133 and 134) were not far behind; 2.9% standard error compared to 3.92% and 3.442%. This small difference is by itself not a major counterargument to using SVM for this application, but as SVM and Gaussian Processes are in this case is allowed a significantly longer (700 and 1960 seconds versus 45 seconds) execution time to specialise the predictor parameters. Random Forest was significantly faster as well as marginally more accurate, and is therefore the best method for predicting the total run-time of a known program on a new design.

Table 6.4 summarises the numbers from figures 6.2(a) to 6.2(n) on pages 132 to 134. From this table it can readily be seen that the most accurate methods for predicting the run-time of a known program on a new application are Gaussian Processes, SVM and Random Forest, which all manage better than 80% improvement in accuracy over randomly guessing the run-time. Gaussian Processes and SVM took much longer then any of the other methods; this was a result of attempting to locate the best parameters using exhaustive search, in the hope that this would improve overall performance. The performance of SVM is as a result very good, but at the cost of a lengthy training time. In comparison, Random Forest performs marginally better in a much shorter time, making it the preferred choice.

6.2.4 Accuracy in predicting total energy for new designs

This section presents the results of predicting the energy consumption of a new, unseen design executing a known program. The data set is the same as the one used in section 6.2.3, but the response value is different; instead of run-time, total energy consumption is used. This has a rather different distribution, which is most evident in 6.3(a) on page 136, with a large concentration in the bottom of the range. It would

thus be expected that this distribution is harder to predict, due to the relative scarcity of outliers, and this is indeed the case.

Table 6.5 summarises the performance of the machine-learning methods tried for predicting the energy of unseen hardware designs. As before, SVM takes the longest time due to searching for good parameters, but when optimising for energy this increase in search time is less warranted. Here, SVM is one of the less successful methods, despite this adaptation step.

Method	Training time	Prediction time	Std. Error	Improvement
Empirical Random	0.00027	0.00018	19.86%	0%
*wk*NN	15	8.5	8.924%	55.06%
Linear Regression	0.052	0.02	10.35%	47.88%
Decision Tree	0.63	0.03	8.775%	55.82%
Random Forest	28	0.28	7.29 %	63.29%
Gaussian Process	1959	1.2	8.67%	56.34%
SVM	450	0.022	10.35%	47.88%

Table 6.5: Summary table of the accuracy scores of predicting total dynamic energy for new designs. Improvement is improvement over the Empirical Random baseline.

In general, the improvement over the random baseline is limited for all the tested methods, suggesting that the data is very noisy and contradictory. Random Forest is again the best method, but at only 63.29% improvement. All other methods manage around 50% improvement in prediction errors. As such, the data recommends using Random Forest for predicting the energy consumption of a new system design, but as the error rate is still around 8% there may be many designs that are incorrectly rated favourably by this predictor.

6.2.5 Accuracy in predicting total run-time for new programs

This section presents the results of predicting the total run-time of unseen programs on known hardware designs. The evaluation is complimentary to Section 6.2.3 and 6.2.4, as they dealt with predicting for known programs on new designs. This data was obtained by splitting the data set so that approximately half of the tested programs were in each group, while all hardware designs were present in both halves. The methodology flow in section 3.6 was then followed for each machine learning method out of random-empirical, SVM, *wk*NN, Decision Trees, Random Forest, and Gaussian Processes. In all cases the algorithms were allowed as much time as necessary to build internal representations and do predictions.

Figure 6.4 on pages 140 to 142 shows the performance of various machine learning methods trying to predict the run-time of new programs. Of note in these are Gaussian Processes and SVM, figures 6.4(k) and 6.4(m) respectively, which show a good correlation with the unit gradient line and have very similar error measurements. Both also show a small leg of mispredictions in the same place, indicating that they learn the same kind of misprediction for these few instances. Decision Trees are before quantising the regression results, showing a typical striped pattern in the prediction distribution. Random Forest also does well on this, but is edged out by SVM and Gaussian Processes.

Method	Training time	Prediction time	Std. Error	Improvement
Empirical Random	0.0028	0.00031	22.16%	0%
*wk*NN	11	5.8	8.74%	60.56%
Linear Regression	0.052	0.019	9.076%	59.04%
Decision Tree	0.5	0.025	10.09%	54.47%
Random Forest	32	0.3	7.889 %	64.40%
Gaussian Process	1930	6.3	5.822%	73.73%
SVM	700	1.3	5.869%	73.52%

Table 6.6: Summary table of the accuracy scores of predicting total run-time for new programs. Improvement is improvement over the Empirical Random baseline.

Table 6.6 shows the summary of the performance of the predictors. Of note are SVM and Gaussian Processes, which both manage accuracy improvements of approximately 73.5%. SVM, due to the parameter search, takes a very long time to train; Gaussian Processes likewise take a long time due to the empirical search for the best predictor parameters. Overall, the best method for this prediction is tied between SVM and Gaussian Processes, but Random Forest is only about 10% behind in accuracy improvement and arrives at this result significantly faster than either.

6.2.6 Accuracy in predicting total energy for new programs

This section discusses the predictor performances when trying to predict the dynamic energy consumption of new programs on known hardware designs. The data sets are the same as for Section 6.2.5, but the outcome value is different.

Figure 6.5 on pages 144 to 146 shows the prediction distributions for predicting these values. Most remarkable in these graphs is how poorly the machine learning methods manage to predict the energy consumption. The distribution is heavily skewed towards the lower left, indicating that most designs have similar, small energy

consumptions. Even the empirical random method in figure 6.5(a) tends to underestimate the energy. This trend of systematical underestimation is visible in all other predictions except Decision tree and Gaussian Processes. Decision tree, despite not systematically underestimating, still quantises the results into the typical striped patters, and so does not fare very well. Gaussian Processes, however, seem to be a good method to use for this task.

Method	Training time	Prediction time	Std. Error	Improvement
Empirical Random	0.00026	0.0002	20.46%	0%
*wk*NN	14	6.4	9.632%	52.92%
Linear Regression	0.056	0.025	10.42%	49.07%
Decision Tree	0.62	0.029	11.63%	43.16%
Random Forest	28	0.29	9.008 %	55.97%
Gaussian Process	1950	1.4	8.372%	59.08%
SVM	472	0.019	10.42%	49.07%

Table 6.7: Summary table of the accuracy scores of predicting total dynamic energy for new programs. Improvement is improvement over the Empirical Random baseline.

The performance figures for this experiment are summarised in table 6.7. From this table it is clear that the best methods for predicting the energy performance of a new program on a known system architecture are Gaussian Processes; random forest is a close second. However, both are only about 55% better than randomly selecting the prediction from the known dataset, and in addition, Gaussian Processes take 32 minutes to train on this dataset due to the exhaustive parameter selection process. In general, predicting the performance of unseen programs is more uncertain than predicting the performance of new designs, indicating that perhaps more features of programs should be captured to improve this metric. Which method is preferred for this prediction will depend on the prediction deadline; Gaussian Processes does give a more accurate result but at a much longer training time.

6.2.7 Accuracy in predicting total run-time of new programs on new designs

This section discusses the results of predicting the run-time of new programs running on new designs. Compared to Sections 6.2.3 through 6.2.6, this section has less training data, and thus performance may be worse.

This section also features the source data split into four quarters, assembled so that there are two sets of cross-validating pairs. Each quarter shares system designs with one other quarter and programs with another quarter; it is paired with the quarter that

it does not share either system designs or programs with. Predictors are thus trained on one quarter and evaluated on the paired quarter and vice versa; thus four training phases and prediction phases are used for this evaluation. Results are then aggregated for display purposes so as to aid comparison to the preceding sections.

Method	Training time	Prediction time	Std. Error	Improvement
Empirical Random	0.0039	0.0006	22.06%	0%
*wk*NN	7.1	3.1	10.97%	50.27%
Linear Regression	0.073	0.033	9.193%	58.33%
Decision Tree	0.45	0.043	11.19%	49.27%
Random Forest	20	0.28	8.307 %	62.34%
Gaussian Process	618	5.8	7.783%	64.72%
SVM	559	0.7	5.9 %	73.25%

Table 6.8: Summary table of the accuracy scores of predicting total run-time for new programs running on new designs. Improvement is improvement over the Empirical Random baseline.

Figure 6.6 on pages 148 to 150 shows the prediction and error distributions of these experiments. As expected, the results of these predictors shows a greater uncertainty due to the more complex prediction needed, but the performance is not so bad as to render the methods unusable. Linear regression does quite well on this noisy data set, with a few outliers that lower the prediction accuracy. In comparison, Decision Tree does even worse, with more and larger outliers, due to its heavy-handed quantization method. SVM however is clearly the best method, but again suffers from a visible systematic misprediction; this was also mentioned in section 6.2.5. The cause is most likely the same; that there is a particular corner in the design space that is not accurately accounted for in the training but is present in the evaluation set. Gaussian Processes in figure 6.6(k) shows the same misprediction but to an even larger degree; it is also clearly visible in the prediction error histograms.

Table 6.8 summarises the performance of these predictors on this data set. It is clear that the loss in predictive power from the smaller training sets is very small, and the improvement over the baseline is similar to that in Sections 6.2.3 and 6.2.5. The best methods for predicting the run-time in these cases is SVM, with Gaussian Processes 10% behind but with 40 times shorter running time.

6.2.8 Accuracy in predicting total energy for new programs on new designs

This section discusses prediction performance when trying to predict the dynamic energy consumption of unseen programs on unseen designs. The data set is the same as that in Section 6.2.7, but the response value is different. As in that section, this evaluation is carried out in quarters with four training and prediction phases. It is therefor likely that the prediction accuracy will go down compared to Sections 6.2.4 and 6.2.6, due to smaller training sets. Figure 6.7 on pages 152 to 154 presents the prediction distributions and error distributions of the prediction methods on this dataset.

Method	Training time	Prediction time	Std. Error	Improvement
Empirical Random	0.00029	0.00018	16.93%	0%
*wk*NN	7.2	3.2	11.34%	33.02%
Linear Regression	0.074	0.03	11.27%	33.43%
Decision Tree	0.53	0.036	11.89%	29.77%
Random Forest	18	0.37	9.8 %	42.11%
Gaussian Process	640	4.5	10.97%	35.20%
SVM	350	0.55	9.841 %	41.87%

Table 6.9: Summary table of the accuracy scores of predicting total dynamic energy for new programs running on new designs. Improvement is improvement over the Empirical Random baseline.

The distribution of the energy consumption is significantly different from the runtime; in addition, due to the four-way prediction scheme, the distribution of the empirical random evaluation looks different to that in Sections 6.2.4 and 6.2.6. As before, Decision Tree is very quantised, but still manages on par of the other methods; possibly due to the other methods also not managing a very impressive performance. Most of the methods lean towards under-predictions, obvious in both the prediction distribution graphs and the prediction error histograms.

Table 6.9 shows the summary of the experiments in Figure 6.7. It is clear here that the smaller learning sets have caused a loss in predictive power; the best method here is Random Forest, but at barely more than 40% improvement, it is not particularly useful even so.

6.2.9 Predicting metrics summary

This section presented the results of several experiments aimed at trying to find the best machine learning method for predicting good points in the space of embedded

multiprocessor processor designs and applications.

In general, it was found that predictions of the run-time of applications had a higher accuracy then predictions of the energy consumption, but that both problems were tractable with the machine learning methods presented. Random Forest in particular performed well at this task, being the best method in half of the experiments and close to the best in two others. If a single method had to be picked for this, the recommendation would be Random Forest; if time allows, also using SVM when predicting run-time would be a good idea, as SVM scored significantly better on two of the three run-time experiments. Unfortunately, SVM has a long run-time for specialising the input parameters to the training set; but conversely, even at 20 minutes run-time this is at least 10 times shorter than a full synthesis and evaluation run of a design.

6.3 Summary

This chapter has detailed experiments and results trying to find 1) if machine learning can be used on the space of embedded designs and applications to predict outside of the learning set and 2) which method is best suited to the task.

It was found that it is possible to find new design points outside the training sets. The accuracy of the best method for predicting synthesis constraint conformance was random forest, with 93% of instances predicted correctly. The accuracy of the best method of predicting if a design will hit timing was also random forest, this time with an accuracy of 83%. It follows that the accuracy of finding new design points that both meet constraints and hit timing is $93\% \times 83\% = 77\%$.

It was also found that characterizing the new design points was possible. When predicting the area of a new design it was found that linear regression was the best method, with a standard deviation of 5.148% and an accuracy of 94.8%. Similarly, the accuracy in predicting the runtime was in the best case 97% for the Random Forest algorithm. The standard error in predicting the energy of a new design was 7.29%, giving an accuracy of 92%, again using Random Forest.

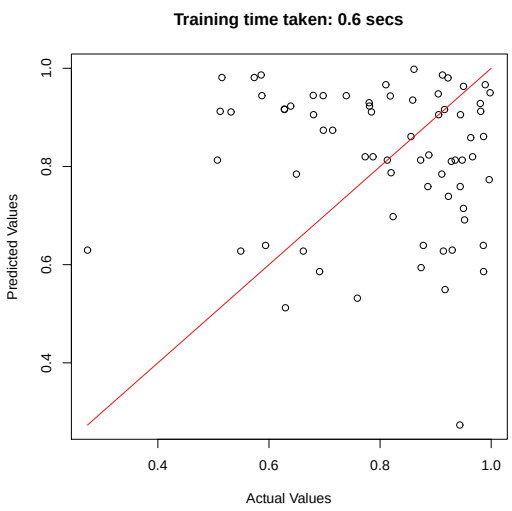
It was generally found that statistical machine learning methods such as those presented in 2.2 are applicable to this design space, with some caveats. Firstly, predicting for energy is harder than predicting run-time, which can be explained by the design space being simpler in terms of run-time response then energy response. Secondly, predicting the area is generally best done through a linear combination of architectural features, and in fact, linear regression is a good tool for this. Thirdly, for predicting run-time, the impact of the training set size is small once a certain predictor accuracy has been reached. This was shown in Section 6.2.7 where the learning set was half of

that in the previous sections but the accuracy similar.

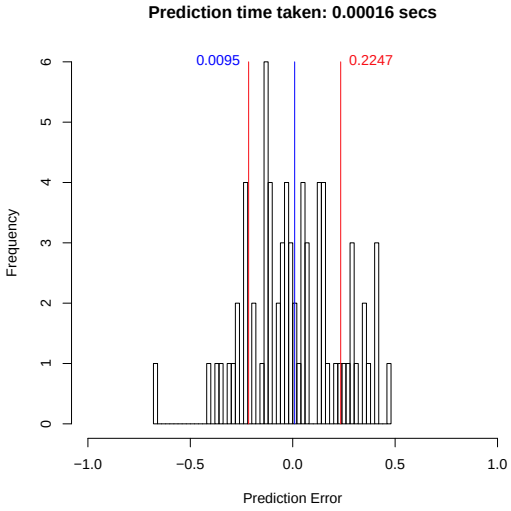
The best method across the experiments in predicting the run-time and energy performance is Random Forest. For certain experiments it is edged out by SVM or Gaussian Processes, but these methods both take additional time. SVM and Gaussian Processes in particular take a long time to select the optimal parameters for the training set. This additional time is still less then the time taken for a full design synthesis and test, and it is therefor still beneficial use these methods in concert with Random Forest.

Predicting area of new designs

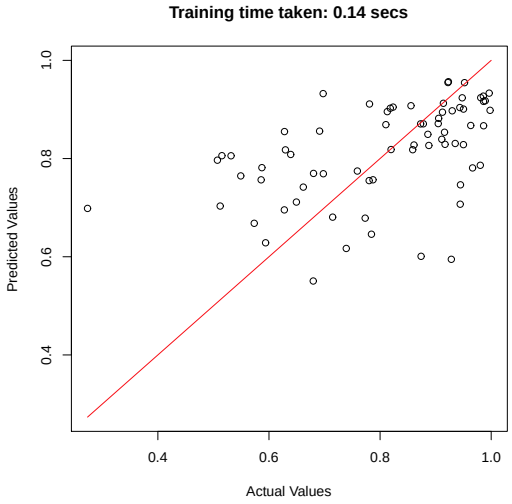
Figure 6.1



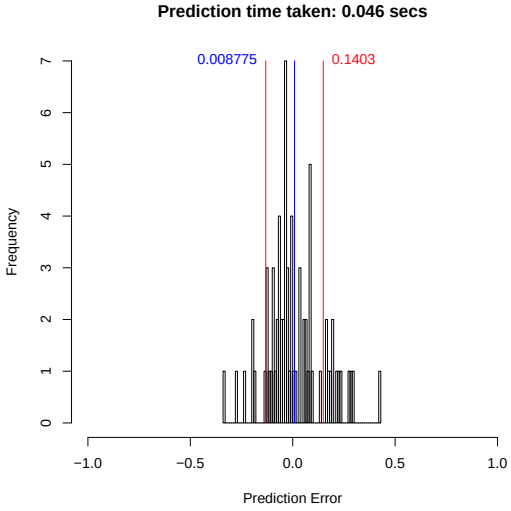
(a) Prediction Distribution, Empirical Random



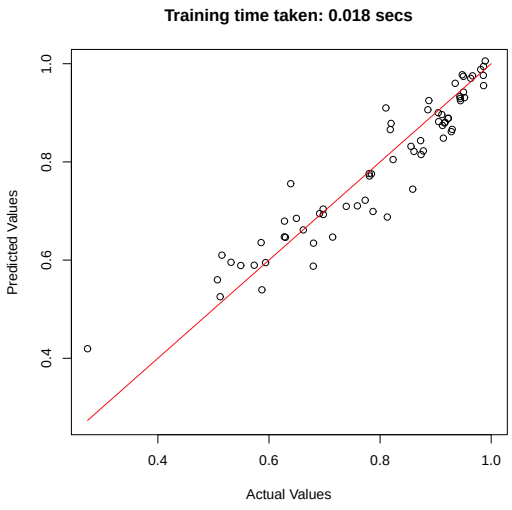
(b) Prediction Error, Empirical Random



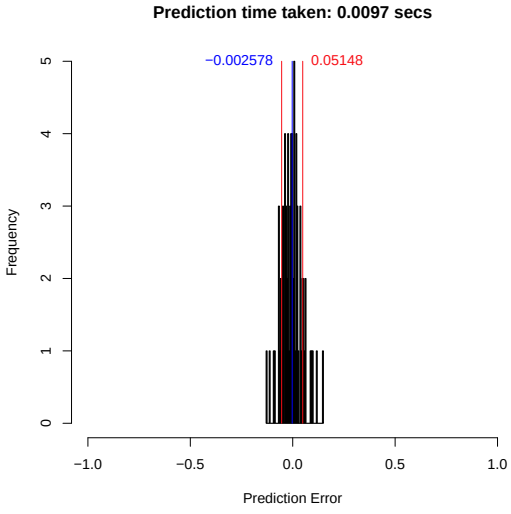
(c) Prediction Distribution, *wk*NN



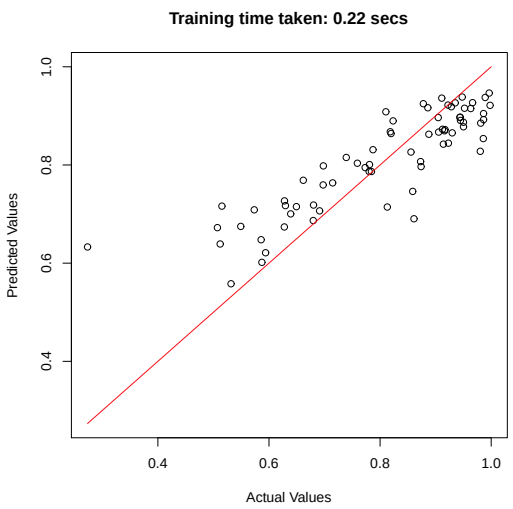
(d) Prediction Error, *wk*NN



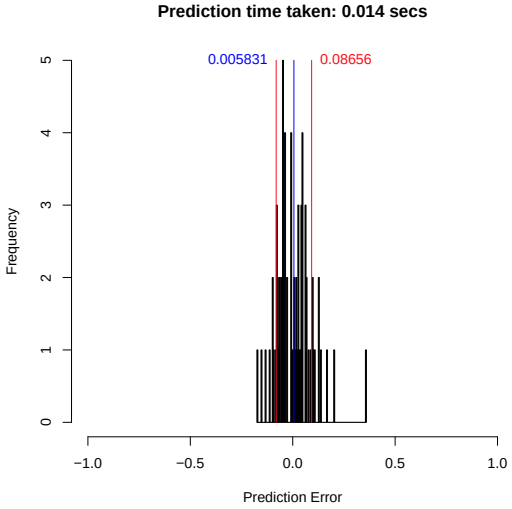
(e) Prediction Distribution, Linear Regression



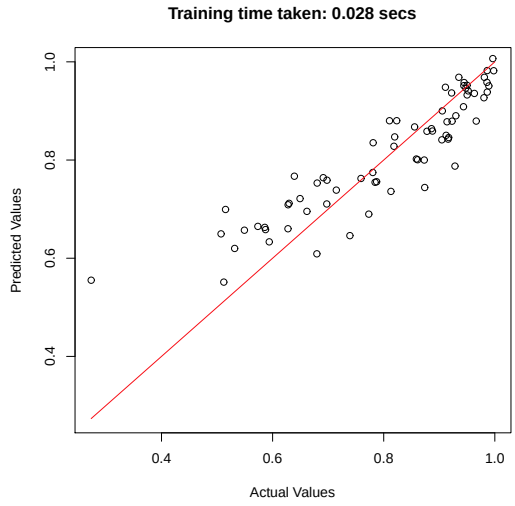
(f) Prediction Error, Linear Regression



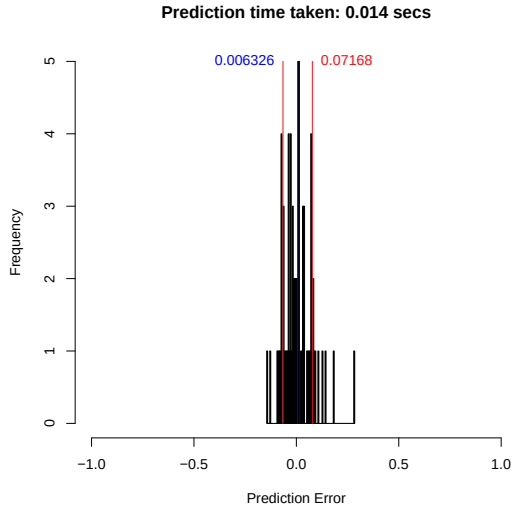
(g) Prediction Distribution, Random Forest



(h) Prediction Error, Random Forest



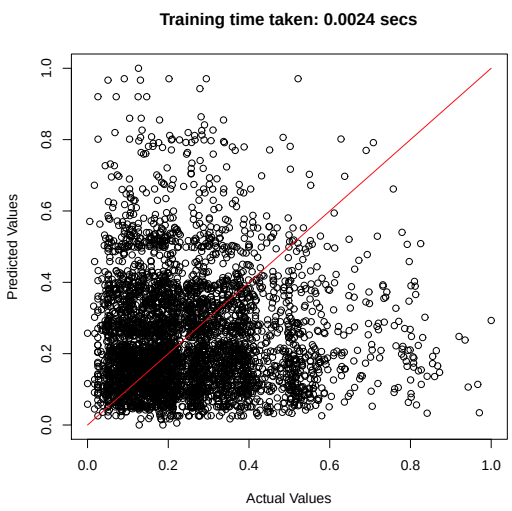
(i) Prediction Distribution, SVM



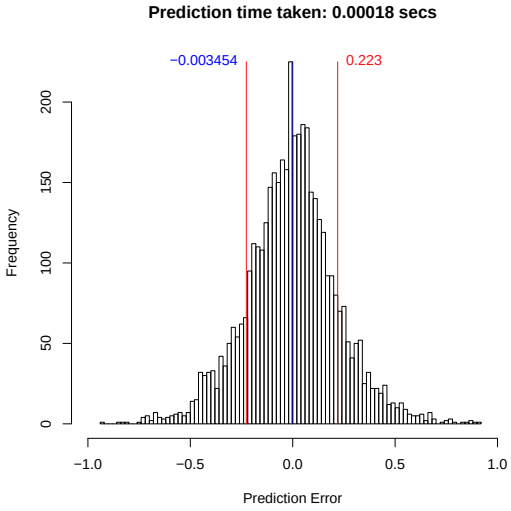
(j) Prediction Error, SVM

Figure 6.1: Graphs showing the performance of statistical machine learning techniques in predicting the area of unseen hardware designs.

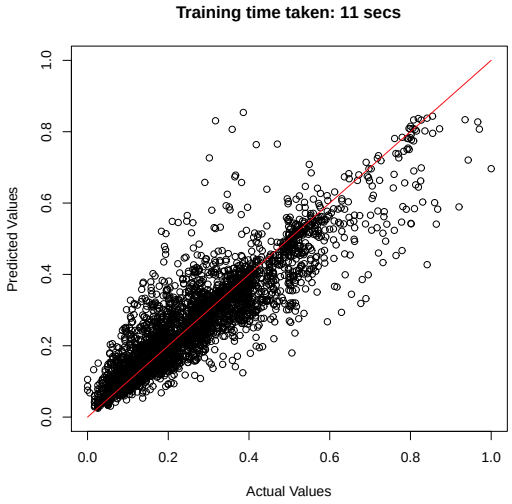
Predicting runtime of known programs on new designs
Figure 6.2



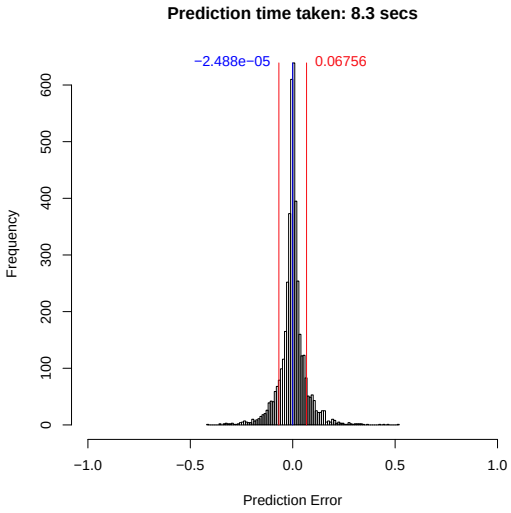
(a) Prediction Distribution, Empirical Random



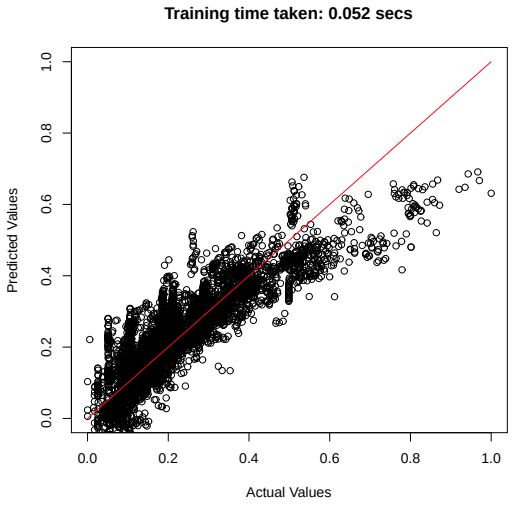
(b) Prediction Error, Empirical Random



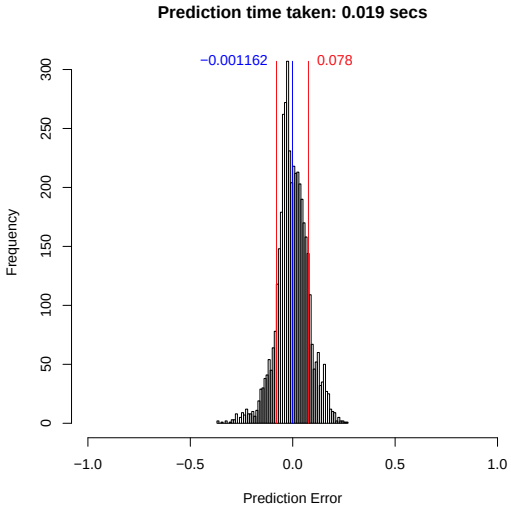
(c) Prediction Distribution, *wk*NN



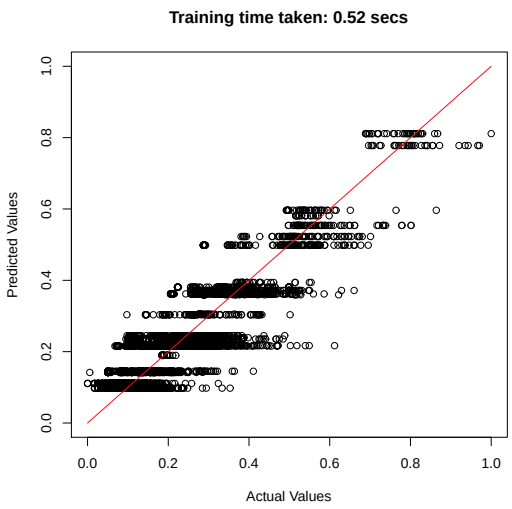
(d) Prediction Error, *wk*NN



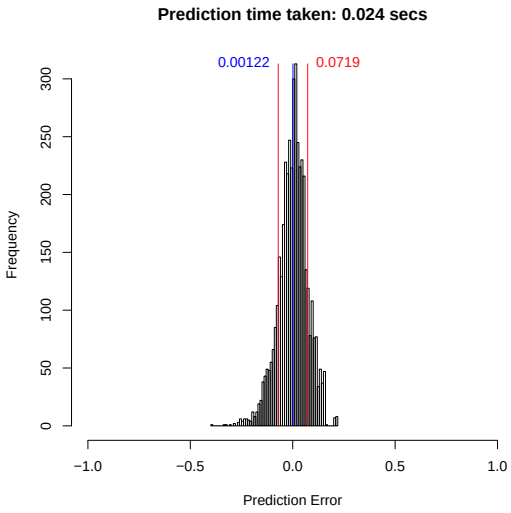
(e) Prediction Distribution, Linear Regression



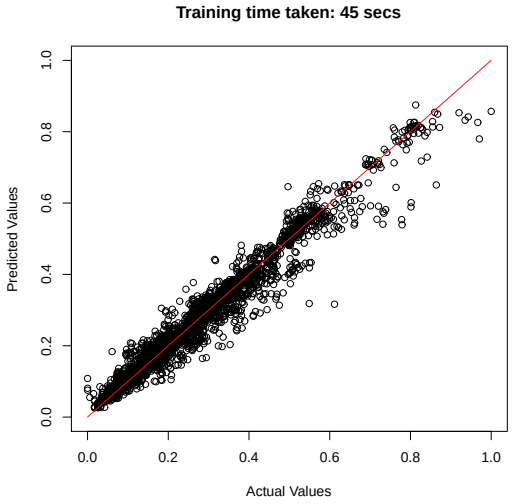
(f) Prediction Error, Linear Regression



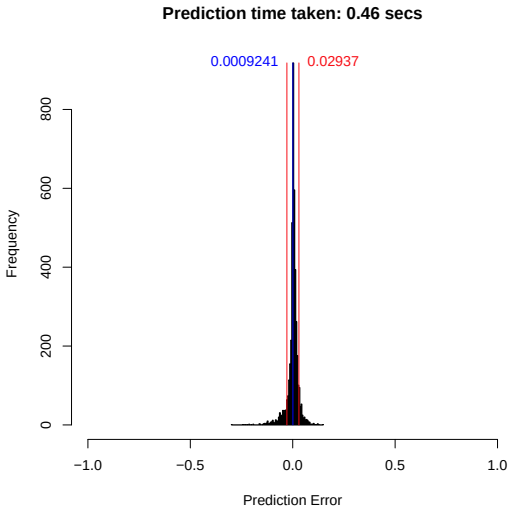
(g) Prediction Distribution, Decision Tree



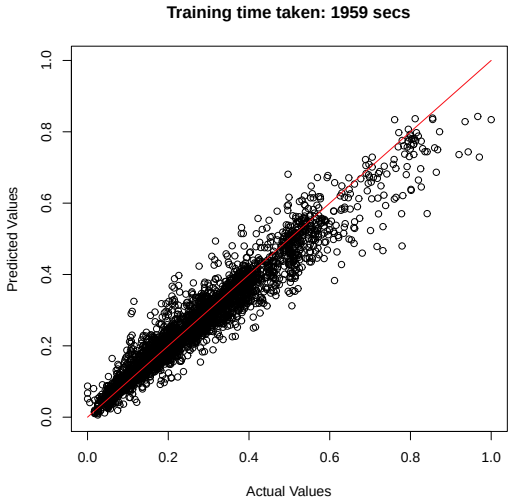
(h) Prediction Error, Decision Tree



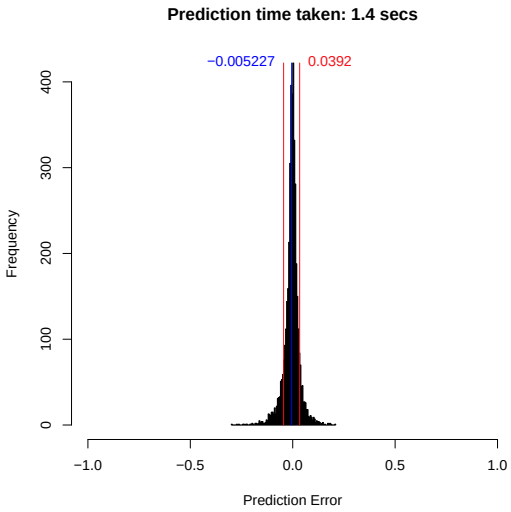
(i) Prediction Distribution, Random Forest



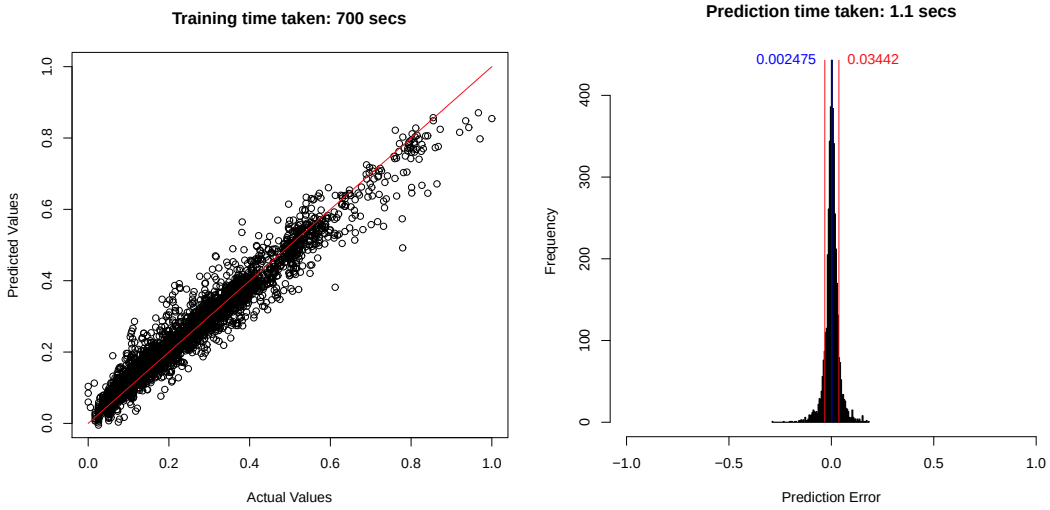
(j) Prediction Error, Random Forest



(k) Prediction Distribution, Gaussian Processes



(l) Prediction Error, Gaussian Processes

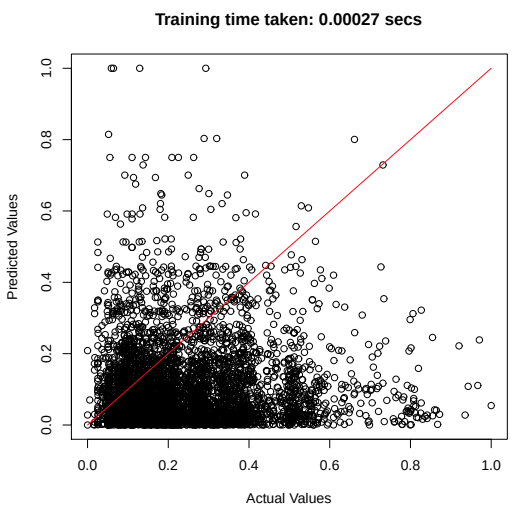


(m) Prediction Distribution, SVM

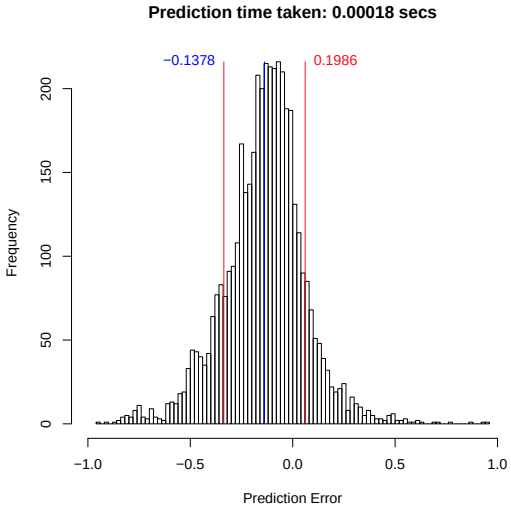
(n) Prediction Error, SVM

Figure 6.2: Graphs showing the performance of various statistical machine learning techniques in predicting the run-times of known programs on unseen hardware designs.

Predicting energy of known programs on new designs
Figure 6.3

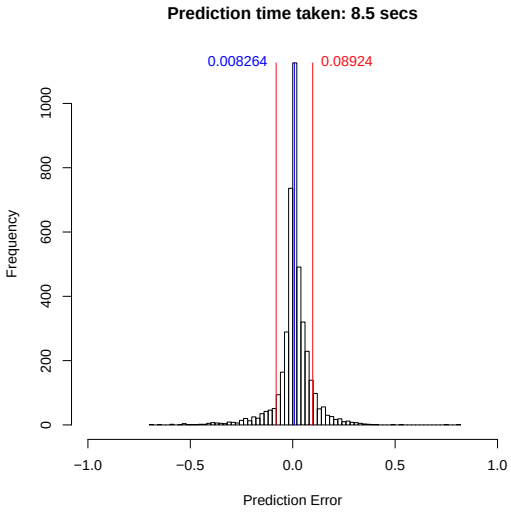


(a) Prediction Distribution, Empirical Random

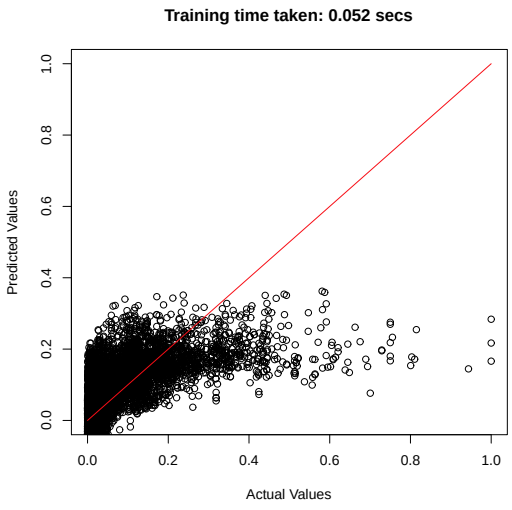


(b) Prediction Error, Empirical Random

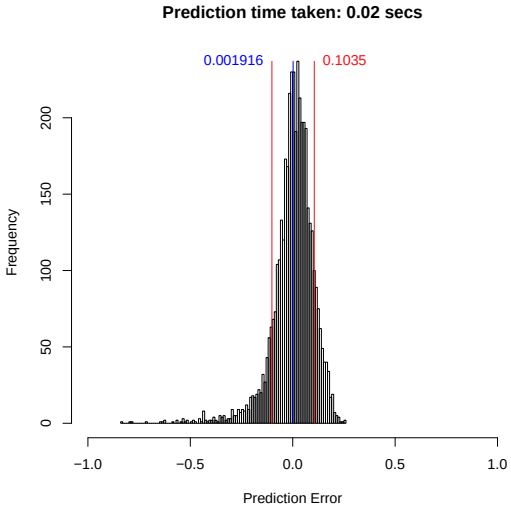
(c) Prediction Distribution, *wk*NN



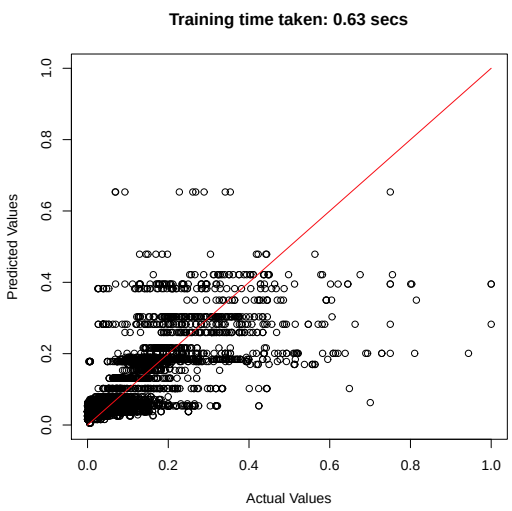
(d) Prediction Error, *wk*NN



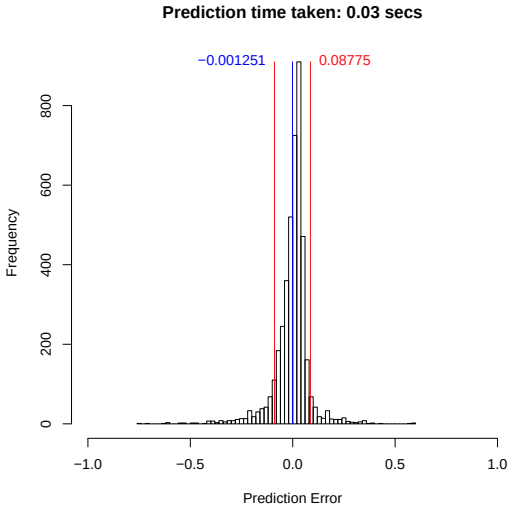
(e) Prediction Distribution, Linear Regression



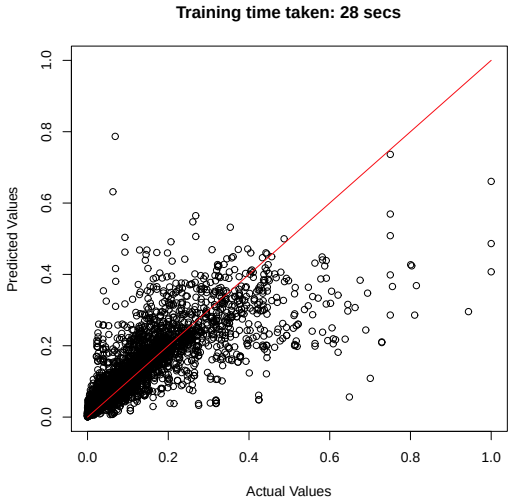
(f) Prediction Error, Linear Regression



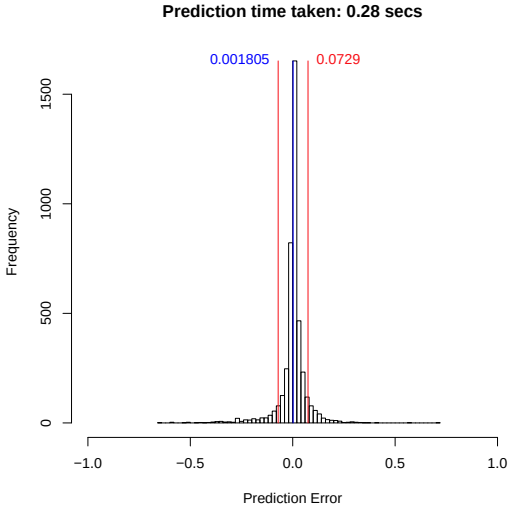
(g) Prediction Distribution, Decision Tree



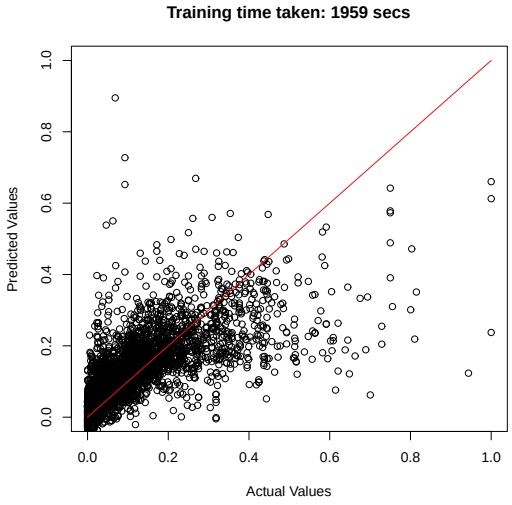
(h) Prediction Error, Decision Tree



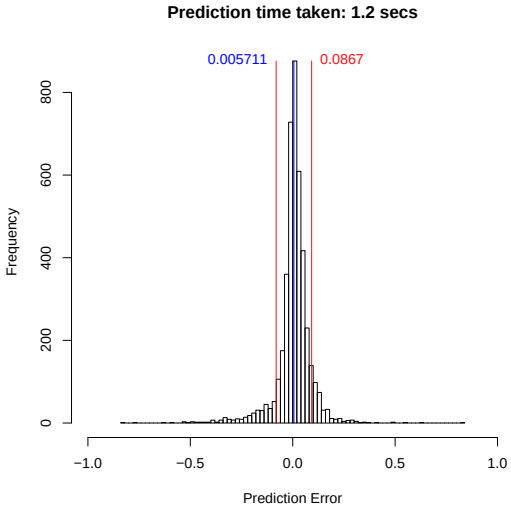
(i) Prediction Distribution, Random Forest



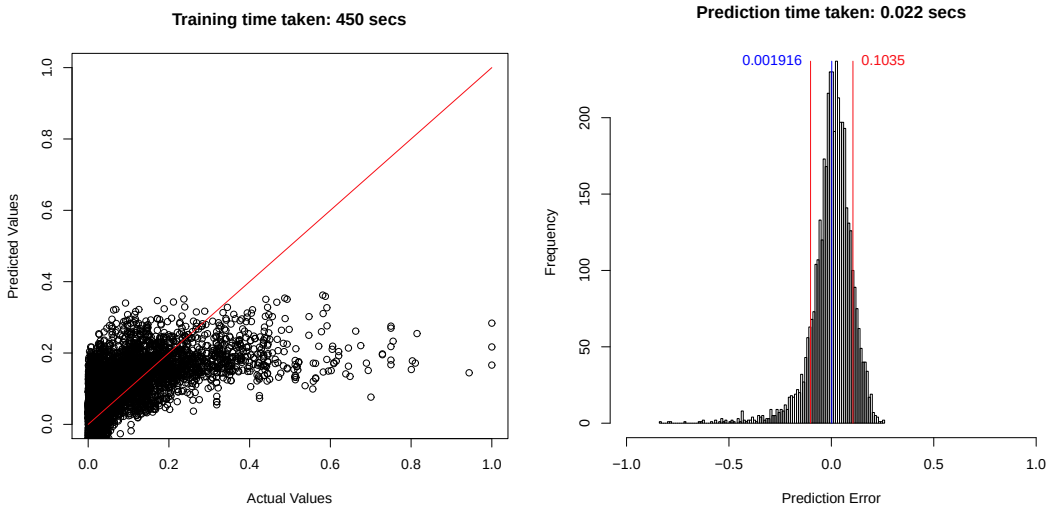
(j) Prediction Error, Random Forest



(k) Prediction Distribution, Gaussian Process



(l) Prediction Error, Gaussian Process

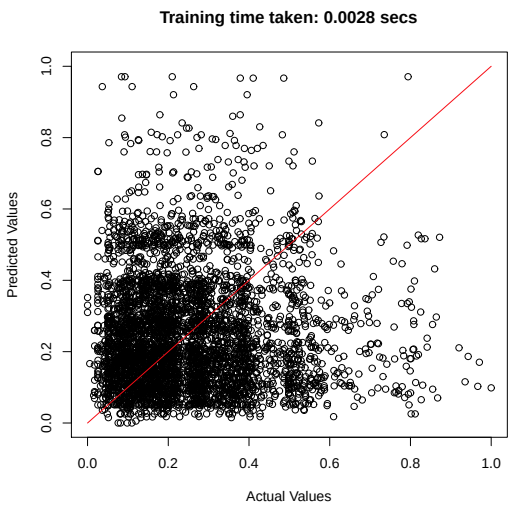


(m) Prediction Distribution, SVM

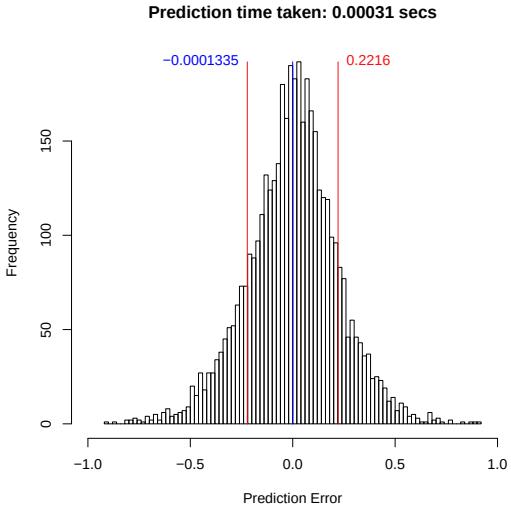
(n) Prediction Error, SVM

Figure 6.3: Graphs showing the performance of various statistical machine learning techniques in predicting the total energy consumption of known programs executing on unseen hardware designs.

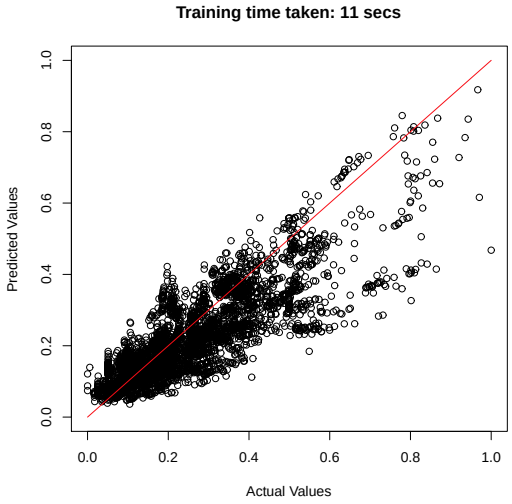
Predicting runtime of unknown programs on seen designs Figure 6.4



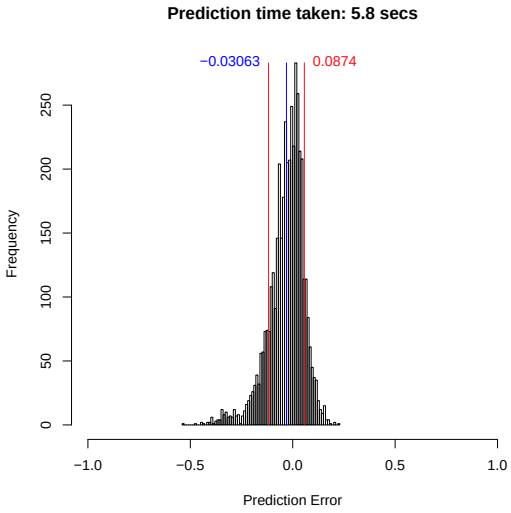
(a) Prediction Distribution, Empirical Random



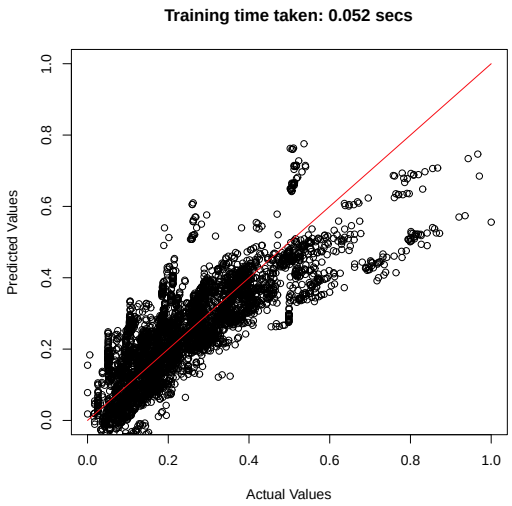
(b) Prediction Error, Empirical Random



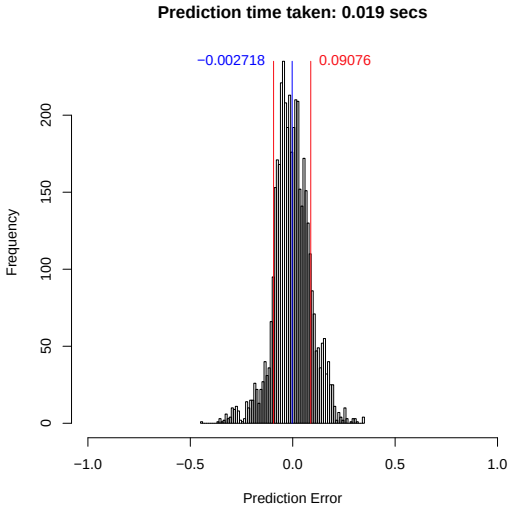
(c) Prediction Distribution, *wk*NN



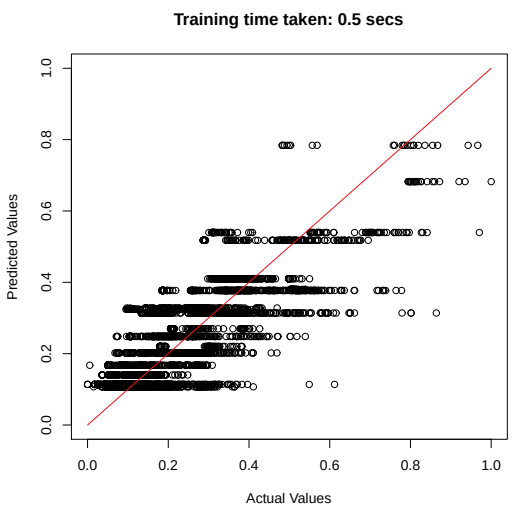
(d) Prediction Error, *wk*NN



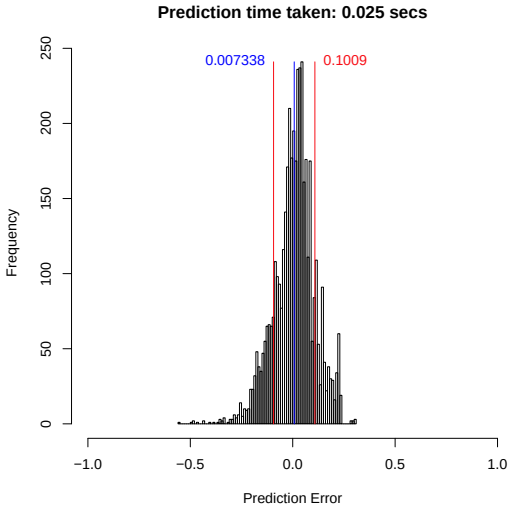
(e) Prediction Distribution, Linear Regression



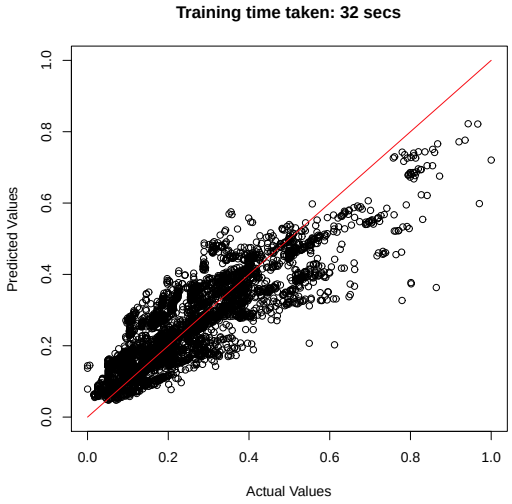
(f) Prediction Error, Linear Regression



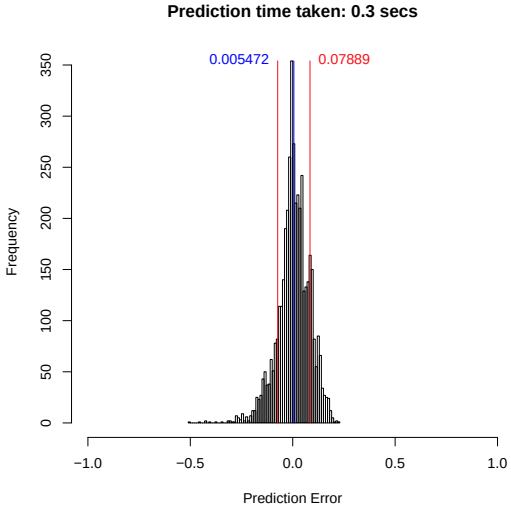
(g) Prediction Distribution, Decision Tree



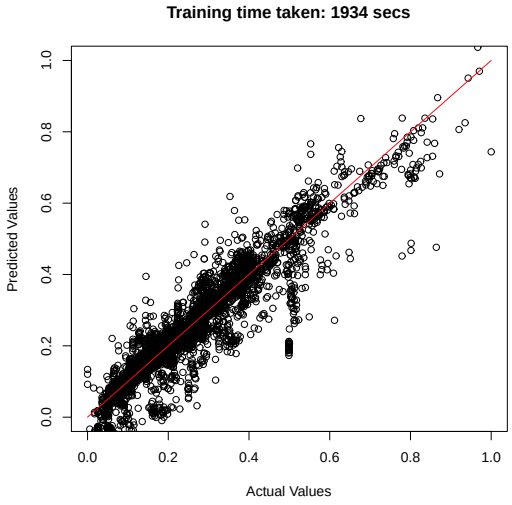
(h) Prediction Error, Decision Tree



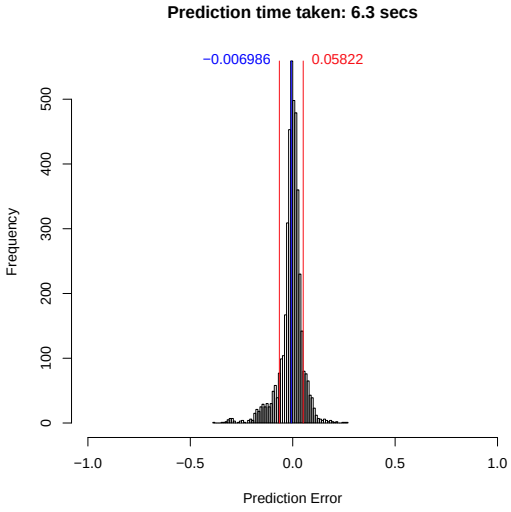
(i) Prediction Distribution, Random Forest



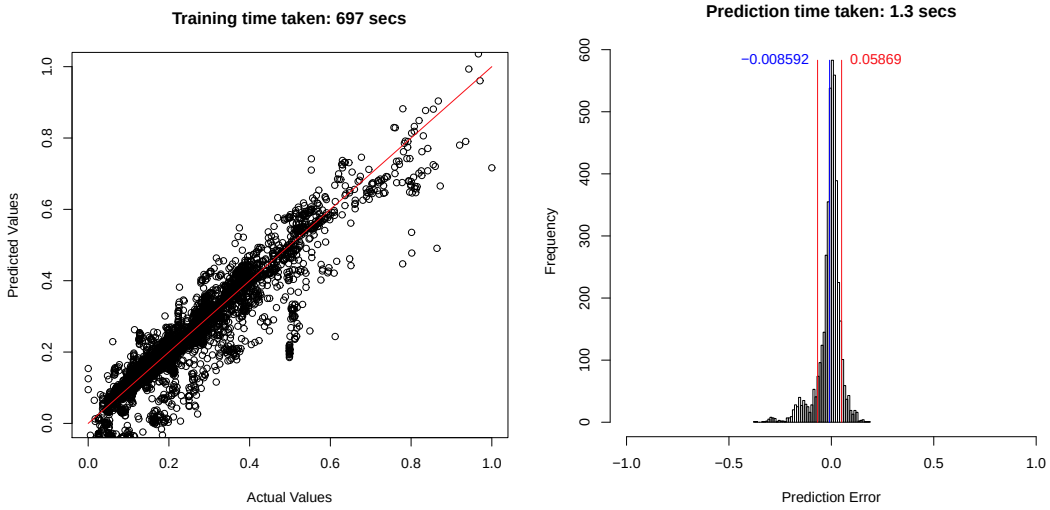
(j) Prediction Error, Random Forest



(k) Prediction Distribution, Gaussian Processes



(l) Prediction Error, Gaussian Processes

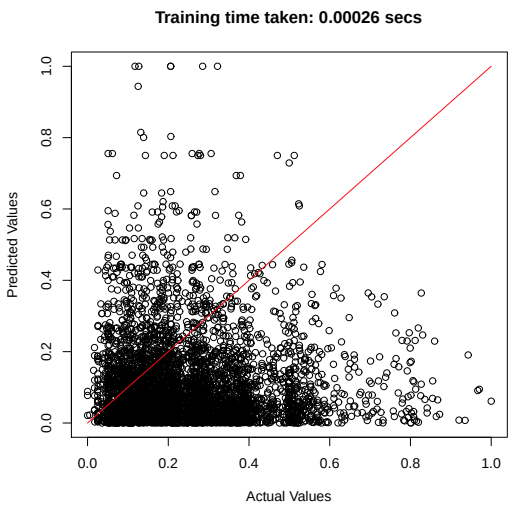


(m) Prediction Distribution, SVM

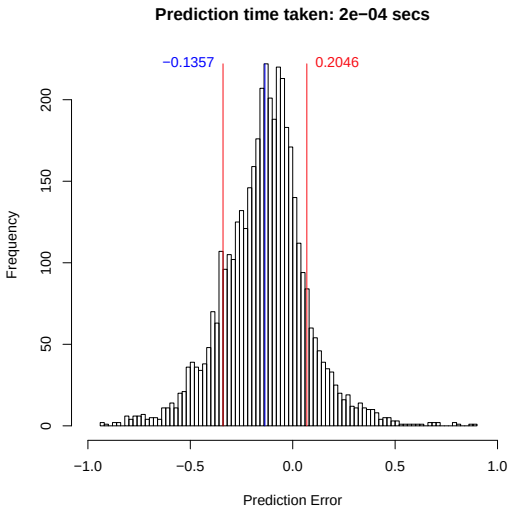
(n) Prediction Error, SVM

Figure 6.4: Graphs showing the performance of various statistical machine learning techniques in predicting the run-times of new programs on seen hardware designs.

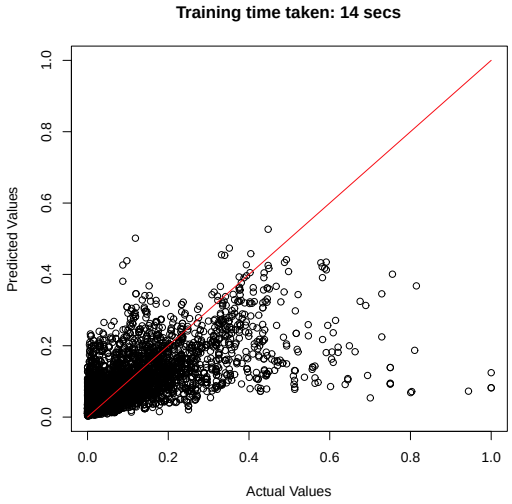
Predicting energy of unknown programs on seen designs
Figure 6.5



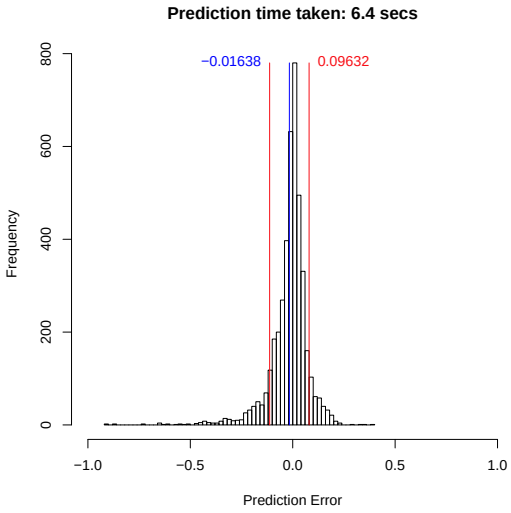
(a) Prediction Distribution, Empirical Random



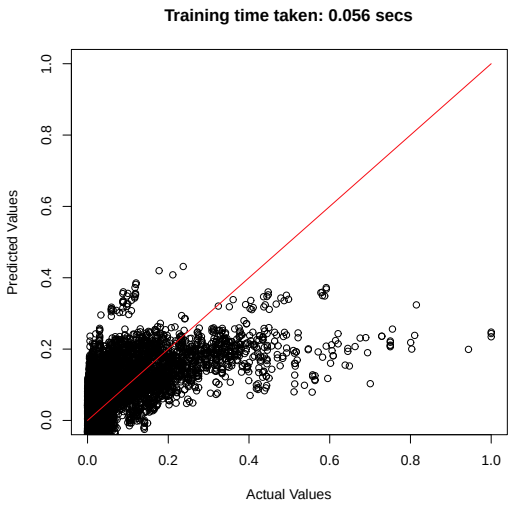
(b) Prediction Error, Empirical Random



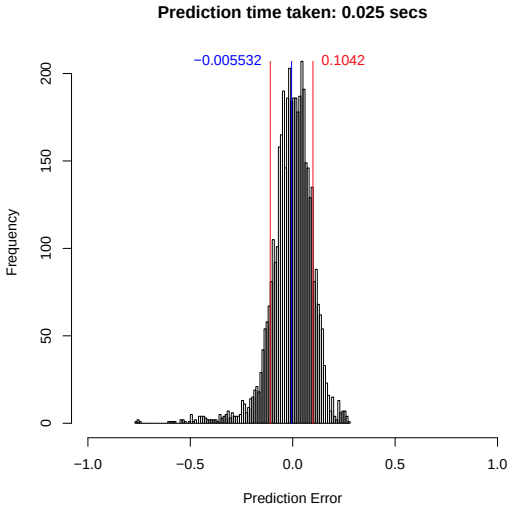
(c) Prediction Distribution, *wk*NN



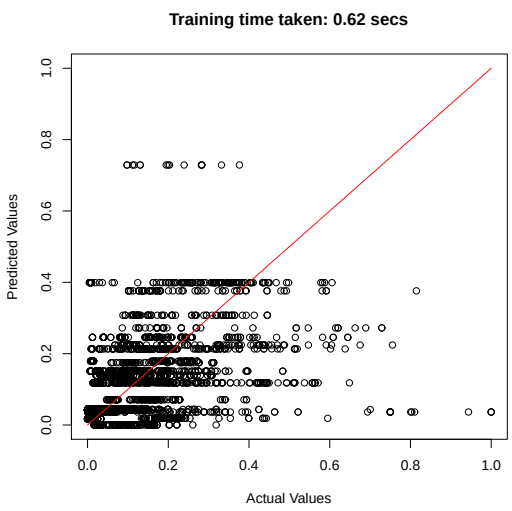
(d) Prediction Error, *wk*NN



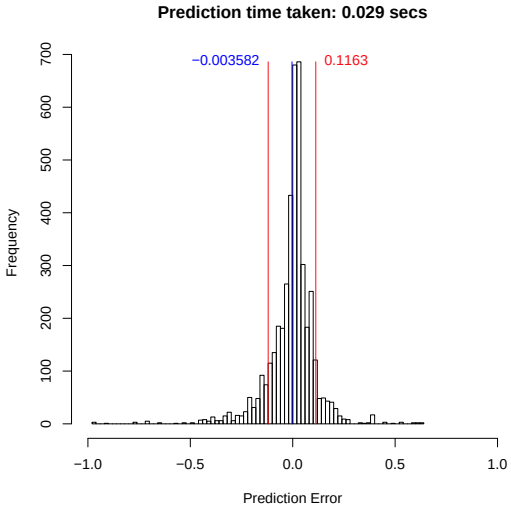
(e) Prediction Distribution, Linear Regression



(f) Prediction Error, Linear Regression

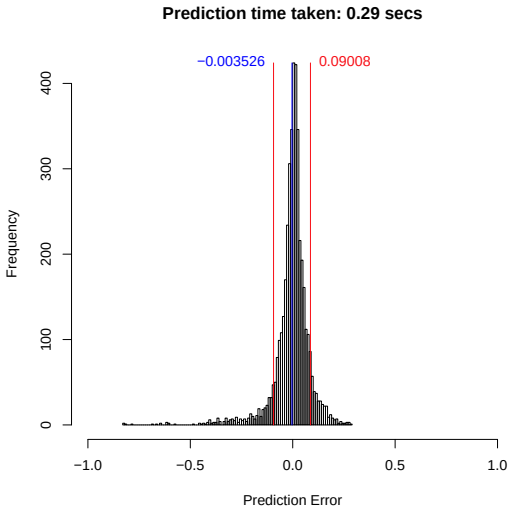


(g) Prediction Distribution, Decision Tree

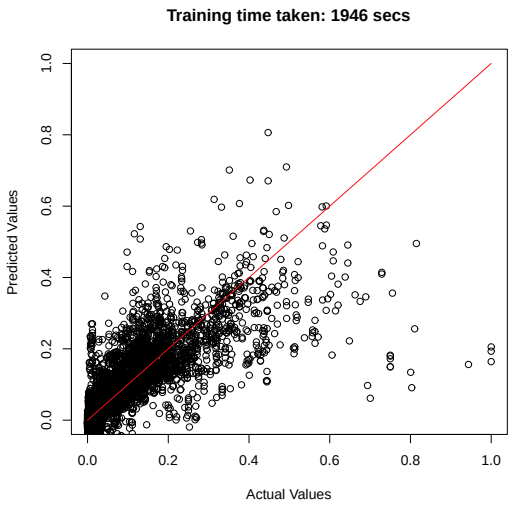


(h) Prediction Error, Decision Tree

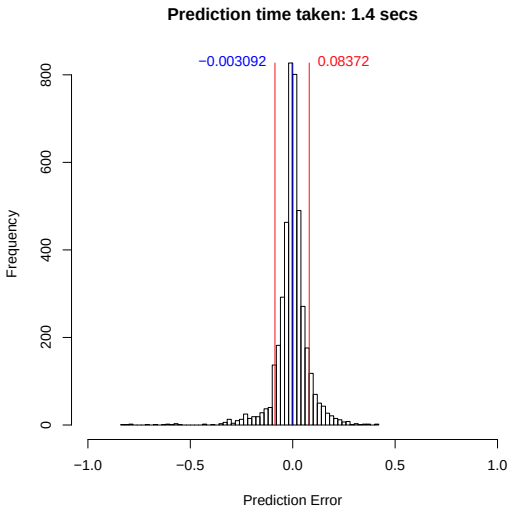
(i) Prediction Distribution, Random Forest



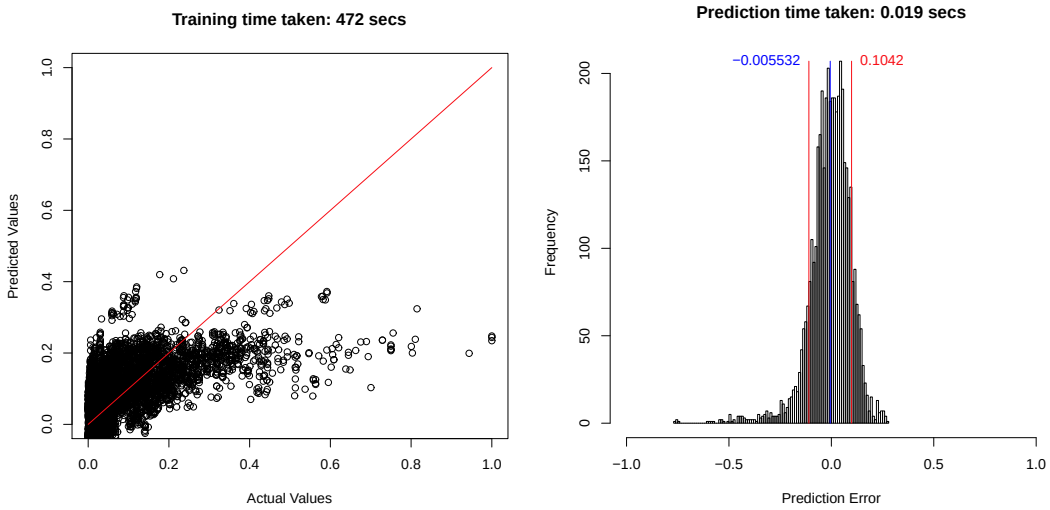
(j) Prediction Error, Random Forest



(k) Prediction Distribution, Gaussian Process



(l) Prediction Error, Gaussian Process

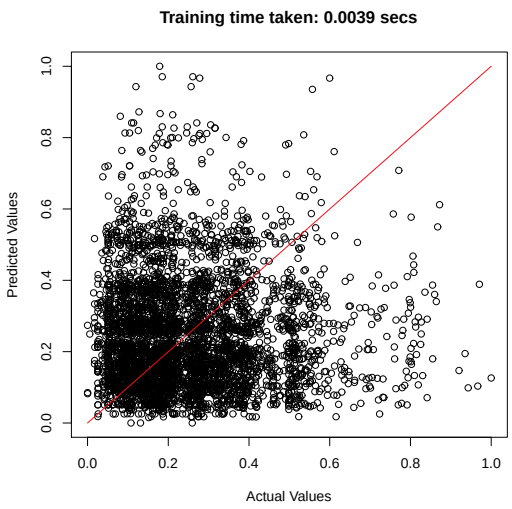


(m) Prediction Distribution, SVM

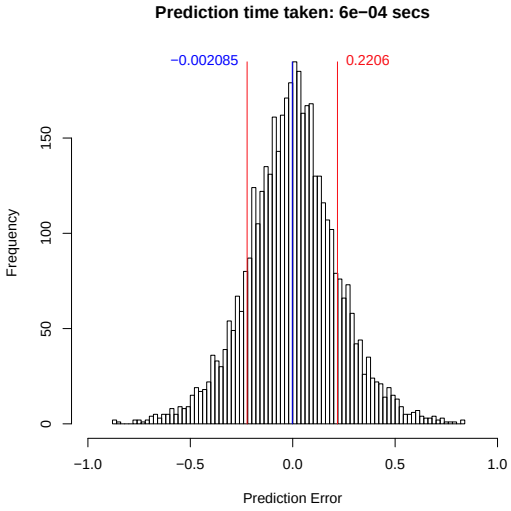
(n) Prediction Error, SVM

Figure 6.5: Graphs showing the performance of various statistical machine learning techniques in predicting the total energy consumption of new programs executing on seen hardware designs.

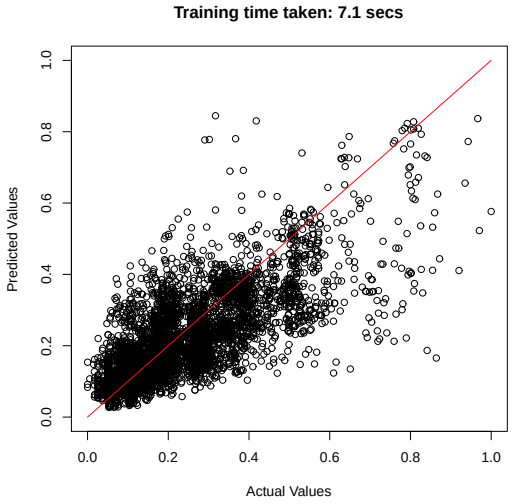
Predicting runtime of unknown programs on new designs
Figure 6.6



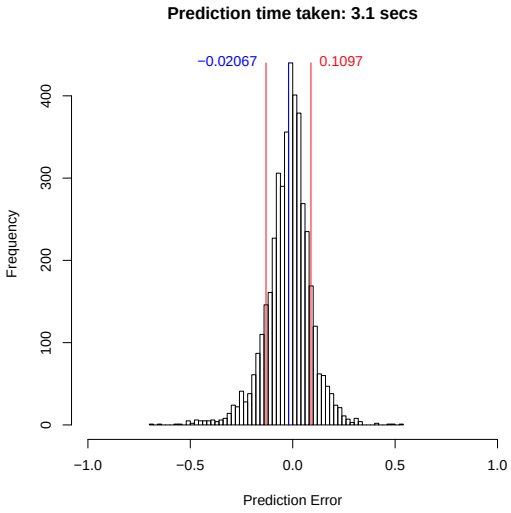
(a) Prediction Distribution, Empirical Random



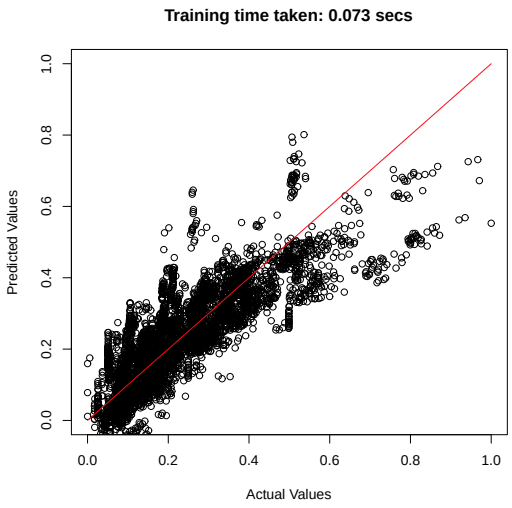
(b) Prediction Error, Empirical Random



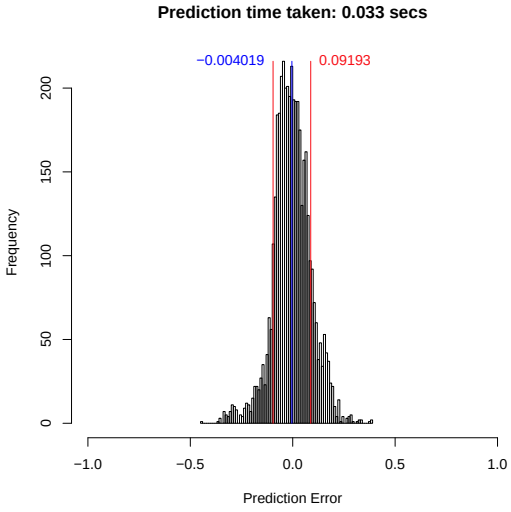
(c) Prediction Distribution, *wk*NN



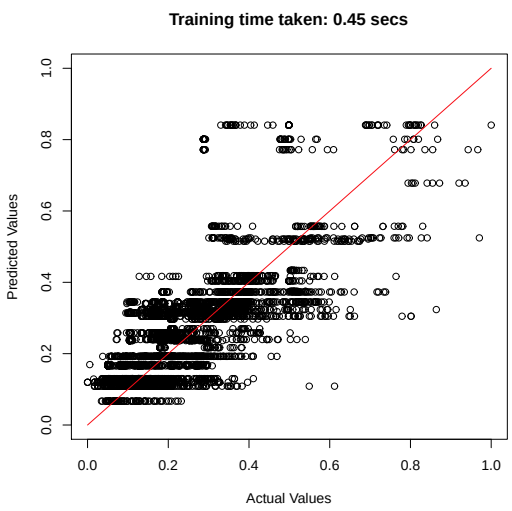
(d) Prediction Error, *wk*NN



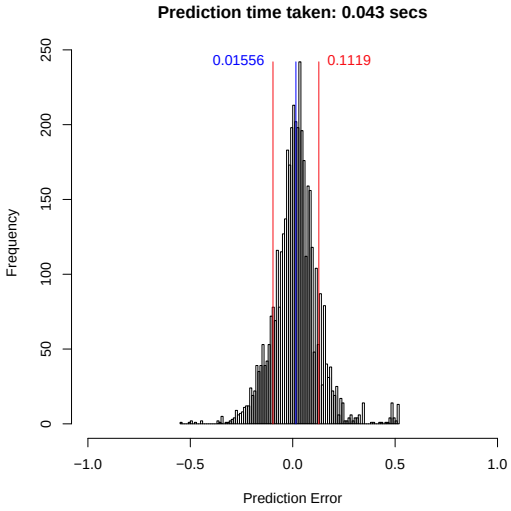
(e) Prediction Distribution, Linear Regression



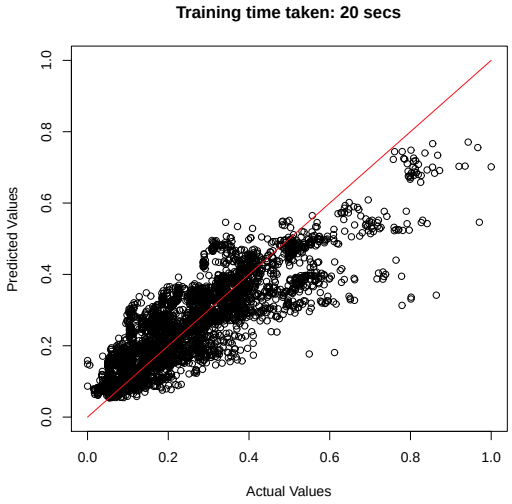
(f) Prediction Error, Linear Regression



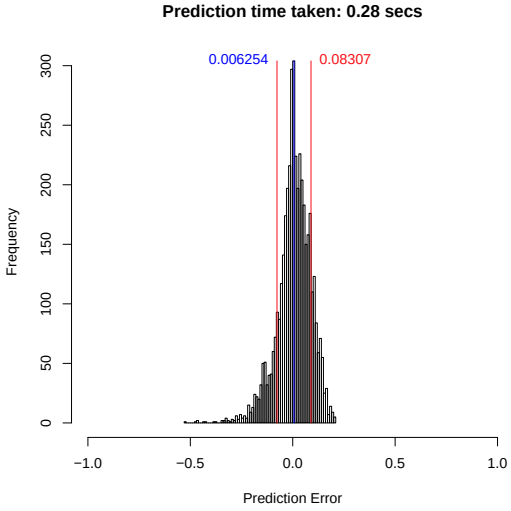
(g) Prediction Distribution, Decision Tree



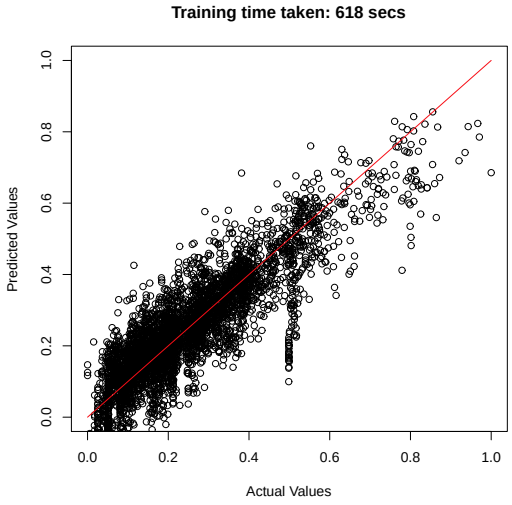
(h) Prediction Error, Decision Tree



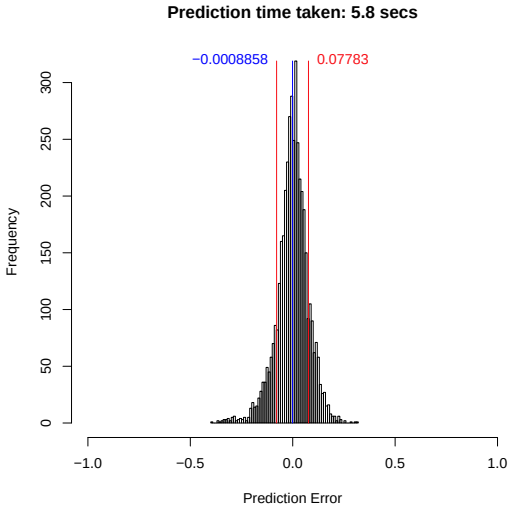
(i) Prediction Distribution, Random Forest



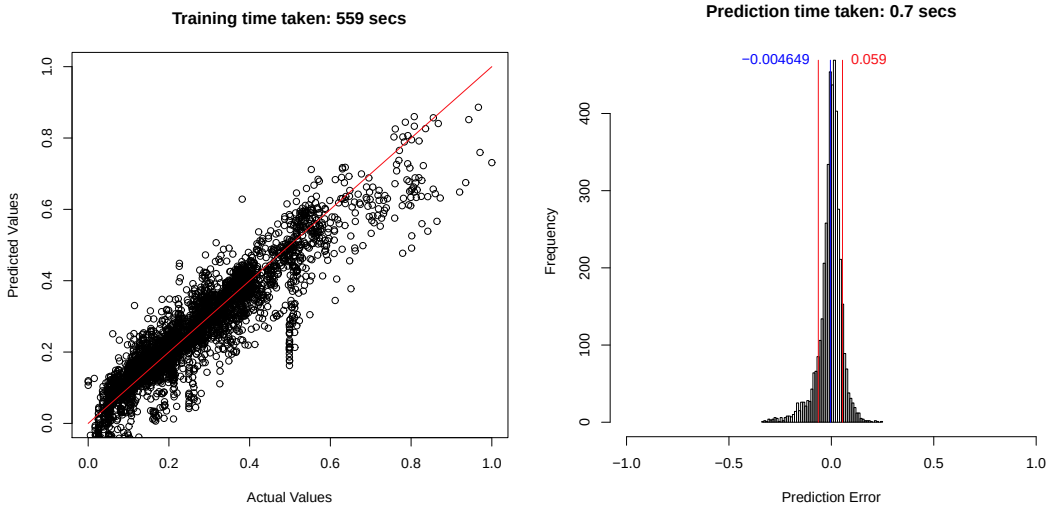
(j) Prediction Error, Random Forest



(k) Prediction Distribution, Gaussian Processes



(l) Prediction Error, Gaussian Processes

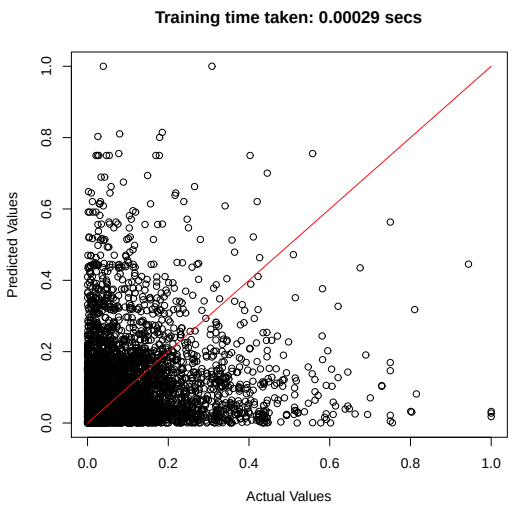


(m) Prediction Distribution, SVM

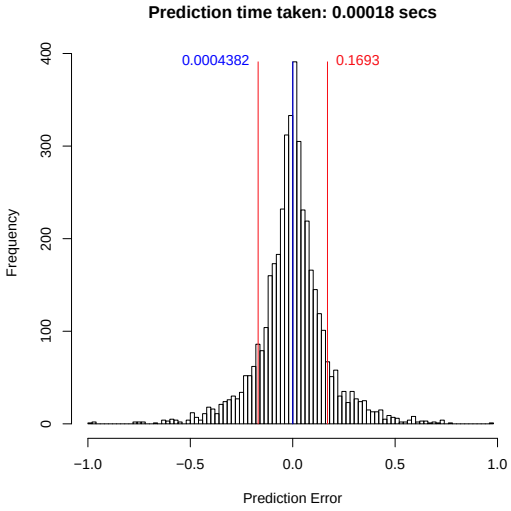
(n) Prediction Error, SVM

Figure 6.6: Graphs showing the performance of various statistical machine learning techniques in predicting the run-times of unknown programs on unseen hardware designs.

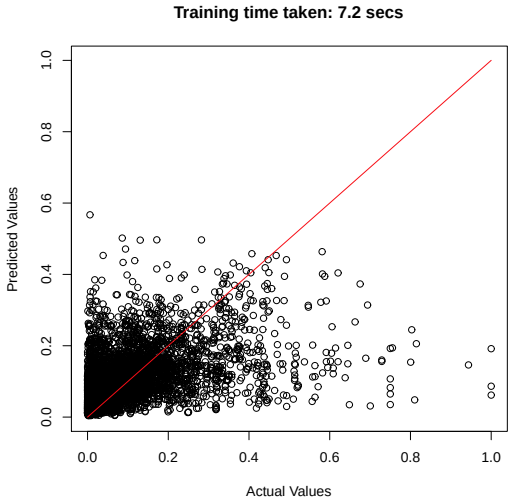
Predicting energy of unknown programs on new designs
Figure 6.7



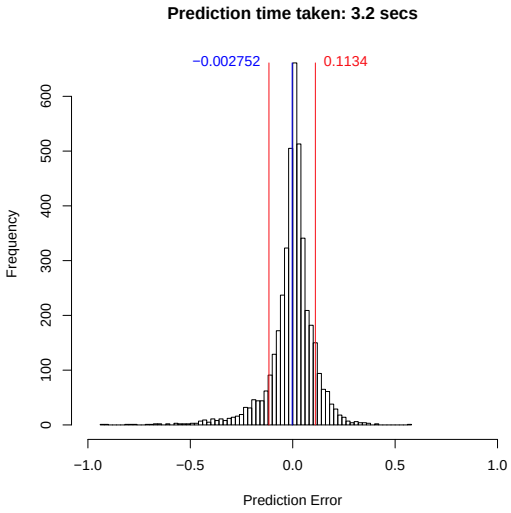
(a) Prediction Distribution, Empirical Random



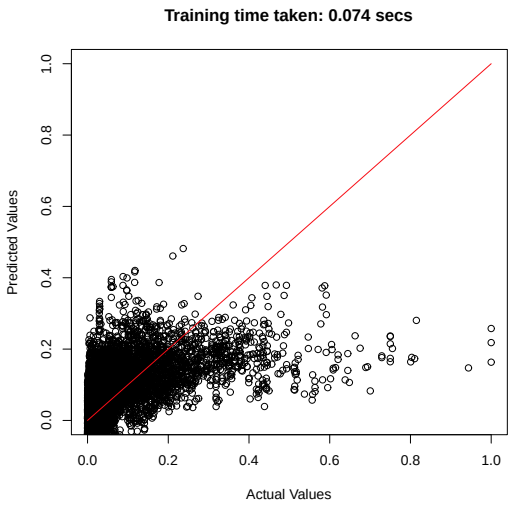
(b) Prediction Error, Empirical Random



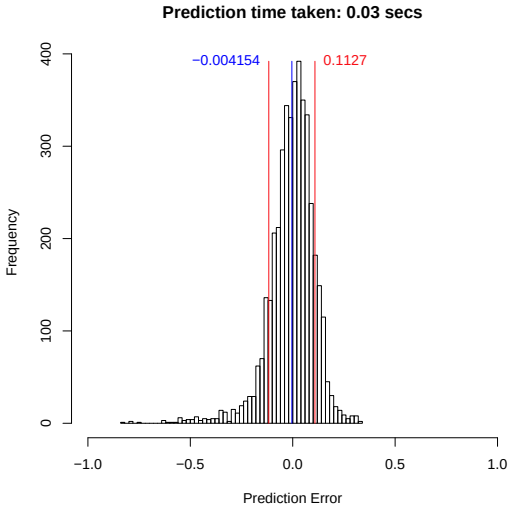
(c) Prediction Distribution, *wk*NN



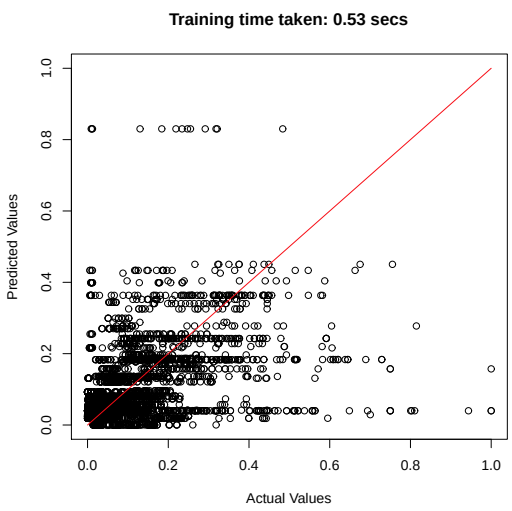
(d) Prediction Error, *wk*NN



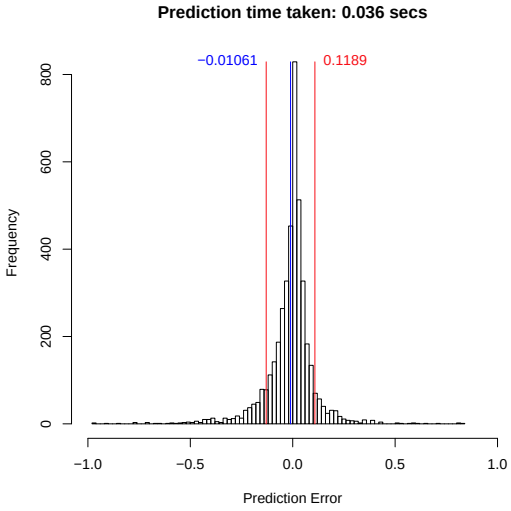
(e) Prediction Distribution, Linear Regression



(f) Prediction Error, Linear Regression

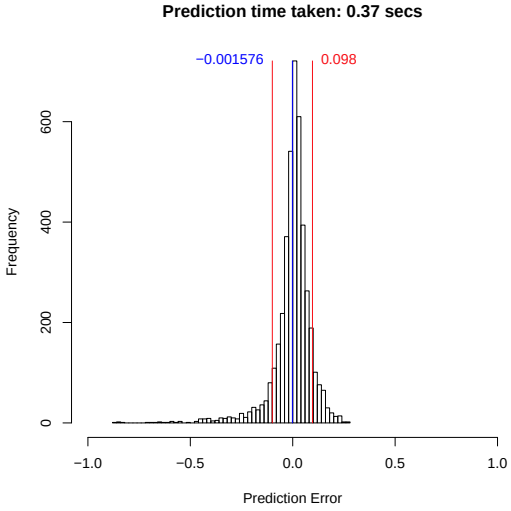


(g) Prediction Distribution, Decision Tree

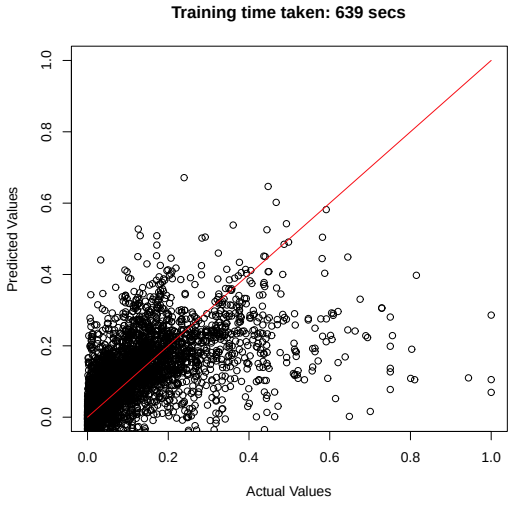


(h) Prediction Error, Decision Tree

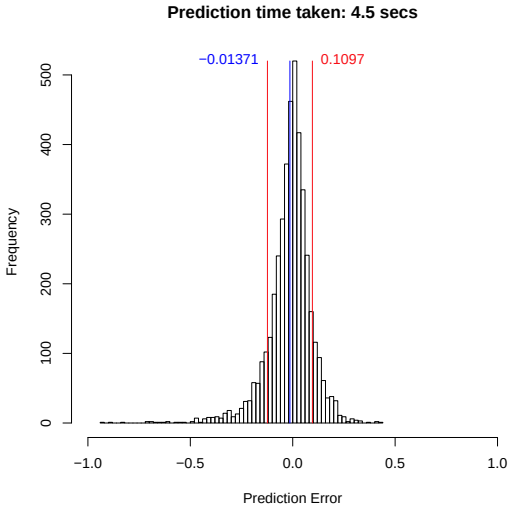
(i) Prediction Distribution, Random Forest



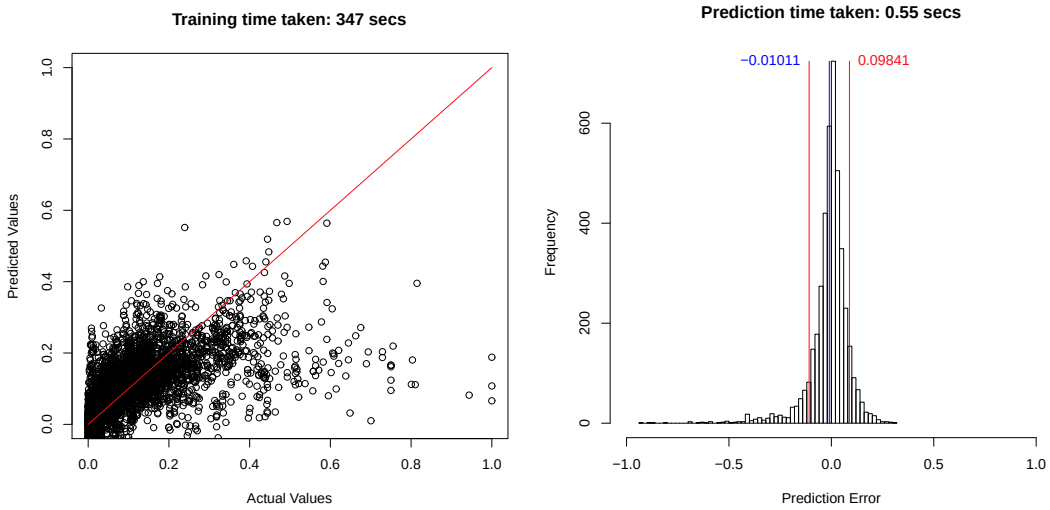
(j) Prediction Error, Random Forest



(k) Prediction Distribution, Gaussian Process



(l) Prediction Error, Gaussian Process



(m) Prediction Distribution, SVM

(n) Prediction Error, SVM

Figure 6.7: Graphs showing the performance of various statistical machine learning techniques in predicting the total energy consumption of unknown programs executing on unseen hardware designs.

7 Summary and conclusion

This thesis has presented methodology, experiments and results of investigations into using machine learning on the design space of interconnects in multi-core systems. This novel approach to managing the tradeoffs in this field has not been previously attempted, and this thesis shows that it is suitable and appropriate. This chapter contains a short summary of the thesis in section 7.1, a discussion of the applicability of the method in an industrial setting in 7.2 the overall conclusions in section 7.3, and a indication of further work that is possible in 7.4.

7.1 Summary

These contributions in this thesis are as follows.

1. Using complete or limited design space exploration over an interconnect design space it is possible to gain understanding of the performance trade-offs in a given implementation.
   (a) A NOC implementation can be varied in ways that generate potentially very large design spaces.
   (b) The design space exploration does not need to be exhaustive to find good designs.
   (c) Good points are generally close by in some dimensions of the design space.
2. By using machine learning on the design space data, predicting points that are optimal in some regard becomes feasible.
   (a) Evaluating several machine-learning and statistical methods, suitable methods to apply in this field can be found.
   (b) Accurate predictions of optimal points in the design space for previously unseen applications can be achieved.
   (c) Accurate predictions the energy and runtime performance of a known application on a new design can be achieved.
   (d) Accurate predictions the energy and runtime performance of an unknown application on a known design can be achieved.

(e) In the presence of design system faults, the level of functionality that is obtained in a new design of this system can be predicted.

3. These methods can be used to significantly reduce search time for new optimal designs in the design space.

   (a) The method is scalable to large design spaces and also handles sparse exploration of these design spaces.

4. Using real synthesisable hardware and not simulations or analytical methods allows for deeper evaluation.

   (a) By using standard benchmark programs, synthesisable hardware designs and appropriate synthesis flow the approach is suitable for the field.

   (b) By using standard benchmark programs, synthesisable hardware designs and appropriate synthesis flow the accuracy of the metrics can be ensured.

5. This is the first time machine learning methods have been applied to the design space of synthesisable SOC designs.

Chapter 3 presented a comprehensive and thorough view of several types of SOC designs, including a NOC type not commonly used. These systems are parameterisable, allowing for a large number of configurations, and synthesisable, which allows for real measurements to be made. In addition, these systems run high-level code, also presented in chapter 2, that while low level allows for the use of standard benchmarks. This chapter shows how contribution 1a), 4a) and 4b) have been accomplished.

Chapter 4 discusses the experiments performed on the first experimental system, which had a design parameter space totalling 280 SOC implementations. As this relatively small, the full dataset could be evaluated with some effort. The results of the experiments on the fully evaluated data set showed that machine learning was usable and effective on the design spaces presented in embedded systems. A version of *wk*NN was able to predict which design to use for a given application with high accuracy, minimizing runtime performance, dynamic energy and ED product. This chapter shows how contribution 1b), 1c), 2b), 2c) and 3) have been accomplished.

Chapter 5 expands substantially from chapter 4 in both system scale and machine learning methodology. This chapter uses another type of experimental system with a larger parameter space of some 510,000 possible designs. Because of this large design space it was not possible to fully or exhaustively profile the design space, so a random sample of design points were used. This chapter then proceeds to use *wk*NN and SVM methods to find the optimal designs within this dataset for new applications, evaluating the performance of these predictors in terms of how close to the optimal choice

they achieved. Again, the runtime, dynamic energy and ED optimisation goals are used, and it is found that SVM is a better predictor overall for this space than *wk*NN. This chapter shows how contribution 1b), 2a), 2b), 2c), and 3) have been accomplished.

Chapter 6 further uses the dataset from chapter 5 to find new optimal points outside of the explored area of the design space. This chapter focuses on identifying trends in the larger dataset, and again uses a larger selection of machine learning methods. First, predicting for functionality of new architectural options is covered, achieving a accuracy of 77%. Secondly, predicting the runtime and energy metrics of new SOC architecture, new programs, and new programs with new SOC architecture is covered. Across these experiments, random forest proves to be a generally valuable method, closely followed by SVM and Gaussian Processes. It is shown conclusively that the time taken in training and prediction is much smaller than the time taken in testing even a single new design. This chapter shows how contribution 1b), 2a-e) and 3) have been accomplished.

7.2 Applicability

This thesis has presented a novel approach to evaluating the NOC of a SOC design via machine learning methods. This section discusses applying this method in an industrial setting, and what changes would be necessary to the method presented for it to be efficient in such a setting.

The method relies on gathering learning data from a number of training instances, essentially disparate points in a design space encompassing the design choices in a SOC. This relies on having a almost-completed SOC architecture from which test instances can be drawn; alternatively, it relies on early design-space exploration of the intended architecture. Early design-space exploration of possible architectures is commonly performed, with various methodologies and at different depths of abstraction. Almost universally the abstraction level is too high for the use of machine learning methods, as the noise and uncertainty of these abstraction methods overshadow the trends of interest; they are effectively lost in abstraction noise. Attempting to use this method with a high-level abstraction design exploration scheme would be an interesting extension. Conversely, using this method with a almost finished parameterisable SOC design has the drawback that most of the SOC hardware needs to be present and in working order before a design-space exploration can take place. Specifically, the parts of the system must be parameterisable in a meaningful way so as to enable exploration of the the design space tradeoffs desired.

Another concern is that the machine learning methods need a large enough train-

ing set to be able to formulate a coherent picture of the design space. This thesis has shown that even when only covering 0.1% of the design space, the machine learning methods can arrive at solutions with high accuracy. This 0.1% is however still 512 separate design points, each of which needs synthesis, testing, and evaluation. The synthesis, testing and evaluation process can be automated, but is still likely to take a long time. The synthesis of a single design of this test run took up to four CPU-hours and a not insignificant amount of RAM, and testing takes a number of minutes per test depending on the system performance. Overall, then, constructing the training cases can take a long time before the predictor generates the high accuracies presented in this thesis. The presented method is still significantly faster than a full design-space evaluation.

In addition, as a design proceeds through completion, it may be possible to implement limited test systems and use the training data from these as parts of the training set; some runs with full system options are still required, it may be possible to reduce the amount of such runs. The part-completed designs would be represented as feature vectors with some options set to 0 to indicate the missing or unexplored dimensions, thus enabling these to be filled in later. Thus completing the training set may take a shorter time.

Given the long times of synthesis, testing and evaluation, using even the slow evaluation steps of Gaussian Processes and SVM is warranted, as they are still an order of magnitude faster than even a single test run. In addition, the predictors can be inserted into a search loop such that prediction results, predicting the 10 best designs for a given application, are fed to a synthesis and evaluation stage, which then forwards the results back to the training set, thus enabling a further improvement and refinement in the predictions. Such a search system could operate autonomically for a number of iterations, given sufficient automation, and produce highly optimised designs.

Another way to utilise the presented approach is in application-specific adaptation of a hardware system in the design phase. If the hardware system components are available, but there are several use cases for the completed system, then application specific optimisation may be desired. The presented approach can easily accommodate this scenario, by using a shared learning set of designs and specialising for each application. Learning the best system design for one application may then feed back several non-optimal design points to the learning set, enhancing further learning tasks with the same overall SOC meta-design. Thus the same components can be effectively and automatically rearranged and reparameterised to fit the application with minimum human involvement, and this process may also benefit several product lines at

once, if the SOC designs are all derived from the same meta system design. Finding a new optimal design to run a given, new application on can take as little a single run of the application on a specifically implemented instance of the meta-SOC design.

The effort required to bring the methods presented in this thesis into an industrial setting is medium to high, as it would need to be integrated with the existing system design processes and modified to use the feature vectors available in the design spaces that are relevant. However, as the experiments in this thesis have been performed using many of the techniques that would be required in such a setting, it is by no means infeasible to do so. The current experiment management software would either need to be rewritten or substantially refactored; the test system evaluation and programming parts would likely need to be improved, and the machine learning flow would need to be integrated in the evaluation flow. Some engineering effort would need to be expended to build an application that would perform the method in an industrial setting, but there are no theoretical hindrances to such a development.

7.3 Conclusion

The experiments presented in Chapters 4, 5 and 6 share several features; importantly, they are using runtime and energy data from a realistic SOC design. This parameterisable design was discussed in detail in Chapter 3, demonstrating that the SOC design chosen has several axes of freedom that can be used to optimise a given design, and importantly, that the SOCs generated are synthesisable and manufacturable. The runtimes extracted from the instances of this SOC design represent full runs of industry standard benchmarks, including start-up times, cache warm-up, real memory and interconnect latencies, and scheduler and runtime system overheads in the form of system call delays. Therefore, resultant data in terms of run-times and switching counts are representative of the workloads in a real SOC system, and the data collected gives an accurate picture of the real performance of the SOC system.

This data is then used to experiment with several aspects of machine learning over the design space represented. Chapter 4 is mainly concerned with three questions; first, showing that the design space is complicated and non-intuitive; secondly, whether using machine learning on this type of data will work and thirdly, how well it will work using a simple method. The complexity of the design space is demonstrated in Figures 4.2(a) and 4.2(b), plotting Performance score versus dynamic energy for the design points. These graphs show the performance tradeoffs in the range of designs, and how there are trends for a given application, but these trends do not necessarily generalise across applications; different applications therefore have different

optimal points. The experiments in this chapter demonstrate that the machine learning approach is suitable, as the *wk*NN method employed managed to on average find a design within 90% of the performance of the optimal design, whether optimising for energy, runtime, or a combination. This demonstrate both that machine learning works over the design space, and that it can work well. The *wk*NN method used is simple to understand, but does not have as much predictive power as some of the more advanced methods tried in later chapters; even so, it manages well on the data and shows the power of the method presented in this thesis.

Chapter 5 answers questions about the scalability of the novel methodology, as well as introducing SVM to the problem as a alternative to *wk*NN. The scalability of the approach is demonstrated through the use of a significantly larger design space (510,000 design points instead of 280) generated through he use of a larger FPGA device and more SOC cores. Consequently, the number of benchmark programs also increased from 5 to 70, resulting in a larger training set as well as in increased demands on the predictors. Figure 5.6 demonstrated the spread in the design space, showing that no single design is absolutely optimal across the range of designs and optimisation goals. Application-specific selection and optimisation will always produce a faster, more power efficient, or both, solution than a monolithic design choice in this design space. Due to the size of the design space, a subset of the available designs were used for experiments, and this chapters experiments covered predictions within this space so as to enable absolute performance comparisons. The experiments here show that even with the substantial changes from Chapter 4, the predictive methods perform well on this dataset, achieving better than 80% average performance when predicting the best design in the data set. Additional experiments in this chapter identify the designs that have the best predictive power, and which should therefor be used for testing new software on in order to render a prediction of the optimal design with high accuracy. This is useful for finding which design to run a new program on in order to locate the best SOC design for that program with as few program runs as possible. The results here indicate that the best design for a given design can be found with a single run of a new program on the design with the most predictive power, enabling a single-step SOC optimisation technique.

The design space generated in Chapter 5 was also used in chapter 6, but the goals here were different. The spread in the design space, and the fact that no design is optimal for all programs, thus remained as the same data set was used. Chapter 6 aimed to find the best machine learning method to use when predicting the metrics of new and untested designs and programs. This can then be used to find optimal designs, by predicting the metric of interest over the entire design space and then sorting by

the value, thus finding completely new designs that should perform well on the new application. Figures 6.1 through 6.7 show the results of these experiments in detail. A few things can be noted about these experiments and the performance of the machine learning approach; firstly, it can be highly accurate, as demonstrated by the 97% accuracy in predicting runtime. Secondly, the choice of machine learning method that works best varies somewhat with metric, but random forest backed by SVM do well. Thirdly, these experiments demonstrate the time savings and benefits of the method, as even the longest training was an order of magnitude shorter than the synthesis time of even a single design for a few thousand elements in the training set. Thus it is possible to use several methods in concert and still perform significantly better than empirical design-space exploration. Fourthly, prediction is demonstrated to be very quick after training, especially in the eager algorithms, and therefore predicting metrics even for an entire 510,000 element design space is feasible, making the method scalable.

In addition, some of the experiments in chapter 5, specifically the experiments in finding working designs, were repeated in chapter 6, using more machine learning methods and a more fine-grained approach. These experiments showed that predicting for functionality is more feasible than indicated in the previous experiments, with an accuracy of 77% from the given dataset.

The end result of the experiments in chapter 6 is that the machine learning approach taken in this thesis is validated as a fast, efficient, and accurate method of finding good system designs. The method is accurate not least because the training and evaluation data was derived from a synthesisable, parameterisable SOC design executing real programs, and thus implicitly taking into account the intricacies of real systems. In addition, specialisation of the system towards an application is accomplished simultaneously with finding a good system, as a good system is only relevant in conjunction with a given application.

The novel methods presented in this thesis have been confirmed as appropriate, accurate and efficient, and represent a practical new approach to finding optimal NOC and SOC designs.

7.4 Further work

The method presented in this thesis has been evaluated and shown to work appropriately, but it could be extended in several ways. Some of these approaches were considered for inclusion in the presented thesis, but were omitted due to complexity and the level of additional engineering required.

The experimental system still has relatively small size limits, and is a multi- rather than a many-core system. Future SOC designs are likely to feature many independent processing elements. The scalability of the method has not been tested beyond half a million design points and a few thousand feature vectors. Using a larger meta design system would allow for more investigations in this regard, and may uncover further efficiencies of the method.

The NOC topology used in this thesis is limited to several different tree-like butterfly topologies. Other topologies may have better properties for some applications, and including these topologies in the design space would be a worthwhile effort. For this, a way to distinguish the topologies as feature vector parameters is needed, but importantly, a large amount of engineering work to implement the new topologies in a synthesisable manner is required. Mesh topology is popular for NOC designs up to 16 cores, and is an obvious candidate for an alternative topology.

The experimental system also used homogeneous switching, where every single switch in the fabric was the same, and importantly, every single switch was either buffered or un-buffered. Some applications may benefit from an inhomogeneous switching fabric, as this would allow for differently and possibly more efficiently clocked and laid out designs. This would filter through to the application level as lower5 memory latency for equivalent clocking and as lower energy cost for transactions. The presented method should be able to take advantage of these improvements if they are implemented in the switch fabric. To evaluate the benefits of an inhomogeneous fabric, the experimental system would have to be enhanced to enable synthesis of such fabrics, and it would also need to be included in the feature vectors. Combining this with the topology variations suggested above is likely to be extremely challenging.

The processor cores in the experimental system are also completely homogeneous. The performance tradeoffs of an inhomogeneous set of processors would likely also be an interesting investigation. Different core clocks as well as different amounts of cache and even pipeline depths allow for different performance of any given core. The mapping of software tasks to processors then become an additional problem that will need to be taken into account by the method, and investigating how this could be accomplished could lead to some interesting work.

The applications and benchmarks used in this thesis are comprised of non-communicating tasks, due to the difficulty of generating different versions of communicating benchmarks. The method presented works well on these non-communicating tasks, and while there is no reason to suspect it will not work on communicating tasks, this has not been examined in detail. Communicating tasks also have greater interdependency,

making it even less likely that a given, random, system configuration will be optimal for the overall task. Developing and evaluating linked tasks using the presented machine learning methodology is a subject for further work.

Further, other machine learning methods may also be tried to find if they are comparable or better for this purpose. Neural networks, in particular, are omitted from the evaluation due to similarities to both SVM and Gaussian Processes. It could however be included in the evaluation with relatively little effort.

Bibliography

[1] Peter Alfke. FIFOs in virtex-5 FPGAs. Xilinx White Paper 333, March 2008. URL `http://www.xilinx.com/support/documentation/white_papers/wp333.pdf`. Referenced on pages x, 42, and 43.

[2] O Almer, R Bennett, I Böhm, A Murray, X Qu, M Zuluaga, B Franke, and N Topham. An end-to-end design flow for automated instruction set extension and complex instruction selection based on GCC. In *Proc. 1st Int. Workshop GROW*, pages 49–60, 2009. Referenced on page iii.

[3] O. Almer, I. Bohm, T.E. von Koch, B. Franke, S. Kyle, V. Seeker, C. Thompson, and N. Topham. Scalable multi-core simulation using parallel dynamic binary translation. In *Embedded Computer Systems (SAMOS), 2011 International Conference on*, pages 190–199. IEEE, 2011. Referenced on pages iii, 31, 57, and 64.

[4] O. Almer, N. Topham, and B. Franke. A Learning-Based Approach to the Automated Design of MPSoC Networks. *Architecture of Computing Systems-ARCS 2011*, pages 243–258, 2011. Referenced on pages iii, 6, and 82.

[5] Oscar Almer, Miles Gould, Björn Franke, and Nigel Topham. Selecting the optimal system: automated design of application-specific systems-on-chip. In *Proceedings of the 4th International Workshop on Network on Chip Architectures*, NoCArc '11, pages 43–50, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0947-9. URL `http://doi.acm.org/10.1145/2076501.2076510`. Referenced on pages iii, 6, and 99.

[6] Federico Angiolini, Paolo Meloni, Salvatore Carta, Luca Benini, and Luigi Raffo. Contrasting a NoC and a traditional interconnect fabric with layout awareness. In *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, pages 124–129, 2006. Referenced on page 14.

[7] *AMBA AXI Protocol Specification*. ARM, March 2004. Referenced on pages 15 and 49.

[8] ARM. ARM products, March 2011. http://www.arm.com/. Referenced on page 19.

[9] ARM. Cortex-A7 processor, January 2012. URL http://www.arm.com/products/processors/cortex-a/cortex-a7.php. Referenced on page 3.

[10] James Balfour and William J. Dally. Design tradeoffs for tiled cmp on-chip networks. In *Proceedings of the 20th annual international conference on Supercomputing*, ICS '06, pages 187–198, New York, NY, USA, 2006. ACM. ISBN 1-59593-282-8. doi: 10.1145/1183401.1183430. URL http://doi.acm.org/10.1145/1183401.1183430. Referenced on page 14.

[11] A. Banerjee, R. Mullins, and S. Moore. A power and energy exploration of network-on-chip architectures. In *Networks-on-Chip, 2007. NOCS 2007. First International Symposium on*, pages 163 –172, may 2007. Referenced on page 15.

[12] Richard Vincent Bennett. *Increasing the Efficacy of Automated Instruction Set Extension*. PhD thesis, School of Informatics, The University of Edinburgh, March 2010. Referenced on page 21.

[13] Davide Bertozzi, Antoine Jalabert, Srinivasan Murali, Student Member, Rutuparna Tamhankar, Student Member, Stergios Stergiou, Student Member, Luca Benini, and Giovanni De Micheli. NoC synthesis flow for customized domain specific multiprocessor systems-on-chip. *IEEE Transactions on Parallel and Distributed Systems*, 16:113–129, 2005. Referenced on page 18.

[14] T. Bjerregaard and S. Mahadevan. A survey of research and practices of network-on-chip. *ACM Computing Surveys (CSUR)*, 38(1):1, 2006. Referenced on page 14.

[15] S. Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *Micro, IEEE*, 25(6):10–16, 2005. ISSN 0272-1732. Referenced on pages 12 and 19.

[16] S. Borkar, T. Karnik, S. Narendra, J. Tschanz, A. Keshavarzi, and V. De. Parameter variations and impact on circuits and microarchitecture. In *Proceedings of the 40th annual Design Automation Conference*, pages 338–342. ACM, 2003. ISBN 1581136889. Referenced on page 19.

[17] M.A. Bramer. *Principles of data mining*. Undergraduate topics in computer science. Springer, 2007. ISBN 9781846287657. URL http://books.google.co.uk/books?id=xVW7NslHNHsC. Referenced on page 22.

[18] L. Breiman, JH Friedman, RA Olshen, and CJ Stone. Classification and regression trees. 1984. Referenced on page 27.

[19] Leo Breiman. Random forests. *Machine Learning*, 45:5–32, 2001. ISSN 0885-6125. URL `http://dx.doi.org/10.1023/A:1010933404324`. Referenced on page 27.

[20] K. Buchenrieder and J. Rozenblit. *CODESIGN: Computer-aided Software-hardware Engineering*. IEEE Press, 1995. Referenced on page 21.

[21] R. Caruana, N. Karampatziakis, and A. Yessenalina. An empirical evaluation of supervised learning in high dimensions. In *Proceedings of the 25th international conference on Machine learning*, pages 96–103. ACM, 2008. Referenced on page 28.

[22] C.C. Chang and C.J. Lin. LIBSVM: a library for support vector machines. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 2(3):27, 2011. Referenced on page 26.

[23] K.S. Chathak, K. Srinivasan, and G. Konjevod. Automated techniques for synthesis of application-specific network-on-chip architectures. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 27(8):1425 –1438, aug. 2008. ISSN 0278-0070. doi: 10.1109/TCAD.2008.925775. Referenced on page 18.

[24] C. Clos. A study of non-blocking switching networks. *The Bell System Technical Journal*, 32(5):406–424, 1953. Referenced on page 15.

[25] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20:273–297, 1995. ISSN 0885-6125. URL `http://dx.doi.org/10.1007/BF00994018`. 10.1007/BF00994018. Referenced on page 26.

[26] Clifford E. Cummings and Peter Alfke. Simulation and synthesis techniques for asynchronous fifo design with asynchronous pointer comparisons. In *SNUG Proceedings*, April 2002. Referenced on pages x, 10, 42, and 43.

[27] W.J. Dally. Performance analysis of k-ary n-cube interconnection networks. *IEEE Transactions on Computers*, 39(6):775–785, 1990. ISSN 0018-9340. Referenced on page 14.

[28] W.J. Dally and B. Towles. *Principles and Practices of Interconnection Networks*. The Morgan Kaufmann Series in Computer Architecture and Design. Morgan Kaufmann Publishers, 2003. ISBN 9780122007514. URL `http://books.google.co.uk/books?id=uyAg3zu_DYMC`. Referenced on page 14.

[29] R. Das, S. Eachempati, A.K. Mishra, V. Narayanan, and C.R. Das. Design and evaluation of a hierarchical on-chip interconnect for next-generation CMPs. In *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, pages 175 –186, feb. 2009. doi: 10.1109/HPCA.2009.4798252. Referenced on page 14.

[30] EEMBC. Coremark, May 2010. URL `http://www.coremark.org`. Referenced on pages 78, 88, and 101.

[31] EIA/TIA-232-F. *Interface Between Data Terminal Equipment and Data Circuit-Terminating Equipment Employing Serial Binary Data Interchange.*, October 1997. Referenced on page 174.

[32] Haytham Elmiligi, Ahmed A. Morgan, M. Watheq El-Kharashi, and Fayez Gebali. Power optimization for application-specific networks-on-chips: A topology-based approach. *Microprocessors and Microsystems*, 33(5-6):343–355, 2009. ISSN 0141-9331. Referenced on page 15.

[33] C. Galuzzi and K.L.M. Bertels. The instruction-set extension problem: A survey. In *International Workshop on Applied Reconfigurable Computing (ARC)*, pages 209–220, March 2008. Referenced on page 21.

[34] R. Ginosar. Fourteen ways to fool your synchronizer. *Asynchronous Circuits and Systems, 2003. Proceedings. Ninth International Symposium on*, pages 89–96, May 2003. ISSN 1522-8681. Referenced on pages x, 10, 40, and 41.

[35] On-Chip Bus Development Working Group. Virtual component interface standard, April 2001. Referenced on page 62.

[36] K. Hechenbichler and K. Schliep. Weighted k-nearest-neighbor techniques and ordinal classification. 2004. Referenced on page 24.

[37] J.L. Hennessy and D.A. Patterson. *Computer architecture: a quantitative approach*. The Morgan Kaufmann Series in Computer Architecture and Design. Morgan Kaufmann Publishers, third edition, 2003. ISBN 9781558607248. Referenced on pages 14 and 15.

[38] Sebastian Herbert and Diana Marculescu. Characterizing chip-multiprocessor variability-tolerance. In *Proceedings of the 45th annual Design Automation Conference*, DAC '08, pages 313–318, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-115-6. Referenced on page 20.

[39] Carles Hernández, Federico Silla, and José Duato. A methodology for the characterization of process variation in NoC links. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, pages 685–690, 3001 Leuven, Belgium, Belgium, 2010. European Design and Automation Association. ISBN 978-3-9810801-6-2. Referenced on page 12.

[40] K. Jeong, A.B. Kahng, B. Lin, and K. Samadi. Accurate machine-learning-based on-chip router modeling. *Embedded Systems Letters, IEEE*, 2(3):62–66, 2010. Referenced on page 17.

[41] John Kim and Hanjoon Kim. Router microarchitecture and scalability of ring topology in on-chip networks. In *Proceedings of the 2nd International Workshop on Network on Chip Architectures*, NoCArc '09, pages 5–10, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-774-5. doi: 10.1145/1645213.1645217. URL `http://doi.acm.org/10.1145/1645213.1645217`. Referenced on page 14.

[42] John Kim, James Balfour, and William Dally. Flattened butterfly topology for on-chip networks. In *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 172–182, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-3047-8. Referenced on page 14.

[43] John Kim, Wiliam J. Dally, Steve Scott, and Dennis Abts. Technology-driven, highly-scalable dragonfly topology. In *ISCA '08: Proceedings of the 35th International Symposium on Computer Architecture*, pages 77–88, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3174-8. Referenced on page 15.

[44] Jongman Kim, Dongkook Park, Chrysostomos Nicopoulos, N. Vijaykrishnan, and Chita R. Das. Design and analysis of an NoC architecture from performance, reliability and energy perspective. In *ANCS '05: Proceedings of the 2005 ACM Symposium on Architecture for Networking and Communications Systems*, pages 173–182, New York, NY, USA, 2005. ACM. ISBN 1-59593-082-5. Referenced on page 14.

[45] Martha Mercaldi Kim, John D. Davis, Mark Oskin, and Todd Austin. Polymorphic on-chip networks. In *ISCA '08: Proceedings of the 35th International Symposium on Computer Architecture*, pages 101–112, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3174-8. Referenced on page 14.

[46] L. Kleeman and A. Cantoni. Metastable behavior in digital systems. *Design Test of Computers, IEEE*, 4(6):4 –19, dec. 1987. ISSN 0740-7475. Referenced on page 10.

[47] K. Lahiri, A. Raghunathan, and S. Dey. Design Space Exploration for Optimizing On-Chip Communication Architectures. *IEEE Transactions on Computer-Aided*

Design of Integrated Circuits and Systems, 23(6):952–961, 2004. Referenced on page 17.

[48] Leslie Lamport. A new solution of Dijkstra's concurrent programming problem. *Commun. ACM*, 17:453–455, August 1974. ISSN 0001-0782. URL `http://doi.acm.org/10.1145/361082.361093`. Referenced on page 76.

[49] Glenn Leary and Karam S. Chatha. A holistic approach to network-on-chip synthesis. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, CODES/ISSS '10, pages 213–222, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-905-3. doi: 10.1145/1878961.1879001. URL `http://doi.acm.org/10.1145/1878961.1879001`. Referenced on page 18.

[50] Charles E. Leiserson. Fat-trees: universal networks for hardware-efficient supercomputing. *IEEE Trans. Comput.*, 34(10):892–901, 1985. ISSN 0018-9340. Referenced on pages 15 and 84.

[51] D. Ludovici, F. Gilabert, S. Medardoni, C. Gómez, ME Gómez, P. López, GN Gaydadjiev, and D. Bertozzi. Assessing fat-tree topologies for regular network-on-chip design under nanoscale technology constraints. In *Design, Automation and Test in Europe Conference and Exhibition, 2009. Proceedings*, pages 562–565, 2009. Referenced on pages 15 and 84.

[52] Debora Matos, Gianluca Palermo, Vittorio Zaccaria, Cezar Reinbrecht, Altamiro Susin, Cristina Silvano, and Luigi Carro. Floorplanning-aware design space exploration for application-specific hierarchical networks on-chip. In *Proceedings of the 4th International Workshop on Network on Chip Architectures*, NoCArc '11, pages 31–36, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0947-9. doi: 10.1145/2076501.2076508. URL `http://doi.acm.org/10.1145/2076501.2076508`. Referenced on page 17.

[53] Giovanni De Micheli and Luca Benini. *Networks on Chips: Technology and Tools (Systems on Silicon)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006. ISBN 0123705215. Referenced on page 14.

[54] K. Mistry, C. Allen, C. Auth, B. Beattie, D. Bergstrom, M. Bost, M. Brazier, M. Buehler, A. Cappellani, R. Chau, C.-H. Choi, G. Ding, K. Fischer, T. Ghani, R. Grover, W. Han, D. Hanken, M. Hattendorf, J. He, J. Hicks, R. Huessner, D. Ingerly, P. Jain, R. James, L. Jong, S. Joshi, C. Kenyon, K. Kuhn, K. Lee, H. Liu, J. Maiz, B. Mclntyre, P. Moon, J. Neirynck, S. Pae, C. Parker, D. Parsons, C. Prasad,

L. Pipes, M. Prince, P. Ranade, T. Reynolds, J. Sandford, L. Shifren, J. Sebastian, J. Seiple, D. Simon, S. Sivakumar, P. Smith, C. Thomas, T. Troeger, P. Vandervoorn, S. Williams, and K. Zawadzki. A 45nm logic technology with high-k+metal gate transistors, strained silicon, 9 cu interconnect layers, 193nm dry patterning, and 100packaging. In *Electron Devices Meeting, 2007. IEDM 2007. IEEE International*, pages 247 –250, dec. 2007. doi: 10.1109/IEDM.2007.4418914. Referenced on page 12.

[55] Enric Musoll. A process-variation aware technique for tile-based, massive multi-core processors. *IEEE Computer Architecture Letters*, 8:52–55, 2009. ISSN 1556-6056. Referenced on page 13.

[56] W.C. O'Mara, R.B. Herring, and L.P. Hunt. *Handbook of semiconductor silicon technology*. Materials science and process technology series. Noyes Publications, 1990. ISBN 9780815512370. URL `http://books.google.co.uk/books?id=COcVgAtqeKkC`. Referenced on page 12.

[57] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2011. URL `http://www.R-project.org/`. ISBN 3-900051-07-0. Referenced on pages 23 and 79.

[58] Carl Edward Rasmussen and Christopher K. I. Williams. *Gaussian Processes for Machine Learning*. The MIT Press, second edition, 2006. ISBN 0-262-18253-X. Referenced on page 28.

[59] R. J. Reidlinger, R. Bhatia, L. Biro, B. Bowhill, E. Fetzer, P. Gronowski, and T. Grutkowski. A 32nm 3.1 Billion Transistor 12-wide-issue Itanium Processor for mission-critical servers. Presentation at ISSCC 2011, February 2011. Referenced on page 20.

[60] E. Salminen, A. Kulmala, and T.D. Hamalainen. Survey of network-on-chip proposals. *white paper, OCP-IP*, pages 1–13, 2008. Referenced on page 14.

[61] S. Scott, D. Abts, J. Kim, and W.J. Dally. The blackwidow high-radix clos network. *Computer Architecture, 2006. ISCA '06. 33rd International Symposium on*, pages 16–28, 2006. ISSN 1063-6897. Referenced on page 14.

[62] IEEE Computer Society. IEEE standard for reduced-pin and enhanced-functionality test access port and boundary-scan architecture. *IEEE Std 1149.7-2009*, pages c1 –985, 10 2010. Referenced on page 66.

[63] IEEE Computer Society. IEEE standard VHDL language reference manual. *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)*, pages 1–626, 2009. Referenced on page 9.

[64] IEEE Computer Society. IEEE Standard Verilog Hardware Description Language. *IEEE Std 1364-2001*, pages 1–856, 2001. Referenced on page 9.

[65] K. Sollins. The TFTP Protocol (Revision 2). RFC 1350 (Standard), July 1992. URL `http://www.ietf.org/rfc/rfc1350.txt`. Updated by RFCs 1782, 1783, 1784, 1785, 2347, 2348, 2349. Referenced on page 175.

[66] Synopsys. DesignWare processor IP, January 2012. `http://www.synopsys.com/IP/ProcessorIP/Pages/default.aspx`. Referenced on page 19.

[67] Igor Böhm Nigel Topham, Björn Franke, Marcela Zualaga, Richard V. Bennett, Alastair Murray, Oscar Almer, Georgios Tournavitis, Xinhao Qu, and Karthik T. Sundararajan. The EnCore embedded processor, May 2010. URL `http://groups.inf.ed.ac.uk/pasta/posters.html`. Best Poster Award at the HiPEAC Innovation Event. Referenced on page 19.

[68] Brian Towles and William J. Dally. Route packets, not wires: On-chip inteconnectoin networks. *Design Automation Conference*, 0:684–689, 2001. Referenced on page 13.

[69] O.S. Unsal, J.W. Tschanz, K. Bowman, V. De, X. Vera, A. Gonzalez, and O. Ergin. Impact of parameter variations on circuits and microarchitecture. *Micro, IEEE*, 26 (6):30–39, 2006. ISSN 0272-1732. Referenced on page 19.

[70] T. van Meeuwen, A. Vandecappelle, A. van Zelst, F. Catthoor, and D. Verkest. System-level interconnect architecture exploration for custom memory organizations. *Proceedings, The 14th International Symposium on System Synthesis*, pages 13–18, 2001. Referenced on page 17.

[71] H.J.M. Veendrick. The behaviour of flip-flops used as synchronizers and prediction of their failure rate. *Solid-State Circuits, IEEE Journal of*, 15(2):169 – 176, apr 1980. ISSN 0018-9200. Referenced on page 10.

[72] Perry H. Wang, Jamison D. Collins, Christopher T. Weaver, Blliappa Kuttanna, Shahram Salamian, Gautham N. Chinya, Ethan Schuchman, Oliver Schilling, Thorsten Doil, Sebastian Steibl, and Hong Wang. Intel© Atom™ processor core

made FPGA-synthesizable. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, FPGA '09, pages 209–218, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-410-2. URL `http://doi.acm.org/10.1145/1508128.1508160`. Referenced on page 11.

[73] Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann Series in Data Management Sys. Morgan Kaufmann, second edition, June 2005. ISBN 0120884070. Referenced on page 22.

[74] Xilinx Datasheet. Virtex-5 Family Overview, v5.0. February 2009. URL `http://www.xilinx.com/support/documentation/virtex-5_data_sheets.htm`. Referenced on page 11.

[75] Xilinx Datasheet. Virtex-6 Family Overview, v2.3. March 2011. URL `http://www.xilinx.com/support/documentation/virtex-6_data_sheets.htm`. Referenced on page 11.

A Appendix: I/O and memory devices

This appendix contains information about some of the I/O devices present in the system. This information is not considered of immediate importance, but is included for completeness and for indications of how some of the engineering challenges were solved.

A.1 Display device

The display device was used to show the system state and other useful software information as the system was running. It is therefore implemented as an AXI slave device that can only be written to.

The display device consists of an AXI slave module, a shared 40 kBytes BRAM, and a rendering pipeline, all implemented in the FPGAs. The rendering pipeline outputs signals for a CH7301C DVI Codec, which is present on both the ML507 and ML605 boards. This Codec, in turn, drives the DVI interface present on these boards.

The rendering pipeline is clocked at 108MHz and outputs signals for a 1280x1024 resolution screen at 24 bits per clock. Due to the large bandwidth requirements a real screen would have, some simplifying abstractions have been made. This resolution is fixed, and cannot be altered in software; it also dictates the amount of video memory to use. The screen uses a 160 x 128 character-mapped mode with four bits of colour for foreground and background, respectively.

In this case, with one byte for colour and one byte for character, I need 40 kBytes of RAM (5 BRAM blocks) to keep the frame. As a comparison, the entire screen in an addressable pixel-by-pixel mode and the same number of colours would occupy 753 kBytes. We adopted the character mode and limited colour palette out of necessity due to resource constraints in the FPGA.

The AXI slave module is attached to the AW, W and W AXI channels from the interconnect, and output data to the shared 40 kBytes BRAM block. The interconnect maps this device at memory address 0xFF010000, and I thus write to the screen by writing to this memory address plus an offset. As the interconnect does not generally run at the odd frequency of the rendering pipeline, I use the dual-ported RAM blocks as synchronising elements. The AXI slave module runs at interconnect frequency and inserts data at that speed, while the pipeline reads the data out at a fixed 108MHz.

The display device can be seen in Figure 3.3 where I demonstrate the text-output capabilities. Of note is the picture fragment displayed by cores 0, 1 and 4; this is in fact software interpolated graphics mapping to the character output capabilities of the display device. I am using the four grey levels in CGA and knowledge about the character set pixel configurations to map the 8-bit greyscale pixels to this display device.

A.2 UART interface

A Universal asynchronous receiver/transmitter (UART) is a device for framing and transmitting data over some distance, usually using the RS-232, RS-422 or RS-485 protocols[31]. Generally, a UART implies a standard framing around a byte, as well as a certain signal period, but does not imply a common clock reference.

The Virtex-5 ML507 board has a physical 9-pin serial connector, with the RX and TX lines brought through a level shifter directly to the I/O pins on the FPGA. The ML605 board, by contrast, has a serial-over-USB ASIC embedded on the board, saving connector space; it still brings through the native TX and RX signals to pins on the FPGA. This ASIC is recognised as a serial device under Linux, and presented no problems in use.

A.2.1 UART device

We needed a UART interface to my AXI interconnect so that I had a faster off-board two-way connection than JTAG. The UART interface was used to transmit debug data for off-board analysis as well as, on some boards, uploading new program code as required (with the assistance of an in-memory loader).

The UART was constructed out of two separate Verilog modules, one for transmit and one for receive, coupled though an asynchronous FIFO to an AXI interface. The AXI interface was logically mapped by the interconnect at 0xFF020000 for reading incoming data and 0xFF020001 for writing outgoing data. No traditional buffer or speed options were exported to the AXI interface, as the modules were made to use fixed size internal BRAM FIFO (4096 entries x 1 byte) and the I/O rate was fixed to 115200 baud. The reason for fixing the FIFO size is that that is the size offered in the FPGA fabric for hard asynchronous FIFO. The reason for fixing the baud rate was that by doing so I did not need to have logic to switch clock counters around unnecessarily.

Both the TX and RX modules are fed from a common 50MHz clock, which drives a state machine / counter arrangement to count the required number of periods between bit transitions. This arrangement is not the most robust for RX, as the 50MHz

clock may be drifting, and as compensation I sampled the middle of each bit period to minimize the effects of such drift. In practice this worked well enough to be of use, but could be improved on by multi-sampling methods.

As the transfer rate through the UART, even at 115200 baud, is only about 10 kBytes/s, it is entirely possible for the cores to output data significantly faster than it can be transmitted. In these cases, data is buffered in the 4096-byte Asynchronous FIFO until it is transmitted, and, if this buffer fills up, further AXI writes to the buffer are kept on hold. This hold will mean that the core issuing the write will not continue (see section 3.2.4.1) and will sit idle until the buffer has cleared enough to allow the write through. This state was encountered in some rare cases during debugging, while dumping internal program data to the UART for external analysis and debugging, but was not seen during the experiments themselves.

A.2.2 Program loader

The program loader was built upon the UART by integrating an IP-over-serial line stack into libmetal (see section 3.5) which handled the framing and deframing of a certain small subset of protocols. Most importantly, it used FTP[65] to transfer raw binary and text data over the serial line, enabling automated loading of binaries for some system configurations.

On the host side, this setup required nothing more than a standard TFTP server and the setup of a SLIP connection to the serial port connected to the board. This software setup was also agnostic to whether the serial was encapsulated in USB or not.

When the loader is included in the system build, a special region of memory is added, using a 16 kBytes RAM controller and mapped at 0xFF040000, that held only a program to load a file and store it starting at 0x00000000. This region was managed and connected through a standard Block RAM controller (Section 3.2.5), and the program was hard-coded into the FPGA synthesis. To enable this to run, a single JTAG command (see section 3.3) was used to insert a jump to 0xFF040000 instead of the normal start-up code, and the loaded software manged the rest. If necessary, JTAG could also be used to update this loader and update the file name in the loader requests.

An obvious extension to this, but which was not implemented, was to also let the loader program obtain its own IP address from a DHCP server, which would also provide the file name to load. This was not deemed necessary, as I can update that through JTAG, and due to the complexities of SLIP and DHCP interacting.

A.3 Keyboard interface

The Virtex-5 ML507 board also has an on-board PS/2 connector for a keyboard, and as the pinout and protocol for this device is not particularly complex, I decided to use it.

The keyboard interface shares some conceptual ideas with the UART device, in that it is a serial protocol with small hardware requirements. The keyboard interface is clocked, and thus somewhat easier to accommodate, and there are other differences. A single keypress can generate up to 4 bytes worth of data, when option keys are included, and this needs to be passed to the software stack as a keypress and not separate bytes. On the other hand, there is no mandatory output channel in the PS/2 protocol, so I did not have to consider reverse data.

The keyboard interface was implemented as a Verilog module connecting the FPGA pins from the PS/2 connector to the AXI interconnect. The PS/2 clock domain logic recognises and handles the keypress symbols as 32-bit entities, and writes them to a 32-bit wide FIFO.

The AXI interface also includes signals for an interrupt, allowing interrupt-driven input handling. The interrupt routines for this are present in libmetal (Section 3.5) and are enabled by default. These interrupt routines read the keypress data from the I/O device as a 32-bit chunk and place it into a software FIFO, which can then be dequeued by whatever program is in the foreground.

A.4 Block RAM controller

Block RAM is the native RAM resource in a Xilinx FPGA, and has a simpler interface than DDR2 RAM. Consequently, I have constructed a controller interfacing these to AXI that implements read, write, atomic operations and concurrent requests appropriately. Figure A.1 shows the pipeline overview of this controller. The figure shows the four-stage pipeline implementing the full controller and including the lock RAM necessary to enforce atomic requests. Logic is abstracted with circles, pipeline stages with flip-flop symbols.

The stages in the controller divide the responsibility of handling the memory transactions. Stage 0 arbitrates between the feedback paths and the AXI input channels, so that the feedback paths always have priority. The priority here is because the feedback paths cannot wait, as this would require locking the pipeline stages and invalidate reads. The AXI channels, by contrast, can wait by asserting the READY as long as there is a feedback path active or the input FIFO is full.

Stage 1 handles picking whether a read or write transaction gets inserted into the

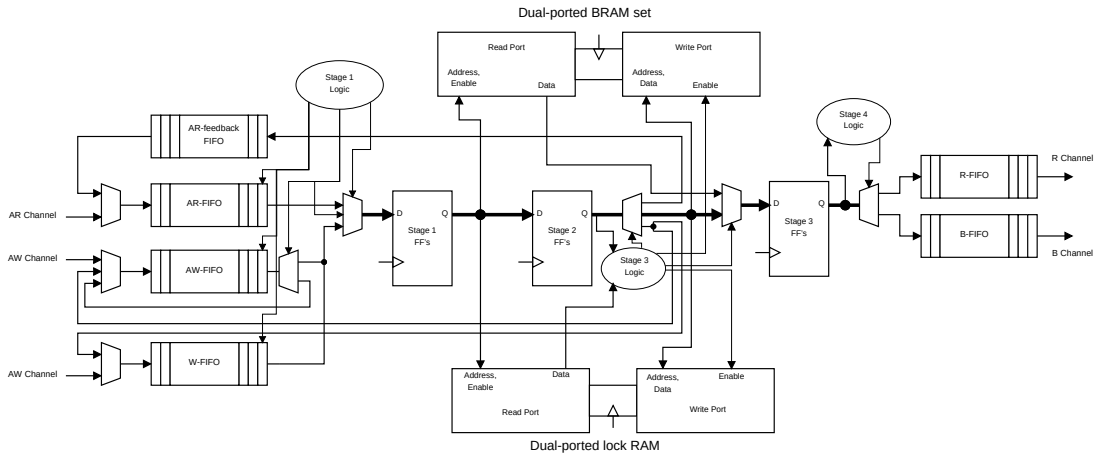


Figure A.1: Overview schematic of the BRAM controller pipeline, including lock RAM.

pipeline; the rest of the pipeline handles both depending on a single bit being set or unset. This stage also handles matching AW channel transactions to W channel transactions; AW channel transactions can be reordered as well as modified and written back, but W channel transactions have to be in-order. The logic therefore circulates the AW buffer to match the current W transaction before issuing it into the pipeline; while this occurs, AR transactions may well be issued instead. This stage is on the system critical path, as I want to serve memory requests as quickly as possible, and I want to keep the memory controller pipeline fed with a new operation every cycle if there are operations pending. These instructions may be fed back later due to lock constraints, but that can be attributed to software wait states rather than system throughput, and is acceptable as well as necessary.

Stage 2 is in place to allow for the one clock cycle delay of reading from both the data RAM and the lock RAM. The data RAM is read for both read and write operations, as the write may have strobe lines set to enable / disable particular bytes. The lock RAM is read to provide stage 3 with the data on whether this operation is trying to access a memory address locked by another operation.

Stage 3 then decides whether the operation should be allowed to go ahead and modify RAM. If this is a multi-operation read or write, I need to feed back AR or AW so that it can be reissued or matched to another W transfer. If this is a write to a locked region, then this stage may feed back both W and AW data, for a later re-attempt at writing (which may again feed back, until the lock release has occurred). This W data is safe to feed back, as lock / unlock operations are always only one W transfer long, because they are non-cached (see Section 3.2.4.1). The assumption here is that there will never be a multi-operation access to a locked address, as such an operation cannot be fed back safely. As it cannot be fed back, it is necessary to let such a operation

commit, and as such, it is possible to break the atomic access scheme. In practice any cache-ignoring requests from the EnCore processor is always one operation long, so this cannot be exploited deliberately, but it can be accidentally triggered if I allow the locked address to be cached. We therefore must ensure, in software, that any lock addresses fall in address ranges that are never cached. The libmetal operating environment (Section 3.5) contains code to ensure this condition is met. In addition, non-lock-requesting reads from a locked address are allowed, as they do not change the memory state and does simplify test-and-then-lock software behaviour, reducing the amount of atomic read-write pairs that may be issued. If the operation is allowed to proceed past stage 3, the potentially strobed data is supplied to the RAMs to be written in this cycle, and the read data is forwarded into the stage 3 flip-flop bank.

Stage 4 is responsible for enqueueing the response data onto the respective output FIFO buffers. Which FIFO to enqueue in is decided by which type of operation was performed. These are linked back into the AXI fabric using FIFO-to-AXI methods as discussed in Section 3.2.2.4.

The extra AR feedback FIFO in Figure A.1 is inserted to handle the expected multitude of AR-locking requests to a single master software lock (implemented through AXI atomic transfers) that is expected to guard the critical sections of the operating system. It could be the case that several cores are trying to get this lock at once, while one core is holding it; these lock requests will circulate through this buffer as long as the lock is held. We therefore want as much capacity in this path as there are cores in the system so that I do not lock up the interconnect when the AR FIFO fills up. In this case, of many circulating lock requests, I also need to make headway when the unlock request does appear; for that reason, the combined AW and W channels have absolute priority in stage 1.

A different problem can occur when several cores issue a lock request for the same address at once. In this case the lock BRAM must forward the data from the write stage to the read stage in one clock cycle; in other words, if the read and write addresses to the lock RAM matches, then the output must be the newly written data. In Xilinx parlance, this is a write-first RAM, as it conceptually does the write 'before' the read, so that the read returns the written data. Internally, this is accomplished by a special forwarding path between the write and read ports. If the lock RAM is not a write-first RAM, then it could well happen that two different cores will receive successfully locked responses.

A similar problem may occur with immediately subsequent differently-masked writes to the same 32-bit aligned address. The first write would commit, altering the in-memory representation in the same cycle as the second write would read it, mean-

ing that if the main RAMs are not also write-first (or has an appropriate forwarding mechanism) the second write will commit stale data. The problem in this case is very rare, as it involves cache-ignoring byte- or word-writes to the same address from two different cores that happen to line up right next to each other in the pipeline, but can nevertheless be triggered by careless programming. As above, the solution is to designate the RAMs as write-first or to introduce the feed-forward circuitry, which has been done in the BRAM instantiation module.

The size of the BRAM bank is variable, and given as a parameter to the top-level controller Verilog module. As such, I have experimentally used the same controller for BRAM block sizes ranging from 64k byte (8 controllers) up to 512k byte (1 controller). The actual low-level BRAM block composition is handled by the Xilinx tools; the behavioural Verilog merely asks for a given size of RAM and lets the tool flow (Section 3.4) handle the details.

This BRAM controller is driven at the same clock frequency as the system interconnect, but this is not an absolute requirement. By substituting asynchronous FIFO instead of synchronous ones, I could run the assembly at different clocks as necessary. This is not implemented, and is left as an extension.