Title	Metacomputing on clusters augmented with reconfigurable hardware
Authors	Healy, Philip D.
Publication date	2006-05
Original Citation	Healy, P. D. 2006. Metacomputing on clusters augmented with reconfigurable hardware. PhD Thesis, University College Cork.
Type of publication	Doctoral thesis
Link to publisher's version	http://library.ucc.ie/record=b1551998~S0
Rights	© 2006, Philip Healy - http://creativecommons.org/licenses/by-nc-nd/3.0/
Download date	2025-01-10 22:46:50
Item downloaded from	https://hdl.handle.net/10468/803

Metacomputing on Clusters Augmented with Reconfigurable Hardware

by

Philip D. Healy, B.Sc.

THESIS

Presented to the Faculty of Science

National University of Ireland,

Cork

for the Degree of

Doctor of Philosophy

May 2006

Acknowledgements

First and foremost, I would like to thank my family and friends for their ongoing support and encouragement. In particular, my parents have helped me in countless ways through many years of education; I couldn't have done it without them. Special thanks are also due to my supervisor, Prof. John Morrison, for his tireless assistance and endless patience during all stages of my postgraduate studies, from suggesting the idea of a Ph.D. through to preparation for the thesis defence. Thank you John.

Although all of my colleagues at the CUC, past and present, deserve credit for creating a work environment that is both enjoyable and interesting, several deserve special mention for being of particular assistance. Padraig O'Dowd collaborated directly on a number of the projects described in this thesis as well as several publications that subsequently arose. Brian Clayton patiently answered countless technical questions and also found time to administer the FPGA cluster despite having almost endless work commitments of his own. Thomas Quillinan was the first to propose a Condensed Graphs implementation of RC5 key search, and answered many questions over the years on a variety of topics. The friendly race to the finish between Tom and I provided a spur whenever it seemed like the end would never come; I hope that our arrangement was of similar benefit to him. A debt of gratitude is owed to Daithí Ó Crualaoich for his assistance with mathematical notation as well as many helpful corrections and suggestions. Finally, I would like to thank Dave Power, James Kennedy and Keith Power for our many lunchtime chats. Apart from providing lively conversation, their collective wisdom and experience was of enormous benefit while preparing this thesis.

Contents

Chapter 1

Introduction

Despite the low price and ever increasing capability of commodity desktop computing hardware, a strong demand still exists in many quarters for computing power beyond that which can be delivered by the desktop PC. Examples of performance-hungry computer applications include 3D image rendering, large-scale simulations and computational biology. Users wishing to execute applications such as these could wait for Moore's Law [1] to deliver better-performing commodity hardware sometime in the future, but this approach is unsatisfactory in many situations, particularly when execution times run to days or even weeks.

An alternative approach is to turn to the field of *High Performance Computing*, i.e., the use of specialized architectures or an aggregation of commodity computing resources to improve application performance. The most common method of achieving higher performance is known as *Parallel Computing* or *Parallel Processing*, and involves the use of multiple processing units working together to solve the problem at hand. Parallel computing was traditionally performed using purpose-built supercomputers developed by companies such as Cray, IBM, Sun Microsystems and Silicon Graphics. These systems, although enormously powerful in comparison to contemporary desktop machines, are highly expensive to purchase and maintain. Furthermore, the ongoing exponential increase in commodity computer performance soon negates the performance advantage of supercomputers. As a result of these factors, High Performance Computing has traditionally been a worthwhile activity only for organizations with acute need and deep pockets.

In recent years, supercomputers have increasingly been replaced by clusters composed of commodity hardware (see Section 1.1 below). The rise in prominence of clusters has led to a drastic reduction in the cost associated with High Performance Computing, and hence increased popularity. Another recent trend is the increasing interest in the use of reconfigurable hardware (FPGAs) to create application-specific coprocessors that improve

application performance (see Section 1.2). Although still regarded as a niche area, it is likely that interest will continue to grow as the capability of reconfigurable hardware increases over time. The focus of this thesis will be on how both techniques (Cluster Computing and Reconfigurable Computing) can be combined in a manner so as to yield portability, reuse and separation of concerns and hence better and faster engineered code. In addition, attention is paid to computational efficiency; all design decisions are cognisant of the resulting effect on application performance.

1.1   Cluster Computing

The notion of creating *compute clusters*, i.e., collections of interconnected computers working together as a single parallel system, is not a new one; IBM created compute clusters from mainframes during the 1960s [2]. However, it was not until the 1990s that the popularity of clusters gathered momentum due to the falling price and improving capability of commercial-off-the-shelf (COTS) components such as microprocessors and networking equipment. Efforts to create *commodity clusters* such as Beowulf [3] and Berkeley NOW (Networks of Workstations) [4] demonstrated that many large-scale and grand-challenge applications that were previously in the realm of supercomputers could be tackled using inexpensive hardware manufactured for the desktop computing market.

Commodity clusters are typically composed of a number of *compute nodes* connected by an Ethernet network. The compute nodes are usually homogeneous desktop machines configured without monitors, keyboards and other peripherals in order to save money and reduce space requirements (see Figure 1.1). Some clusters also contain a dedicated *head node*, which can perform a number of roles: running a firewall to separate the cluster
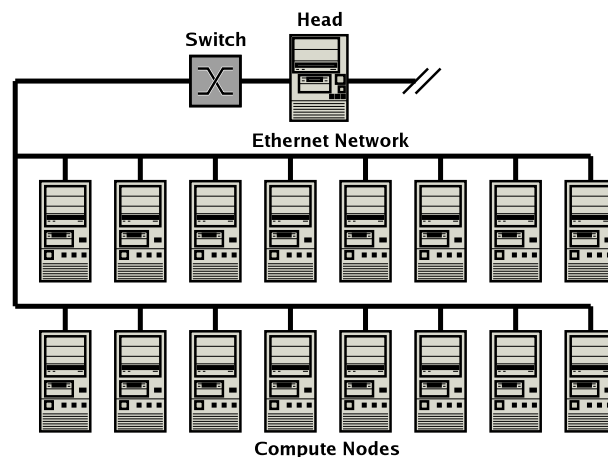

Figure 1.1: A typical commodity compute cluster, composed of 16 compute nodes, an Ethernet network and a head node.

from the outside world, hosting home directories that the compute nodes can access using NFS, running job management systems or even acting as a master in master/slave type computations. Clusters designed for applications that deal with large amounts of data may also contain *network attached storage* (NAS) units capable of storing Terabytes of data.

Although the majority of clusters are composed of dedicated machines, they may also be constructed in an *ad hoc* fashion by exploiting the idle cycles of underused workstations. The most simple method of constructing such clusters is by rebooting a collection of machines into dedicated disk partitions during periods when they are out of use [5]. More sophisticated schemes involve the use of daemons that allow machines to perform work during periods of inactivity (see Section 1.1.1).

Despite the relatively poor performance characteristics (i.e., processor performance, network latency and network bandwidth) of commodity clusters when compared with dedicated parallel processing machines, they offer much higher price/performance ratios due to the economies of scale achieved during the manufacture of their constituent components. Notwithstanding their shortcomings, clusters have been shown to be capable of executing a wide variety of High Performance Computing problems [6]. The increasing popularity of clusters is illustrated by the rapid rise in recent years of the number of clusters on the list of the world's top 500 supercomputers (see Figure 1.2). In any case, the performance gap between clusters and dedicated parallel machines has narrowed significantly over time. Specialized minicomputers and workstations are almost a thing of the past; modern desktop microprocessors are up to all but the most demanding tasks. The advent of Gigabit Ethernet has made cheap, high performance networking a reality. Higher performance cluster interconnects that offer improved latency and scalability, such as Myrinet [7], are available if even Gigabit Ethernet does not suffice for the task at hand. However, many tightly-coupled, latency-sensitive applications (such as some real-time visualization applications [8]) still lie outside the realm of commodity hardware, and will require the use of dedicated parallel machines for some time to come.

1.1.1   Programming Models for Cluster Computing

A variety of APIs, dedicated programming languages and middlewares are available for developing applications on clusters. The most suitable choice for any given project may depend on a number of factors, such as a desire for maximal performance, dependence on existing libraries, or ease of development. The wide range of choice available can be explained by the fact that no method has emerged that is best for all possible situations; each has associated advantages and disadvantages, and it falls to the programmer to choose the most appropriate model for the task at hand.

The most widely used technique is the use of traditional low-level programming languages such as ANSI C and Fortran in conjunction with message passing libraries such as PVM [9] and MPI [10]. The use of these libraries involves the explicit scheduling of networking communications through the sending and receiving of messages. Although this method is somewhat cumbersome and error-prone, the efficiency of a well-written message passing application is difficult to match using higher-level techniques. The continuing popularity of message passing libraries can also be explained by their high performance, relatively long history (and subsequent buildup of expertise), portability and popularity on non-cluster architectures (e.g., multiprocessors).

Another approach is to use *distributed shared memory*; each machine has access to the memory of all the others participating in the computation, creating a global virtual memory. Network communications are performed implicitly and are triggered by reads and writes to the shared memory space. Some shared memory systems, such as Treadmarks [11] and OpenMP [12], use APIs for existing programming languages. Others, such as Linda [13] and Unified Parallel C (UPC) [14], are utilized through programming language extensions. Shared memory systems require a mechanism for keeping data consistent. That is, a method of determining how local updates to the shared memory space are reflected on the other machines in the system. The simplest approach is to maintain sequential consistency by immediately transmitting any changes to the memory of a machine to the memories of all the other machines in the cluster. Unfortunately, the relatively poor performance characteristics of commodity networking equipment makes this approach impractical. As a result, most implementations use relaxed memory models. Reading the contents of a virtual shared memory location in these models does not necessarily return the last value written to that location. Knowledge of the message sequencing in the application will allow the programmer
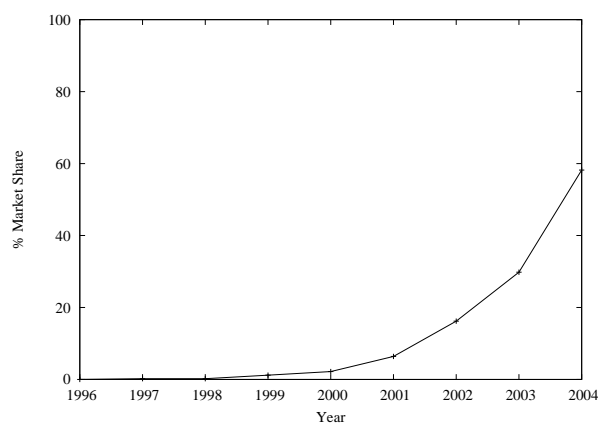

Figure 1.2: The growth of cluster market share as a percentage of the world's top 500 supercomputers *[Source: top500.org]*.

to extract performance when it is clear that consistency is not an issue. On the other hand, the onus is on the application programmer to maintain global consistency by explicitly using synchronization barrier and lock constructs.

Dedicated parallel programming languages offer a more appealing environment for the application developer to work with compared to the traditional approach of using a sequential language in combination with a parallel processing library. Parallel languages fall into two broad categories: extensions to existing sequential languages, and new languages designed specifically with the goal of facilitating parallel processing. Extensions to existing languages, such as High Performance Fortran (HPF) [15] and Charm++ [16], are comprised of parallel programming constructs such as `forall` loops and new keywords or directives (such as `DISTRIBUTE` and `ALIGN`) to facilitate the exploitation of distributed shared memory. Languages designed specifically for parallelism such as Occam [17] and Sisal [18] can express parallel programming functionality in a more elegant manner since they can be designed without the constraints of maintaining backward compatibility. Occam, for instance, features `seq` and `par` statements that specify sequential and parallel sections, as well as channels that allow processes to communicate easily in a fashion that minimizes deadlocks. Although these languages certainly simplify the job of the application developer to some extent, they are essentially syntactic sugar for underlying message passing or shared memory environments. As such, responsibility for parallelizing the program remains with the application developer.

Compilers for sequential programming languages often take advantage of chip-level parallel architectural features such as multiple ALUs and vector processing. Compilers such as SUIF [19] have extended this idea further through the development of compilers that attempt to extract the intrinsic parallelism of the application during compile-time and produce executables that can make use of multiple processors with no intervention on the part of the programmer. This approach carries obvious appeal; the burden of parallelizing the application is lifted from the developer and, even more importantly, preexisting code and libraries can be utilized in a parallel fashion with little effort. Unfortunately, code written in traditional sequential languages tends to be difficult to parallelize automatically due to the proliferation of side-effects caused by destructive assignment and limited inherent parallelism. Functional languages such as Haskell [20] tend to be more amenable to parallel compilation [21] since, due to the nature of these languages, side-effects are typically used only when necessary. However, the use of functional languages has not found wide acceptance within the HPC community because appropriate compilers have not been developed and because the necessary library codebase does not exist. All in all, parallelizing compilers are more suited to tightly-coupled architectures than clusters, due to the very fine-grain

nature of the parallelism they are able to exploit. Programs that express parallelism, either explicitly or through implicit parallel constructs, can result in a more efficient HPC implementation than a sequential equivalent that relies on extensive compiler support to find and exploit any inherent parallelism.

All of the parallel programming schemes mentioned above present the cluster of individual machines, to some extent, as a *single system image* (SSI) to the end user. Message passing libraries, for example, allow the cluster to be programmed as a loosely coupled parallel machine. Some projects have taken this idea to its logical extreme by allowing the cluster to be utilized as a single virtual machine. Examples of this approach are MOSIX [22] and Kerrighed [23]. These systems provide modified kernels allowing the user to see the cluster as a single UNIX workstation. No recompilation of existing applications is necessary; processes, and in the case of Kerrighed, threads, are migrated automatically across the cluster. Memory can be shared between nodes as needed, allowing virtual memory to page to the physical memory of the other nodes as well as to disk. Although undoubtedly the easiest method of using a cluster, the limitations inherent in this approach impose severe restrictions on the types of applications that can be parallelized. Mosix is only suited to applications that can execute as a set of independent processes. Even though Kerrighed allows for threads to be migrated across the cluster, the need to maintain global memory consistency and the performance overheads imposed by commodity networking hardware are likely to negate many of the advantages gained through thread migration. The lack of any method of discovering and taking advantage of the fine-grain parallelism found in many scientific applications (e.g., matrix multiplication) means that those wishing to execute such applications efficiently must turn to other techniques.

In contrast to dedicated clusters, Networks of Workstations can be used for distributed computing using a technique known as *cycle stealing* [24]. With this technique, any idle CPU time can be exploited in an opportunistic fashion without impacting on the individual machine users, taking advantage of an abundant resource that would otherwise be wasted. Examples of cycle-stealing systems, in addition to the SSI systems mentioned above, include Condor [25], Cyclone [26], and LinuxNOW (see Section 2.8.4).

The term *metacomputing* has recently come into vogue. In [27] it is defined as *"the use of powerful computing resources transparently available to the user via a networked environment"*. However, this definition does not distinguish metacomputing from the SSI systems mentioned above. In practice, metacomputing has come to mean the use of middlewares to present a collection of potentially diverse and geographically distributed computing resources transparently to the user as a single virtual computer. Perhaps the most well-known metacomputing environment is the Globus Metacomputing Toolkit [28], which provides a

middleware for constructing computational grids from distributed, heterogeneous computing resources. The term *grid* is most often used to mean an infrastructure for the sharing of resources across administrative domains. Therefore, a *computational grid* is a hardware and software infrastructure that provides dependable, consistent and pervasive access to high-end computational capabilities, despite the geographical distribution of both resources and users [29]. A computational grid may be composed of a number of clusters, or parallel processing machines, or both. Globus is designed to be as flexible as possible, and as such does not enforce the use of any particular programming model. Programming languages and APIs supported include MPI, Java, Compositional C++, RPC and Perl.

Besides Globus, other metacomputing projects of interest include Legion [30] and WebCom [31]. Legion is an object-oriented metacomputing system that implements computations as collections of distributed objects. Programming languages supported include MPL (a parallel version of C++ that was also used as the implementation language), Fortran and Java. Emulation libraries are also provided for PVM and MPI, allowing legacy applications to be supported. WebCom is a metacomputing environment that allows computations expressed as *Condensed Graphs* (see Section 2.3) to be executed on a variety of platforms in a secure, fault-tolerant manner. Load-balancing is also performed over the computing resources available without requiring any intervention on the part of the programmer. Originally designed as a means of creating *ad hoc* metacomputers from Java applets embedded in web pages, WebCom has since been developed into a general-purpose distributed computing environment suitable for the creation of grids. An extended version of WebCom, entitled WebCom-G [32], allows for interoperability with other Grid Computing platforms and includes support for legacy applications.

1.2   Reconfigurable Computing

Reconfigurable Computing is defined as *"the use of systems that can alter their hardware configuration in response to changing context or data content"* [33]. In effect, this requires the utilization of *reconfigurable hardware*, i.e., logic devices that can be reprogrammed with different hardware designs. Although different types of reconfigurable hardware are available (see Section 1.2.1), *field programmable gate arrays* (FPGAs) are of most interest, since they allow complex, register-heavy and pipelined designs to be executed at comparatively high clock speeds. These interesting properties have led to FPGAs finding applications in a number of areas, such as ASIC design prototyping, Embedded Computing and, most notably, High Performance Computing.

The notion of creating application-specific coprocessors using reconfigurable hardware has gained popularity in recent years as FPGAs have increased in speed and density. Using

this technique, applications may be accelerated by delegating some of their most commonly used functionality (usually an innermost loop) to an FPGA coprocessor configured with a suitable hardware implementation. Not all applications are amenable to acceleration with reconfigurable hardware; the part of the application to be accelerated must typically exhibit a high degree of intrinsic parallelism and data locality in order to make the conversion to hardware worthwhile. For some classes of applications, however, the benefits are impressive, with orders-of-magnitude increases in performance common in fields such as cryptography [34], image processing [35], data compression [36] and neural networks [37].

1.2.1   Programmable Logic

Unlike *application-specific integrated circuits* (ASICs) which are *fixed-function*, i.e., the gates implementing the hardware design are physically etched into the silicon die, reconfigurable hardware is characterized by its ability to be reprogrammed with different configurations. A variety of devices fitting this description are available, but all fall into three broad categories: SPLDs, CPLDs and FPGAs [38].

The term *programmable logic device* (PLD) is often used to describe all types of programmable and reconfigurable hardware, but was initially used to refer to simple reconfigurable devices capable of replacing a small collection of combinatorial logic (i.e., AND, OR and NOT) chips on a circuit board. More recently, these devices have been referred to as *simple programmable logic devices* (SPLDs) in order to avoid confusion with the other classes of programmable devices described below. The advantage of using SPLDs when designing circuits is that they require less board area, power and wiring than a collection of simpler chips. SPLDs are typically implemented as a collection of fully-connected macrocells, each containing the logic necessary to implement a simple Boolean equation and a flip-flop to store the result until the next clock transition [39]. Other terms for SPLDs include *programmable logic array* (PLA) and *programmable array logic* (PAL). Most SPLDs cannot be reprogrammed; although they are capable of implementing a variety of logic functions, the function is typically fixed by the manufacturer and cannot be changed thereafter. The advent of SPLDs capable of being reprogrammed, referred to as *generic array logic* (GAL), marked the beginning of the field of Reconfigurable Computing.

Complex programmable logic devices (CPLDs) can be thought of as a number of SPLDs integrated on the same silicon die, along with a programmable switch matrix that connects them. This configuration allows more complex designs to be implemented, and allows dozens of individual logic chips to be replaced by a single device. Unlike the interconnects within the individual PLDs, the switch matrix in a CPLD is often not fully connected, making some designs impossible to implement even when there are sufficient logic gates available.

Although an overlap exists between the capabilities of CPLDS and low-end FPGAs, the simpler structure of CPLDs allows for shorter delays and hence higher clock speeds. As a result, CPLDs are often chosen for circuits where high performance is a priority.

The most complex type of reconfigurable hardware devices are *field programmable gate arrays*[1] (FPGAs). FPGAs are composed of a two-dimensional array of logic blocks surrounded by routing resources, with I/O resources at the periphery. The principle application of FPGAs has traditionally been the prototyping of ASIC designs. In recent years they have also made inroads into the embedded computing market, where their reconfigurable nature offers the advantage of allowing hardware designs to be upgraded over time. The cost savings achieved by mass producing low-end FPGAs have also led to them increasingly being used in place of low-volume runs of ASICs. As the complexity of high-end FPGAs has increased over time, they have become capable of executing many applications faster than even top-of-the-range microprocessors, creating an interest in the use of FPGAs for High Performance Computing purposes [40].
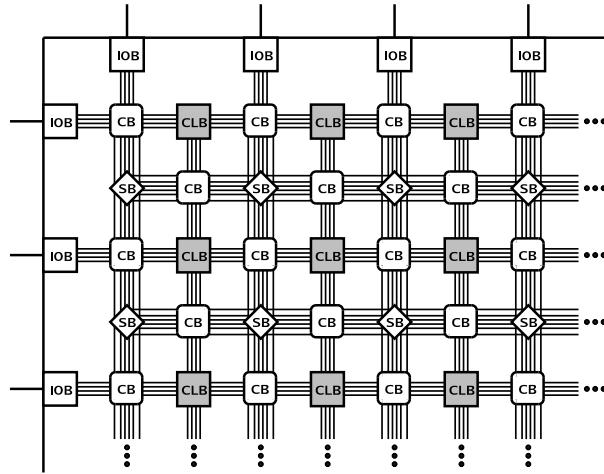
Figure 1.3: Simplified diagram of the internal structure of a typical FPGA, composed of I/O blocks (IOBs), configurable logic blocks (CLBs), connection blocks (CBs) and switch boxes (SBs).

FPGAs are composed of a collection of *configurable logic blocks* (CLBs) sitting in a "sea" of routing resources. The routing resources are surrounded by a collection of *I/O blocks* (IOBs) which are connected to I/O pins. Other resources, such as memory banks and ALUs, may also be embedded in the FPGA fabric. CLBs are capable of implementing simple logic functions and are typically implemented using lookup tables (LUTs). The routing resources are composed of *wires*, *connection blocks* and *switch boxes*. Wires carry signals between the various components. Connection blocks contain multiplexers that select which signals are to

---

[1]The *field programmable* part of the title refers to the fact that the devices can be programmed "in the field" rather than any electromagnetic properties they possess.

be connected to the terminals of nearby CLBs, and also connect short local wires to longer routing resources. Switchboxes are used to change the direction of signals from horizontal to vertical routing resources or *vice versa*. Although the architecture described here is by far the most common, a number of alternative approaches have been developed [41].

Of the three distinct types of reconfigurable hardware described here, FPGAs are by far the most commonly used by those wishing to create application-specific coprocessors. In light of this, the term *reconfigurable hardware* will hereafter be used to refer only to a collection of one or more FPGAs.

1.2.2   Benefits and Limitations of Reconfigurable Computing

Performance increases attained through the use of reconfigurable hardware result from the massive levels of parallelism that exist within FPGAs. While CPUs are limited to sequential operation by the fetch/decode/execute cycle, FPGAs can carry out many operations in parallel. Even though modern CPUs are parallel to an extent through the use of pipelining and multiple ALUs, FPGAs offer much greater parallelism due to their ability to have many data paths, each of which can carry out operations of much more complexity than a CPU instruction. Unfortunately, the performance characteristics of FPGAs are in many respects far inferior to those of CPUs, particularly in terms of clock speed and floating-point performance (see below). The relatively low bandwidth and latency of many FPGA connection topologies (see Section 1.2.3) further degrades performance. Application developers wishing to improve performance by using reconfigurable hardware must be sure that the application in question plays to the strengths of FPGAs if any improvement is to be seen.

Suitable applications for FPGA acceleration must be CPU-intensive and exhibit a high degree of intrinsic parallelism and data locality. Applications that work with large, complex memory structures involving many non-local memory accesses are unlikely to benefit [42]. FPGAs are also at a signicant disadvantage for applications that make heavy use of floating-point arithmetic. The complexity associated with performing floating-point operations leads to long delays within resulting FPGA configurations and hence poor performance. Modern CPUs improve floating-point performance through the use of *floating point units* (FPUs); FPGAs under development that contain embedded FPUs are likely to ameliorate this situation in the future. Another factor that application developers must take into account is the sequential bottleneck imposed by the limited number of I/O pins that connect FPGAs to the outside world. Applications may be unable to take advantage of parallelism if they are starved of data due to the limited amount of data they can read per clock cycle. The highest-performing applications are those that can unroll loops and execute them in parallel inside the FPGA, with little communication occurring with the outside world. Research

by Sun Microsystems into the development of microchip architectures that use capacitive coupling rather than pins [43] to increase bandwidth may improve this situation in future generations of FPGAs.

The most significant factor mitigating against the use of FPGAs is the significant difference between the clock speeds of even the highest-performing FPGA and contemporary CPU (see Figure 1.4(a)). The problem of low FPGA clock speeds is further exacerbated by the fact that maximum clock speeds are rarely attained in practice due to longest path delays in hardware designs. At the time of writing, the most advanced FPGA available is the Xilinx Virtex-4 family, which has a maximum clock speed of 500 MHz *[Source: xilinx.com]*. This figure stands in stark contrast to the clock speed of the highest-performing Intel Pentium 4 desktop CPU, which is clocked at 3.6 GHz *[Source: intel.com]* – over seven times faster. This divergence in clock speeds is likely to increase over time as the rate of increase of CPU clock speeds is greater than that of FPGAs. Although the exact date for the expiration of Moore's Law is the subject of ongoing speculation (the International Technology Roadmap for Semiconductors currently extends to 2016 [44]), the laws of Physics dictate that a limit on microprocessor clock speed must be reached at some point in the future. Once this limit has been reached, FPGA clock speeds could begin to converge with those of CPUs, but the degree to which this convergence can occur is, again, the subject of speculation.

Given the increasingly large divergence between FPGA and CPU clock speeds, pessimism about the future of Reconfigurable Computing for high performance purposes is understandable. However, such pessimism ignores another important performance characteristic: transistor count. The number of transistors embedded in FPGAs (and hence the complexity of the hardware designs they can accommodate) has increased exponentially (see Figure 1.4(b)), a trend that looks likely to continue into the future as smaller-scale
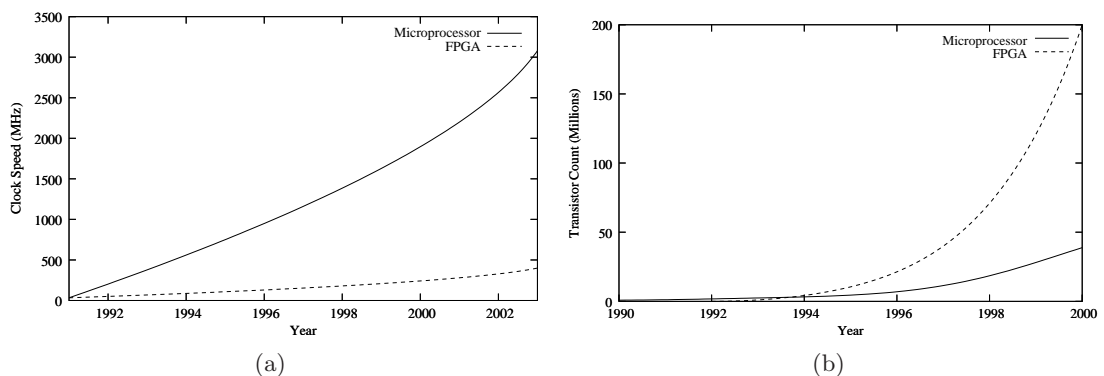

Figure 1.4: The divergence between (a) the clock speeds and (b) the densities of the Intel Pentium family of microprocessors compared with the Xilinx Virtex family of FPGAs during the 1990s.

manufacturing processes are used. An increasing divergence, similar to that between clock speeds, exists between the transistor counts of CPUs and FPGAs. It is therefore likely that Reconfigurable Computing will deliver even greater speedups relative to CPUs in the future, but only for those fine-grained, compute-intensive parallel applications to which they are already suited [45].

1.2.3   Reconfigurable Computing Topologies

A variety of methods for attaching reconfigurable hardware to host processors have been proposed. The most tightly-coupled arrangement is the integration of microprocessors and reconfigurable hardware on the same silicon die (see Figure 1.5(a)). This approach can involve augmenting an existing microprocessor architecture with a *reconfigurable functional unit* (RFU) that has direct access to the CPU's registers, and is utilized through extensions to the microprocessor's instruction set. The Chimaera RFU [46] is an example of this approach. A similar approach is the integration of reconfigurable hardware in the form of a tightly-coupled coprocessor, for example the REMARC [47] architecture (see Figure 1.5(b)).

Other methods involve the attachment of reconfigurable computing boards to the local bus of the motherboard, allowing the FPGA to communicate with the same latency and bandwidth as main memory (see Figure 1.5(c)). FPGA boards are available that can plug into unused processor slots on AMD Opteron motherboards [48]. Unfortunately, most motherboards, especially the commodity models typically found in clusters, are not capable of supporting such devices. An ingenious workaround for this limitation is the creation of reconfigurable computing boards that can be fitted to the memory slots of commodity motherboards, for example Pilchard [49] and SmartDIMM [50] (see Figure 1.5(d)). Such boards have only ever been manufactured in limited quantities, however, and are not commercially available at the time of writing[2].

The most common means of attaching reconfigurable hardware to the host processor is to use dedicated parallel processing boards that are fitted to the PCI bus of standard motherboards (see Figure 1.5(e)). This approach is less than ideal due to the performance penalty incurred by sending data back and forth over the PCI bus. The largest market for high-end FPGAs is currently the prototyping of ASIC designs, an application much less sensitive to the limitations imposed by the PCI bus than the creation of application-specific coprocessors. As a result, PCI-based reconfigurable computing boards are manufactured in large quantities and are hence widely available at relatively low cost. Manufacturers of reconfigurable computing boards include Celoxica, Nallatech, Xilinx and Annapolis Microsystems.

---

[2]One company (Nuron) did briefly manufacture such devices during the late 1990s, but has since ceased trading.

(a) Combined CPU/FPGA

(b) FPGA coprocessor

(c) Local Bus Interface

(d) Memory Slot Interface

(e) PCI Board
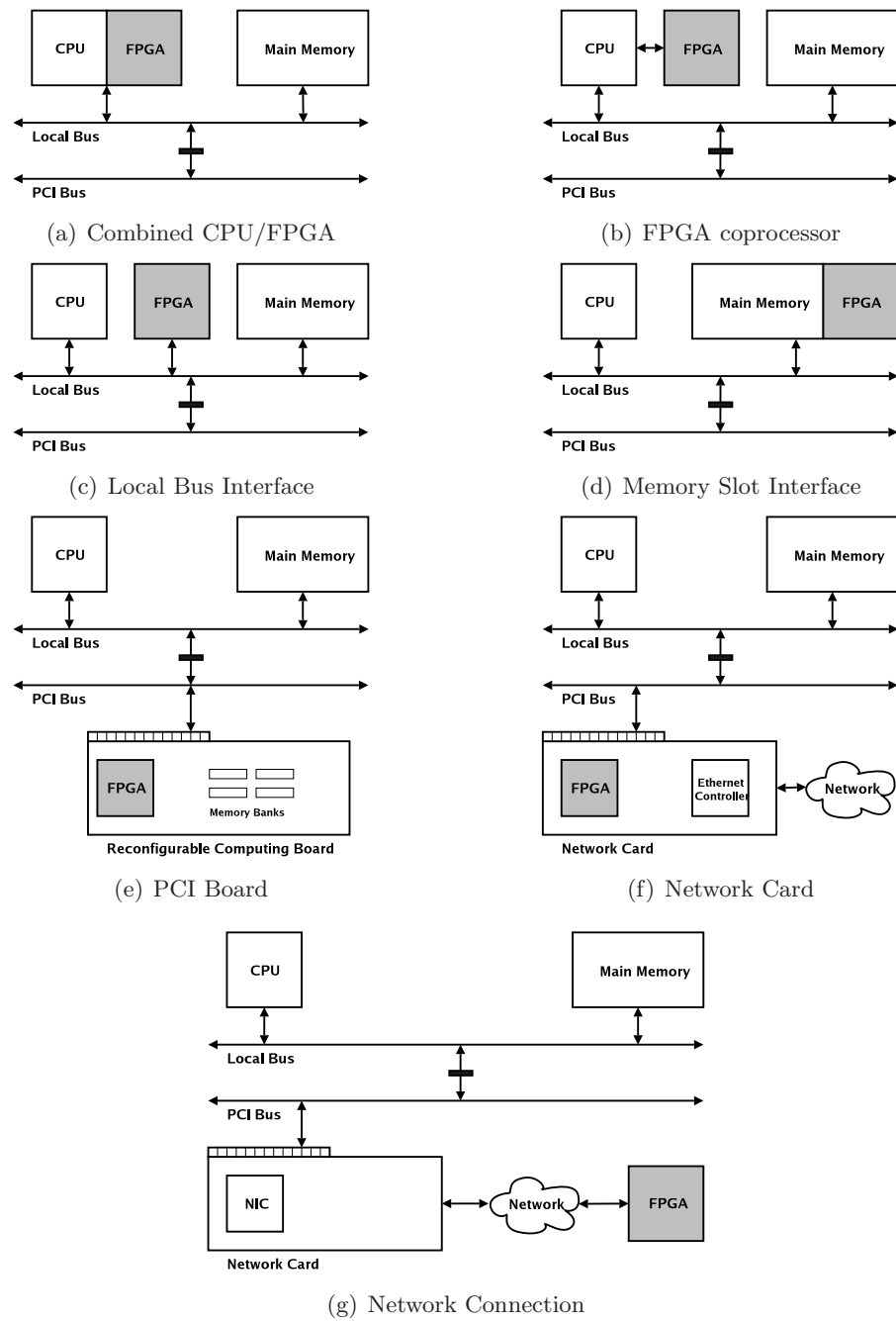
(f) Network Card

(g) Network Connection

Figure 1.5: Reconfigurable Computing Topologies

Alternative methods of attaching FPGAs to desktop PCs include using the serial or USB buses, but the poor performance of these buses makes this approach unsuitable in situations where high performance is a consideration.

Another PCI-based approach is the integration of reconfigurable hardware with network interface cards (NICs), for example the custom cards created in the construction of the Tower of Power [51], a cluster created from commodity machines fitted with FPGA/NIC hybrids (see Figure 1.5(f)). These cards are designed for Digital Signal Processing (DSP) applications, where data arriving over the network is piped through the FPGA on the network card before being passed to the host CPU for further processing.

The most loosely-coupled connection topology is direct communication with reconfigurable hardware over a network (see Figure 1.5(g)). For example, QinetiQ market a system composed of a desktop workstation as well as a heterogeneous collection of FPGAs and individual CPUs, all of which communicate directly with a Myrinet packet-switched network [52]. Although very modular and adaptable, the performance limitations of even the best-performing networking equipment limits the range of applications that can be tackled on systems such as this. This is due to the fact that without the supervision of a locally-attached CPU, every communication with the FPGAs must be performed over the network, degrading the performance of many applications. As such, network-attached FPGAs are suited only to the most coarse-grained of applications that are amenable to FPGA acceleration.

Although the majority of work relating to reconfigurable computing involves the integration of FPGAs with host processors, as described above, other work has focused on the creation of *custom computing machine*s. These are specialized computing architectures integrating many discrete programmable logic components into a single computing board [53] [54]. Despite being expensive to design and construct, architectures such as these are useful in situations where throughput requirements would limit the usefulness of a single, complex, FPGA. For example, matrix multiplication, although not normally amenable to acceleration using a single FPGA (see Section 1.2.2 above), could be accelerated using a number of relatively simple devices with shared memory operating in parallel.

1.2.4   Creating FPGA Configurations

The creation of FPGA configurations (or *bitstreams*) is typically a two-step process. The first step is to create a description of the logic gates and their interconnections that comprise the hardware design, using a standard format such as Electronic Design Interchange Format (EDIF) [55]. The hardware description is then passed through a *place-and-route* tool specific to the target model of FPGA. The place-and-route tool uses a floorplanning algorithm to

map the logic blocks contained in the hardware description to a bitstream that can be sent to an FPGA using a suitable software driver or API. Although little choice usually exists in the selection of place-and-route tool and support software (these are typically vendor-specific), numerous techniques are available for the creation of hardware descriptions.

The most widely used means of creating hardware descriptions is the use of traditional hardware description languages such as VHDL [56], Verilog [57] and ABEL [58]. When using these languages the designer must explicitly specify the logic elements (such as gates, flip-flops and latches) and interconnections that comprise the design. This approach is comparable to assembly language programming for microprocessors in that it has the advantage of allowing for the creation of extremely efficient designs but at the cost of increased development time and reduced design maintainability.

Rather than using a dedicated language, low-level hardware designs may also be created programatically using *circuit generators*, i.e., APIs for existing software programming languages. Perhaps the most popular of these is JBits [59], an API for the Java language. Other APIs include PAM-Blox (C++) [60], PyHDL (Python) [61] and Lava (Haskell) [62]. These APIs can be used either for the creation of parameterized low-level hardware designs, such as variable-width adders, or can be used at a higher level to create designs by linking together predefined logic blocks.

Graphical hardware design tools, such as Simulink [63] and Viva [64], aim to reduce the complexity associated with specifying low-level hardware designs by providing a more user-friendly environment than text-based interfaces to hardware design languages or APIs. When using these tools, circuits are specified by dragging and dropping predefined logic blocks from a palette and "drawing" the connections between them. This technique is particularly powerful when a large selection of high-level predefined components such as adders, multipliers and counters are available. The drawbacks of these systems are that they emphasise structural rather than behavioural development, and the fact that large designs with many non-local interconnections can become extremely complex to develop and maintain in a graphical fashion.

High-level general purpose hardware design languages such Handel-C [65], SystemC [66] and Transmogrifier C [67] have also become available. All are based upon the ANSI C language, but with some features (such as pointers) removed and others (such as parallel programming constructs) added. By starting with a simple, poorly-performing design, analysis with place and route tools reveals the longest paths in the resulting hardware design. Through a process of iterative refinement, various optimizations can be performed until an acceptable level of speed/efficiency is reached. Small changes in the high-level source code can result in major changes in the resulting logic, allowing different design strategies

to be evaluated quickly. Although these languages offer the advantage of reduced development time and increased agility in comparison with their lower-level counterparts, the resulting designs cannot match the efficiency of implementation attainable using traditional languages.

The creation of FPGA configurations directly from existing sequential languages is a field under active investigation. Some systems map complete programs directly to FPGA configurations. This approach has been attempted for a variety of languages, such as Occam [68] and Smalltalk [69]. Other systems allow the developer to identify the part of an existing sequential program most amenable to hardware acceleration (perhaps using a profiler) and then delegate the necessary hardware/software co-design to the compiler. The Galadriel [70] compiler, one example of such a system, converts Java byte code to FPGA configurations (via VHDL) and modifies the byte code to transmit data to and from the attached reconfigurable hardware. Similarly, the NAPA C compiler [71] allows parts of an ANSI C program to be automatically delegated to reconfigurable hardware through the use of `#pragma` directives. Although sequential compilers such as these allow reconfigurable hardware to be exploited with very little effort on the part of the application designer, the resulting designs are usually far less efficient than those created using more traditional methods. In particular, dedicated hardware design languages and APIs allow the width of data paths to be specified explicitly to avoid wasting logic resources, a feature that sequential languages lack. NAPA C provides directives for specifying the bit-width of variables, although doing so significantly increases development time and erodes the advantage associated with automatic compilation.

A more unusual approach is the evolution of hardware designs using genetic algorithms [72]. Starting with a random configuration, the effect of small perturbations are measured against a fitness function to determine how close they come to meeting the desired behaviour. Through this evolutionary process, those designs that perform well are more likely to be chosen for the next iteration until eventually a configuration emerges that meets all of the design requirements. The resulting designs are often very efficient and can take advantage of characteristics of FPGAs unavailable to conventional design tools. For example, some evolved configurations contain areas of logic completely cut off from the main circuit, causing them at first to appear redundant. However, if these areas of logic are removed the main circuit may cease to function. Possible explanations for this behaviour include electromagnetic coupling or subtle interactions through the power supply or silicon substrate. Although unorthodox circuit behaviour such as this may be advantageous in terms of performance and efficiency, care must be taken to ensure that the design does not rely on the exact conditions in which it was evolved (e.g., the precise voltage used, the individual FPGA used for evaluation and the presence of electromagnetic interference) by including

tests for robustness in the fitness function.

1.3   Augmenting Clusters with Reconfigurable Hardware

Given the large performance increases attainable using Cluster Computing and Reconfigurable Computing, it is not surprising that interest has developed in using a combination of both techniques, i.e., the creation of clusters where the compute nodes are augmented with reconfigurable hardware in order to improve performance [73][74][75] (see Figure 1.6 below). This field is still in its infancy, and referred to by a number of synonymous terms in the literature, including *Distributed Reconfigurable Computing*, *Distributed Adaptive Computing* and *High Performance Reconfigurable Computing*. In order to avoid confusion, it should also be noted that the term *FPGA cluster* is generally used to refer to groups of logic blocks within FPGAs [76] rather than compute clusters augmented with reconfigurable hardware.
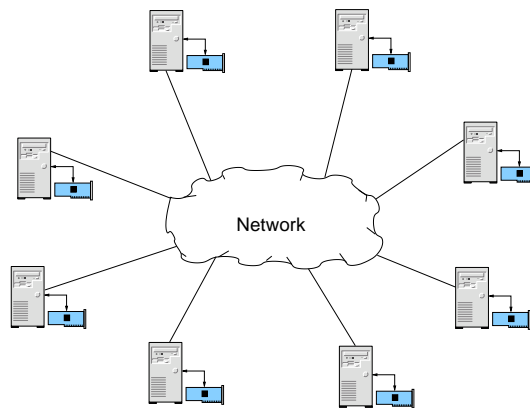
Figure 1.6: A cluster augmented with reconfigurable hardware in the form of PCI FPGA boards.

The most common methodology for developing applications for clusters augmented with reconfigurable hardware is to use MPI in conjunction with high-level hardware design languages, with communications to and from the reconfigurable hardware performed using vendor-supplied drivers. The FPGAs are typically connected to the cluster nodes using either PCI cards or combined FPGA/NICs. A number of applications have successfully been implemented using this technique, such as Parallel Brutus [77] (a chess playing application), fast Fourier transforms [78], data processing for physics applications [79], and an application for solving the n queens problem [80].

Libraries have been developed to simplify the task of developing applications for clusters augmented with reconfigurable hardware. The Adaptable Computing API [81] is a connection-oriented library that allows stream-based applications to be created that utilize distributed reconfigurable computing boards in a portable fashion. When using this library,

all reconfigurations and communications must be specified explicitly, and as such this approach can be regarded as a reconfigurable computing analogue of message passing libraries for clusters. Similarly, an extension has been added to the low-level GRIM (General-purpose Reliable In-order Messages) protocol in order to exploit clusters augmented with various resources, such as network attached storage and FPGAs, through the creation of logical channels [82]. Another approach is the use of existing job management systems (JMSs), such as LSF and CODINE, to target reconfigurable hardware embedded in distributed machines at the process level [83].

Considerable effort has been expended in developing tools for compiling and scheduling applications both for individual custom computing machines containing multiple FPGAs and clusters augmented with reconfigurable hardware. MATCH [84] is a Matlab compiler that can target a collection of distributed heterogeneous FPGAs. An automated mapping and scheduling tool has been developed [85] that statically partitions a graph description of an application based on the distributed computational resources available and produces an execution schedule. A similar approach is the notion of *resource pools* [86], which are abstract representations of the computing resources available, allowing a high-level description of an application to be mapped to both hardware and software as necessary.

CARMA (Comprehensive Approach to Reconfigurable Management Architecture) [87] is a management framework designed to facilitate the development of applications, system services, programming models and middlewares that utilize distributed reconfigurable hardware. The services provided include board-independent application mapping, dynamic job scheduling, distributed configuration management [88], performance monitoring [89] and board-independent modules for interfacing with FPGAs.

On the theoretical side, a *performance model* has also been developed for clusters augmented with reconfigurable hardware [90]. The performance model allows the behaviour of various cluster configurations to be accurately simulated, and as such can be of significant benefit when designing clusters before purchase. Related work includes an analysis of the cost effectiveness of clusters augmented with reconfigurable hardware [91].

Despite the advances planned and supplied by many of the systems above, it can be noted that these applications and systems are relatively simple to construct and do not consider issues such as task heterogeneity and the management of multiple FPGA configurations. Active consideration for these important characteristics was a prime motivator for the ARC system.

Although the majority of work related to the use of reconfigurable hardware within clusters is concerned with executing portions of applications on the reconfigurable hardware, as described above, other roles for reconfigurable hardware have also been examined. A significant body of research has been conducted on the use of FPGAs to improve the performance

of networks for HPC purposes [92][93]. Higher networking performance is typically achieved through the implementation of custom protocols designed specifically for that purpose. An implementation of the Condensed Graphs (see Section 2.3) model of computation has been developed [94] with the aim of reducing the performance overhead associated with identifying subtasks. Unfortunately, the latencies associated with communicating over the PCI bus and the sequential bottleneck imposed by access to the development board's memory banks resulted in poor performance. Nevertheless, this concept may become viable if adapted to more advanced FPGA architectures and connection methods. A similar approach is the automatic delegation of grid services to reconfigurable hardware in a platform-independent fashion [95].

1.4   Parallel Computing Metrics

A number of useful metrics can be applied to parallel processing systems, either before development in order to evaluate potential benefits, or afterwards so that the performance of the system can be measured. Since the goal of parallel processing is typically to minimize execution time, perhaps the most useful metric is the ratio of the times taken to execute the application sequentially and in parallel. This ratio, called the *speedup factor* (S), is given by

$$S = \frac{t_s}{t_p}$$

where t_s is the sequential execution time and t_p is the parallel execution time. In multiprocessor systems the speedup factor is typically limited by the number of processors (n). *Superlinear speedup*, where $S > n$, is sometimes observed in multiprocessor systems, often due to the extra memory available in systems with distributed shared memory. However, in the case of clusters augmented with reconfigurable hardware, when utilized effectively, superlinear speedup should be the norm rather than the exception if the FPGAs are not included in the processor count.

Calculating the *maximum speedup* attainable for an application is an important aspect of parallel application design; there is little point in attempting to parallelize an application unless a significant speedup will result. Any parallel application not composed of completely independent processes is composed of some parts that, by their nature, must execute sequentially. Amdahl observed in 1967 that the sequential fraction of a computation limits the maximum speedup that can be achieved, even if all parallel processing resources are fully utilized during the parallel parts. As Amdahl put it: *"the effort expended on achieving*

high parallel processing rates is wasted unless it is accompanied by achievement in sequential processing rates of very nearly the same magnitude" [96]. This observation has become known as *Amdahl's Law*, and although Amdahl's original paper did not contain an equation form of the law, it is usually formulated in terms of multiprocessor systems and expressed as follows:

$$S(n) = \frac{1}{f + \frac{1-f}{n}}$$

where $S(n)$ is the maximum speedup attainable for n processors and f is the fraction of the computation that must be executed sequentially. Even with an infinite number of processors, the maximum speedup is limited to $\frac{1}{f}$:

$$\lim_{n \to \infty} S(n) = \frac{1}{f}$$

It should be noted that this limitation applies equally to Reconfigurable Computing; the fact that the parallel processing resources available are logic gates rather than processors makes no difference. Despite the fact that the law was originally published as an attack on the viability of parallel systems, proponents of parallel processing have argued that any significant speedup is of benefit, particularly in long-running computations.

Others have questioned the reasoning behind Amdahl's Law, in particular the assumption that the amount of computation to be performed is fixed and not dependent on the number of processors available. Gustafson observed in 1988 [97] that in practice users tend to scale the amount of computation performed to the resources available. For example, the grid resolution of a weather prediction application may be increased to improve accuracy. In many cases, it is therefore execution time that is held constant rather than the amount of computation to be performed. Furthermore, it is frequently the case that the sequential fraction of such applications is constant rather than scaling with the amount of computation to be performed. In this case, the *scaled speedup* is given by

$$S_S(n) = n(1 - f) + f$$

where $S_S(n)$ is the scaled speedup for n processors and f is the fixed sequential fraction of the program. Note that the scaled speedup increases linearly with the number of processors. This observation has become known as *Gustafson's Law*.

When deciding on the number and complexity of subtasks in a parallel application, it is essential that inter-task communication does not dominate the overall computation time. The application designer must therefore strike the right balance between maximizing

parallelism in the application and minimizing time spent communicating. A useful metric in this case is the *granularity* (G) or "work/talk" ratio

$$G = \frac{t_{comp}}{t_{comms}}$$

where t_{comp} is the computation time and t_{comms} is the communication time.

Computational Efficiency (E) is a measure of how effectively an application takes advantage of the parallel computing resources available to it, and is given by

$$E = \frac{t_s}{t_p n}$$

where t_s is the sequential execution time, t_p is the parallel execution time and n is the number of processors in the system. Note that if efficiency of utilization of FPGAs as well as CPUs in the system is being considered, then the number of FPGAs present should be included in the processor count. This definition of efficiency does not take into account the limitations imposed by Amdahl's Law; in most situations full efficiency (according to definition above) is unattainable due to the presence of sequential sections in the application. As a result, the following alternative definition of computational efficiency is sometimes given:

$$E = \frac{S_a}{S_i}$$

where S_a is the actual speedup and S_i is the ideal speedup.

1.5 Research Motivation and Objectives

1.5.1 Shortcomings of Current Techniques

The current standard practice when targeting commodity clusters is to leave the task of extracting parallelism from the application and performing network communications to the programmer. Existing sequential languages coupled with message passing libraries are used to reduce the learning curve associated with parallel programming. This approach, although somewhat primitive and labour intensive in comparison with some of the more advanced programming models outlined in Section 1.1.1, is the methodology of choice for the majority of cluster applications. Reasons for the continuing popularity of the message passing approach over alternative methods include better performance, portability and resistance to change due to the large body of existing code and expertise accumulated through many

years of use. However, the resulting applications are large and intricate where the implementation of the desired algorithm is scattered amongst frequent calls to the message passing library.

The complexity imposed by the message passing approach is manageable in simple scatter/gather type applications where the parallelism can be divided evenly over the available resources and all parallel subtasks take the same amount of time to execute. Unfortunately, not every application can be decomposed into identical parallel subtasks, and there may be many more tasks to execute than processors available. In this case, *load balancing* must be performed in order to efficiently utilize the computational resources, whereby unassigned tasks are sent to the most *lightly loaded* nodes, i.e., the nodes with the least amount of work left to perform. Message passing libraries by their nature offer no support to the programmer in situations such as this – load balancing must be performed explicitly.

The lack of load balancing support in message passing libraries is particularly disadvantageous to those wishing to exploit reconfigurable hardware efficiently within cluster nodes. If the resources available (CPUs and FPGAs) are to be exploited most effectively, then they must be utilized as much as possible during the lifetime of the computation. In order to achieve this, the application must partition the computation and balance the load between the CPUs and FPGAs available both on each node and over the cluster as a whole. This situation is further exacerbated by computations where multiple FPGA configurations are in use throughout the cluster, in effect requiring load balancing across heterogeneous resources. Furthermore, this heterogeneity is dynamic in that FPGA reconfigurations take place and need to be planned with a view to minimizing execution time.

Given the difficulties associated with scheduling tasks across clusters augmented with reconfigurable hardware on an application-by-application basis, prudent programmers will trade off this cost with the potential speedups that could be achieved. Complex applications may require a disproportionate effort to load balance effectively. The load balancing effort dominates the application development costs when the number of processors exceeds a particular threshold and/or when the application processes are heterogeneous in size or function. These limitations stem from the lack of expressiveness of the message passing libraries, making it difficult to succinctly express complex policies; the purpose of message passing libraries is to move data from one node to another, they do not assist in deciding where to send it. Furthermore, message passing libraries do not address the sending of data to and from reconfigurable hardware – this task is beyond the scope of their design.

1.5.2   Research Question and Thesis Overview

The prospect of developing complex applications for clusters augmented with reconfigurable hardware can be a challenging one, as evidenced by the previous section. A reasonable question to ask is the following: Would the average programmer, aware of the potentially large speedups attainable using reconfigurable hardware embedded in clusters, attempt to develop for such a system using the techniques currently available?

The answer would appear to be largely in the negative because of the lack of uptake of the technique despite its availability for a number of years. It follows that a more developer-friendly environment may be beneficial if Reconfigurable Computing is to find wider acceptance within the field of Cluster Computing. However, any system attempting to provide a higher-level view by abstracting details will invariably incur a performance overhead. There is a lesson to be learned here from the continued popularity of the message passing paradigm in Cluster Computing – any system that exhibits what is deemed to be an unacceptably high degradation in performance is unlikely to find favour among parallel programmers since the ultimate goal in the field is, after all, high performance. As a result, it can be concluded that any system designed to simplify the task of developing applications for clusters augmented with reconfigurable hardware should make efficient use of the computational resources available as well as the programmer's time.

The Accessible Reconfigurable Computing (ARC) system, which is the subject of this thesis, is a collection of software tools created with the aim of providing the application developer with an efficient high-level abstraction of a cluster augmented with reconfigurable hardware. The overriding design goal during the development of the ARC system was to create an environment where the implementation of applications for clusters augmented with reconfigurable hardware is cleanly separated, insofar as possible, from the details of the hardware upon which they are to be executed. Using this approach, application logic is not cluttered with calls to message passing APIs or to FPGA drivers. Instead, the distinct tasks of implementing the application logic, specifying high-level parallelism and accelerating parts of the application using reconfigurable hardware are treated separately. The most appropriate tools and methodologies can then be used for each of these tasks in isolation, instead of specifying all of these activities as a large and intricate program written in a sequential language, as is the current practice.

The remainder of this thesis is organized as follows: Chapter 2 describes in detail the design, implementation and operation of the ARC system. An example application (cryptographic key search) is presented in Chapter 3. The development of a performance model of the ARC runtime environment is described in Chapter 4, along with a program that automatically generates ARC applications. The ARC load balancing framework is presented

in Chapter 5, and various load balancing algorithms and optimizations are evaluated with the aid of the performance model. Finally, conclusions and future work are presented in Chapter 6.

Chapter 2

ARC: A Distributed Reconfigurable Metacomputing System

The Accessible Reconfigurable Computing (ARC) system [98] is a collection of software tools created with the aim of providing the application developer with an efficient high-level abstraction of a cluster augmented with reconfigurable hardware. The following sections present the motivation behind the project as well as a detailed description of the architecture and operation of the system itself, both at compile-time and at runtime. The process of developing applications for the system is also described through the development of a sample application. A more detailed example of application development will be presented in Chapter 3.

2.1   Design Philosophy

ARC's overriding design goal was to create an environment where the implementation of applications for clusters augmented with reconfigurable hardware is cleanly separated, insofar as possible, from the details of the hardware upon which they are to be executed. Using this approach, application logic is not cluttered with calls to message passing APIs or FPGA drivers. Instead, the distinct tasks of implementing the application logic, specifying high-level parallelism and accelerating parts of the application using reconfigurable hardware are treated separately. The most appropriate tools and methodologies can then be used for each of these three tasks in isolation, in contrast to specifying all of these activities as a single large and intricate program.

1. ANSI C was chosen as the most appropriate language for implementing the application

logic. Despite the increasing popularity of higher-level languages such as C++ and Java, ANSI C is still considered the best choice for high performance applications due to the existence of very efficient compilers and a large body of existing code and libraries that can be leveraged when implementing applications. Performance considerations aside, and despite the fact that ANSI C lacks many of the features (such as exception handling and object-oriented constructs) found in many modern languages, it is still regarded as a very capable procedural language, much more so than its predecessor, Fortran, in the field of High Performance Computing. Although the language has its detractors [99], the continuing popularity of ANSI C is illustrated by the fact that it has consistently ranked highly in the TIOBE Programming Community Index[1], a monthly independent survey of programming language popularity. In the period from January 2001 to the time of writing it has never dropped below second place in the survey.

2. The Condensed Graphs model of computation (see Section 2.3) was chosen as the means of specifying the high-level parallelism within applications. Unlike message passing libraries, the Condensed Graphs model does not require network communications to be explicitly specified within application code; communications can be performed implicitly by the runtime environment based on the high-level relationships contained in the application's definition graphs. Other graph-based means of expressing parallelism and data dependency, such as control-flow and data-flow graphs [100], lack the flexibility of the Condensed Graphs model, especially its ability to allow large, complex graphs to be modularized using condensation and its ability to express multiple sequencing constraints such as demand-driven, data-driven and control-driven modes within the same application. XML was chosen as the file format for specifying graph definitions (see Section 2.4). This format was chosen for its flexibility; XML specifications can be easily created manually using a text editor or high-level GUI. Furthermore, XML documents are simple to generate automatically using a variety of XML APIs on many different platforms.

3. In keeping with the theme of providing the application developer with as much flexibility as possible in terms of implementing the various application components, no particular method of creating FPGA configurations was mandated. Developers are free to use any of the hardware design methodologies outlined in Section 1.2.4, and as such may choose to maximize performance at the cost of increased development time using low-level hardware design languages such as VHDL. Alternatively, they may decide to use a more agile development system such as Handel-C.

---

[1]See http://www.tiobe.com/tpci.htm

Unfortunately, the goal of abstracting the application from the underlying execution environment breaks down somewhat when FPGA configurations are considered. Although well-written ANSI C code and the XML Condensed Graphs file format are completely portable across cluster installations, in most cases FPGA configurations are targeted towards a specific model of FPGA. This drawback is not caused by any defect in the hardware tools; indeed, EDIF files are completely cross-platform and in theory could be compiled for any logic device containing enough gates to implement the logic specified therein. When trying to maximize performance, the parallelism within a hardware design is tuned so as to take advantage of as much of the available resources on a particular FPGA model as possible. One can only hope that in the future this lack of available abstraction can be remedied through the availability of FPGAs containing more logic gates than most applications can leverage.

Linux was chosen as the sole target operating system due to its position as the cluster operating system of choice in the majority of high performance computing centres. Other UNIX variants were not considered; the extra effort involved in writing portable code and performing comprehensive testing across multiple platforms was deemed to be too great when compared with the questionable benefits. One particular obstacle to portability is the limited availability of drivers for reconfigurable boards on more niche platforms such as FreeBSD. In fact, very little Linux-specific functionality is used in the implementation of ARC, so a portable version could certainly be developed without undue effort in the future. A Microsoft Windows port would represent a more formidable undertaking, but this task could be simplified through the use of the Linux-emulating Cygwin tools and libraries.

Once an (albeit imperfect) abstraction of clusters augmented with reconfigurable hardware is available, any such cluster can be regarded from the developer's perspective as a single virtual machine rather than a collection of discrete entities. The task of maintaining the illusion of a single system falls to the compile-time and runtime environments that comprise the ARC system (subsequent sections will describe the operation of these in some detail). If the illusion proves successful, developing for the ARC system can be regarded as a form of declarative programming, where the developer specifies *what* is to be computed rather *how* it is to be computed.

The application development process can be summarized as follows: Starting from a sequential implementation of the application, this implementation is broken down into a collection of atomic operators bound together by a collection of implicitly parallel graph definitions. Next, those atomic operators that lend themselves to hardware acceleration are reimplemented with hardware equivalents. Finally, a compiler is used to convert these elements to a format capable of execution by the runtime environment. Upon execution of the application, the runtime environment performs all communications between cluster nodes

and data transfers between host CPUs and attached reconfigurable hardware. Therefore the runtime environment, rather than the developer, is responsible for ensuring that the computational resources available are utilized in the most efficient manner possible.

It should be noted that the ARC system does not automatically create FPGA configurations. For now, the FPGA configurations necessary for acceleration must be implemented using standard hardware development tools. Nevertheless, the design of the system does not preclude an automation of this task. If, at some point in the future, hardware compilation techniques and the capabilities of reconfigurable hardware advance to the point where ANSI C hardware compilation becomes feasible, then the ARC system will be in a position to take advantage of this development. In this scenario, ARC applications could be specified simply as collections of graph definitions and ANSI C functions, and the ARC compiler would automatically convert the ANSI C portions to hardware configurations as necessary.

2.2   System Overview

The ARC system is composed of a number of individual components, which when taken as a whole implement the vision outlined in the previous section, i.e., the ability for application developers to treat clusters augmented with reconfigurable hardware as a single virtual machine. This section provides a high-level view of the operation of the system, and each of the major components is examined in some detail in subsequent sections. Individual components fall into two categories: those that comprise the compile-time environment and those that comprise the runtime environment.

2.2.1   Compile-time Environment

The compile-time environment is responsible for converting the various application source files to a format capable of execution by the runtime environment. A number of third-party tools are required to complete the compilation process: an ANSI C compiler, a hardware description language compiler and a place-and-route tool. ARC-specific functionality is provided by the Condensed Graphs Compiler and the header files required to compile the native and FPGA operators. The final output is one or more shared object files (suffixed with `.so`, these are the Linux equivalent of Windows DLLs) and one or more FPGA configurations (suffixed with `.bit`, an abbreviation of *bitstream*). Figure 2.1 illustrates the relationship between the various tools that implement the ARC compilation process, showing input files, intermediate outputs and the files comprising the finished application.

The shared object files of an application contain the implementation of operators intended for execution on the host processors of the cluster. These operators can be either

atomic, and implemented using ANSI C, or graph definitions, specified as XML. The Condensed Graphs Compiler (see Section 2.5) is responsible for compiling the graph definitions and incorporating the implementations of the atomic operators into the resulting shared object files. Although the compiler operates directly on the XML graph definition files, the atomic operators must first be compiled to object code (`.o`) format using an ANSI C compiler before being linked into a shared object file. Each object file can contain multiple operator definitions and ancillary functions.

To reduce the complexity of the Condensed Graphs Compiler's command-line interface, the decision was taken to require that all atomic operators be precompiled before the CGC is invoked; it was deemed counterproductive to add an extra layer of indirection to already convoluted compiler command line interfaces. The GNU C Compiler (GCC) [101] was used for the compilation of all atomic ARC operators developed so far, and for the implementation of the metacomputer itself. GCC was chosen for its ubiquity on Linux platforms across machine architectures rather than for performance reasons. Better-performing compilers, such as Intel's C++ Compiler [102], can be used where available.

Those operators destined for execution on reconfigurable hardware contained in the cluster are developed using standard hardware development tools. Header files that simplify communications with the ARC runtime environment are provided for Handel-C; similar headers for other hardware description languages could be implemented with little effort. Whatever the choice of hardware description language, the resulting EDIF (`.edf`) files must then be passed through a place-and-route (P&R) tool to produce a bitstream compatible with the specific model of FPGA contained in the cluster. Although no technical reason exists to prevent bitstreams from being linked directly into application shared object files, it was deemed more convenient to distribute them separately due to their relatively large size (up to several Megabytes each).
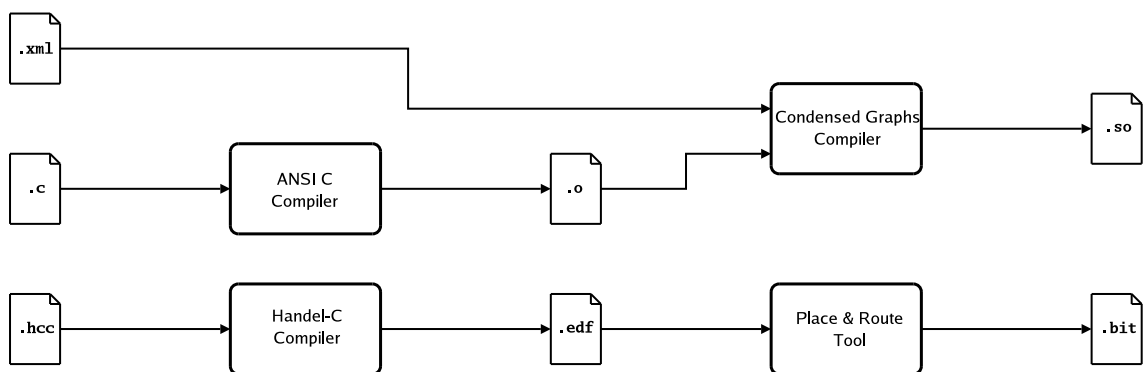

Figure 2.1: The process of compiling a ARC application from its constituent source files. Handel-C was chosen as an example hardware description language; in practice any HDL could be used.

2.2.2   Runtime Environment

The role of the runtime environment is to physically execute ARC applications, and to present a simple interface that preserves, as far as is practical, the illusion of a single virtual machine. This illusion is realized by the ARC daemon running on each cluster node, and the computation processes spawned by the daemons when applications are executed. A user interface program is also provided for starting, stopping and monitoring ARC application instances (see Figure 2.2).

A typical use case for the runtime environment is as follows: First, the user copies the executable files produced by the compile-time environment and any required data files to a shared directory in the cluster. Next, the UI program (see Section 2.6) is used to start computation process instances (see Section 2.8) on the requisite number of cluster nodes. Each application contains a single top-level graph instance (the ARC equivalent of a C `main` function), and application execution commences with the creation of a top-level graph instance in a computation process on an arbitrary cluster node. Any parameters required by the top-level graph are specified by the user and passed by the UI program to the cluster node on which the top-level graph is created.

As nodes in the top-level graph fire, *instructions* are generated, with each representing either an atomic operation or an unevaluated subgraph[2]. These instructions are distributed in a peer-to-peer fashion amongst the computation processes executing on the cluster. The scheduling component of the computation process is responsible for spreading the computation load as evenly as possible across the participating cluster nodes, minimizing overall execution time and thus maximizing computational efficiency. Once an instruction has been evaluated, the resulting data value is plugged back into the graph instance that created it, progressing the computation and possibly uncovering further parallelism.

Instructions are of three categories, depending on the type of operator present at the node in the graph instance that created them (see Section 2.3 below). triple manager (TM) instructions represent node manipulation operations to be performed on a particular graph instance; these are always executed locally. Condensed graph instructions represent new graph definition instances and can be executed either locally or on another cluster node, but always on the host processor. Primitive instructions represent atomic operations that can be evaluated anywhere. The corresponding primitive operator can have an implementation in software (as a native operator), or in hardware (as an FPGA operator), or both. If an operator has implementations both in software and hardware, evaluation can take place either on a host CPU or FPGA, with the scheduling component of the supervising computation process deciding which is most appropriate in order to minimize overall execution time.

---

[2]In Condensed Graphs terminology, the term *instruction* denotes an operation of arbitrary complexity (or *task*) rather than the more common meaning of individual microprocessor instructions.

Execution of a computation continues until the computation as a whole has been evaluated, i.e., the top-level graph has completed execution. This process can be stopped prematurely through the use of the `allDone` TM operator. The use of `allDone` is somewhat inelegant but necessary in some cases. For example, when one of the parallel branches in a search application finds a goal state, exiting prematurely avoids the delay of waiting for all the other branches to continue searching fruitlessly. Execution may also be disrupted through unforeseen circumstances, such as a segmentation fault produced by an incorrectly coded native operator. Finally, the user may halt execution manually via the UI program.

The operation of the ARC runtime environment can now be summarized as follows: an executing computation is composed of a distributed collection of graph instances. Any of these instances can generate instructions, which are in turn distributed throughout the cluster for execution. Instructions, once evaluated, produce results that are returned to the appropriate machine and incorporated into the graph instance that created the corresponding instruction. Result values become operands for other graph nodes and thus trigger the creation of new instructions. This process continues until evaluation is completed, an error occurs, or the computation is halted by the user.

While a computation is in progress, its state can be monitored in real time using the UI program. Extensive log messages are sent to the UI program by the computation processes that comprise a metacomputer instance, and, if present, from application code. By examining these messages, the user can follow the evaluation of the computation by watching where instructions are generated and consumed. Problems such as logic errors in graph definitions, native operators and FPGA operators can be diagnosed by isolating the exact moment during the life of the computation at which the problem arises. For example, a fault in an FPGA operator could be discovered by observing that operands were passed to an instance of the operator but no corresponding result was received. If the user is confident in the correctness of the application, remote logging can be disabled to improve performance and conserve network bandwidth, although this may alter the behaviour of the application

Figure 2.2: At runtime, users start, stop and monitor application instances via the UI program. ARC daemon instances and computation processes executing on each cluster node create the illusion of a single virtual machine.

due to the nondeterministic nature of the runtime environment.

The distributed debugging example presented above demonstrates that it is desirable in some cases to relax the virtual machine abstraction and expose execution details directly to users of the system. The abstraction, after all, is intended as an aid to the application developer rather than an end in itself.

2.3   The Condensed Graphs Model of Computation

The *Condensed Graphs* ($\mathcal{CG}$) model of computation [103] plays a fundamental role in the operation of the ARC system by providing the means through which the high-level parallelism in ARC applications is expressed. Although superficially similar to classical dataflow [104] in that computations are represented by directed acyclic graphs, the Condensed Graphs model is unique in that it allows three different evaluation strategies (availability driven, coercion-driven and control-driven) to be incorporated within the same computation. As a result, various sections of a computation may execute in strict sequential order (control-driven), eagerly attempt to evaluate all available execution paths in a speculative fashion (availability-driven), or lazily perform work only when absolutely necessary (coercion-driven).

The ability of the model to incorporate these different evaluation strategies stems from the central notion of a *triple*. A triple is formed when a complete set of *operands*, an *operator* and one or more *destinations* are present at a node (see Figure 2.3). Once a triple has been formed, the node can *fire*, i.e., the operands are passed to the operator function and the resulting datum is passed to the destinations in order to progress the computation. Restricting the availability of the different triple elements (operands, operator and destinations) of a node results in the three different possible evaluation strategies. For example, a node that has an associated operator and a destination but lacks a full complement of operands cannot fire until the missing operands become available, resulting in an availability-driven execution strategy. Similarly, restricting the availability of operators and destinations results in a control-driven or data-driven strategy, respectively.

The various triple elements attach to a node at a predetermined number of *ports*. Each node contains a single operator port and destination port[3], with the number of operand ports present determined by the arity of the operator function. Each operand port has a *strictness* property that specifies whether or not simple data values are required for the evaluation of the operator function. A *strict* operator port therefore requires that an operand consisting of an unevaluated subgraph must be reduced to a simple data value before the node can fire. On the other hand, *nonstrict* operand ports allow unevaluated subgraphs to be operated on or passed through by operators.

---

[3]Enter nodes (see below) are the sole exception to this rule.

In order to allow subgraphs to be evaluated lazily, a mechanism is required that allows subgraphs to be associated with an operand port but prevented from executing in a speculative manner. This mechanism is known as *stemming*. Rather than the typical speculative arrangement where one node lists the operand port of another as a destination, a stemmed node is created by removing this destination and specifying the node itself as an operand to the destination node. The desired association between the two nodes is therefore specified, but the first node cannot fire because it lacks a destination. Once the second node becomes fireable, any operands that are stemmed nodes are *grafted* by removing them from the destination node's operand port and recreating the destination from the first node to the second. The first node is now fireable, and once the subcomputation it is associated with has been evaluated, the resulting datum will be placed onto the operand port to which it was attached.

Figure 2.4 illustrates the concepts of strictness and stemming by showing how the `Ifel` branching operator, when used in conjunction with nonstrict operand ports, can be used to select between a pair of (unevaluated) subcomputations. The `Ifel` operator takes three operands; a boolean value that determines which branch should be selected and two branch values, which may be of any type. The simplest use of the `Ifel` operator is to select between two simple data values. In this case, all the `Ifel` node's operand ports will be strict, with the value on the branch indicated by the boolean operator placed on the destination arc(s) upon firing. However, when selecting between two subcomputations the branches must be stemmed in order to prevent evaluation before the `Ifel` node fires. Although the subcomputations in Figure 2.4 are represented by single nodes, large unevaluated subgraphs can be built up by "daisychaining" collections of nodes using stemming. Once the last node in the chain becomes fireable, it will coerce its operand nodes to fire by grafting (assuming that the operand port is strict), which in turn coerces others and so on. This rippling effect supplies all the destinations required to make the whole subgraph executable. In some cases, it may be desirable to pass subgraphs unevaluated through `Ifel` nodes in order to create a
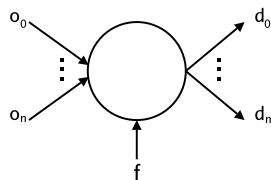

Figure 2.3: A triple is formed when a node has a full complement of operands $(o_0..o_n)$, an operator (f) and one or more destinations $(d_0..d_m)$.

larger subgraph for later potential execution. This behaviour is achieved by specifying the branch operand ports of the node as nonstrict, and thus preventing the evaluation of the chosen subgraph. Note that the Boolean operator port must always be specified as strict because a simple Boolean data value is required in order to make the branching decision.

Rather than representing computations as a single large graph, the $\mathcal{CG}$ model incorporates (and, indeed, is named for) the notion of *condensation*. $\mathcal{CG}$ computations are expressed as collections of graph definitions, each of which can be represented by a *condensed node* which upon firing expands (or *evaporates*), incorporating an instance of the associated graph definition into the computation. Memory management is simplified by the fact that subgraphs can be allocated as needed and deallocated when no longer required, allowing the computation graph to grow and shrink as necessary. The problems of seeding new graph instances with operand values and returning results are solved by requiring that each graph definition contains both a single *Enter node* and a single *Exit node*, denoted by E and X respectively. Enter nodes have a number of operand ports corresponding to the arity of the function implemented by the graph definition and a corresponding number of destination ports (each of which can have multiple destinations). Upon firing, the operand values are simply copied to the corresponding destination ports where the destination arcs distribute them throughout the graph as needed. Exit nodes have a single operand port and a single destination port. Upon firing, the single operand value is copied to the destinations associated with the condensed node that produced the graph instance, and the graph is deallocated.

In order to implement an application using the $\mathcal{CG}$ model, developers must specify a collection of operators. Operators are divided into three categories: condensed graphs, primitive operators and TM operators. Condensed graph operators represent graph definitions and are used to create condensed nodes as described above. Primitive operators are used to create atomic value-transforming instructions. TM (Triple Manager) operators are
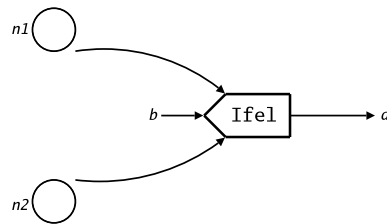

Figure 2.4: A node with an associated Ifel operator used to select between two unevaluated nodes ($n1$ and $n2$) depending on the value of the boolean operand b. Stemming (denoted by unconnected arcs) is used to prevent evaluation of the nodes before the Ifel node fires. The use of strict or nonstrict operand ports determines whether the chosen node is evaluated on selection or passed to the destination (d) unchanged.

similar to primitive operators but operate on nodes rather than tuples of operands. As such they are used for node manipulation operations, such as the implementation of the `Enter`, `Exit` and `Ifel` operators. For the vast majority of applications there is no need to create custom TM operators; the predefined TM operators described above should suffice.

Any graph-based model of computation should ideally provide some means of handling side-effects. Although the $\mathcal{CG}$ model incorporates the notion of state through its control-driven semantics, other means of expressing side effects are also possible. One approach, from functional programming, is the use of monads to enforce sequencing constraints. However, although the use of monads in purely functional languages is desirable due to the inability of these languages to guarantee execution order, the $\mathcal{CG}$ model *does* allow sequencing constraints to be enforced through data dependency and stemming. It is therefore possible to incorporate state-changing operators into $\mathcal{CG}$ computations provided that care is taken to sequence the execution of nodes making use of these operators appropriately.
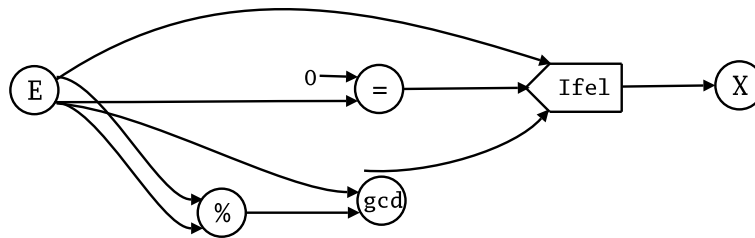

Figure 2.5: A recursive condensed graph implementation of the Euclidean greatest common divisor algorithm. The graph definition accepts two parameters (a quotient and a remainder) and recursion continues until the remainder is zero. Note how the multiple destination ports of the Enter node (`E`) are used to seed the graph with operand values and the single operand port of the Exit (`X`) node is used to specify the return value.

2.4   XML Definition Files

Given the central role of the Condensed Graphs model within the ARC system as the "glue" that binds together lower-level application components (e.g., native and FPGA operators), the development of a file format for specifying graph definitions was a necessary part of the development of the system as a whole. The tool chosen for this task was the Extensible Markup Language (XML) [105], a restricted form of SGML developed by the World Wide Web Consortium (W3C). XML allows structured data to be simply represented as marked-up text documents, a format easily created, edited and parsed both by humans and computers. Although superficially similar to HTML by structuring documents using elements and attributes, XML is more restrictive in terms of syntax (i.e., elements must always occur in pairs) but much more general in terms of expressiveness in that all elements

and attributes are user-defined.

Humans can work with XML documents using either a simple text editor or dedicated XML editors that present the user with a tree-like view of documents. Similarly, a wide variety of tools and techniques are available to developers who wish to work with XML programmatically, from simple event-driven parsers such as SAX [106] to tree-based representations that allow documents to be created and manipulated in memory, such as the Document Object Model (DOM) [107]. The availability of generalized XML APIs such as these eliminate the need to develop the custom parsers and in-memory representations associated with dedicated languages, whilst offering much more flexibility and expressiveness than simpler data representation schemes such as comma separated values (CSV).

Another notable feature of XML is that documents can be *validated*, i.e., automatically verified to conform to a user-defined set of constraints. For example, restrictions can be placed on the number of each type of element, the attributes they may contain and how they may be nested within one another. Two principle methods of validation are available: Document Type Definitions (DTDs) and XML Schema. DTDs allow simple rules, such as those described above, but are limited in terms of the complexity of the rules that may be expressed. For example, it is not possible to allow a type of element to have different behaviour depending on its parent element. XML Schema was designed to address issues such as these, and allows practically any type of restriction to be expressed. Unfortunately, many XML APIs do not yet support validation using XML Schema, with the result that for now DTD is still the *de facto* standard. The benefits of validation are manifold; from the developer's point of view validation acts as a "sanity check" on incoming data, reducing the amount of error-checking code required when parsing. Applications that generate XML can ensure that their output is structured correctly. End users also benefit through the use of XML editors that automatically validate documents as they are typed, preventing the accidental creation of erroneous documents.

The platform-agnostic nature of XML confers many advantages over custom binary file formats. Tools for working with XML are available for a wide variety of operating systems and programming languages, facilitating the movement of data across platforms. Apart from the editors and APIs mentioned above, users can leverage many other XML-specific tools for data-processing tasks. Dedicated XML compressors such as XMill achieve compression rates far in excess of those attainable using standard compression algorithms [108]. Documents can be easily translated to other XML formats such as XHTML using the XSL Transformations (XSLT) framework [109]. Tools have been developed that automatically generate code for converting XML to and from data objects, for example Sun's Java-to-XML Binding (JAXB). These examples represent a small sample of the myriad of tools available, most of which are released under liberal licenses.

The widespread adoption of XML for data representation has led to criticism from a number of quarters. Much of this criticism, particularly relating to security concerns, relates specifically to the use of XML for implementing Web Services, a topic not relevant here. Criticism directed towards XML for data representation purposes often focuses on the relatively large size of XML documents compared to equivalent binary representations, and the performance overhead introduced by the need to parse large text files. Proponents of XML argue that if size is a consideration, XML compresses very well due to its strict syntax rules, and that the parsing overhead is a necessary tradeoff for the convenience of human-readability. Others have noted that the functionality of XML is already attainable through the much older technology of LISP S-expressions; XML advocates respond that XML has become a widely-adopted standard, a position that S-expressions never achieved.

The XML file format devised for ARC is similar to that used in the WebCom system [110][111], but with modifications where necessary to facilitate ARC-specific features. As specified by the XML standard, each document contains a single root element (`<definitions>` in this case) within which all other elements are nested. Three types of elements may be children of the root: graph definitions, operators and user-defined types.

Graph definitions are specified as collections of named nodes. Each graph definition (`<graphdef>`) element has a `name` attribute that uniquely identifies the graph definition within the application. Similarly, each `<node>` element contains a `name` attribute that uniquely identifies each node within the graph definition. Node names provide a means for nodes within a graph definition to reference one another. Each node element also contains one `<operandport>` element for each operand required by the node's operator, optional

```
<definitions>
    <graphdef>
    ⋮
    </graphdef>

    <operator>
    ⋮
    </operator>

    <type>
    ⋮
    </type>
</definitions>
```

Figure 2.6: High level-structure of ARC XML definition document. The document root element (`<definitions>`) may contain three types of child element: graph definitions, operators and user-defined types.

`<operand>` elements for hardcoded operand values, one `<operator>` element specifying the node's operator, and at least one `<destinationport>` element.

`<operandport>` elements contain a `strictness` attribute that specifies whether the operand port in question is strict or nonstrict. `<operand>` elements contain three attributes: `operandport`, `type` and `value`. The `operandport` attribute specifies the number of the operand port with which the operand value is to be associated. `type` specifies the type of the operand. `value` specifies the actual operand value. The `type` attribute may be the name of one of the ANSI C primitive types, or `graph`, a special value denoting a graph operand (i.e., a stemmed node). If the operand type is `graph`, then the `value` attribute specifies the name of the node to be used as the operand value. `<operator>` elements contain a single attribute specifying the name of the operator to be associated with the node's operator port, and as such may be either the name of a graph definition or an atomic operator declared elsewhere. Recursive graph definitions can be constructed simply by using the name of the graph as the operator value. Although the `<operator>` element is mandatory, nodes can be created without operators by omitting the `name` attribute, allowing for control-driven semantics. With the exception of `Enter` nodes, each node element contains only one `<destinationport>` element, which in turn contain zero or more `<destination>` elements. `<destination>` elements have two attributes; the name of the node that the destination points to (`nodename`) and the number of the target operand port of the destination node (`operandport`).

`<type>` elements are used to declare ARC types. At compile-time types enable the structure of graphs to be verified by preventing mismatches between the types of the return values of operators and the operand ports to which they are connected. At runtime, type information is used both for debugging purposes by allowing any value present within the system to be converted to an equivalent text representation, and for serializing and deserializing data values for transmission either over the network (to peers) or over a local

```
<node name="equals1">
      <operandport strictness="strict"/>
      <operandport strictness="strict"/>
      <operand operandport="1" type="float" value="1"/>
      <operator name="float_equals"/>
      <destinationport>
            <destination nodename="ifel" operandport="0"/>
      </destinationport>
</node>
```

Figure 2.7: An example node element that uses the `float_equals` operator to test whether the first operand value is equal to 1. The resulting data value is sent to the first operand port of the node named `ifel`.

bus (to FPGAs). A type declaration is therefore composed of a name, a string conver-
sion function, a serialization function and a deserialization function. Each `<type>` element
contains a mandatory `name` attribute, specifying the name of the type. A default naming
convention is in place for each of the other functions: the name of the type, followed by an
underscore which is in turn followed by `tostring`, `serialize` or `deserialize`, depending
on the function category. Additional child elements may be used to specify the names of
these functions explicitly if the naming convention is not followed.

```
<!-- List type declaration -->
<type name="List">
      <tostring>List_tostring</tostring>
      <serializer>List_serialize</serializer>
      <deserializer>List_deserialize</deserializer>
</type>
```

Figure 2.8: An example type declaration for the `List` type. Although the `tostring`,
`serializer` and `deserializer` function names are shown for clarity, they could be omitted
because they follow the default naming convention.

`<operator>` elements are used to declare the atomic operators referenced by graph
definitions. Two mandatory attributes are used to specify the name of the operator and its
category. Allowed values for the `category` attribute are `primitive` and `TM` for primitive and
Triple Manager operators, respectively. Condensed graph operators are defined implicitly
through their graph definitions. Elements declaring primitive operators may also contain
a `<typesig>` child element, and may also contain an optional `<fpgaimpl>` element. Both
categories of operator may contain a `<nativeimpl>` child element. `<typesig>` elements
enclose lists of comma separated type names. In a scheme similar to that used by the
Haskell programming language, one type name is required for each operand that the operator
requires, along with one more representing the type of the return value. TM operators do
not require type signatures because they operate on nodes rather than on tuples of operand
values; they have an implicit type signature of one node mapping to another. `<nativeimpl>`
elements specify the name of the ANSI C function that implements the operator natively
using a mandatory `name` attribute. If the `<nativeimpl>` is omitted, a native implementation
of the same name as the operator is assumed unless an FPGA implementation is also
provided, in which case the `<nativeimpl>` element may be required for disambiguation
purposes (see below). `<fpgaimpl>` elements are used to indicate that a primitive operator
has an implementation in hardware. Two mandatory attributes, `name` and `clockspeed`, are
required to indicate the name of the file containing the FPGA bitstream and the speed (in
MHz) at which it should be clocked. If a primitive operator has implementations both in
hardware and software then both must be explicitly declared even if the naming convention is

followed; this is to allow primitive operators to be declared with hardware implementations only. Note that this scheme allows only stream-based communications with the FPGA, as the operands are serialized and the result is deserialized. In some situations, finer control over the FPGA may be required for performance purposes. For example, it may be necessary to manage on-board resources such as memory banks, or have finer control over what is communicated to the FPGA and whether or not DMA is required. An extension to ARC to accommodate schemes such as this is left as future work.

```
<!-- mandelbrot_draw operator declaration -->
<operator name="mandelbrot_draw" category="primitive">
     <typesig>int, int, int, int, float, Image</typesig>
     <nativeimpl name="mandelbrot_draw"/>
     <fpgaimpl name="mandelbrot.bit" clockspeed="40"/>
</operator>
```

Figure 2.9: Declaration of a primitive operator that has both a native and an FPGA implementation.

2.5   The Condensed Graphs Compiler

The Condensed Graphs Compiler (CGC) provides the means through which applications expressed as XML documents according to the schema described above are converted to a format capable of execution by the ARC runtime environment. The operation of the compiler proceeds in two distinct steps. First, an ANSI C file is created that contains, amongst other things, implementations of the graph definitions described in the XML document. Next, the resulting file is compiled to object code format and linked with the implementations of the native operators referenced by the graph definitions to produce a shared object that can be distributed and executed at runtime (see Figure 2.10).

Unlike the runtime environment, where high performance is a necessity, correctness and maintainability were deemed to be the overriding design goals during development of the compiler. For this reason, the compiler was implemented using the Python Programming Language [112]. The performance overhead incurred through the use of an interpreted scripting language was more than compensated for by the clarity and conciseness of the resulting code. In particular, the advanced string handling capabilities of the Python language compared to traditional languages greatly simplified the development of the code responsible for translating the XML documents to ANSI C.

The compiler is invoked from the command line using the cgc command. The only mandatory parameter is the XML definition file to be compiled. In practice, an additional

parameter specifying the names of the object code files containing the native implementations of the operators referenced by the graph definitions is also supplied (it is possible, but of limited usefulness, to create graph definitions using only the standard operators automatically loaded at runtime). Additional parameters allow the name of the output file to be specified, the verbosity of the compiler to be adjusted and parameters to be passed to the linker upon invocation. An example invocation of the compiler is as follows:

```
# cgc -i factorialops.o factorial.xml
```

This command specifies that the definition file `factorial.xml` should be compiled and that the resulting object code file should be linked with `factorialops.o` to produce the default output file `factorial.so`.

Upon execution, the compiler begins by parsing the input XML files to an in-memory representation using a SAX parser. The in-memory representation is used to facilitate graph verification and rewriting operations, which would be difficult to implement using a tree-based representation such as DOM. The in-memory representations of the graph definitions are then checked to ensure that the graphs are well-formed, i.e., that instances of the graphs terminate correctly due to the presence of a full complement of valid operands, operators and destinations at each node, and that no unstemmed circular references between graph definitions are present. Once these checks are complete, ANSI C functions are generated implementing each graph definition. Each function generated shares the name of the XML
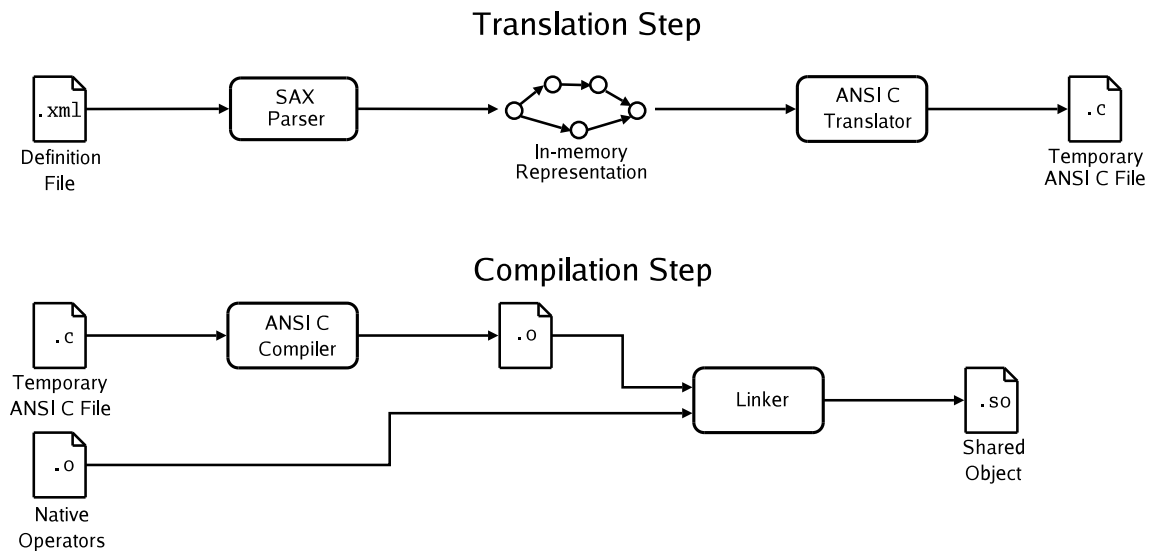

Figure 2.10: An overview of the operation of the Condensed Graphs Compiler. The translation step converts the definition file to a temporary ANSI C file via an in-memory representation. This file is then compiled and linked with the required native operators to produce a shared object capable of execution by the runtime environment.

definition from which it was created, and uses an API available at runtime to build and return an instance of the specified graph. These functions are written to a temporary ANSI C file along with four *registration functions* (one for each operator category, and one for types). The purpose of the registration functions is to allow the runtime environment to determine the operators and types provided by a shared object file once it has been loaded. The registration functions follow a strict naming convention, and when invoked by the runtime environment use a series of API calls to add entries to the operator and type information tables of the computation process.

Once the temporary ANSI C file has been created, the GCC compiler is invoked to produce a corresponding object code file. The GCC compiler is then invoked again, but in linking mode, to link the newly-created object code file with the object code files containing the native implementations of the operators declared in the XML file. Any linking options specified when the CGC was invoked are passed directly to the linker, allowing the user to include libraries if necessary. The result is a shared object file that can be distributed at runtime and dynamically loaded and executed by the runtime environment.

The `Square` graph definition shown in Figure 2.11, although very simplistic and impractical, serves as a suitably concise example application to demonstrate the operation of the compiler. The graph definition uses the primitive `int_mul` operator to square its single integer operand. The single destination port of the `Enter` node has two destinations, replicating the operand and placing copies on the operand ports of the `int_mul` node. A graph definition document describing `Square`, and the resulting ANSI C file generated by the condensed graphs compiler are shown in Figures 2.13 and 2.12 respectively.
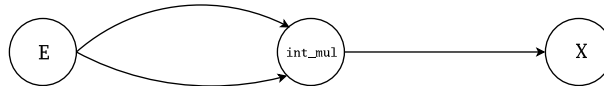

Figure 2.11: The `Square` graph definition.

2.6   The ARC User Interface Program

The ARC User Interface Program (UIP) allows ARC applications to be launched and monitored by users. Applications may be launched from the command line using the UIP in non-interactive mode, or by issuing a series of interactive commands at the UIP command prompt. Applications may also be launched without the use of the UIP via an API. Any executing application may be monitored by the UIP, regardless of how it is launched. Applications are monitored by watching log messages arriving in real-time from the cluster nodes participating in the computation.

```
<definitions>
    <graphdef name="Square">
        <node name="E">
            <operandport strictness="strict"/>
            <operator name="Enter"/>
            <destinationport>
                    <destination nodename="mul" operandport="0"/>
                    <destination nodename="mul" operandport="1"/>
            </destinationport>
        </node>
        <node name="mul">
            <operandport strictness="strict"/>
            <operandport strictness="strict"/>
            <operator name="int_mul/">
            <destinationport>
                    <destination nodename="X" operandport="0"/>
            </destinationport>
        </node>
        <node name="X">
            <operandport strictness="strict"/>
            <operator name="Exit"/>
            <destinationport/>
        </node>
    </graphdef>
</definitions>
```

Figure 2.12: A definition file implementing the `Square` graph definition.

2.6.1   Executing Applications

Once the UIP is launched in interactive mode (the default), the user is presented with a prompt at which commands may be entered. The first task when launching an application interactively is typically to inform the UI program of the number and identity of the cluster nodes that will be participating in the computation. This is achieved via node files – simple text files with one machine name or IP address per line. Assuming that a text file called `allnodes` containing the names of all the nodes in the cluster to be used is present, the following command will inform the UI program that the specified nodes are to be used:

```
-> setnodes allnodes
```

The availability of each of the nodes specified can be determined using the `nodes` command. Each address is checked to ensure that it resolves and that each machine can be reached using `ping`:

```
-> nodes
```

```
Nodes Square() {
     Nodes nodes;
     Node node_array[3];
     int i;

     /* Instantiate nodes */
     for(i=0; i<3; i++) {
          node_array[i] = NodeMake();
     }

     /* Add operand ports */
     NodeSetPort(node_array[0], PortSet(PortMake(), operand_port, 0, strict));
     NodeSetPort(node_array[1], PortSet(PortMake(), operand_port, 0, strict));
     NodeSetPort(node_array[1], PortSet(PortMake(), operand_port, 1, strict));
     NodeSetPort(node_array[2], PortSet(PortMake(), operand_port, 0, strict));

     /* Add operator ports */
     NodeSetPort(node_array[0], PortSet(PortMake(), operator_port, 1, strict));
     NodeSetPort(node_array[1], PortSet(PortMake(), operator_port, 2, strict));
     NodeSetPort(node_array[2], PortSet(PortMake(), operator_port, 1, strict));

     /* Add destination ports */
     NodeSetPort(node_array[0], PortSet(PortMake(), destination_port, 2, strict));
     NodeSetPort(node_array[1], PortSet(PortMake(), destination_port, 3, strict));
     NodeSetPort(node_array[2], PortSet(PortMake(), destination_port, 2, strict));

     /* Add operands */

     /* Add operators */
     NodeSetOp(node_array[0], OperatorSet(OperatorMake(), TMInstr, "enter"));
     NodeSetOp(node_array[1], OperatorSet(OperatorMake(), Primitive, "int_mul"));
     NodeSetOp(node_array[2], OperatorSet(OperatorMake(), Primitive, "Exit"));

     /* Add destinations */
     NodeSetDest(node_array[0], NodeDest(node_array[1], 0));
     NodeSetDest(node_array[0], NodeDest(node_array[1], 1));
     NodeSetDest(node_array[1], NodeDest(node_array[2], 0));

     /* Insert nodes into the node list */
     nodes = NodesMake();
     for(i=0; i<3; i++) {
          nodes = NodesInsert(nodes, node_array[i]);
     }

     /* Return node list */
     return nodes;
}
```

Figure 2.13: The ANSI C implementation of the `Square` graph definition produced by the Condensed Graphs Compiler.

```
node01.cuc.ucc.ie      143.239.211.107      OK
node02.cuc.ucc.ie      143.239.211.108      OK
node03.cuc.ucc.ie      143.239.211.109      OK
node04.cuc.ucc.ie      143.239.211.110      OK
node05.cuc.ucc.ie      143.239.211.111      OK
node06.cuc.ucc.ie      143.239.211.112      OK
node07.cuc.ucc.ie      143.239.211.113      OK
node08.cuc.ucc.ie      143.239.211.114      OK
```

Next, the cluster nodes must be prepared for application execution by first ensuring that each is running an ARC daemon and then ordering each daemon to spawn a computation process:

```
-> startall
```

If any of the nodes in question do not have an ARC daemon running, then the user will be prompted for a password and one will be started using `ssh`. The cluster is now ready to commence executing an application. First, the shared object containing the application implementation must be loaded by all the participating nodes. It is assumed that the application is available in some shared network location. Any other shared objects required by the application are loaded automatically by the runtime environment:

```
-> load /usr/share/arc/examples/gcd/gcd.so
```

Finally, execution is commenced by specifying the name of a graph definition and any arguments it requires. As the metacomputer operates in a peer-to-peer fashion, the cluster node on which execution commences is irrelevant. The UI program chooses one node at random and informs the computation process on that node to create and execute an instance of the specified graph. Any text printed to standard output by the application on any cluster node is redirected and printed after the prompt. The return value of the computation is also printed automatically on program termination.

```
-> execute GCD int:24 int:32
Result: 8
```

Applications may be terminated prematurely using the `stopall` command, which shuts down the computation processes executing on all participating cluster nodes. This feature is particularly useful during testing when errors in applications can lead to the system hanging or crashing.

Although the ability to interactively manage computations using a prompt is useful in many respects, the constant re-typing of commands each time an application is launched quickly becomes tedious. For this reason, features are in place to allow applications to be started non-interactively. The simplest method of doing this is to use command-line arguments when starting the UIP. Alternatively, commands may be grouped into batch files which are executed sequentially by the UIP. For example, the following shell command has the same effect as all the UIP commands above combined:

```
# uip -n allnodes -l /usr/share/arc/examples/gcd/gcd.so -e "GCD int:24 int:32"
Result: 8
```

2.6.2   Monitoring Applications

Upon startup, the UI program automatically starts another process that listens for logging messages on a specified port. When the UI program starts a computation process on a remote machine, it informs the computation process that all log messages should be redirected to the specified port on the machine executing the UIP. In this way, all logging messages generated throughout the computation may be stored and viewed in aggregate on the user's machine. These messages may be viewed as they arrive either by using the `tail` command on the resulting aggregate log file, or by using the log viewing GUI (see Figure 2.14). When using the GUI, the messages may be viewed either in aggregate or organized by the cluster nodes from which they originated. The ability to view distributed log messages in real-time is an invaluable aid to debugging applications, as the user can observe which instructions executed on each machine as well as the order of their execution. A compile-time option is available that includes the creation time in log messages, allowing the order of execution to be determined across machines. However, this feature is of limited usefulness unless the Network Time Protocol daemon or similar is used to synchronize the clocks of the cluster machines.

2.7   The ARC Daemon

The ARC daemon is a lightweight process that executes on each cluster node. The role of the daemon is to spawn more heavyweight computation processes when requested to do so. Separate processes are used for computation in order to isolate failures; an unrecoverable error within a computation (e.g., a segmentation fault caused by a poorly-written native operator) should not lead to the failure of the entire system. When an error does occur, the daemon is still running, allowing the existence and nature of the error to be communicated to the UI program and another process started if requested. The modular nature of computation processes also allows for multiple computations to proceed simultaneously, although
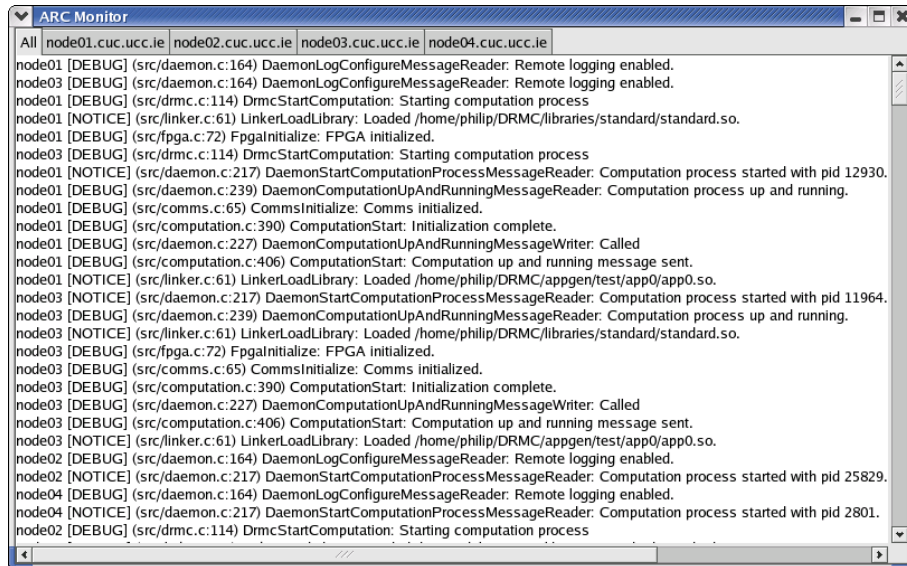
Figure 2.14: A screenshot of the ARC log viewer displaying the aggregate output of the four participating cluster nodes.

they would have to contend with one another for the computational resources available.

The daemon is written in ANSI C and follows the standard UNIX daemon paradigm described in [113]. The daemon responds to two types of message from the UI program executing on the user's machine. The first is to redirect logging messages to the log listener on the user's machine, allowing important information such as computation process failures to be relayed back to the user in real-time. The second type of message is to spawn a computation process. Messages are sent and received using a `Messenger` object identical to that used by communications module of the computation process (see Section 2.8.4). Once a computation process commences executing, the UI program communicates directly with it and its log messages are routed back to the user's machine without passing through the daemon. The daemon and the computation process also intercommunicate via the same socket mechanism used by the UI program. For example, the computation process informs the daemon once all initialization has completed successfully. Limited intercommunication also occurs via the UNIX signaling mechanism; this is how the daemon detects the occurrence of a computation process failure and determines the cause.

2.8   Computation Process Architecture

Computation processes are responsible for executing ARC applications across clusters in a manner that makes efficient use of the computational resources available. As performance was a design priority, ANSI C was used for implementation, in conjunction with a number
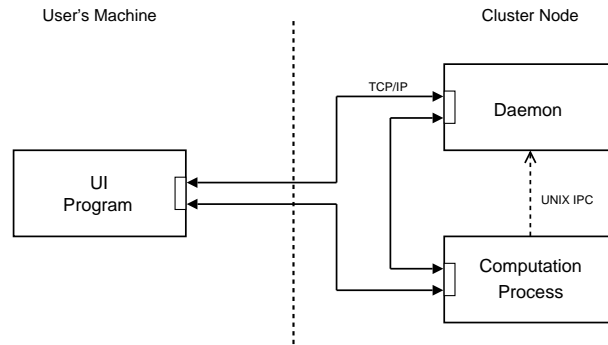
Figure 2.15: The UI program communicates directly with daemons and computation processes using sockets. The daemon and the control process intercommunicate using both sockets and UNIX signals.

of external libraries such as the Condensed Graphs API, POSIX Threads and FPGA device drivers. Computation processes are spawned by ARC daemons on the user's request, and terminate either when the computation is complete, an unrecoverable error has occurred, or the user halts the computation. One computation process is spawned on each participating cluster node for every computation.

Individual computation processes are composed of a number of independent modules, each of which runs in a separate thread (see Figure 2.16). The Condensed Graphs Engine is responsible for maintaining the computation graph by generating instructions and incorporating results. The Native and FPGA Instruction Execution Threads, as their names suggest, execute native and FPGA instructions, respectively. The Communications Module handles the exchange of instructions and results with computation processes on other cluster nodes, as well as messages to and from the UI program and daemon. Finally, the Scheduler is responsible for deciding where instructions are executed (i.e., locally or remotely, natively or on an FPGA) as well as the order in which they are executed. Each of these modules will be examined in detail in subsequent sections.

The fundamental data objects used by computation processes are *instructions* and *results*. Instructions represent unevaluated portions of the computation. Note that the term "instruction" is used here to refer to operations of arbitrary granularity rather than individual microprocessor instructions. Since the Condensed Graphs Model is used as the underlying computation model, instructions are composed of triples, i.e., an array of operand data values, an operator function and a destination to which the result should be sent. Additional information, for distribution purposes, includes the address of the machine on which the instruction was created, and, if the instruction originated on another machine, a reference to the original instruction on the remote machine (see Figure 2.17). Results are composed of a data value, the address of the machine on which the associated instruction was created,

and a reference to the original instruction object on that machine. Both instruction and result objects contain references that may be invalid on the machine on which the object currently resides. Access to the fields of these structures is restricted by ensuring that the fields of the structures are kept private and are accessed only through functions that contain appropriate assertion checks. An instruction is said to be *local* if it exists on the machine on which it was originally created, or *remote* if it originated on another machine.

The various components communicate with each other by passing instructions and results through thread-safe queues. This arrangement enforces component modularity and reduces the possibility of race conditions arising through design or coding errors. Since all the components execute in the same memory space, there is no performance penalty incurred by copying instructions, results, or their associated data values between modules. Instruction and result objects and associated data are serialized for transmission to other cluster nodes, and operands may also be transmitted to FPGAs via some system bus (typically PCI). Similarly, incoming instructions and results from other cluster nodes, as well as result values from FPGA operators, are deserialized and added to the memory space of the computation process.

Data values are stored as a generic `Operand` datatype, which can be a placeholder for any of the ANSI C primitive types, including pointers. The type information structure associated with each type is used to perform serialization, deserialization and string conversions on type instances. Type information structures are stored in the type information table (see below) and are keyed on integer values, stored in instances of the `OperandType` datatype. An (`Operand`, `OperandType`) pair is therefore required when any type-specific behaviour is invoked on a data value.

Once a collection of computation processes has been initialized across a cluster, an application must be loaded before execution can commence. The processes receive a signal
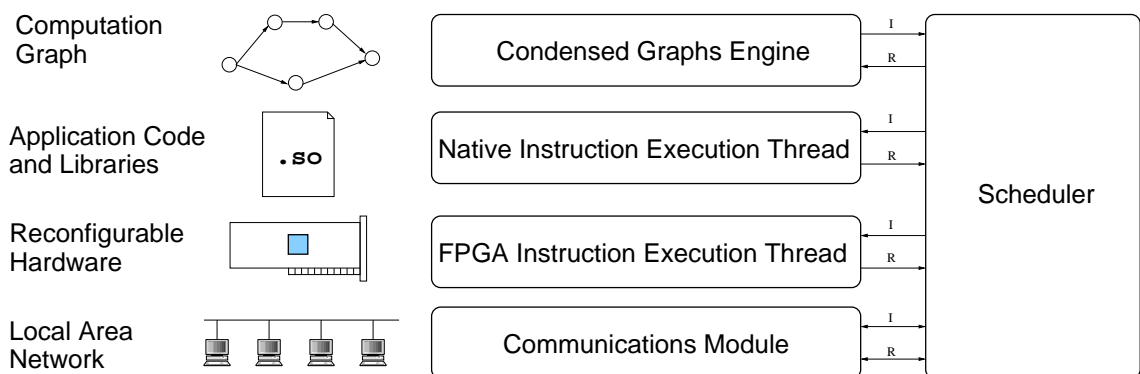

Figure 2.16: An overview of the various components comprising a ARC computation process, along with the resources managed by each. Arrows indicate the flow of instructions (I) and results (R).

```
typedef struct Instruction_tag {
      InstructionLocality locality;
      Instruction remoteReference;    /* Valid only when remote */
      GraphInstance instance;         /* Valid only when local */
      char *creator;

      /* Operand information */
      int numOperands;
      OperandType *operandTypes;
      Operand *operands;

      /* Operator information */
      char *operatorName;
      OperatorType category;

      /* Destination information */
      DestinationsGroup dests;        /* Valid only when local */
} InstructionStruct;

typedef struct Instruction_tag* Instruction;
```

Figure 2.17: The `Instruction` structure and datatype declarations. Individual fields are invisible to the rest of the program, allowing access to potentially invalid references to be controlled.

from the UI program informing them to load a shared object at a specified location. The UNIX `libdl` (dynamic linking) library is then used to load the library and dynamically link a collection of specially-named registration functions. These functions, when called, add entries to the information tables maintained by the computation process for types and operators. One of the functions also informs the computation process of any other shared objects that may be required, which are subsequently loaded in turn.

2.8.1   Condensed Graphs Engine

The Condensed Graphs Engine is responsible for creating and managing the Condensed Graphs component of an executing ARC application. This is achieved by executing instances of graph definitions by generating instructions and incorporating the corresponding results back into the graph instances that created them. The entire graph representing the state of the computation (or *V-Graph* in Condensed Graphs terminology) is not stored in its entirety on any of the cluster nodes; instead, it exists as a collection of individual graph instances scattered across the cluster.

Whenever a graph instance is created or modified, the engine checks for fireable nodes, and fires them by creating instruction objects. Once a node has been fired, it has its

destinations removed according to the deconstruction semantics described in Section 2.3. The destinations of a node are preserved in the corresponding instruction object so that the result value (when generated) can be forwarded to the correct destinations. Instructions with TM operators are always processed locally because they operate directly on local node instances. Instructions with operators of either of the other two types (condensed graphs or primitive) are passed to the load balancing component of the Scheduler for processing.

Irrespective of where a primitive instruction is executed, the result will eventually reach the Condensed Graphs Engine in which the instruction originated. Instructions with primitive operators return a result object containing a single data value. Instructions with condensed graph operators generate two result objects. The first contains a reference to a new instance of the specified definition that is added to the graph collection managed by the Condensed Graphs Engine on the machine on which the instruction was executed. Once the resulting graph has finished executing (i.e., the Exit node fires), another result object containing a single data value is returned to the machine that generated the original instruction.

Execution of an application commences with a message from the UI program to one of the computation process instances, specifying a graph definition and the operand values required to populate it. In response, the messaging component of the Communications Module (see below) creates an instance of the `Toplevel` graph, comprised of a condensed instance of the specified graph definition combined with another node (`allDone`) that halts the computation when executed. This condensed node is fireable immediately, and evaporates to produce an instance of the requested graph definition. As the operand values flow into this graph instance, more nodes become fireable, uncovering more work. Eventually, the computation results in a value being returned to the UI program.

2.8.2   Scheduler

The Scheduler is responsible for routing instructions between modules, both local and remote. Local instructions arrive at the Scheduler from the local Condensed Graphs Engine, and from remote engines via the Communications Module. The Scheduler in turn targets instructions to the local execution threads or to the remote Condensed Graphs Engines, again via the Communications Module. Instructions are sent either to one of the execution threads or to the Communications Module for distribution. Results arrive from the instruction execution threads and from the Communications Module, and are directed to the Condensed Graphs Engine from which the corresponding instruction originated.

Instructions may be executed on any machine partaking in the computation, and those with primitive operators may be executed either by the native or the FPGA execution

threads (if the appropriate implementations are available). The Scheduler uses appropriate load balancing algorithms when assigning an instruction to an execution thread. The algorithms used by the Scheduler that decide where incoming instructions should be sent for execution play a crucial role in the efficiency of the ARC system as a whole. Ideally, the algorithms should ensure that the work available is distributed as evenly as possible over the available computational resources. These algorithms must respect the constraints imposed by the communications overhead incurred when transporting instructions and results. In the case of instructions designated for execution on FPGAs, instruction sequencing should be managed in a fashion that minimizes time lost due to reconfiguration. If the algorithms fail to take advantage of the computational resources available, then one of the primary motivations for using the system in the first place is lost. Chapter 5 will examine the development and evaluation of load balancing algorithms that attempt to maintain an optimal ARC execution.

2.8.3   Native and FPGA Instruction Execution Threads

An ARC implementation has an instruction thread for every available computational resource. Each thread has an associated priority queue into which instructions are placed by the Scheduler for execution. An execution thread is put to sleep when its associated queue is empty and automatically woken up to process an instruction immediately upon its arrival. An arriving instructions takes up a position in the queue in a manner dictated by the Scheduler. In general, instructions will join the queue in first in, first out (FIFO) manner but occasionally FPGA instructions will need to take a priority position to minimize the occurrence of hardware reconfigurations (see Section 5.3.1).

In the case of native execution, an instruction is processed by looking up the function implementing the operator from the primitive or condensed graph operator information table, depending on the operator category. The operator function is then invoked with the supplied operand values to produce a result value (in the case of primitive instructions) or a new graph definition instance (in the case of condensed graph instructions). In either case the result value is used to create a result object.

All instructions are executed in-process. This arrangement can cause the failure of the entire computation process if an error (such as a segmentation fault or infinite loop) is encountered while executing a badly-behaved operator. However, the cost of spawning a new process for every instruction execution, and the associated interprocess communication overhead, was deemed to be too high a price to pay for isolating errors. The log files generated by computations allow users to immediately identify the operator that has caused a computation to fail. If error isolation through out-of-process execution is required, operator

implementations are free to spawn their own processes.

The FPGA Execution Thread operates identically to its native counterpart except that it executes instructions using an FPGA rather than the host CPU. Once an instruction is ready for execution, the FPGA must first be configured with the appropriate operator implementation. If the FPGA is already configured with the correct operator then no further action is necessary. Otherwise, the filename and clock speed of the bitstream implementing the operator are retrieved from the primitive operator information table and the FPGA is reconfigured. The FPGA is now ready to receive the operand values it requires for execution. This is achieved on the host side by invoking the serialization function associated with the type of each operand value and sending the resulting data to the FPGA using the FPGA's device driver. The FPGA operator implementation is responsible for decoding this information correctly as it is received. Once the FPGA has completed execution of the instruction, it sends the resulting data value back to the execution thread. The deserialization function of the operator's return type is invoked to decode the incoming data and convert it to an operand value. The code that interfaces with the FPGA driver is written in a modular fashion that allows drivers for different FPGA models to be used by implementing a well-defined set of functions, avoiding dependencies on any particular driver.

Once an instruction has been executed by either execution thread, a corresponding result object is produced. In the case of instructions with primitive operators, the result object contains a reference to a data value. Instructions with condensed graph operators produce result objects with references to new graph definition instances. In either case, result objects pass from the execution thread to the Scheduler. What happens next depends on the type and locality of the instruction that generated the result. The results of primitive operators are forwarded either to the Condensed Graphs Engine or to the Communication Module, depending on whether the instruction is local or remote. Graph instances are always sent to the Condensed Graphs Engine on the local machine. If the instruction that created a graph instance is remote, then the result of the newly-created graph is returned to the original machine once it has finished executing. This is achieved by mutating the graph's Exit node to use a `RemoteExit` operator, causing the graph to produce an appropriate result object.

2.8.4   Communications Module

The Communications Module is responsible for handling all communications between the collection of computation processes and the outside world. Computation processes communicate with three types of entities: their peers on other cluster nodes partaking in the computation, the ARC daemon executing on the same machine and the UI program executing on the user's machine. Communications with peers is performed using the LinuxNOW

library, while a custom TCP/IP messaging scheme is used to communicate with the daemon
and UI program. The Communications Module is implemented as a number of concurrent
threads, all of which are either listening on sockets, waiting for instructions or results to
arrive on queues or blocking on calls to the LinuxNOW library.

LinuxNOW [114] is a general-purpose library for creating computations that are dis-
tributed across a network of workstations (NOW). The library can handle many networking
issues (such as peer discovery, information gathering, load balancing and fault tolerance)
that would otherwise have to be implemented by the application developer. Ethernet broad-
casts are used by each library instance to maintain a table of connected peers. Developers
may implement custom information gathering functionality via callback functions, with
Ethernet broadcasting again used by each peer to advertise this information at predefined
intervals. Custom load balancing schemes may be implemented, again through callback
functions that implement a chosen load balancing algorithm. Fault tolerance is achieved by
reassigning work in the event of a node failure. Work is distributed using instruction and
result objects that act as envelopes for user-defined data structures.

LinuxNOW was chosen as the delivery mechanism for ARC instructions and results
because of the close fit between its functionality and that required by ARC computation
processes. When the Communications Module is initialized, it in turn initializes the Lin-
uxNOW library, using API calls to initiate peer discovery and information gathering (the
information required by the ARC system is the number of unevaluated instructions in each
execution thread queue). Since the computation process handles its own load-balancing
using the Scheduler, the LinuxNOW load-balancing functionality is disabled. Threads are
created to listen for incoming instruction and result objects from the Scheduler, and in-
coming LinuxNOW instructions and results via blocking API calls. Computation process
instruction and result objects are serialized and prepended with LinuxNOW instruction
headers before being handed over to the LinuxNOW library for transportation. Incoming
LinuxNOW instructions and results are stripped of their headers and deserialized before
being passed back to the Scheduler.

At startup, a `Messenger` object is also started in its own thread. This thread listens
for incoming messages using the ARC custom messaging scheme. The thread accepts con-
nections both from the UI program and the ARC daemon running on the same machine
as the computation process. Several types of messages, such as requests to start or stop
computations and redirect logging messages, can be received. Each message type has an as-
sociated handler function that decodes the message and performs any actions subsequently
required. Once a message arrives, its header is decoded to determine the message type and
the message body is then passed to the associated message handler. Functions are also
present that allow messages to be safely sent to the daemon or UI program from anywhere

in the computation process.

2.9   Application Development

In order to illustrate the ARC application development process, a simple divide-and-conquer application is presented. For simplicity and without loss of generality, only a single, primitive, operator is used apart from the standard `Enter` and `Exit` operators, and a binary reduction tree is formed. It must be stressed that this implementation is terribly inefficient, given the very small grain sizes involved and the latency issues inherent in both commodity networking equipment (used for inter-node communication) and PCI buses (typically used for communicating with the FPGAs). Nevertheless, the application's concise nature serves as a simple proof of concept.

The single graph definition of the application uses a series of addition operators to sum its eight integer inputs using a simple binary reduction tree. This is easily expressed as the condensed graph illustrated in Figure 2.18. The XML document describing the graph is relatively straightforward and repetitive (see Section 2.5 for an example of a ARC XML definition file). The same graph could be defined more concisely by creating a special TM operator for constructing graphs consisting of binary reduction trees.
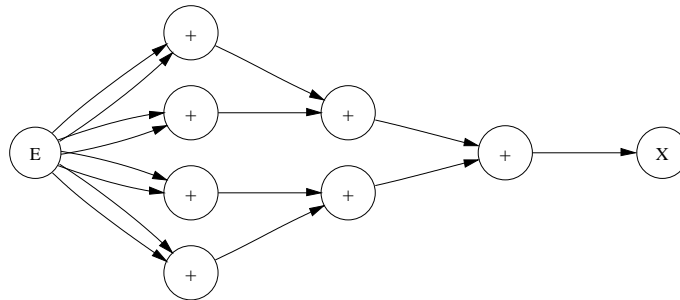
Figure 2.18: An example binary reduction application graph.

Although a native operator that performs integer addition already exists in the ARC standard library (`int_add`), another implementation is presented here for illustration purposes. The operator, as with all native operators, is implemented in ANSI C with all parameters and the return type being of the generic `Operand` type (defined in `arc.h`). Macros are available to cast instances of the `Operand` type to any of the ANSI C primitive types and vice versa. The `add` operator shown below simply casts both arguments to integer values, and returns a new `Operand` value created from their sum.

A Handel-C implementation of the same operator is shown below. FPGA operators are implemented as an infinite loop that processes one invocation at a time. A Handel-C header file is available to simplify the task of reading and writing instances of the ANSI

```
#include "arc.h"

Operand add(Operand o1, Operand o2)
{
      int a = OperandToInt(o1);
      int b = OperandToInt(o2);

      return OperandFromInt(a + b);
}
```

Figure 2.19: ANSI C implementation of the native addition operator.

C primitive types when communicating with the host processor. Each loop iteration reads
the two integer operands from the host before writing back their sum.

```
#include "arc.hch"

int main(void)
{
      while(1) {
            int a = ArcReadInt();
            int b = ArcReadInt();
            ArcWriteInt(a + b);
      }
}
```

Figure 2.20: Handel-C implementation of the FPGA addition operator.

Since the application uses only integer values, no new types need to be declared in the
XML definition file. However, an operator declaration similar to that in Figure 2.9 must
be added to the definition file along with the graph definition. Once the definition file has
been compiled and linked with the native implementation of the add operator, the result
is an application that can execute transparently in a distributed fashion across a cluster
augmented with reconfigurable hardware.

The process of developing more sophisticated applications follows exactly the same pat-
tern as described above: the specification of the high-level parallelism in the application as
a collection of graph definitions, and the implementation of the primitive operators refer-
enced by the graph definitions in software and hardware as appropriate. The application
development process will be examined in more detail in Chapter 3.

Chapter 3

An Example ARC Application

This chapter describes the development of an example application in order to demonstrate the ARC application development process, as described in the previous chapter, in practical terms. The application chosen, a brute-force cryptographic key search [115], represents perhaps the easiest class of application to accelerate using distributed reconfigurable computing; the computation required is CPU-intensive rather than memory-intensive, the amount of data to be transferred is small, and the subtasks produced can be of arbitrary granularity. The remainder of this chapter is organized as follows: 3.1 introduces cryptographic key search in general, while Section 3.2 describes the particular cryptographic algorithm being considered here. Section 3.3 describes the development of an FPGA configuration that accelerates the key searching process. Section 3.4 describes how an ARC application was created that allows key searches to be performed using distributed resources and considers the performance benefits attained by the application through the use of clusters augmented with FPGAs.

3.1 Cryptographic Key Search

The aim of cryptographic key search ("key crack") applications is to determine the value of a secret key by repeatedly encrypting a piece of known plaintext with varying candidate key values. The output at each iteration is compared with a piece of known ciphertext, with a match indicating that that the key used in that iteration was the one used to create the known ciphertext, and hence the value of the secret key has been found (see Figure 3.1). A *brute force* attack works by examining every possible key value, while others operate more selectively by choosing key values based on information specific to the algorithm being considered. For longer key lengths (112 bits or greater), brute force attacks are impractical, requiring millions of years to yield the secret key [116].

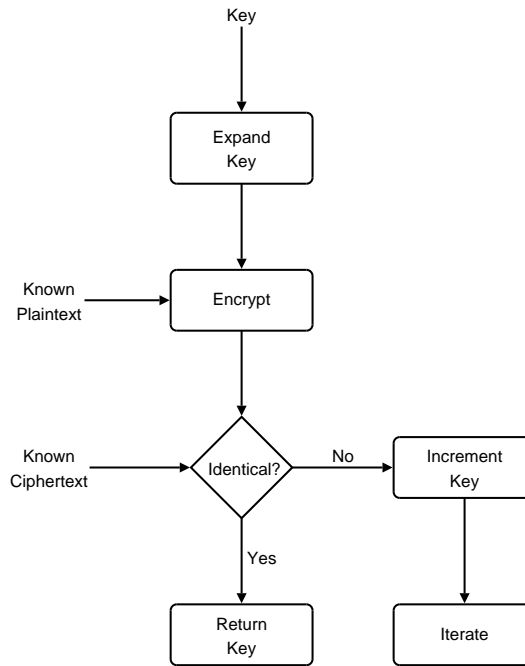Key search applications are good examples of *embarrassingly parallel* computations [117],

Figure 3.1: Operation of a cryptographic key search application. Successive key values are used to encrypt a piece of known plaintext until output matching the corresponding known ciphertext is found.

i.e., they can be divided into completely independent parallel tasks that require no inter-communication. As a result, they are often executed on clusters of commodity machines, or even large scale distributed computing projects[1]. Cryptographic applications in general are also ideal candidates for acceleration using reconfigurable hardware because of their high degree of intrinsic parallelism and data locality. Furthermore, cryptographic algorithms typically employ bitwise rather than floating point operations, so the poor floating-point performance of FPGAs is not an issue. Reconfigurable computing has been used previously to implement DES [118] and RC4 [119] key search applications.

As both Cluster Computing and Reconfigurable Computing can be used individually to accelerate key search applications, significant speedups should therefore be attainable through the use of a combination of both techniques by partitioning the keyspace across the cluster and accelerating the search on individual cluster nodes using the attached FP-GAs. Furthermore, these applications play to the strengths of both techniques as the amount of data communicated during execution is small and individual tasks are completely independent. These characteristics allow the capabilities of both types of hardware to be fully exploited for the duration of the computation, since the bottlenecks traditionally associated with each (Ethernet latency/bandwidth in the case of clusters and PCI

---

[1]See http://www.distributed.net.

bus latency/bandwidth in the case of reconfigurable hardware) are not limiting factors to performance. The speedups attained should therefore scale up with faster processors and faster/denser FPGAs without requiring a corresponding improvement in the network or system bus.

3.2   The RC5 Encryption Algorithm

RC5 is a simple and fast symmetric block cipher first published in 1994 [120]. The algorithm requires only three operations (addition, XOR and rotation), allowing for easy implementation in hardware and software. Data-dependent rotations are used to make differential and linear cryptanalysis difficult, and hence provide cryptographic strength (see Figure 3.2(b)). The number of rounds and the key length may be varied, allowing the user to determine the most appropriate tradeoff between speed and security [121].

RC5 forms the basis of a family of algorithms determined by three parameters: the word size (w) in bits, the number of rounds (r) and the number of bytes (b) in the secret key. A particular (parameterized) RC5 algorithm is denoted RC5-$w/r/b$, with RC5-32/12/16 being the most common. Plaintext is processed in blocks of size $2w$ bytes, producing $2w$ bytes of ciphertext. As 64 bit chip architectures become the norm, it is likely that 64 bit word sizes will increase in popularity. In that case it is suggested that the number of rounds be increased to 16.

Variable length keys are accommodated by expanding the secret key to fill an *expanded key table* of t subkeys of one word each, where $t = 2(r + 1)$. The subkeys are created by initializing an array $S_{0..t-1}$ with pseudorandom numbers determined by the values of w and r. Next, the b byte key is copied to array L, padding if necessary so that the length of L, c, is evenly divisible by the word size. Finally, a mixing operation, requiring three passes over S, is performed that combines that values in L with the initial values in S to produce the final value of S (see Figure 3.2(a)). This relatively expensive key expansion process makes the brute-forcing of keys difficult, but allows encryption to take place relatively quickly once the table has been created.

RC5 is extremely resistant to linear cryptanalysis, and is widely accepted as being secure (notwithstanding certain pathological examples that could yield to differential cryptanalysis and timing attacks) [122]. A brute force attack is only feasible in this case because a deliberately short key length – 40 bits – was chosen. As noted above, for longer key lengths (128 bits or greater), the brute-force approach is unlikely to succeed within any practical timeframe. Despite this, brute-force RC5 key searching is an application worthy of interest because it provides a simple, easily parallelizable real-world application that is amenable to acceleration with reconfigurable hardware.
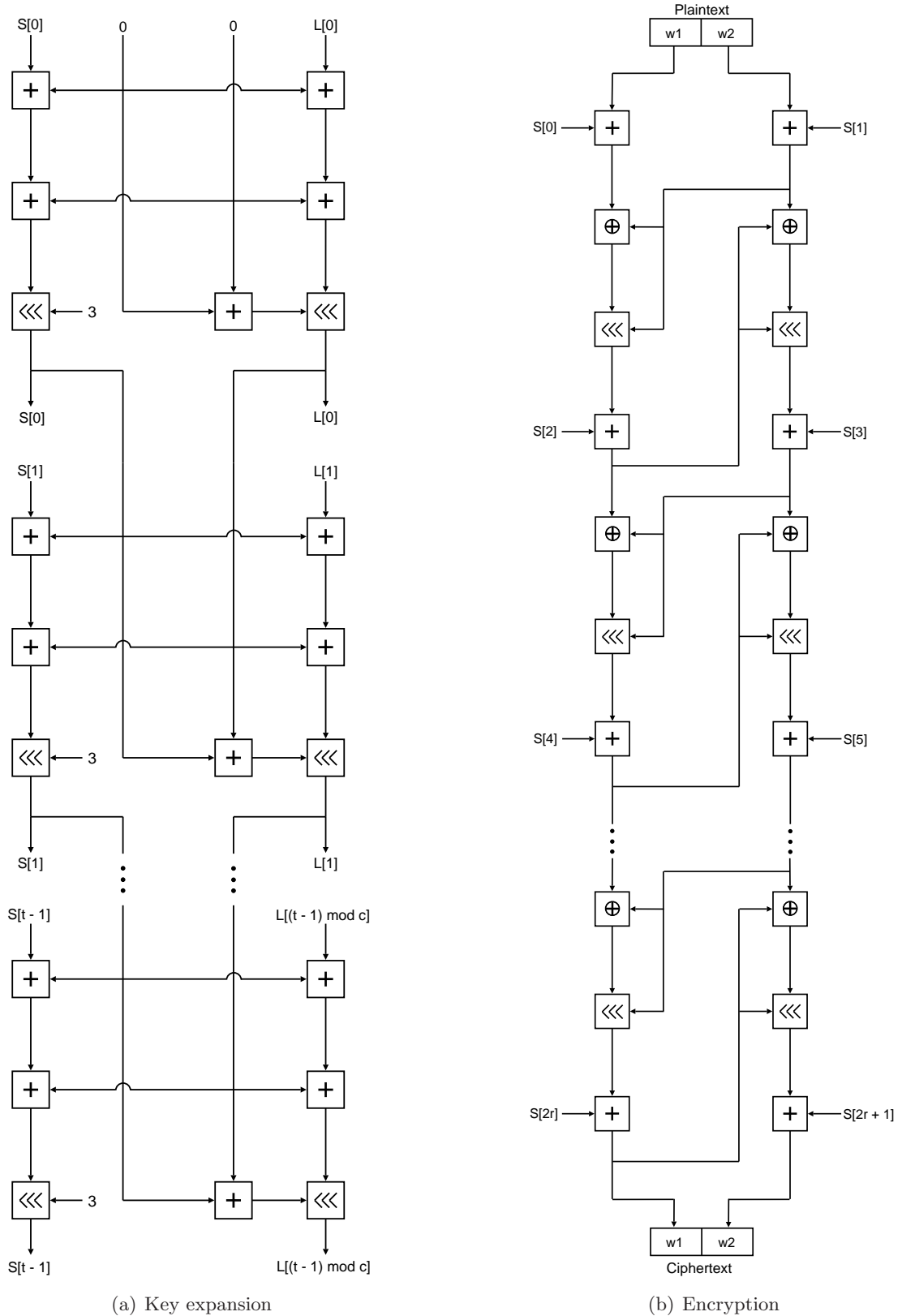
(a) Key expansion

(b) Encryption

Figure 3.2: The RC5 (a) key expansion and (b) encryption processes. The key expansion process is performed three times before the expanded key table, S, is fully initialized.

3.3   Development of the Hardware Implementation

The development of the RC5 key search application[2] began with the creation of an FPGA configuration that accelerates the key searching process. By performing this step first, the amount of application logic capable of fitting on the FPGA model used could be determined, and hence those parts of the application requiring software implementation later could be identified. Handel-C was chosen as the most suitable language to work with when creating the FPGA configuration, as its conciseness and behavioural nature allows different implementation strategies to be evaluated quickly.

The precise details of the operation of the encryption algorithm were determined through analysis of the reference ANSI C implementation of the RC5 algorithm in IETF RFC 2040 [123]. The algorithm parameters (RC5-32/12/5), known plaintext and known ciphertext were the same as those used by the distributed.net project's 40-bit key search effort. Development commenced with the implementation of the encryption algorithm in Handel-C, using the parallelization features of the language where appropriate in order to improve performance. The correctness of the hardware implementation was verified by comparing its output with that of the reference software implementation. The presence of a simulation mode in the Handel-C IDE that allows individual parts of a hardware design to be tested in isolation using inlined ANSI C code assisted in this process.

Once it was confirmed that the parallelized form of the encryption algorithm would fit on the target model of FPGA (a Xilinx Virtex XCV2000E [124] mounted on a Celoxica RC1000 [125] PCI board), the next step was determine whether the known plaintext and ciphertext could be incorporated into the hardware logic. This optimization would prevent the same values from being passed into the FPGA upon every invocation, significantly reducing the application's communications overhead, and would also increase the configuration's clock speed by reducing the number of registers in the design. A drawback of this approach is that it necessitates the recompilation of the configuration for distinct target key values. However, it was felt that the improved clock speed and reduction in communications overhead compensated for this inconvenience. Upon investigation, it was found that the known plaintext and ciphertext would fit on the FPGA along with the logic implementing the encryption algorithm. This reduced the number of parameters to the algorithm to one: the candidate key.

Next, the possibility of including a number of copies of the encryption algorithm in parallel on the same configuration was examined. It was found that twelve copies of the algorithm could be run in parallel, with each searching a portion of the specified key range.

---

[2]The development of the hardware implementation was performed by Padraig O'Dowd.

The key range was fixed at 10 million keys in order to minimize the communications over-head, with the sole parameter being the first key in the key range. Upon completion, the configuration returned the secret key, if found, or a zero value indicating failure (assuming, of course, that the secret key is not a zero value). This configuration that could run at 50 MHz on the target FPGA model, allowing approximately 240,000 keys to be checked per second.

Analysis of the operation of the algorithm revealed that further performance improvements could be made by pipelining the design, i.e., breaking the design up into stages, where each stage can operate independently of the others. Although this approach may lead to an overall reduction in clock speed, once the pipeline has been filled then processing is performed on multiple keys per clock cycle, improving overall throughput. As the performance of a pipeline is limited by the performance of the slowest stage, care had to be taken when breaking the design up into individual stages. It soon became apparent that the encryption of the plaintext was a relatively simple operation, warranting only a single stage, with the
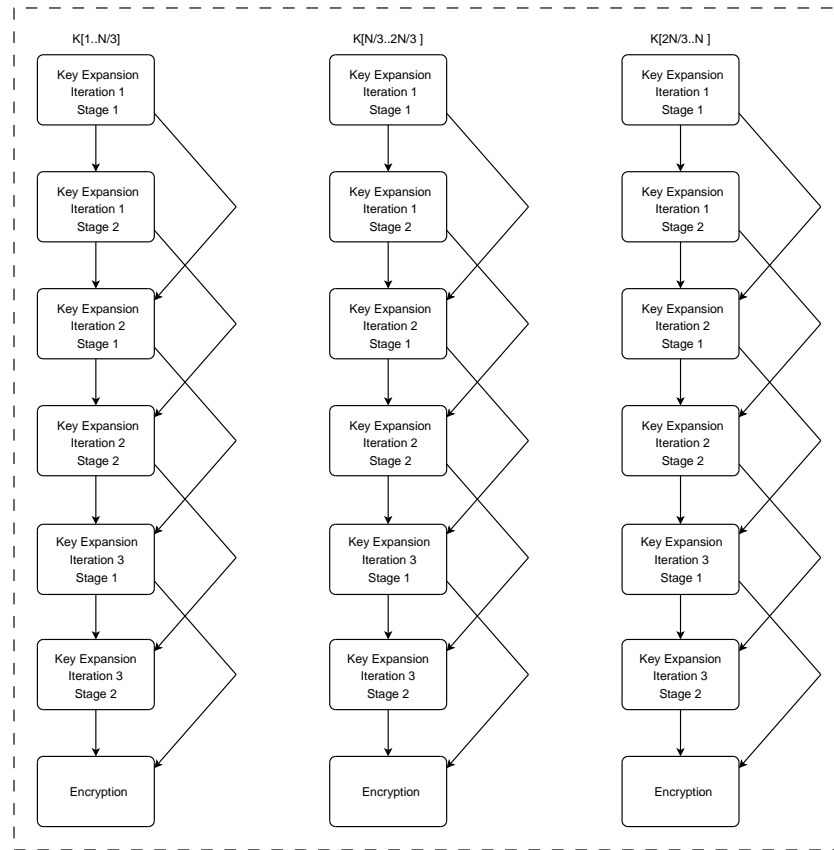
Figure 3.3: Overview of the final version of the RC5 key search FPGA configuration. Each of the three iterations over S in the key expansion process is divided into two stages, with the encryption of the plaintext comprising the last stage. Three pipelines, searching different regions of the key space (K), composed of N keys, operate in parallel.

remainder devoted to the key expansion process. Initially, key expansion was broken up into three stages (one for each iteration over S), resulting in four stages overall. As each half of a key expansion iteration depends on the previous value of S rather that any newly-computed values, it was determined that each of these stages could be broken down further into two semi-independent operations. Even these smaller key expansion stages were slower than the encryption stage, so no further divisions were possible. The overall result was a configuration composed of three pipelines running in parallel, with each pipeline composed of seven stages (see Figure 3.3).

The resulting configuration ran at 41 MHz and was capable of checking over 1.7 million keys per second; a remarkable achievement for a model of FPGA that is over a decade old. This represented a 42-fold speed increase over the CPUs used in the cluster (350 MHz Intel Pentium IIs) when these were executing the optimized distributed.net RC5 implementation. Although these CPUs were not the most modern available at the time, a greater than order-of-magnitude speedup was also recorded when comparing against the best performing CPU then available for comparison (a 2.4 GHz Intel Pentium 4). As the design is not I/O-bound, the performance achieved should scale up when adapted to more modern FPGAs that contain more logic gates (allowing the design to accommodate more parallel pipelines) and run at higher clock speeds (allowing individual pipelines to process more keys per second).

3.4   Development of the ARC Application

Although the creation of the hardware implementation of the search algorithm demonstrated that significant performance benefits could be attained through the use of individual FP-GAs, realising these benefits on a cluster-wide basis required the creation of a suitable ARC application. This application would partition the key space and search the resulting partitions using FPGAs distributed throughout a cluster. As noted above, the hardware implementation searches over a fixed number of keys, precluding the exact partitioning of the key space based on the number of nodes in the cluster. The ARC application would therefore have to operate by dividing the key space into partitions of the size hardcoded in the hardware implementation and use the load balancing capabilities of the runtime environment to ensure that these partitions were distributed as evenly as possible across the participating cluster nodes. This approach leads to a slight inefficiency when the number of partitions is not divisible by the number of cluster nodes, resulting in some nodes potentially being idle at the end of the computation. However, the increased key throughput achieved through the use of a hardcoded key count in the hardware implementation leads to greater overall performance for searches over larger key ranges. In any case, if the secret key is within the specified overall key range then the problem of load imbalance at the end

of the computation will only arise if the secret key is within the last few partitions to be searched.

The development of the ARC application commenced with the design of the top-level graph definition, titled `Main`. This graph is responsible for creating the list of partitions to be searched and exposing these partitions as a collection of nodes that can be executed in parallel. `Main` therefore requires two parameters: the initial key, serving as a starting point for the search, and the number of fixed-size partitions that comprise the key range (see Figure 3.4(a)). These parameters are passed directly from the enter node to a primitive node, `createPartitions`, responsible for converting this information into a list of partitions. The result of this operation is a value of type `List`, containing elements of type `Key` representing the initial key values in the partitions. Instances of `List` are parameterized with the common data type of the elements that they contain; the data type is required during transmission of `List` instances in order to invoke the correct serialization/deserialization functions on the elements in the list. Although the `List` type is part of the ARC standard library, the `Key` type is specific to the application at hand. Instances of `Key` are references to a five-byte block of memory containing a 40 bit RC5 key.

The list resulting from the evaluation of `createPartitions` is passed to a `forAll` node. `forAll` is a TM operator that accepts two parameters: an instance of `List` and the name of an operator that accepts a sole parameter of the same type as that of the list. The operation of `forAll` is similar to the concept of mapping in functional languages, except that the outputs of the operator invocations are discarded instead of merged into a new list. This is achieved by creating a new node with the specified operator for every element in the
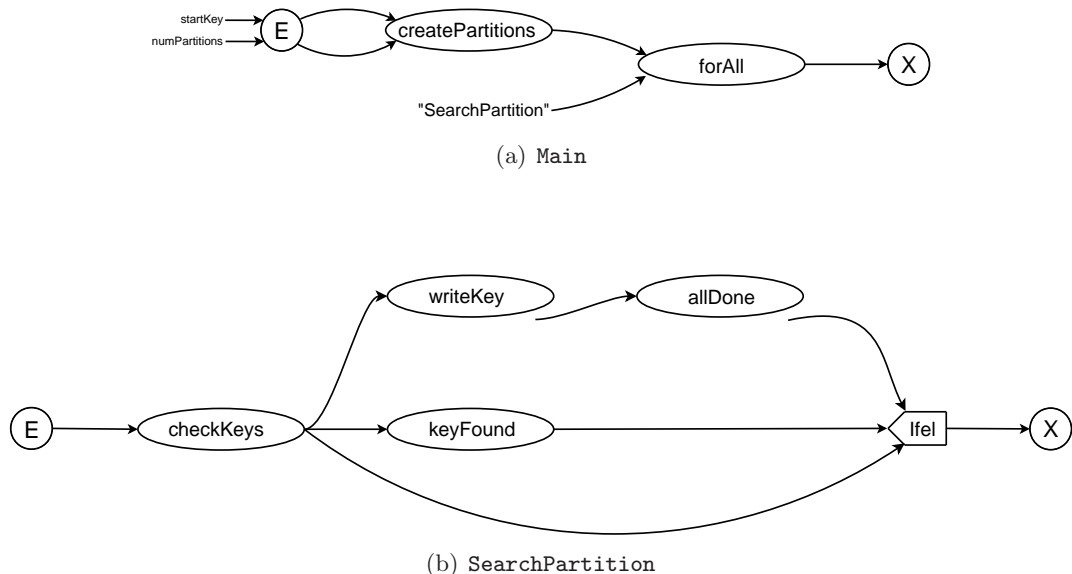

(a) `Main`

(b) `SearchPartition`

Figure 3.4: The graph definitions used in the RC5 key search ARC application.
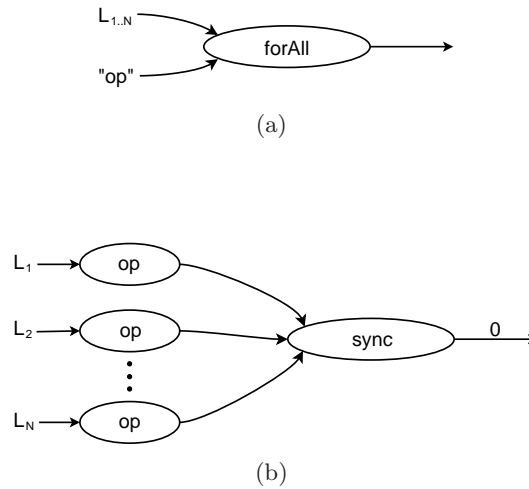
(a)

(b)

Figure 3.5: A `forAll` node (a) before and (b) after evaluation.

list, with the corresponding list element used as the operand to each new node (see Figure 3.5). A single node with a `sync` operator is also created, and acts as the sole destination of all the other newly-created nodes. `sync` is a TM operator that simply returns a zero value; its purpose is to prevent the computation from proceeding until all its operands have arrived, i.e., the operator parameter has been applied to all elements of the list. In this instance, the `forAll` node is used to apply the `SearchPartition` condensed graph operator to the elements of the list. The `sync` node will only fire if the key has not been found, causing a zero value to be passed from the `sync` node to the exit node and hence ending the computation.

The `SearchPartition` graph definition (see Figure 3.4(b)), applied by the `forAll` node to every key partition, is responsible for determining whether or not the secret key lies within the partition specified by its sole operand. If the secret key is found, then the key is written to a file and the computation is terminated. Otherwise, a zero value is returned. The `Key` argument is passed to a primitive node, `checkKeys`, that uses the hardware implementation of the search algorithm to evaluate the key partition starting at the specified key value. The result is an instance of type `Key`, with the bytes of the referenced memory buffer containing the secret key, if found, or zero bytes otherwise. This key value passes in turn to another primitive operator, `keyFound`, that returns a boolean value used to activate the appropriate branch of an `Ifel` node that is its sole destination. If the secret key was not found, then the exit node is fired, returning the zero key value back to the parent graph. In the event that the secret key was present within the specified partition, a "daisychaining" arrangement of stemmed nodes (see Section 2.3) is used to fire a node that writes the secret key to a file (as noted earlier, it is assumed that a common networked file system is available to all participating nodes) before terminating the computation through an invocation of the

`allDone` operator.

The correct operation of the application was verified through execution on a cluster composed of eight nodes, where each node contained a single 350 MHz Pentium II, 256 MB of RAM and a single Celoxica RC1000 reconfigurable computing board. The nodes were connected by a 100 Mb Ethernet switch. This resulted in a throughput of over 13 million keys per second, almost 350 times faster than the throughput of one host CPU and enough processing power to search the entire 40-bit RC5 key space in under 22 hours.

Chapter 4

Performance Model of the ARC Runtime Environment

Before work could commence on the development of the load balancing algorithms presented in Chapter 5, an objective method of evaluating their effectiveness was required. Ideally, every decision made by a load balancing algorithm based on the limited information available to it should be compared to the best decision possible based on all the relevant information. To this end, a performance model of the ARC runtime environment was developed that allows the execution time of instructions with known characteristics to be approximated with reasonable accuracy on any of the available computational resources. The execution time of an instruction resulting from the assignments made by load balancing algorithms could then be compared with the least possible execution times as predicted by the model.

4.1   Application Generation

Extensive information about the application being executed is required ahead of time if the behaviour of the runtime environment is to be predicted accurately. This information includes, for every instruction created, the cost of the associated operation, the amount of input and output data and, if the instruction's operator has a hardware implementation, the speedup attained through hardware acceleration. Unfortunately, this information is difficult or impossible to determine ahead of time for most real-world applications.

In order to work around this problem it was decided to create a method of generating applications where all the information described above is known ahead of time. As the operation of the runtime environment, rather than the function of the applications themselves was of interest, it was decided to randomly generate the form of the graph definitions, and to use primitive operators that simply wait for a specified amount of time.

This scheme was chosen for its flexibility, since new applications with desirable properties could be easily created in order to test various aspects of the performance model or load balancing algorithms. It was also decided that the randomness present in the applications should be introduced statically rather than at runtime in order to facilitate reproducibility when conducting experiments. The availability of randomly-generated but fixed-form applications allows alterations to the runtime environment, such as modified load balancing algorithms, to be evaluated under a single set of conditions and hence objective performance comparisons can be made.

As described in Section 2.2.1, ARC applications are composed of a collection of graph definitions and a collection of operators, with the operators having implementations either as machine code, FPGA configurations, or both. Typical real-world graph definitions also contain a number of operations that are trivial to execute and hence wasteful to delegate. In order to mimic this behaviour as closely as possible, it was decided that the generated applications should be composed of multiple graph definitions and should utilize a number of different operators. Some of the operators should have only native implementations, others should have only FPGA implementations, with the remainder having implementations of both type but with varying degrees of native:FPGA speedup.

With these requirements in mind, a Python program (`appgen.py`) was created that generates applications with user-defined characteristics. Upon invocation, the program begins by reading a parameter file passed as its sole argument. Parameter files are composed of a collection of attribute/value pairs that specify the behaviour of various aspects of the application generation process. An example parameter file is shown in Figure 4.1, and a description of the attributes provided in Table 4.1. Next, the program generates the number of applications specified by the `numApps` parameter. The application generation process is composed of three distinct steps: generation of the primitive operators, generation of the graph definitions and application output. These stages are discussed individually in subsequent subsections.

4.1.1   Generation of Primitive Operators

The program begins by generating the set of primitive operators that will be referenced by the graph definitions. The number of primitive operators in an application is chosen randomly using a normal distribution, depending on the mean and standard deviation values specified in the parameter file (`numPrimOperatorsMean` and `numPrimOperatorsSigma`). Primitive operators are created with some or all of the following attributes:

1. *Name.* Primitive operators are named as they are generated by suffixing `op` with the number of the operator. The numbering of operators start from 0.

Attribute Name	Allowed Values	Description
numApps	$1 \ldots 100$	The number of applications to generate.
numPrimOperatorsMean	$1 \ldots 100$	The mean number of primitive operators per application.
numPrimOperatorsSigma	$1 \ldots 10$	The standard deviation of the number of primitive operators per application.
trivialityProbability	$[0, 1] \subset \mathbb{R}$	The probability that a newly created primitive operator is trivial.
nativeOnlyProbability	$[0, 1] \subset \mathbb{R}$	The probability that a newly created nontrivial primitive operator has a native implementation only.
fpgaOnlyProbability	$[0, 1] \subset \mathbb{R}$	The probability that a newly created nontrivial primitive operator has an FPGA implementations only.
operationCostMean	$1 \ldots 10^7$	The mean cost of nontrivial primitive operations.
operationCostSigma	$1 \ldots 10^6$	The standard deviation of the cost of nontrivial primitive operations.
dataSizeMean	$0 \ldots 2^7$	The mean number of bytes of data returned from primitive operations.
dataSizeSigma	$0 \ldots 2^6$	The standard deviation of the number of bytes of data returned from primitive operations.
fpgaSpeedupMean	$1 \ldots 100$	The mean speedup of FPGA v. native operator implementations.
fpgaSpeedupSigma	$1 \ldots 10$	The standard deviation of the speedup of FPGA v. native operator implementations.
numGraphsMean	$1 \ldots 100$	The mean number of graph definitions per application.
numGraphsSigma	$1 \ldots 10$	The standard deviation of the number of graph definitions per application.
nodesPerGraphMean	$1 \ldots 100$	The mean number of non-TM nodes per graph definition.
nodesPerGraphSigma	$1 \ldots 10$	The standard deviation of the number of non-TM nodes per graph definition.
nodeCondensedProbability	$[0, 1] \subset \mathbb{R}$	The probability that a newly created non-TM node is assigned a graph, rather than a primitive, operator.
nodeParallelProbability	$[0, 1] \subset \mathbb{R}$	The probability that a newly created non-TM node is placed in parallel, rather than in sequence, with another node.

Notes

1. The probability that a newly created nontrivial operator has implementations in both software and hardware is defined implicitly as 1 - (`nativeOnlyProbability` + `fpgaOnlyProbability`).

2. The values of all parameters for which a mean and standard deviation are specified are generated using a normal distribution.

Table 4.1: The tunable parameters accepted by the ARC application generation program.

2. *Triviality.* Real world applications typically contain a number of primitive operators that perform $O(1)$ operations such as arithmetic and comparison. Operators such as these whose cost is trivial are known as *trivial operators.* The presence of these operators can be imitated in generated applications using the `trivialityProbability` parameter that specifies the probability that a newly created primitive operator will be trivial. Trivial costs are given the value of 1, and operators of this kind are prevented from having an FPGA implementation.

3. *Native and FPGA Implementations.* The assignment of implementations to nontrivial operators is performed in a single step to ensure that every operator has at least one implementation. The probability that a newly created nontrivial operator has an implementation of only one type is specified by the `nativeOnlyProbability` (`nop`) and `fpgaOnlyProbability` (`fop`) parameters. The probability that a newly created operator has implementations of both type is therefore defined implicitly as 1 - (`nop` + `fop`).

```
# Example appgen.py settings file


numApps=1


# Operator related options
numPrimOperatorsMean=10
numPrimOperatorsSigma=0
condensedProbability=0.2
trivialityProbability=0.3
nativeOnlyProbability=0.3
fpgaOnlyProbability=0.3
operationCostMean=1000
operationCostSigma=100
dataSizeMean=1024
dataSizeSigma=100
fpgaSpeedupMean=10
fpgaSpeedupSigma=2


# Graph related options
numGraphsMean=4
numGraphsSigma=1
nodesPerGraphMean=10
nodesPerGraphSigma=1
nodeCondensedProbability=0.2
nodeParallelProbability=0.5
```

Figure 4.1: An example application generation parameter file.

4. *Cost.* All nontrivial operators are assigned a nontrivial cost value depending on the operator cost mean and standard deviation parameters. The assigned cost represents the cost of execution relative to the cost of execution of a trivial operation.

5. *FPGA Speedup.* Operators with FPGA implementations are assigned a speedup value controlled by the speedup mean and standard deviation parameters. The speedup value specifies the performance increase attainable through hardware acceleration, and includes data transfer times.

6. *Output Data Size.* The amount of output data (in bytes) created by the invocation of an operator is specified by the `dataSizeMean` and `dataSizeSigma` parameters. The amount of input data received by any particular operator invocation is dependent on the topology of the graph instance that created the associated instruction.

4.1.2   Generation of Graph Definitions

Once the primitive operator table has been created, the next step in the application generation process is to create the set of graph definitions. The number of graph definitions created for each application depends on the `numGraphs` mean and standard deviation parameters. The generation of each graph definition then proceeds in two stages: the creation of the graph topology and the assignment of operators to nodes.

The application generation program creates graph definitions that utilize only a small subset of the features of the full Condensed Graphs model: all primitive operators and graph definitions were restricted to arity 1, only a single operand type representing blocks of memory is used, and neither lazy evaluation nor the mobility of subgraphs, operators and destinations was considered. Nevertheless, these graphs form a proper subset of Condensed Graphs. These restrictions do not imply any loss of generality for performance modelling and load balancing purposes (see below). The generation of applications that utilize the features of the model more fully is left as future work.

The applications created by the application generation program are similar to classical dataflow graphs and do not exhibit the more complex behaviour found in general Condensed Graphs applications. Although ignoring the characteristic features of the model results in simplistic application behaviour, this simplification can be justified if the sole purpose of the applications is to observe the behaviour of the runtime environment and evaluate load balancing algorithms. From this perspective, all that is required is the creation of instructions across the cluster of varying cost and data size; introducing the more complex features of the model into generated applications would alter the number and execution order of the instructions created, but not the behaviour of the modules of interest.

The restriction of non-TM operators to arity 1 was put in place because only TM operators may operate on a variable number of operands, and the task of matching the number of incoming connections to each node to the arity of the associated operator adds significant complexity to the process of generating graph topologies. When generating graphs using the simplified dataflow scheme, the presence of non-TM nodes with more than one operand port is emulated through the use of `merge` nodes. `merge` is a TM operator that accepts a variable number of operands representing memory blocks and coalesces the contents of the multiple memory block arguments into a single block. When the new, larger, block is associated with a non-TM node, the resulting instruction has the same amount of associated data as it would if the merge had not been performed and the operands had been passed to it directly. The limitation on non-TM operator arity therefore results in identical behaviour from a performance modelling or load balancing perspective.

Graph definition topologies are generated randomly, with variable numbers of nodes and variable degrees of parallelism. An example of the topology generation process in action is provided in Figure 4.1.2. The generation of a graph commences with the creation of an `Enter` and an `Exit` node. The next node is then added in between these two (see Figure 4.2(a)). Additional nodes are then added to randomly chosen non-TM nodes until the predefined quota for that particular graph definition is reached. The node quota applies to non-TM nodes and is determined using the `nodesPerGraph` mean and standard deviation parameters.
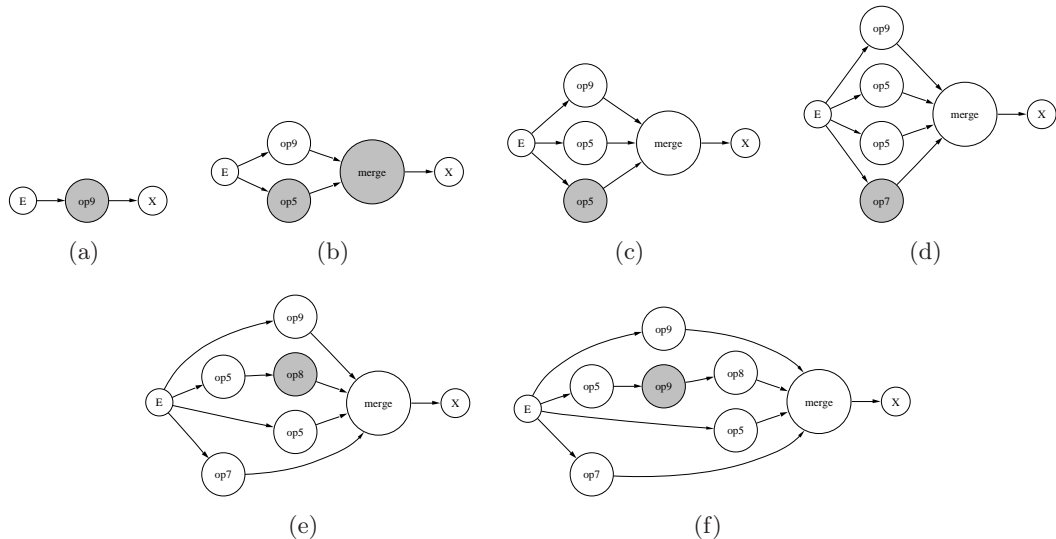
Figure 4.2: Steps in the generation of a graph definition containing six non-TM nodes. Newly added nodes are shaded in grey. Although operators are not assigned to non-TM nodes until topology generation is complete, the subsequent operator assignments are shown at each step for clarity.

A probability value, `nodeParallelProbability`, is used to decide whether new nodes are added in sequence or in parallel to existing nodes. New nodes are added sequentially by copying the destinations of the existing node to the new node before setting the new node as the existing node's sole destination (see Figures 4.2(e) and 4.2(f)).

Nodes are added in parallel by adding a destination to the new node from the origin of the existing node's incident arc. As noted above, the original implementation of the application generation program restricted all non-TM operators to arity 1. The outputs of the new node and existing node must therefore be merged into a single value. If the existing node already has a `merge` node as its destination, then the available `merge` node is reused (see Figures 4.2(c) and 4.2(d)); otherwise a new `merge` node is created (see Figure 4.2(b)).

Once the structure of a graph definition has been defined, each non-TM node in the graph must be assigned an operator. In order to ensure that each graph definition is used at least once, the operator assignment process begins by choosing a non-TM node at random and assigning the next graph definition as its operator. The sole exception to this rule is the case where the current graph definition is the last to be generated. For subsequent nodes, a probability value (specified by `nodeCondensedProbability`) is used to determine whether the node in question should be assigned a condensed graph operator or a primitive operator. Primitive operators are assigned simply by choosing an operator uniformly at random from the set of primitive operators created earlier. However, care must be taken when assigning condensed graph operators that circular dependencies are not introduced into the graph definitions, as these would result in infinite recursion at runtime. Condensed graph operators are therefore always chosen uniformly at random from the set of graph definitions generated after the current one. In keeping with this scheme, no condensed graph operators are assigned to the last graph definition to be generated.

At this point, all the characteristics of the application have been specified. The result is a table of primitive operators and a collection of graph definitions. An example of a completed application composed of ten primitive operators and five graph definitions is provided in Figure 4.3.

4.1.3   Application Output

The next stage in the generation process is the conversion of the in-memory representation of the primitive operator table and graph definitions to a set of files that implement the application as described. First, a directory is created to contain the application files. The application directory is named by suffixing `app` with the application number.

The sole datatype used by generated applications is `Block`, instances of which are references to blocks of memory, with the size of a block stored as an integer at the first address.
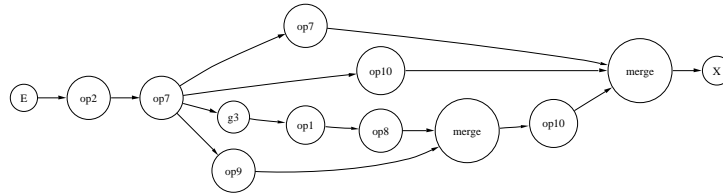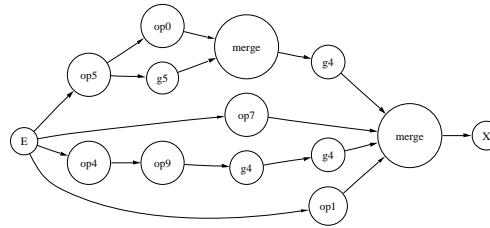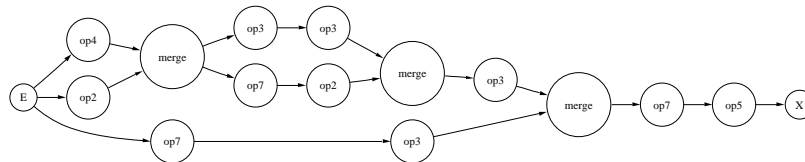
g0
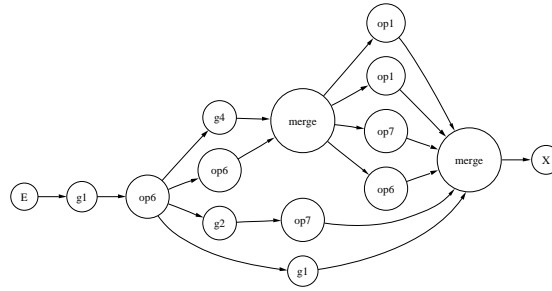
g1
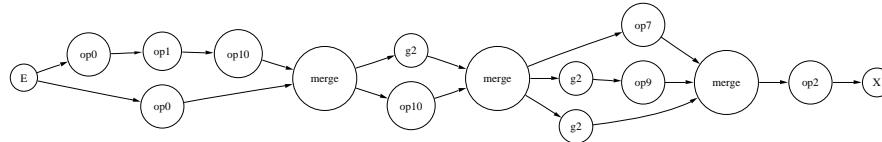
g2

g3

g4

Primitive Operator Table

Op#	Trivial?	Native?	FPGA?	Cost	Speedup	Data Size
0	✓	✓	×	1	N/A	928
1	×	✓	×	10095	N/A	1096
2	×	✓	×	9980	N/A	927
3	×	✓	✓	9894	10	1138
4	×	✓	✓	9845	9	1179
5	×	×	✓	10023	8	963
6	×	✓	×	9914	N/A	947
7	×	✓	×	9965	N/A	986
8	✓	✓	×	1	N/A	974
9	×	✓	×	9973	N/A	949

Figure 4.3: The graph definition set and primitive operator table of an example application generated by the application generation program. Primitive operator names are prefixed with op, while condensed graph operator names are prefixed with g.

Application output commences with the creation of an ANSI C file (`ops.c`) containing the serialization, deserialization and string conversion function for the `Block` datatype, as well as the implementation of the `merge` TM operator. Next, the primitive operator implementations are added to the file using a template ANSI C function parameterized with the name of the operator and the values of local variables specifying the amount of time to wait upon invocation and the amount of data to return. This template is filled in for every primitive operator with a native implementation and the resulting function is added to the file. All of the operator functions created accept a single `Block` operand and also return a `Block` result.

As the compilation of FPGA configurations takes a considerable amount of time, it was decided to avoid, if possible, the generation of a unique hardware implementation for every relevant operator. The use of homogeneous configurations for operators of varying simulated complexity does not affect the realism of the resulting applications because reconfiguration time was found to be a constant value rather than a variable dependent on the complexity of the function implemented (see Section 4.3.2 below). A consequence of assuming a fixed execution cost is that the data transfer rates are not modelled explicitly and hence cannot be reasoned about in this framework. Dynamic determination of data transfer costs may result in different load balancing decisions when considering specific individual instructions. However, the basis of these decisions is not altered if one assumes a fixed cost model.

To avoid generating different hardware implementations, and hence the associated cost, copies of a canonical implementation are made to files in the application directory with filenames matching the operators that they implement. However, the use of homogeneous configurations means that information not part of the operator's type signature (i.e., the amount of time to wait upon invocation) must be passed to the FPGA configuration at runtime. Furthermore, invocation of serialization and deserialization functions for the `Block` datatype must be circumvented if the cost of data transfer is to be incorporated. As a result, a modification was made to the implementation of the FPGA execution thread in the runtime environment that traps calls to generated operators, preventing the data transfer specified by the operator's type signature and transmitting instead the operator's adjusted cost (calculated by dividing the native cost by the speedup factor) to the FPGA.

Once the FPGA operator implementations have been created, application output proceeds with the creation of the XML file containing the high-level description of the application using the definition format described in Section 2.4. The set of graph definitions is added by converting individual graphs from the in-memory representation to XML using a simple scheme that assigns names to nodes based on their position in the graph definition's node list. Next, an `operator` element, with corresponding attributes and sub-elements, is added for each primitive operator. Finally, an `operator` element is added for the `merge`

operator along with a `type` element for the `Block` datatype.

At this stage, all the files required to compile and execute the application have been created. However, the application generation program also outputs a number of files for convenience. The first is a compilation script, allowing applications to be generated on one platform but compiled on another. Scripts are also created for running the application across 1-8 cluster nodes. Finally, visualizations are created of the application's graph definition set and primitive operator table via intermediate formats. The application's graph definitions are converted to the format used by the DOT graph visualization application [126], with the primitive operator table being converted to LaTeX [127] format. The resulting DOT and LaTeX files are then converted to PostScript, producing images such as those shown in Figure 4.3. This output is useful for examining the form of generated applications, particularly when experimenting with different parameter values.

4.2 Modifications to ARC to Support Performance Model Development

The process of developing the performance model required that the accuracy of the model be verified experimentally at each stage of development. These experiments were typically performed by recording the time taken by the runtime environment to perform particular tasks and then comparing the recorded values with the predictions produced by the model. Modifications to both the ARC compile-time and runtime environment were necessary in order to facilitate the recording of the required values.

Many of the experiments conducted relied on the availability of information about the executing application that could not normally be determined: instruction costs, output data sizes and FPGA speedups. This information cannot be determined in general for real-world applications, but is available for applications created using the application generation program. Modifications were therefore necessary to the compile-time environment to allow this additional information to be imparted by generated applications to the runtime environment. The XML application definition format was modified to allow `operator` elements and their child elements to support the additional attributes provided by the application generation program (see Figure 4.2 below). Corresponding changes were made to the Condensed Graphs Compiler to allow these attributes to be parsed and added to compiled applications.

The runtime environment was modified to support the new attributes described above by including them in the operator information table. Corresponding extra parameters were added to the callback functions used by applications to register operators. All the additional application information required for experimentation was then available at runtime. Other modifications to facilitate experimentation included the addition of a general-purpose

logging module, allowing tuples of recorded data to be stored as comma separated values in files. Additional temporary modifications to the runtime environment were made as necessary to record the values required for individual experiments and save them using the logging module.

```
<operator category="primitive" name="op4" cost="836" datasize="1067">
    <typesig>Block,Block</typesig>
    <nativeimpl name="op4"/>
    <fpgaimpl bitfile="test/app0/op4.bit" clockspeed="10" speedup="9.0"/>
</operator>
```

Figure 4.4: An example `<operator>` element created by the application generation program containing `cost` and `datasize` attributes, with its `<fpgaimpl>` child element containing a `speedup` attribute.

4.3   Development of the Performance Model

Performance modelling is loosely defined as *"the capture and analysis of the dynamic behaviour of computer and communication systems"* [128]. In effect, this involves the development of a model that predicts the behaviour of the aspect(s) of a system's performance that are of interest. In this case, the aspect of system performance of interest is the amount of time required to execute a newly created ARC instruction on any of the available computational resources. Probabilistic techniques commonly used for performance modelling such as stochastic modelling [129] were found to be unsuited to this task because they are used to reason about aggregate system characteristics, such as throughput, rather than individual system states as is required in this instance.

In light of this observation, it was decided that an analytical model of the relevant aspects of the ARC runtime environment was required. As described in Section 1.3, an analytical model of some aspects of clusters augmented with reconfigurable hardware is presented in [90]. However, this model is specific to the analysis of "synchronous iterative algorithms", and does not capture the more general nature of the ARC runtime environment. It was therefore decided that an ARC-specific model should be developed that captured only those aspects of system behaviour required for the evaluation of load balancing algorithms, and that the accuracy of the model would be tested during development where appropriate using the application generation program.

4.3.1   Approximating Native Instruction Completion Cost

The first step in the development of the performance model was to derive a formula that would approximate the *completion cost* (C) of an instruction executed by the CPU. Completion cost is derived from the elapsed time from when the result of an instruction is required to when the result is usable. This contrasts with the *execution cost* (E) of an instruction which is derived from the time spent by the CPU executing the instruction. This can be estimated by adding the cost required to complete the currently executing instruction, the sum of the execution costs of the other instructions in the instruction queue and the execution cost of the instruction in question:

$$C_C(n) = (E_C(c) - t_c) + \sum_{i=1}^{|Q|} E_C(Q_i) + E_C(n)$$

where n is the instruction being considered, Q is the native instruction queue, c is the currently executing instruction and t_c is the amount of time elapsed since execution of c commenced. E returns the cost value for the operator associated with its instruction argument (as specified by the application generation program) if the operator is primitive. Zero is returned in the case of a TM or condensed graphs operator. The reasoning behind this definition was as follows:

1. TM instructions represent simple node-manipulation operations, and the execution of a condensed graph instruction represents the evaporation of a graph definition rather than the evaluation of the nodes contained therein.

2. The application generation program cannot specify costs for these operations because their execution time is entirely dependent on the machine on which execution takes place.

3. The costs of both of these operations (node manipulation and graph evaporation) was assumed to be relatively small.

Before development of the performance model could continue it was deemed necessary to determine the accuracy of the part of the model presented above. This required the design and execution of a suitable experiment. This was achieved by using the application generation program to create applications with varying instruction costs and comparing the actual execution times of instructions with the times predicted by the C_C function. Variety in instruction costs was achieved by generating applications with mean instruction costs of every power of 10 from 1 to 4. Five applications were generated for each power value. The standard deviation of the costs was set to 10% of the mean cost used in each case.

After execution of all the applications was complete, the results were collated using a script that processed the output files for each mean cost value in turn. First, the data sets contained in the five files relating to the cost value currently being processed were appended in memory to form a single data set. Next, the predicted execution time of each instruction in the data set was subtracted from the corresponding actual execution time, resulting in a set of execution time differences. The minimum value, first quartile, median, third quartile and maximum value was then computed for the set of differences. A box plot of the resulting data is shown in Figure 4.5.
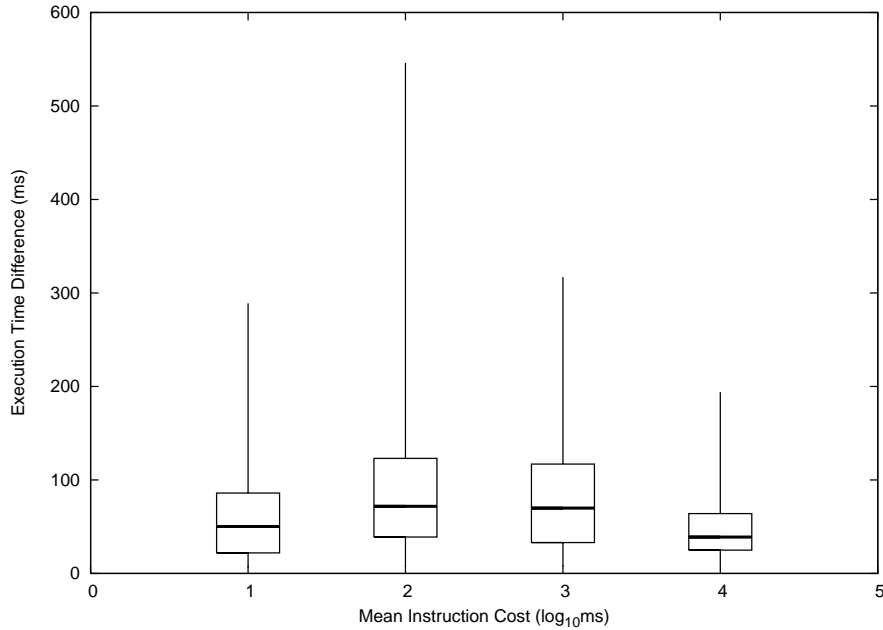
Figure 4.5: A box plot showing the differences observed between instruction execution times predicted by C_C and actual instruction execution times.

Reflecting on the results obtained, a number of conclusions were reached. Despite the presence of large outlying values, the majority of differences are clustered between 20 and 120 ms. This observation, coupled with the lack of significant variation of the median difference values across mean cost values, suggests that the model is approximating completion cost to some extent. On the negative side, the model underestimated the actual completion cost by a significant margin in every recorded case. The cause of this underestimation could not be determined simply by examining the data presented above. Further investigation was therefore required if the accuracy of the model was to be improved.

Based on the conclusions drawn above, the potential causes of completion cost mismatches were examined with a view to potentially refining the model. The following factors were identified as possible contributors to execution time underestimation:

1. As the measurements were made using wall-clock time, some underestimation is expected due to overheads present in the underlying environment. These delays could take the form of contention with other processes for CPU time or OS-related events such as page faults. Although care was taken while the experimental applications were executing to ensure that the cluster machines were as lightly-loaded as possible, these factors could not be ruled out as a potential source of significant delays in instruction execution. Unfortunately, these factors would be impossible to model accurately as they are machine-specific and essentially random events. The only way to account for such overheads within the model would be to introduce a system-dependent efficiency factor that would compensate for their presence in general but does not account for the variation of their effects on individual instructions.

2. Underestimation of instruction completion cost can also be expected due to overheads within the runtime environment. Possible causes of such overheads are the allocation of result objects, contention for locks, and activity in concurrent threads such as the uncovering of instructions in the Condensed Graphs Engine and the processing of load balancing information from other cluster nodes. If these overheads were identified as the source of significant delays, then further investigation would be required to determine if they take the form of a relatively fixed cost per instruction (and hence are linearly related to instruction queue length) or essentially random, as in the case of the underlying system overheads described above.

3. The `usleep` system call used in the implementation of the primitive operators created by the application generation program is not accurate to the millisecond. To quote the relevant entry from the Linux Programmer's Manual: *"The sleep may be lengthened slightly by any system activity or by the time spent processing the call or by the granularity of system timers"*. If "oversleeping" was found to be the primary cause of underestimation then alterations would be required to the application generation program rather than the performance model.

4. As stated above, the time spent executing TM instructions and evaporating condensed instances of graph definitions was not considered when estimating the total cost of the instructions awaiting execution. The assumption that these activities do not contribute significantly to overall execution time could be faulty, even more so if the processing of each instruction introduces an additional overhead, as considered above. Although the execution time of these instructions is system dependent, this could be accounted for through the introduction of additional parameters into the cost estimation function, provided that the cost of individual instances of these activities

does not vary significantly.

The next step in refining the model was to determine the roles of each of the factors described above in contributing to execution time underestimation. To this end, the applications used in the previous experiment were rerun, but with different characteristics being measured. Instead of the difference between predicted and actual instruction execution times, measurements were taken that would allow the significance of the overheads described in Points 1 and 2 above to be compared to the role of TM and condensed graphs instructions as described in Point 3. These were: delays observed between the completion of one instruction (including the creation of the corresponding result) and the commencement of the next in cases where another instruction is available on the queue, differences observed between the specified and actual execution times of operator implementations, the amount of time spent executing TM instructions and the amount of time spent executing

Other 25ms

Oversleep 992ms

Mean Cost 10^1 ms

Other 34ms

Oversleep 1559ms

Mean Cost 10^2 ms

Other 30ms

Oversleep 1359ms

Mean Cost 10^3 ms

Other 27ms

Oversleep 1053ms

Mean Cost 10^4 ms
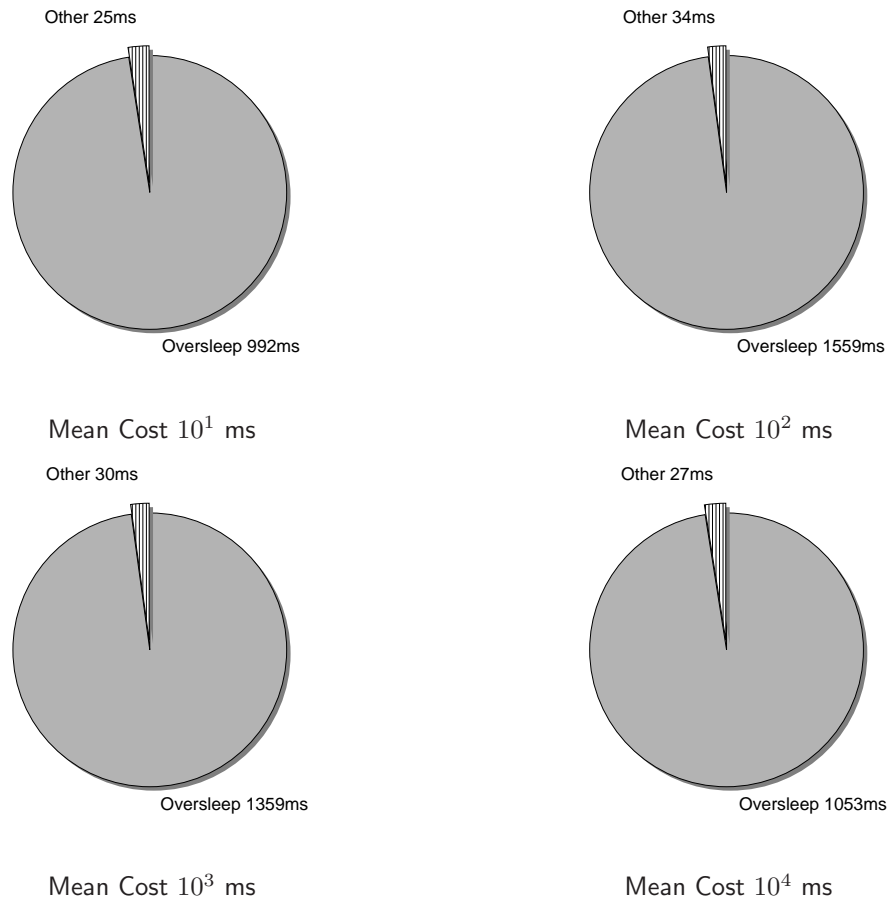
Figure 4.6: Pie charts showing the overall cost of activities not accounted for in the performance model. Due to the dominance of oversleeping, the other factors (executing TM instructions, evaporating graph definitions and instruction/result processing) are grouped into a single slice. The values shown represent the average observed over five applications.

CG instructions. The results are shown as a collection of pie charts in Figure 4.6.
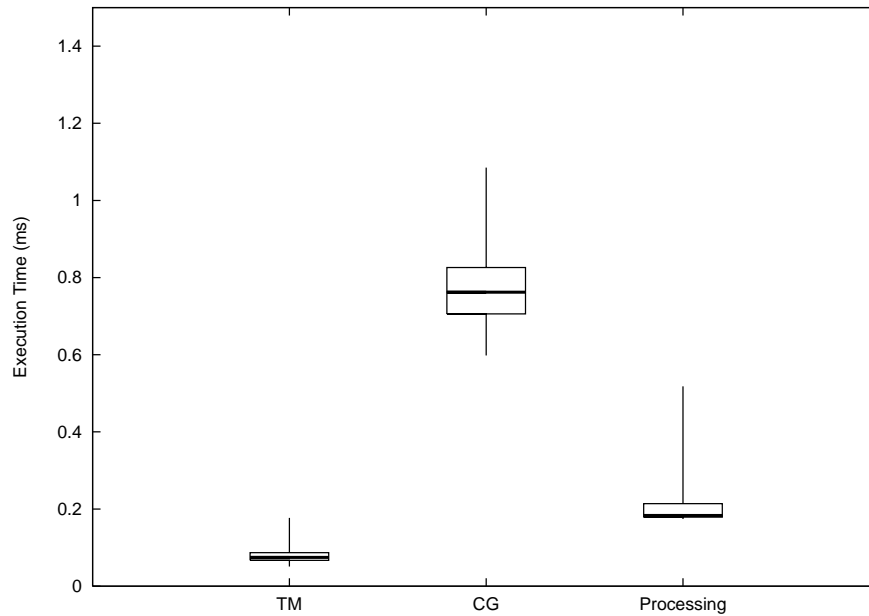
Figure 4.7: A box plot showing the values recorded for activities not accounted for by C_C

As the pie charts make clear, the inaccuracy of the `usleep` system call is by far the dominant factor causing execution time underestimation. This conclusion is underlined in Figure 4.7, showing that the overheads associated with factors not covered by the performance model are typically under 1 ms, justifying their omission during development of the model. Further research into this issue revealed that similar issues with `usleep` have arisen in the field of network simulation. The cause of the problem (timer interrupt frequency) is described in detail in [130]. Quoting verbatim:

"The standard Linux kernel sets the frequency of the timer interrupt to 100 Hz at boot time that corresponds to 10ms interrupts... The interval length between timer interrupts is called a jiffy. Since the kernel checks for expired timers only when a timer interrupt occurs, the smallest meaningful sleep request time is one jiffy. The POSIX standard for `select` *system calls states that the process must sleep for at least the time requested. To guarantee this, the kernel adds one jiffy to the requested sleep time in jiffies."*

From this analysis it would appear that the current implementation of the `usleep` system call is subject to considerably more inaccuracy than its manual page suggests. In effect, an inaccuracy of up to 20 ms can be associated with each invocation. Various workarounds are proposed for the lack of resolution in `usleep`, most of which require modifications to the kernel. The most suitable workaround not requiring kernel modification was found to be the use of *busy waiting*, i.e., a loop that continuously checks the system time (using the `gettimeofday` system call) until the target time is reached. Of course, the accuracy of this
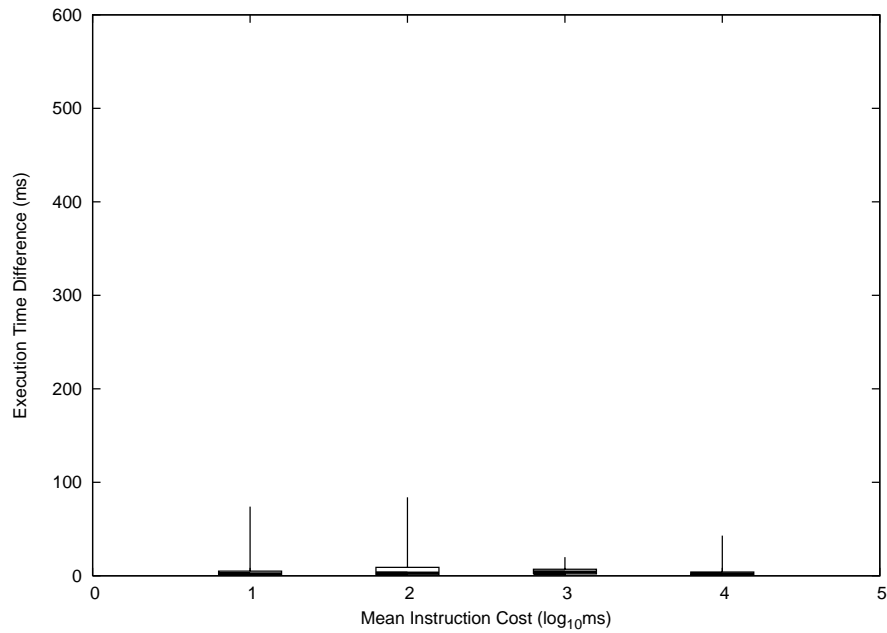
(a)
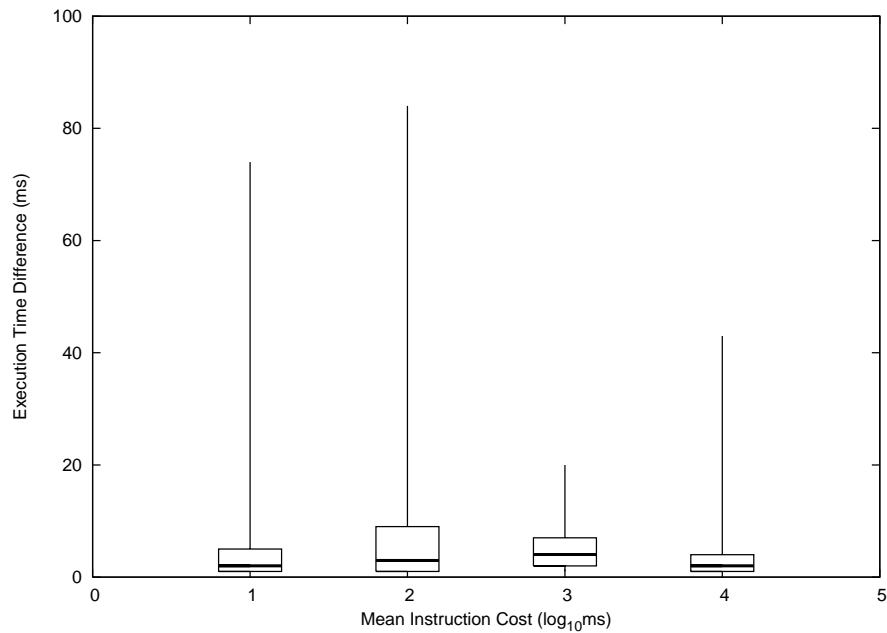
(b)

Figure 4.8: Box plots showing the differences observed between instruction execution times predicted by C_C and actual instruction execution times where the implementations of the primitive operators used busy waiting rather than calls to `usleep`. The scale used in Figure 4.5 is preserved in Subfigure (a) for comparison.

method depends on the granularity of system time updates; if the system time is updated using timer interrupts, then `gettimeofday` could also suffer from inaccuracy. Fortunately, this is not the case. Again referring to [130]:

"The modern Linux kernel `gettimeofday` *provides nearly microsecond accuracy by employing a time stamp register (TSR) available on Pentium processors that is incremented on each clock cycle. Earlier kernel versions returned the time-of-day value updated only at a timer interrupt".*

In light of this information, the application generation program was modified to use busy waiting rather than sleeping in the implementations of primitive operators. A new set of applications was generated according to the same criteria used in the earlier experiments. The differences between predicted and observed execution times were then measured for the newly-created application set. The results are shown in Figure 4.8. The scale employed in Figure 4.5 is maintained for comparison in Subfigure (a).

Examination of the data reveals a significant improvement in prediction times, with the vast majority of execution time differences tightly clustered about the 4 ms mark (see Figure 4.8(b)). The larger outlying values can be explained by preemption of the execution thread by other processes. Prediction accuracy could presumably be improved further by incorporating the other overheads examined into the performance model. However, it was decided that the CPU part of the model was accurate enough at this point to facilitate the load balancing experiments to follow, and so the focus turned instead to modelling the behaviour of the FPGA execution thread.

4.3.2   Approximating FPGA Instruction Completion Cost

The total cost of executing an instruction using the local FPGA execution thread was modelled by adapting the existing model for the native execution thread to account for differences in behaviour resulting from the use of FPGAs. The most important factor is the reduction in execution time attained through the use of the hardware rather than the software implementation of the operator associated with the instruction. The FPGA execution cost of an instruction is given by

$$\mathrm{E_F}(n) = \frac{\mathrm{E_C}(n)}{\mathrm{S}(n)}$$

where S is the speedup factor of the associated operator. The other factor modelled was time lost due to reconfigurations (R), which can potentially occur before the execution of every pending instruction:

$$\mathrm{R}(n) = \begin{cases} 0 & \text{if } m \text{ is defined and } \mathrm{op}(m) = \mathrm{op}(n) \\ r_n & \text{otherwise} \end{cases}$$

where m is the instruction executed immediately ahead of n (m is undefined if n is the first

instruction to be enqueued), *op* returns the operator associated with an instruction and r_n is the reconfiguration time associated with the hardware implementation of $op(n)$. The resulting completion cost formula is as follows:

$$\mathrm{C_F}(n) = (\mathrm{E_F}(c) - t_c) + \sum_{i=1}^{|Q|} (\mathrm{R}(Q_i) + \mathrm{E_F}(Q_i)) + \mathrm{R}(n) + \mathrm{E_F}(n)$$

where c is the currently executing instruction and t_c is the amount of time elapsed since execution of c commenced.

Before evaluation of the FPGA model could begin, a suitable means of calculating the reconfiguration time of each operator had to be found. To this end, a collection of five FPGA configurations was assembled to determine if a linear relationship existed between configuration size and reconfiguration time. The configurations used were drawn from both the set of example applications provided with the RC1000 boards used for evaluation and applications developed for ARC. These configuration files implemented functions ranging in complexity from simple integer addition to complex image manipulation operations. Surprisingly, the configurations were found to be all approximately the same size (1.2 MB ± 30 bytes). The size of the FPGA configurations was therefore found to be unrelated to the complexity of the function that they implement. As a result, it was concluded that the configuration files store the uncompressed state of all gates in the FPGA, not just those that are actively used by the configuration. The reason for the presence of the small variations in file sizes was not determined. The time taken to reconfigure an FPGA with each configuration was recorded and averaged over five runs. The mean reconfiguration time was found to be 173 ms with a standard deviation of 16 ms, with the variations occurring across multiple reconfigurations with homogeneous as well as heterogeneous configurations. Reconfiguration time was therefore found to be a constant value unrelated to the complexity of the configuration itself. The R function was then simplified as follows:

$$\mathrm{R}(n) = \begin{cases} 0 & \text{if } m \text{ is defined and } op(m) = op(n) \\ r & \text{otherwise} \end{cases}$$

where r is the constant reconfiguration time.

An experiment was conducted to verify the accuracy of the predictions produced for the execution times of instructions assigned to FPGAs. The application generation program was used to create application sets with mean instruction costs ranging from 10^2 to 10^5, with each set containing five applications. All the nontrivial operators were specified as having FPGA implementations only, with mean speedup specified as 10, with a standard deviation of 1, in each case. This resulted in FPGA instructions with mean adjusted costs ranging

from 10 to 10^4. The differences observed between predicted and actual FPGA instruction execution times are shown in the form of a box plot in Figure 4.9 below.
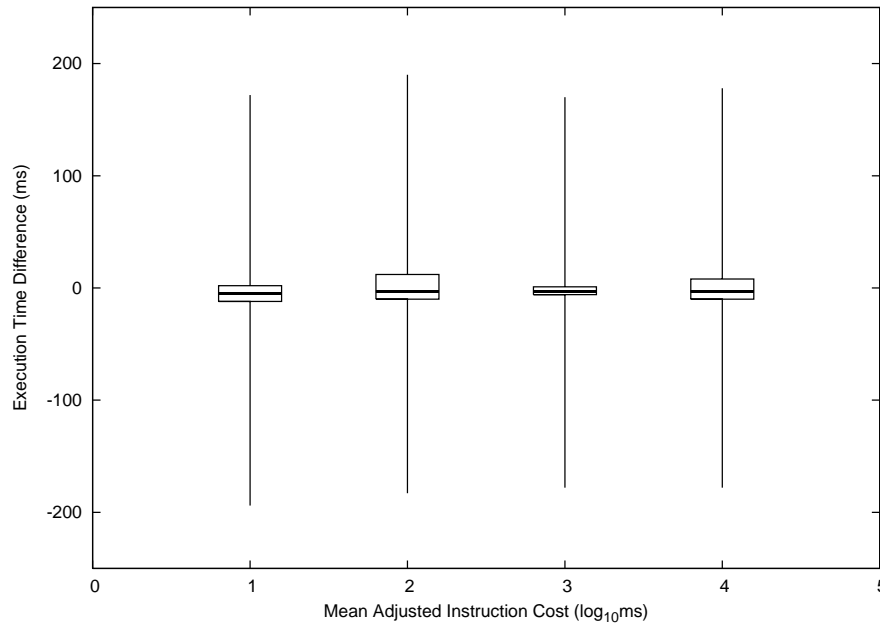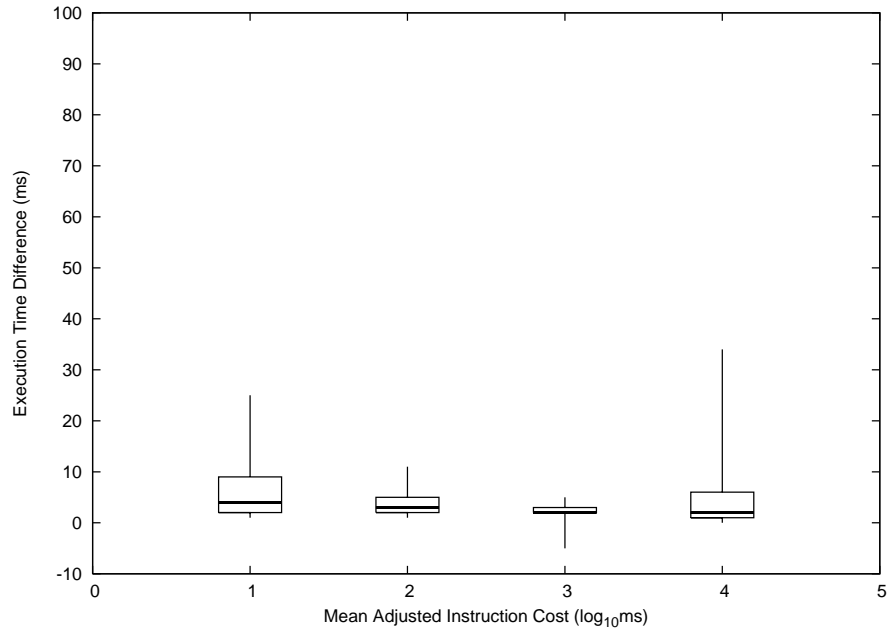

Figure 4.9: A box plot showing the differences observed between instruction execution times predicted by C_F and actual instruction execution times.

Examination of the results obtained reveals that although the predictions are reasonably accurate for the most part across varying mean cost values, large outlying values are present. These inaccuracies represent both over- and underestimation, are present in similar magnitude across all mean cost values, and are significant – up to ± 200 ms. The variation in reconfiguration times observed above were identified as the most likely cause. In order to test this hypothesis, the experiment was repeated with the amount of primitive operators confined to one per application, preventing the occurrence of reconfigurations. The results are shown in Figure 4.10 below.

The lack of similar outliers in this case confirms that variation in reconfiguration time is the primary cause of the prediction inaccuracies observed above. As noted earlier, variations in reconfiguration times are random, and hence unpredictable. The accumulation of significant numbers of instructions requiring reconfigurations in the FPGA instruction queue can create cumulative over- or underestimation in some cases, resulting in the occasional large inaccuracy as observed above. Possible explanations for the variation in reconfiguration times include PCI bus contention and the fact that reconfigurations are performed indirectly using the DMA controller, rather than direct communications over the PCI bus when reading or writing operand values in generated applications. Although this seems to be a credible hypothesis, the same variations were not observed during the networking

Figure 4.10: A box plot showing the differences observed between instruction execution times predicted by C_F and actual instruction execution times when the number of primitive operators was restricted to one in order to prevent reconfiguration.

experiments performed below, thus casting doubt on the cause of the observed jitter.

4.3.3   Approximating Instruction Delegation Overhead

Once the behaviour of the local instruction execution threads had been accurately modelled, the next step in the development of the performance model was to develop a method of approximating the overhead incurred when an instruction is delegated to another machine. The delegation process is composed of three distinct parts:

1. The transmission of the instruction and its associated operand data from the local machine to the remote machine.

2. The execution of the instruction on the remote machine.

3. The transmission of the corresponding result object and its associated data from the remote machine back to the local machine.

In order to approximate the delegation overhead of an instruction, it was necessary to model the behaviour of the runtime environment when performing Steps 1 and 3. The transmission of an instruction commences with the opening a connection to the remote machine. The cost of this activity was assumed to be a system-dependent constant. Next, the instruction object is serialized and transmitted. Although the exact size of a serialized

instruction is dependent on a number of factors (such as the length of the name of the operator), it was decided that the extent of these differences was insignificant. As a result, instruction object serialization was also treated as a constant value. The only factor that varied across instructions was therefore transmission of operand data, a function of the size of the operand values and the bandwidth of the networking equipment used. Expressed as a formula:

$$\mathrm{T_I}(n) = conn + \frac{s_\mathrm{I} + \sum_{i=1}^{|O_n|} \mathrm{size}(O_{ni})}{b}$$

where $conn$ is the system-dependent connection overhead, s_I is the instruction object size constant, O_n is the set of operand values associated with n, $size$ returns the size of an operand value and b is the network bandwidth. Before proceeding with a model of result transmission, it was decided to verify the accuracy of $\mathrm{T_I}$. The first step in this process was to determine the values of c and b on the cluster used for experimentation as well as a suitable value for s_I.

The connection overhead was determined by generating a single application and executing it across two cluster nodes. In order to ensure that a large number of connections were opened between both machines, the application was created with ten graph definitions, each definition in turn containing ten nodes. During execution, the system time was recorded whenever a machine opened a connection to transmit an instruction. Once the remote machine had opened a corresponding connection and was ready to receive data, the system time on the remote machine was also recorded. The system clocks on participating machines were synchronized using the Network Time Protocol (NTP). The results were processed by calculating, for each connection, the time elapsed between the first and second time recordings. The values observed are shown in Figure 4.11.

Examination of the results reveals that the connection times for each machine are tightly clustered about median values. However, a significant divergence is present between the median values observed. Furthermore, the values observed for the second node were less than zero. This phenomenon could be explained by a difference of a few milliseconds in the system times of the two machines. Although the NTP daemon was executing on both machines, it was concluded that the resulting system times were not accurate to the millisecond on the cluster used. The value of $conn$ was determined by calculating the distance between the two medians, resulting in a value of 4 ms.

The size of instruction objects was determined by executing the application used in the previous experiment but recording instead the size (in bytes) of every newly-created delegable instruction (i.e., condensed graphs and primitive operations). All the values in the resulting data set were found to be either 44 or 45. This lack of variation of instruction size
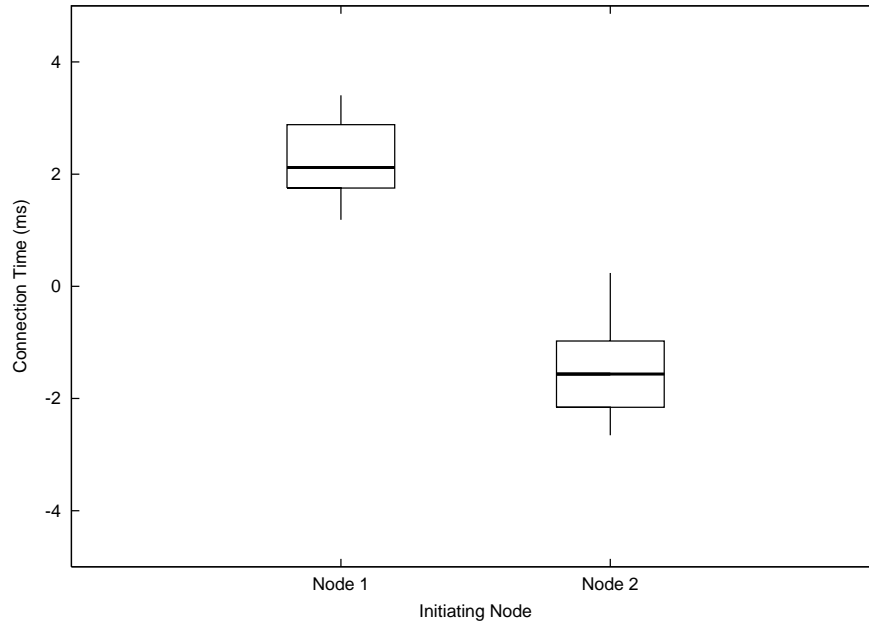
Figure 4.11: A box plot showing the time differences observed between the initiation of a connection on one cluster node and opening of the corresponding connection on another. Both the difference in median values between the two machines and the presence of negative values for node 2 are caused by system time differences.

can be explained by the fact that the application generation program generates delegable instructions with a single operand port and similar operator names. Although significant variation can exist in the number of destinations contained by these instructions, this information is not transmitted during the delegation process. Based on these observations, 44 was used as the value of s_I in subsequent experiments.

The value of the bandwidth parameter b depends on the networking equipment connecting the cluster used for application execution. In this instance, a 100 Mb/s switched Ethernet LAN was used. Converting this bandwidth to bytes per millisecond resulted in a value of 12,500 for b.

The next step in modelling the delegation overhead was to develop a method of approximating the transmission time of results. The resulting formula is similar to T_I, but less complex because only a single data value must be considered:

$$T_R(r) = conn + \frac{s_R + \text{size}(o_r)}{b}$$

where $conn$ is the connection overhead, s_R is the result object size constant, o_r is the data value associated with r and b is the network bandwidth. The values of $conn$ and b are the same as those used for T_I. The only other parameter value that must be known ahead of time is s_R. Observations similar to those taken to determine the value of s_I revealed that

all result objects are 8 bytes in size.

Once the models of instruction and result transmission were complete, the completion cost of an instruction on the CPU of a remote machine p could then be expressed as follows:

$$C_{\mathrm{RC}}(n, p) = T_{\mathrm{I}}(n) + C_{\mathrm{C}}(n, p) + T_{\mathrm{R}}(r_n)$$

where r_n is the result created by the execution of n, and C_{C} is parameterized with p. Similarly, the completion cost of an instruction on the FPGA of a remote machine was given by:

$$C_{\mathrm{RF}}(n, p) = T_{\mathrm{I}}(n) + C_{\mathrm{F}}(n, p) + T_{\mathrm{R}}(r_n)$$

At this point it was decided that an experiment was required to determine the accuracy of the instruction delegation aspect of the performance model. As the communication overhead was of sole interest for this experiment, the application set used was generated accordingly. The applications were generated with parameter values that resulted in a single graph definition per application, with the graph definitions composed of nodes arranged in a linear fashion. This arrangement was chosen in order to prevent the formation of queues of instructions, ensuring that the execution times of instructions on remote machines could be predicted using information local to the delegating machine. The cost of all primitive instructions was held constant at 1,000 ms, with mean data sizes ranging in powers of 2 from 1 to 12. The primitive operators used were generated with implementations in software only, again in order to simplify cost calculations. Hence, only C_{RC} was evaluated, with 1,000 used as the predicted value of C_{C} in every case. The resulting differences between predicted and actual execution times were recorded, with the results shown as a box plot in Figure 4.12 below.

Reflecting on the results obtained, two conclusions were reached:

1. The observed execution time differences are tightly clustered about median values, and the accuracy of the predictions is not affected by increasing data size. This suggests that the model of delegation cost is accurate and that the value chosen for the bandwidth parameter b is correct.

2. The model consistently underestimates the actual execution time by approximately 6 ms across all mean data sizes. This indicates that the value determined above for the connection overhead constant *conn* was incorrect.

Based on these conclusions, it was decided to adjust the value used for *conn* in order to improve prediction accuracy. As the connection overhead constant appears twice in C_{RC}
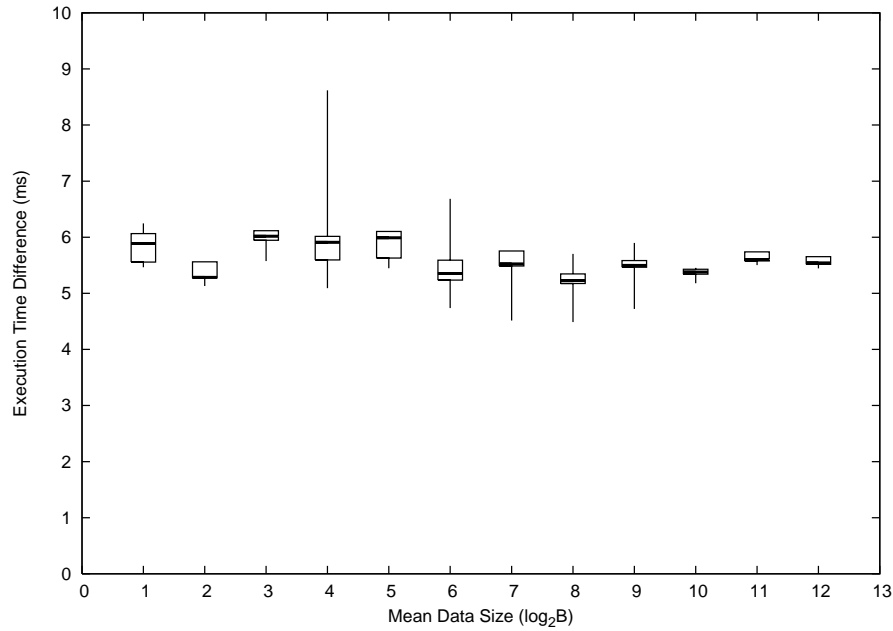
Figure 4.12: A box plot showing the time differences observed between the execution times of instructions as predicted by C_{RC} and actual execution times.

(once in T_I and once in T_R), the value of *conn* was adjusted by halving the mean of the medians observed for low data sizes and adding this to the existing value, resulting in a new value of 7 ms. The experiment was repeated using the new value of *conn*, with the results obtained shown in Figure 4.13 below.

Examination of the new results reveals that the adjusted value of *conn* increases prediction accuracy to within 1 ms in the majority of cases. At this point, all the aspects of runtime environment behaviour required for the conduct of load balancing experimentation had been modelled.

4.4 Estimating Minimum Instruction Completion Cost

The various aspects of ARC runtime environment behaviour modelled above allow the completion costs of instructions to be predicted with reasonable accuracy both locally and remotely. By combining the formulas presented above, the least possible completion cost of an instruction across the whole cluster can be estimated by comparing the lowest possible local and remote completion costs:

$$C_{min}(n) = min \; C_{minL}(n), C_{minR}(n)$$

The lowest possible local completion cost is calculated by comparing the completion
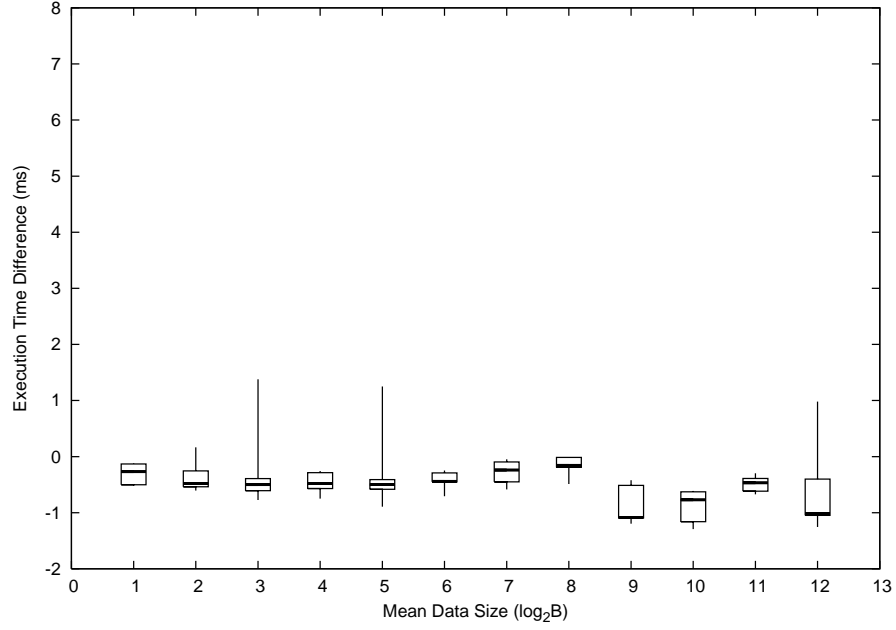
Figure 4.13: A box plot showing the results obtained when the experiment shown in Figure 4.12 was repeated with an adjusted value for *conn*.

costs of the instruction on the local CPU and FPGA, provided that both hardware and software implementations of the operator are available:

$$C_{minL}(n) = \begin{cases} C_C(n) & \text{if op}_{FPGA}(n) \text{ is undefined} \\ C_F(n) & \text{if op}_{CPU}(n) \text{ is undefined} \\ \min C_C(n), C_F(n) & \text{otherwise.} \end{cases}$$

Similarly, the lowest possible remote completion cost is calculated by taking the least possible completion cost available on the CPUs and FPGAs of the set of peers P:

$$C_{minR}(n) = \begin{cases} \min_{p \in P} C_{RC}(n, p) & \text{if op}_{FPGA}(n) \text{ is undefined} \\ \min_{p \in P} C_{RF}(n, p) & \text{if op}_{CPU}(n) \text{ is undefined} \\ \min_{p \in P} C_{RC}(n, p), C_{RF}(n, p) & \text{otherwise.} \end{cases}$$

4.5   Benefits and Limitations of the Performance Model

The performance model presented above is a useful tool for examining various aspects of ARC behaviour, and has a number of potential applications. Apart from its primary intended purpose as a benchmark for load balancing effectiveness, the model (with minor modifications) could also be used to examine how the runtime environment would behave under conditions that cannot be created with the equipment at hand. Potential scenarios

include:

1. Observing execution behaviour on clusters comprised of nodes of SMP machines and/or the presence of multiple FPGAs per cluster node.

2. Observing execution behaviour on clusters where heterogeneity is present across the set of CPUs and/or FPGAs.

3. Evaluation of the performance gains achieved through the use of alternative FPGA connection topologies or networking equipment.

4. Determining the optimal cluster configuration for an application before purchase.

Despite its usefulness, some caveats should be observed regarding the accuracy of the predictions produced by the model.

1. The prediction formulas are not accurate to the millisecond. As most of the formulas are accurate only to within 10 ms, the model cannot be meaningfully applied to very fine-grained computations.

2. Examination of the graphs comparing predicted and actual instruction execution times reveals the presence of significant outlying values (up to tens of milliseconds). These inaccuracies result from unpredictable events occurring in the underlying execution environment, such as CPU contention and networking delays. The model should not therefore be expected to be accurate in every case.

3. Consideration should be taken of system time differences between cluster nodes when comparing measurements taken on different machines.

In light of these observations, the values produced by the model should only ever be regarded as estimates rather than firm predictions. On the positive side, in the vast majority of cases the predictions produced by the model are accurate to within 10 milliseconds. It can therefore be concluded that the model provides reasonably accurate predictions of the behaviour of applications created by the application generation program, provided that the granularity of the application under observation is suitably coarse.

Chapter 5

Development of the ARC Load Balancing Framework

Load balancing can be defined as *"the distribution of computations fairly across processors in order to obtain the highest possible execution speed"* [131]. Load balancing techniques can improve performance in situations where an application to be executed does not decompose neatly into a collection of identical parallel subtasks, there are more subtasks to be executed than processors available, the processors used are heterogeneous, or a combination of these conditions is present. Any ARC application potentially meets all of these conditions: it may be composed of non-identical subtasks, may generate more tasks than there are processors available, and is always executed on a heterogeneous collection of processors. Heterogeneity of processors arises due to the presence of FPGAs, irrespective of whether or not all the host processors are identical. Given that the conditions above are met, it follows that the implementation of an effective load-balancing scheme is a highly desirable feature if the ARC system is to execute applications efficiently.

In general, load balancing is a variant of the *bin packing* problem [132], i.e., the placement of objects in boxes with the aim minimizing the number of boxes required. In the case of load balancing, the boxes represent task queues, the packing occurs in one dimension only, the number of boxes is fixed and the aim is to minimize the size of the largest box (see Figure 5.1). The bin packing problem has been shown to be NP-complete [133] as a decision problem and is therefore NP-hard as an optimization problem [134]. As such, attempting to find optimal solutions is computationally prohibitive. Algorithms designed for the problem typically rely on heuristics to find near-optimal solutions in the search space.

The highest-level division of load balancing techniques is between those that operate
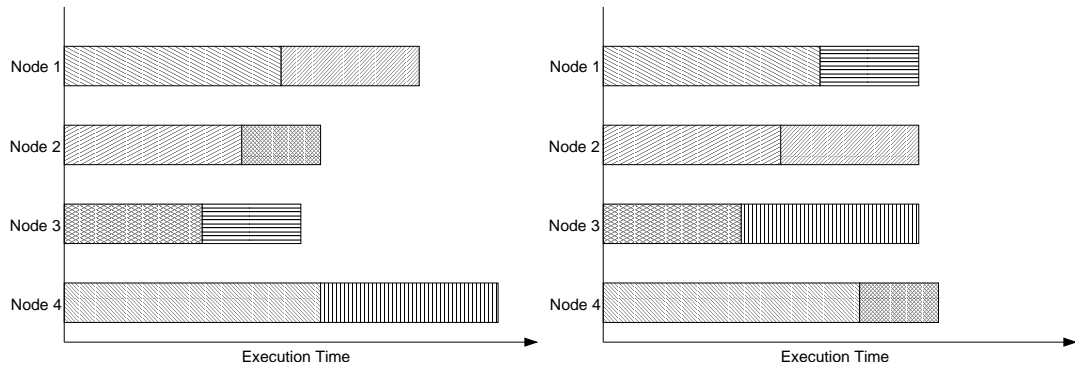
Figure 5.1: The application of load balancing techniques can significantly increase computational efficiency and hence reduce overall execution time. The shaded boxes represent discrete tasks.

statically and those that operate dynamically. Static load balancing algorithms [135] attempt to find a solution before execution commences, resulting in a map of tasks to resources, or *work plan*, that is followed at runtime. The *min-min* heuristic, where the tasks are ordered by increasing cost before being assigned in turn to the resource where they will complete soonest, has been shown to be highly effective, performing almost as well as work plans evolved using genetic algorithms. Crucially, the information required to produce a near-optimal work plan includes the cost of executing each task as well as the amount of data generated at each step. Determining this information ahead of time can be difficult, if not impossible. Dynamic load balancing algorithms, on the other hand, improvise the work plan as the computation progresses and new tasks arrive. As such, they may be able to leverage information that only becomes available after execution has commenced. However, in general they are not as effective as static load balancers given the same amount of information, and have the added drawback of imposing a performance overhead at runtime.

The following section describes how the load balancing requirements for ARC were determined within the context of an existing taxonomy of dynamic load balancing algorithms. The resulting load balancing framework and results from the evaluation of candidate algorithms is presented in Section 5.2. The algorithms described are not specific to the ARC system; they are generally applicable to any situation where load balancing over distributed reconfigurable hardware is required. Section 5.3 describes how the effectiveness of the basic framework was improved through the application of various optimizations.

5.1   Identification of ARC Load Balancing Requirements

During the initial design stages of the load balancing scheme for ARC it was immediately apparent that the static approach would be unsuitable because the necessary information,

such as the execution cost of tasks and even the number and type of the tasks themselves, is not available ahead of time. The focus was therefore on dynamic techniques from the outset. However, the field of Dynamic Load Balancing encompasses a wide variety of methodologies, many of which were developed for multiprocessors or are otherwise unsuited to the task at hand.

A taxonomy, described in Section 5.1.1 below, is available that allows dynamic load balancing algorithms to be classified based on their behaviour. It was decided that this taxonomy would serve as a useful starting point for determining ARC load balancing requirements. By examining the various behaviours available in each category in the taxonomy, the applicability of each behaviour to the context of the ARC runtime environment could be considered, and hence the most suitable behaviour in each category could be determined. Once this activity was completed, i.e., the entire taxonomy tree was traversed, then a significant portion of the required behaviour of the ARC load balancing scheme could be finalized. Experimentation could then be used to determine the most suitable behaviour for the remaining undefined behavioural aspects.

5.1.1 Taxonomy of Dynamic Load Balancing Algorithms

A taxonomy of dynamic load balancing strategies was proposed in [136] that allows all dynamic load balancing algorithms to be classified according to four main characteristics: *initiation*, *location*, *information exchange* and *load selection*.

Initiation describes the mechanism through which the load balancing algorithm is invoked, and may be *periodic* or *event-driven*. Periodic strategies invoke the load balancing algorithm at predetermined time intervals. When an event-driven strategy is employed, the algorithm is invoked based on the load state of individual processors. *Sender-initiated* event-driven strategies trigger load balancing behaviour when an "overloaded" threshold value is exceeded on a processor, causing work to be offloaded to another, more lightly loaded, processor. *Receiver-initiated* event-driven strategies take the opposite approach, using an "underloaded" threshold value to cause work to be migrated to processors that have become underloaded. A combination of sender and receiver-initiated strategies may also be used, where each processor has both an overloaded and an underloaded threshold value.

Location describes where the algorithm is executed upon invocation. The broadest distinction is between *centralized* and *decentralized* algorithms. Centralized algorithms execute on a single master processor that is responsible for assigning tasks to the other (slave) processors based on their load information. Distributed algorithms execute on every processor

participating in the computation, with each individual processor responsible for determining when and where to delegate work. Distributed algorithms can be further classified as *synchronous* or *asynchronous*. Synchronous algorithms execute simultaneously on all processors, with task execution suspended until load balancing is complete. Asynchronous algorithms, on the other hand, can execute at any time on individual processors regardless of the state of the others.

Information exchange specifies how load information is transmitted between processors, and how load information is acted upon. *Decision making* describes whether load balancing decisions are made *locally*, i.e., based on information from individual or neighbouring processors, or *globally*, i.e., based on information from all processors. *Communication* describes both the *topology* used for information exchange and the locality of task exchange. The communications topology may be *uniform*, where each processor communicates with a fixed set of neighbours, or *randomized*, where processors are assigned neighbours at random. *Task exchange* describes whether work is exchanged locally, i.e., between neighbouring processors, or globally between all processors.

Load selection describes how processors are matched in order to exchange work (*processor matching*) and how work is subsequently divided between processors (*load matching*). Although many policies exist for both subcategories, processor matching typically involves determining overloaded and underloaded processors according to a load average and exchanging work accordingly, while load matching is typically carried out by moving work packets from overloaded processors until their load drops below a specific threshold such as the load average.

5.1.2   Classification of Required Load Balancing Behaviour

Each aspect of the taxonomy was examined in turn in order to identify firm requirements for the desired behaviour of the ARC load balancing scheme. The first requirement identified was that the load balancing initiation mechanism used should be event-driven rather than periodic. Periodic load-balancing may cause the load balancing algorithm to be invoked unnecessarily when the load is evenly distributed. Furthermore, when load imbalance does occur there may be a delay before the algorithm is invoked, leaving some processors overloaded and others starved of work. It was decided that in order to avoid this scenario, individual computation processes should invoke the load balancing algorithm as soon as they detect that they have become overloaded, requiring a sender-initiated event-driven algorithm.

Due to the relatively poor latency and bandwidth of commodity networking hardware, the delay incurred in centrally gathering all the required information, such as the number

and type of tasks present at each cluster node, could become a performance bottleneck. One potential solution to this problem would be to use a master-slave approach by exposing parallelism (i.e., evaluating condensed nodes) only on one machine. However, this approach would make disproportionate use of the memory of the master machine while underutilizing the memory present at the slaves. The second requirement identified therefore was that the location of the chosen algorithm should be decentralized.

The peer-to-peer nature of the ARC computation process design naturally results in a flat communications topology. Furthermore, since the proximity of cluster nodes in a switched Ethernet network has no bearing on communication times, assigning nodes to groups would be creating an artificial distinction. Furthermore, the LinuxNOW library used for communications allows cluster nodes to easily broadcast their load to, and exchange work with, all their peers. As a result, it was determined that in terms of information exchange, decision making should be global, the communications topology should be uniform and and that task exchange should be global. Although the use of a flat topology potentially raises scalability issues if very large clusters are used, it was decided that global information sharing would result in better efficiency for the vast majority of cluster configurations.

In summary, the requirements identified were that initiation should be sender-initiated and event-driven, the location should be decentralized, the communications topology should be uniform and that task exchange should be global. These characteristics served as the starting point from the which the development of the ARC load balancing framework, as described in the following section, began.

5.2   Development of Basic Load Balancing Framework

An ARC load balancing framework was created in order to satisfy the requirements identified above while still allowing the ability to experiment with different approaches to initiation and processor selection. The high-level strategy is based on the observation in [137] that in general the best dynamic load balancing results are obtained when new tasks are mapped to the machines that can complete their execution soonest (i.e., the *min-min* heuristic applied to individual instructions). In order to approximate this behaviour, participating nodes must be able to decide whether a newly created instruction can be executed soonest by assigning it to a local resource or delegating it to another machine. If the instruction is to be delegated, then the machine with the lowest completion cost must be determined. In order to facilitate these decisions, some method of approximating the workload associated with a resource had to be found. Furthermore, the decentralized nature of the runtime environment necessitates the distribution of this load information across the cluster and the transmission of instructions from one machine to another upon delegation. The main

components of the framework are therefore the *load metric*, *load advertisement*, *initiation* of load balancing behaviour, *processor selection* and *task exchange*.

In order to determine which machines are overloaded and underloaded, a load metric was required. The ideal metric would be the true execution costs of all the tasks awaiting execution on a particular machine. As noted above, this would be difficult, if not impossible, to determine automatically. Alternatively, requiring the programmer to provide a means of specifying the execution costs of tasks would result in significant extra human effort and would be impractical even if it were possible. In the absence of cost information, the length of the instruction queue associated with a resource was chosen as the most suitable load metric. If the resource in question is executing an instruction when the instruction queue length is being calculated, then the resulting value is incremented by one. A load value is made available for every processing resource present at each cluster node. Typically, there are two, representing a single host processor and FPGA.

Load advertisement is performed through callback functions passed to the LinuxNOW library that specify what information should be sent in outgoing load information messages and the actions that should be performed when a new load information message arrives. Load information messages contain the length of the native instruction queue, the length of the FPGA instruction queue and the current FPGA configuration. Each computation process instance maintains a table containing load information about its peers. A peer's entry in the load information table is updated whenever a new load information message from the peer arrives. The load information table is also updated whenever a task is delegated to a peer by incrementing the load value associated with the appropriate resource. The LinuxNOW library is responsible for periodically broadcasting the load information for each machine at user-configurable time intervals. Longer broadcast intervals can be used when executing coarse-grained applications in order to conserve bandwidth, while shorter intervals minimize load information staleness when executing fine-grained applications.

The scheduling component of a computation process invokes the load balancing algorithm whenever a new instruction arrives, either from the Condensed Graphs Engine or the Communications Module. If the operator associated with the instruction is a TM operator, then the instruction is immediately enqueued on the native instruction queue. If the instruction originated remotely or the current machine is underloaded relative to its peers, then the operator information table is consulted to determine whether the instruction's operator is implemented in software or hardware, and the instruction is enqueued on the appropriate queue. If the machine is found to be overloaded, then the *remote processor selection algorithm* is invoked to determine which peer will be used for task delegation. The instruction is then enqueued on the outgoing instruction queue, and is subsequently serialized and communicated by the LinuxNOW library. These decisions are represented
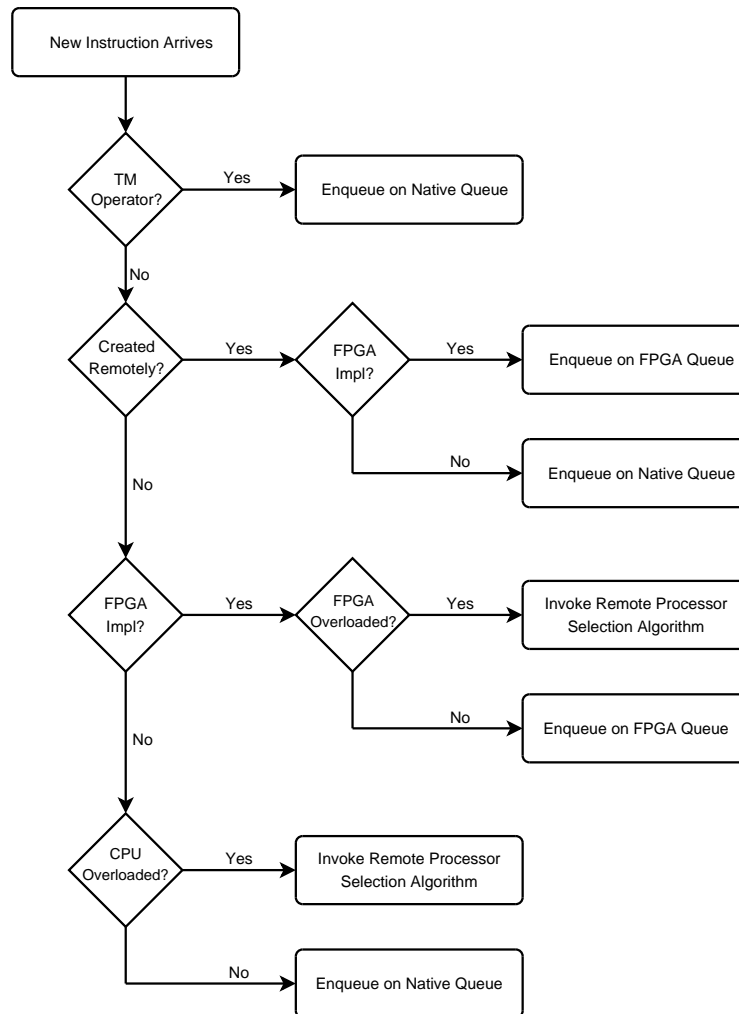
graphically in Figure 5.2 below.

Figure 5.2: A flow chart showing the decisions taken and actions performed by the basic ARC load balancing framework when a new instruction arrives.

Task migration is performed by the LinuxNOW library in conjunction with the ARC type serialization system. When an instruction is ready for transmission, LinuxNOW opens a socket to the remote machine. A callback function is then invoked with the socket's file handle as an argument. The callback function invokes ARC's instruction serialization functionality to serialize the instruction and its associated operands. Similarly, the LinuxNOW instance on the remote machine invokes another callback function to deserialize the instruction before sending it to the Scheduler.

The framework presented above encapsulates the basic functionality required for the implementation of ARC load balancing schemes. However, some aspects of its operation are simplistic and could be improved upon. For example, the default queuing scheme does not attempt to avoid reconfigurations by ordering FPGA instructions and instructions with

dual operator implementations are always assigned to FPGAs. The reasoning behind these design decisions was that extensions to the basic behaviour should be added based on demonstrable benefit only. A number of optimizations to the basic framework that address these issues are examined in Section 5.3.

The current framework does not consider the states of the application graph instances when making load balancing decisions, and neither does the Scheduler consider the states of the available processing resources when choosing the next fireable node to be evaluated. The modular nature of the ARC runtime environment as it is implemented at present precludes this; communications between the Scheduler, Condensed Graphs Engine and Load Balancing Component are solely in the form of instructions and results. A more advanced framework could involve the creation of a work plan in the Scheduler and the creation of a feedback loop between the various modules. Information about upcoming instructions would be obtained from the Condensed Graphs Engine through analysis of the computation graphs. Branch prediction could be used to account for the presence of `Ifel` nodes and nondeterministic behaviour. The ordering of instructions in the work plan would be modified on the fly based on the arrival of load balancing information from other nodes. The feedback element would arise from the fact that the work plan is altered based on the state of the remote nodes as reflected by the load balancing module, and the state of nodes is in turn altered by the work plan, with the resulting changes being fed back to the load balancing module via load information messages. Similar work has been performed in the field of optimizing compilers [138], and would serve as a useful starting point. However, this arrangement would necessitate significant alterations to the runtime environment and is left as a topic of future research.

Although the framework specifies the high level operation of the ARC load balancing scheme, load balancing behaviour can be altered significantly through the use of different subalgorithms, i.e., the load categorization algorithm and the remote processor selection algorithm. Candidate algorithms for each of these subalgorithms are presented and evaluated in subsequent sections.

5.2.1   Candidate Load Categorization Algorithms

The load categorization algorithm is consulted to determine whether the machine is overloaded or underloaded relative to its peers, and hence whether the current instruction under consideration should be enqueued locally or sent to the communications module for remote execution. The following candidate algorithms were considered:

- *Always Delegate.* When invoked by the scheduler, the current machine is always considered overloaded, resulting in the delegation of all suitable instructions. The

most obvious drawback of this approach is that the locally available resources that can be exploited without incurring a communications overhead are ignored.

- *Not Least Loaded.* When invoked on a machine m with a set of peers P, the machine is found to be overloaded if $\exists p \in P : \mathrm{load}(p) < \mathrm{load}(m)$. This will cause the instruction to be delegated if any less loaded peer is available, resulting in aggressive task delegation.

- *Threshold.* When invoked on a machine m, the machine is found to be overloaded if $\mathrm{load}(m) > t$, where t is the specified threshold value. Threshold values of 0, 1 and 2 were considered.

- *Above Average.* When invoked on a machine m, the machine is found to be overloaded if $\mathrm{load}(m) > a$ where, for a fully connected network,

$$a = \frac{\mathrm{load}(m) + \sum_{p \in P} \mathrm{load}(p)}{|P| + 1}$$

is the average workload.

5.2.2   Candidate Remote Processor Selection Algorithms

The remote processor selection algorithm is invoked when an instruction has been designated for remote execution, and is responsible for selecting the most appropriate peer for task delegation. Since it is being assumed that the processing resources within each category (CPU or FPGA) are homogeneous, they can be distinguished based for load balancing purposes based on their current load alone. The following remote processor selection algorithms were considered:

- *Random.* A peer is chosen uniformly at random from the set of peers. No account is taken of the load information available, and therefore no attempt is made to deliberately match overloaded and underloaded machines when delegating work. As a result, this scheme was expected to perform poorly but was chosen as a benchmark for performance in the absence of deliberate load balancing.

- *Random Less Loaded.* When invoked on a machine m, a peer is chosen uniformly at random from the subset of peers $Q = \{p \in P : \mathrm{load}(p) < \mathrm{load}(m)\}$. If Q is empty then the set of peers sharing the lowest load value is used instead. This operates in a similar fashion to the random algorithm, except that only less loaded peers are considered for task delegation. Although this approach guarantees that tasks are always delegated to less-loaded machines when possible, the degree to which machines are underloaded is not considered.

- *Least Loaded.* A peer is chosen uniformly at random from the subset of peers $Q = \{q \in P : \text{load}(q) = \min_{p \in P} \text{load}(p)\}$. Only those peers sharing the lowest load value are selected, ensuring that work is only sent to the most underloaded machines. One potential drawback of this approach is that load information staleness can result in the creation of "hotspots", i.e., processors that become suddenly overloaded due to excessive task delegation by their peers.

- *Localized Round Robin.* A peer is chosen by using the current value of the variable $c \geq 0$ to index term $c \bmod |P|$ of the sequence $\{s_n\}_{n=0}^{|P|-1}$ composed of the elements of P sorted by host name. The initial value of c is zero and it is incremented after every invocation. Although this algorithm ensures that work is distributed fairly from a local point of view, it does not take the actions of the peers into account by examining the available load information.

- *Mirror.* When invoked on a machine m, the sequence $\{s_n\}_{n=0}^{|P|}$ is created by ordering the elements of $P \cup \{m\}$ partially by load then totally by host name. The jth element of s is then chosen as the delegation target, where j is calculated in terms of i (the position of m in s) as follows:

$$
j = \begin{cases}
|P| - i & \text{if } i \neq |P| - i \\
|P| - i - 1 & \text{if } i = |P| - i \text{ and } i \neq |P| \\
|P| - i + 1 & \text{otherwise.}
\end{cases}
$$

  The aim of this approach is to match overloaded machines with comparatively underloaded peers, so that most overloaded machine is matched with the least loaded, the next most overloaded is matched with the next most underloaded, and so on. This method of processor matching was adapted from a synchronous load balancing algorithm, titled *rendezvous*, described in [139].

5.2.3 ARC Modifications to Support Load Balancing Experimentation

In order to determine the best combination of load categorization and remote processor selection algorithms for use with the ARC load balancing framework, some method of objectively ranking the candidate algorithms in each category was required. To this end, experiments were conducted in order to gather empirical data on the effectiveness of each algorithm. It was decided that the estimates of lowest possible execution time produced by the performance model presented in previous chapter was the most suitable benchmark for evaluating the correctness of load balancing decisions. By executing applications created using the application generation program, the computing resource or resources with the

lowest completion cost could to be estimated for any newly created instruction. Assignment to one of these resources would result in behaviour equivalent to the *min-min* heuristic applied to individual instructions. These "ideal" assignments of instructions to resources could then be compared with the actual assignments made by the load balancing algorithm in question.

In order to estimate the least possible execution time for an instruction across a cluster using the performance model, detailed information about the state of each cluster node (such as the number and costs of the instructions in queues) must be known. This information relating to all the nodes participating in the cluster is not available to individual cluster nodes at runtime; the distributed nature of the ARC runtime environment precludes the instantaneous determination of non-local information. Although the load advertisement mechanism could be extended to include the required information from each node in load information broadcasts, the information provided would suffer from staleness and hence reduce prediction accuracy. It was therefore decided that the most accurate estimates could be obtained by recording the required information in real-time at each cluster node during runtime, with analysis being performed post-execution.

Estimating instruction execution times post-execution required that a detailed log be maintained of the state of the instruction queues on each cluster node during execution. This was achieved by modifying the instruction queue and execution thread implementations so that the following information was recorded whenever an instruction was enqueued or dequeued:

1. A timestamp composed of the two `long` values (representing the second and microsecond components of the current system time) obtained from a call to `gettimeofday`.

2. The operator name and execution time remaining of the currently executing instruction. If no instruction was executing when the entry was recorded, then `NULL` and 0 are used for these values respectively. When writing log entries for FPGA queues, the adjusted cost value taking FPGA speedup into account is used.

3. A list containing the operator names and costs of the instructions in the queue.

An excerpt from an example queue state log file is provided in Figure 5.3(a) below. The total cost of the instructions in a queue at any moment in time can be determined by calculating the total cost at the time of the latest entry in the log file with a timestamp lower than the time of interest, and then subtracting the difference between the time of interest and the timestamp. If the result of this subtraction is less than or equal to zero, then it can be concluded that the queue was empty at the time of interest.

```
1132586822, 274638, g6, 0.000000 [op9, 1003.000000]
1132586822, 276804, op9, 1002.998000 []
1132586822, 293219, op9, 986.583000 [op10, 977.000000]
1132586822, 313180, op9, 966.622000 [op10, 977.000000: op9, 1003.000000]
1132586822, 552942, op9, 726.860000 [op10, 977.000000: op9, 1003.000000:
                                     op0, 950.000000]
1132586823, 280470, op10, 976.999000 [op9, 1003.000000: op0, 950.000000]
1132586823, 598897, op10, 658.572000 [op9, 1003.000000: op0, 950.000000:
                                     op2, 1.000000]
1132586824, 258246, op9, 1002.998000 [op0, 950.000000: op2, 1.000000]
1132586825, 262007, op0, 949.998000 [op2, 1.000000]
1132586825, 992898, op0, 219.107000 [op2, 1.000000: op1, 1.000000]
1132586826, 212685, op2, 0.998000 [op1, 1.000000]
1132586826, 216915, op1, 0.997000 []
```

(a) Queue States

```
1132586822,213691,op5,1.000000,-1.000000,201.000000,CPU,OVERLOADED
1132586822,215849,op8,962.000000,4.000000,191.000000,FPGA,UNDERLOADED
1132586822,216318,op9,1003.000000,-1.000000,202.000000,CPU,UNDERLOADED
1132586822,272331,op5,1.000000,-1.000000,190.000000,CPU,OVERLOADED
1132586822,292833,op10,977.000000,-1.000000,190.000000,CPU,UNDERLOADED
1132586822,312802,op9,1003.000000,-1.000000,98.000000,CPU,UNDERLOADED
1132586822,313299,op3,1214.000000,-1.000000,102.000000,CPU,OVERLOADED
1132586822,315296,op6,956.000000,5.000000,95.000000,FPGA,UNDERLOADED
1132586822,492310,op8,962.000000,4.000000,174.000000,FPGA,UNDERLOADED
1132586822,532802,op2,1.000000,-1.000000,201.000000,CPU,OVERLOADED
1132586822,534925,op5,1.000000,-1.000000,186.000000,CPU,OVERLOADED
1132586822,552561,op0,950.000000,-1.000000,206.000000,CPU,UNDERLOADED
1132586823,293809,op3,1214.000000,-1.000000,200.000000,CPU,OVERLOADED
```

(b) Load Categorization Decisions

```
1134572960,854655,g9,0.000000,-1.000000,93.000000,node04.cuc.ucc.ie,CPU
1134572968,076178,op6,1.000000,-1.000000,193.000000,node02.cuc.ucc.ie,CPU
1134572968,095054,op1,1.000000,-1.000000,182.000000,node02.cuc.ucc.ie,CPU
1134572968,196591,op7,1155.000000,5.000000,196.000000,node04.cuc.ucc.ie,FPGA
1134572968,198476,op3,1103.000000,4.000000,178.000000,node04.cuc.ucc.ie,FPGA
1134572968,200447,op3,1103.000000,4.000000,178.000000,node04.cuc.ucc.ie,FPGA
1134572968,395800,op6,1.000000,-1.000000,297.000000,node03.cuc.ucc.ie,CPU
1134572969,014369,g9,0.000000,-1.000000,198.000000,node03.cuc.ucc.ie,CPU
1134572970,236341,op9,1046.000000,6.000000,199.000000,node03.cuc.ucc.ie,FPGA
1134572970,256498,g8,0.000000,-1.000000,190.000000,node04.cuc.ucc.ie,CPU
1134572970,514762,op8,938.000000,-1.000000,198.000000,node02.cuc.ucc.ie,CPU
1134572972,335502,op6,1.000000,-1.000000,405.000000,node02.cuc.ucc.ie,CPU
1134572972,337546,op8,938.000000,-1.000000,404.000000,node04.cuc.ucc.ie,CPU
```

(c) Remote Processor Selection Decisions

Figure 5.3: Excerpts from the log files used for load balancing algorithm evaluation. The first two columns in each log file format represent the second and microsecond components of the timestamp value. Some entries in Subfigure (a) have been indented for clarity.

Given the ability to reconstruct the state of instruction queues post-execution, the next modification required was the recording of load balancing decisions and details of the instructions in question. In light of this, the load balancing framework was modified to record the relevant information whenever a load balancing algorithm is invoked:

1. A timestamp value following the same format as that used in instruction queue state logs.

2. A value representing the decision made by the load balancing algorithm in question. The type of value recorded depends on the category of the algorithm. Load categorization algorithm invocations result in a value indicating whether or not the resource in question was found to be overloaded (see Figure 5.3(b)). Remote processor selection algorithms result in an assignment to a remote processing resource (see Figure 5.3(c)).

3. The operator name, cost, speedup and data size (including the size of the operands) of the instruction in question. A speedup value of -1 indicates that the operator associated with the instruction does not have an implementation in hardware.

The information relating to the different subalgorithms is logged to separate files. Entries to the relevant files are only made when the subalgorithm in question is invoked; as Figure 5.2 illustrates, the arrival of a new instruction does not necessarily result in the invocation of any of the subalgorithms.

Following the implementation of the modifications described above, all the information necessary for load balancing decision evaluation was now available post-execution. Processing of decisions could be performed by iterating through the entries in the log for the subalgorithm of interest. For each entry, the timestamp value was used to reconstruct the state of the instruction queues across the cluster at the time that the algorithm was invoked. The performance model, invoked with the relevant parameter values, was then used to determine the correctness of the load balancing decision in question.

5.2.4   Evaluation of Load Categorization Algorithms

In order to determine the most effective load categorization algorithm amongst those described in Section 5.2.1, a suitable metric that allows objective comparison had to be found. Since each invocation of a load categorization algorithm results in a decision that can be shown to be correct or incorrect, *decision accuracy*, defined as the ratio of the number of correct decisions to the total number of decisions, was chosen as the most objective means of comparison.

The application generation program was used to create a common set of applications for evaluating the correctness of the candidate load categorization algorithms. In order to gain an understanding of the upper bound to load categorization decision accuracy in practice, the initial experiment was designed to be as favourable as possible to the load balancing process; operation costs were fixed at 1,000 ms, and only CPUs were used. An application set consisting of ten applications with these properties were generated and executed repeatedly using the various load categorization algorithms.

Unfortunately, anomalies were soon observed in the results obtained. For example, examination of the decision accuracy of invocations of *Threshold* $(t = 0)$ where the result was an underloaded categorization revealed that these categorizations were only correct approximately 80% of the time. However, *Threshold* $(t = 0)$ will only produce an underloaded categorization if the CPU in question is idle, i.e., no instruction is currently being executed and the instruction queue is empty. In this situation, an underloaded classification for a CPU should *always* be correct due to the transmission overhead incurred when delegating instructions. It was clear that the application of the performance model post-execution was not working as well as had been hoped, and that further investigation would be required to ascertain the cause of the anomalous behaviour.

Close examination of the load categorization logs together with the relevant queue state log files revealed two distinct but related issues: clock skew and clock granularity. Clock skew had been an issue during the evaluation of the performance model when the use of the Network Time Protocol (NTP) [140] daemon had been found to produce unsatisfactory results. This was resolved by running the NTP daemon on a single cluster node, with the others synchronizing their clocks with this node before application runs with calls to the `ntpdate` command. These calls included an argument specifying that the time should be set immediately rather than gradually adjusted, which is the default behaviour. Although this arrangement had proven sufficient for the experiments in Chapter 4, even small differences in system clock times lead to significant inaccuracies when performing post-execution queue state analysis. For example, an overloaded categorization may cause an instruction to be delegated to a machine with a slower clock. When analyzing this decision later, differences in clock times may cause the instruction that was delegated to appear in the queue of the receiving node, causing the queue to appear to be longer than it was in reality. Although the differences between the clocks may be insignificant compared to the mean instruction cost (in the order of a few milliseconds), even small differences may have a significant impact on the reliability of queue state reconstruction due to the short periods of time involved in task delegation. Furthermore, task delegations often happen in close succession as the processing of a result uncovers further instructions, compounding the inaccuracies created. The problem of clock granularity arises from the fact that the analysis program

uses timestamps with a granularity of one millisecond to order events. However, several changes to a queue may take place in a single millisecond; attempting to match the correct queue state to a decision logged at the same millisecond value is a futile exercise.

The difficulties described above are manifestations of the more general problem of correctly ordering events in distributed systems. Fortunately, consultation with the literature revealed that this problem is not new and has been studied extensively over the years. Although techniques are available that can synchronize distributed clocks to within very small margins of error, it was clear that *any* inaccuracy would adversely affect the reliability of queue state reconstruction. Furthermore, even if the clocks of the cluster nodes could be synchronized perfectly, this would not address the issue of timing granularity.

One technique that could address both issues was the creation of a *logical clock* that is independent of the physical clocks in the participating machines. Originally proposed in [141], the idea has since been extended by others [142], although further developments have been incremental with the fundamentals remaining unchanged. A synopsis of the technique is as follows: a distributed system is modelled as a collection of processes that communicate using messages. Each process is composed of a sequence of events, with the sending or receiving of messages considered to be events in their own right. The "happened before" relation $\rightarrow$ can be used to provide an irreflexive partial ordering of all the events in such a system. If a and b are events in the same process and a appears before b in the process's sequence of events, then $a \rightarrow b$. If a represents the sending of a message by a process and b represents the receipt of that message by another process, then $a \rightarrow b$. If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$. These simple rules suffice to create a partial ordering of the events in the system. Two events a and b are considered to be concurrent ($a \parallel b$), and hence not ordered, if $a \nrightarrow b$ and $b \nrightarrow a$. As no causality relationship exists between concurrent events, an arbitrary method of ordering them (such as by host name) can be used to convert the partial ordering of events to a total ordering.

The notion of a logical clock follows naturally from the observations above. Here, the word *clock* simply means a method of assigning a number to an event rather than the more familiar meaning of a timing mechanism. Every process P_i maintains a nonnegative clock variable C_i that is incremented between successive events. Whenever a process sends a message, the current value of the process's clock is used as a "timestamp" and included in the message. Upon receipt of a message, the receiving process sets its clock to a value greater than or equal to the current value and greater than the timestamp contained in the message. It has been shown that if this system is followed, then for any two events a (executed by P_i) and b (executed by P_j) in the system, if $a \rightarrow b$ then $C_i(a) < C_j(b)$. Note that some method of breaking ties must still be used to order concurrent events.

Given the availability of such a simple and elegant method to totally order the events in

a distributed system, all that remained was the question of how to apply the method to the ARC runtime environment. This issue was complicated somewhat by the fact that an ARC computation process is not composed of a neat, sequential series of events as is assumed by the model described above. Instead, a computation process is itself composed of number of intercommunicating threads. This observation raises a number of further questions; for example, as CPU and FPGA instructions execute concurrently, should each resource have its own clock? If so, which messages should be used to increment which clock? On reflection, it was decided that each computation process should maintain a single clock variable, and that this clock should be incremented upon the completion of an instruction of either type. A computation process's clock should be moved forward on the receipt of a message of any type. The clock should also be incremented whenever a load balancing decision is made, so that decisions and queue states could be ordered correctly, eliminating the granularity issue. The runtime environment was modified to incorporate these changes, and the load balancing decision and queue state log formats were changed to incorporate the "time" according to the logical clock as well the physical clock. The physical time stamps were still required in order to calculate the total cost of the instructions in a queue at any given time during a computation.

Although the extent to which anomalies were present in the experimental results was drastically reduced using the new logical ordering scheme, some anomalous behaviour was still observed, albeit on a much smaller scale. The cause of this behaviour was traced to a concurrency issue within the runtime environment. The load associated with a resource was calculated by taking the length of the associated instruction queue and incrementing the resulting value if the resource was currently executing an instruction. The status of the resource was determined by checking a flag that was set upon commencement of the execution of an instruction, and unset afterwards. Occasionally, the state of the flag would be queried when an instruction had been dequeued but had not commenced execution. In this case, the existence of the pending instruction cannot be determined using the method described above, and an erroneous load value results. This problem was solved by associating a counter rather than a flag with each resource. The counter is incremented whenever a new instruction is enqueued, and decremented whenever the execution of an instruction is completed by the execution thread. The implementation of this scheme eliminated all the observed anomalous behaviour.

The results obtained with varying execution parameters are presented in the following pages. In order to aid analysis of the performance of the individual load categorization algorithms, individual results are visualized as a pair of charts representing various execution characteristics. Load balance and decision accuracy are represented by a pie chart containing both dividing lines and a shaded area. The dividing lines represent the ratio of

the number of underloaded states to overloaded states. These were determined as follows: for every load categorization decision, the states of the instruction queues of the cluster nodes were reconstructed, post-execution, for the time at which the decision took place. These states were then categorized as either underloaded or overloaded based on whether or not the instruction in question could be completed soonest on the local machine or a remote machine, respectively. The position of the dividing lines within a chart therefore indicates how effectively the system has been load balanced; when execution across two machines is considered, then a 1:1 ratio of overloaded states to underloaded states would indicate that the load had been well balanced, i.e., in half the instances of a new instruction being considered for delegation the local machine was underloaded and in the other half the local machine was overloaded. The shaded area represents how successfully the load categorization algorithm under consideration correctly identified these overloaded and underloaded states; the white and grey areas within the underloaded pie slice represent the ratio of correct load categorizations to incorrect load categorizations when the machine was underloaded, as determined by the performance model. Similarly, the white and grey areas within the overloaded pie slice represent the ratio of correct load categorizations to incorrect load categorizations when the machine was underloaded. As stated above, a decision is classified as correct if its behaviour matches that of the *min-min* heuristic, i.e., the instruction is assigned to the machine where it will complete execution soonest. The shaded areas within the slices are arranged so that they are contiguous; the overall decision accuracy of the algorithm can be determined by examining the proportion of the pie chart as a whole that is unshaded.

The speedup observed compared to execution on a single machine is represented as a bar underneath each pie chart. The NP-complete nature of the problem at hand precludes the determination of the best possible speedup, which is dependent upon the characteristics of the applications and the machines used to execute them. Instead, the scale of the bar is determined by the number of nodes over which the applications were executed; the possibility of superlinear speedup is not considered. The speedup bar is useful in that it provides a hard metric of the performance of the algorithm being considered, and also provides a means of examining the extent to which decision accuracy and load balance impact actual execution performance.

The results observed when applications with CPU implementations only and a fixed operation cost of 1 second were executed across two cluster nodes are shown in Figure 5.4. Execution across two cluster nodes, although a relatively small number compared to the amount commonly used in practice, is particularly revealing when considering load categorization algorithms as instructions can only be delegated to one machine. The choice of processor selection algorithm is therefore irrelevant, and the load categorization algorithm

can be considered in isolation. Examination of the results reveals that *Always Delegate* and *Not Least Loaded* performed poorest, exhibiting low decision accuracy, load balancing and speedup compared to the other candidate algorithms. Both algorithms misclassify the load as being overloaded a significant proportion of the time. This would suggest that these algorithms are overly aggressive when delegating work, resulting in poor load balancing and hence poor speedup.

The *Threshold* algorithms performed surprisingly well when the fact that they do not take advantage of remote load information is considered. Significantly higher decision accuracies, load balancing and speedups were observed compared to the worse-performing algorithms described above. Although the three threshold values performed similarly in terms of decision accuracy, each exhibited different characteristics. When the threshold value, t, is zero the algorithm misclassifies over 50% of underloaded states while correctly classifying practically all overloaded states, suggesting that delegation is being performed too aggressively. When $t = 1$, the proportion of incorrectly classified underloaded states falls significantly, while a small proportion of overloaded states are misclassified. When $t = 2$, the proportion of misclassified underloaded states falls further, while the proportion of misclassified overloaded states increases. Progressively better load balancing and speedups were observed with increasing threshold values.

The results obtained for *Not Least Loaded* were similar to those for *Threshold* where $t = 2$ in that both exhibit similar decision accuracies and speedups. However, *Not Least Loaded* appears to balance the workload more effectively and examination of the distribution of misclassified load states would indicate that *Not Least Loaded* tends to be more conservative in task delegation. Despite the similarity of the results obtained, it was hoped that *Not Least Loaded* would prove to be a more effective general-purpose algorithm as it takes advantage of remote load information and hence should be able to adapt to differing application and hardware characteristics. It was decided that further experimentation was required in order to test this hypothesis (see below).

The results indicate that decision accuracy alone does not, as was hoped, provide an accurate measure of the performance of a load balancing algorithm as the differing threshold values produced different speedup values but with almost identical decision accuracies. Neither does load balance provide an accurate metric; comparing *Threshold (t = 0)* and *Above Average* reveals that although their load balance appears to be similar, the two algorithms differ significantly in terms of speedup obtained. On closer examination, it would appear that a combination of factors is at work, and that a high decision accuracy coupled with effective load balancing appears to be the best combination. Furthermore, it would appear that the system as a whole can function well despite the presence of a significant proportion of individually poor categorization decisions. It is difficult to see how the 1.9
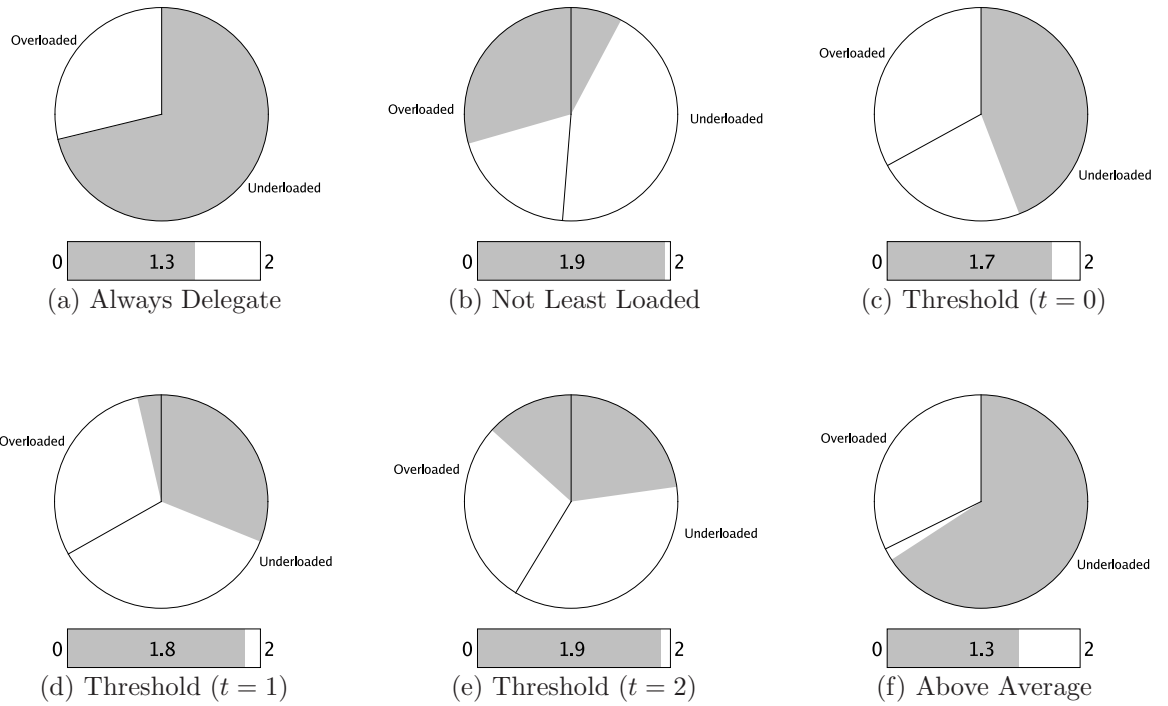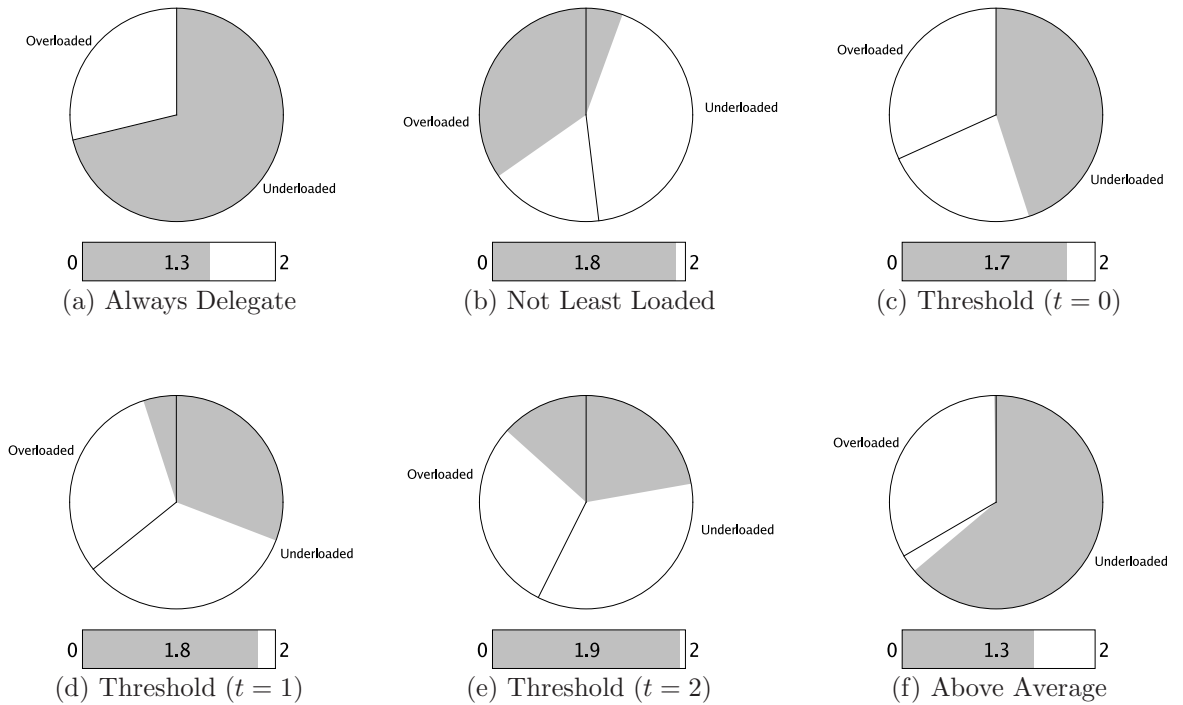
Figure 5.4: The decision accuracies and speedups observed during experimental evaluation of the candidate load categorization algorithms across two cluster nodes.
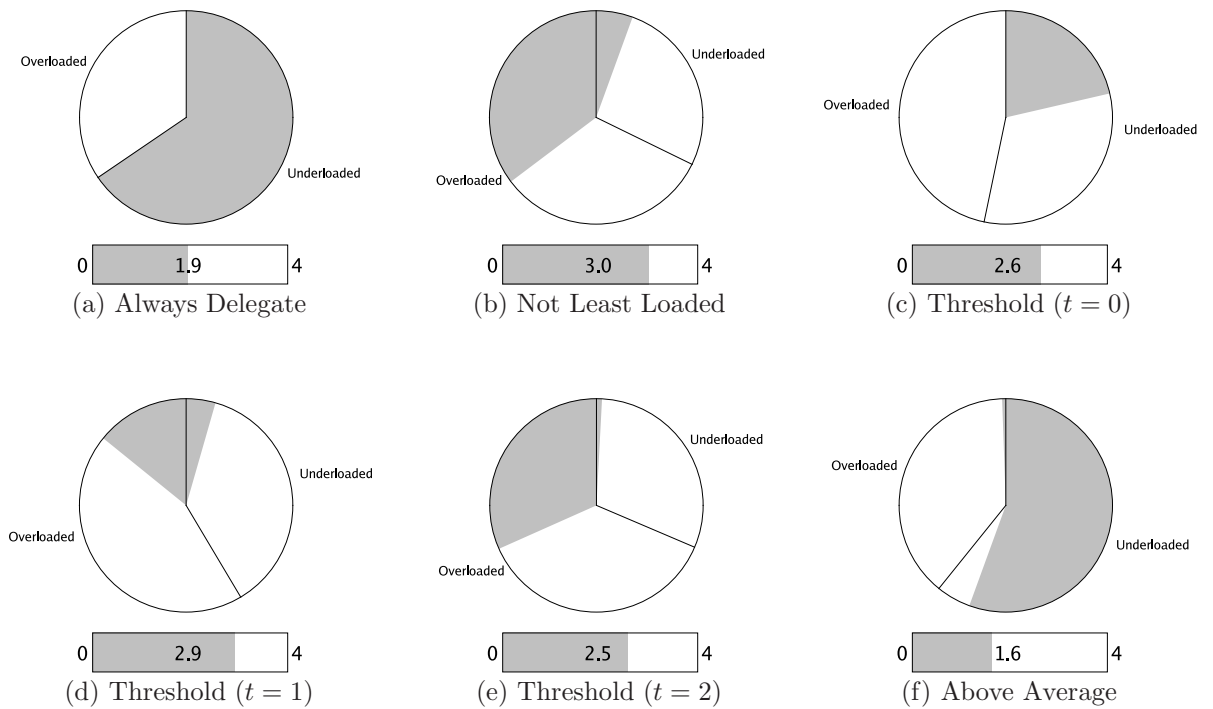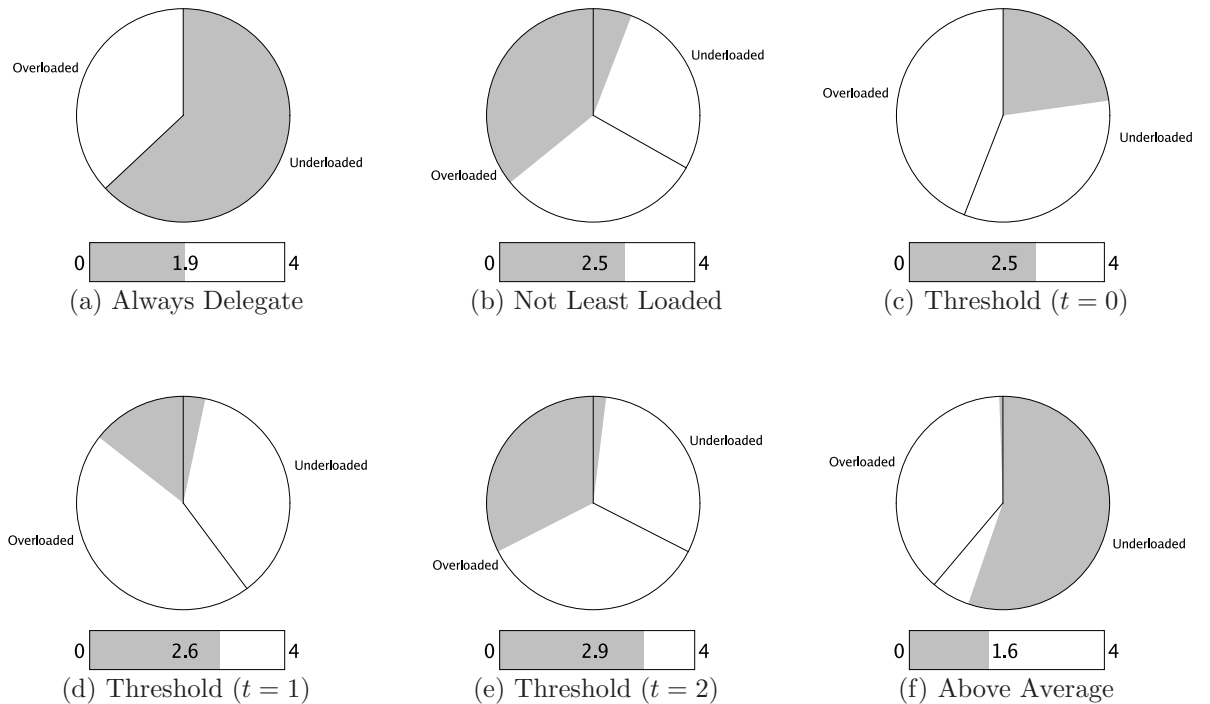

Figure 5.5: The decision accuracies and speedups observed during experimental evaluation of the candidate load categorization algorithms across four cluster nodes.

speedups obtained by *Not Least Loaded* and *Threshold (t = 2)* could be improved upon despite the fact that approximately one third of the decisions made by each were classified as incorrect. One explanation for this could be the definition of a "correct" decision, i.e., one that matches the behaviour of the *min-min* heuristic when applied to individual tasks. Many decisions that are classified as being incorrect according to the heuristic may be neutral or even more correct in terms of achieving the highest possible speedup, although the NP-complete nature of the problem precludes verification of this. Another explanation stems from the observation that the highest speedups were obtained using algorithms that divided their erroneous classifications between both categories of system state; it is possible that the incorrect classifications in one category have the effect of compensating for incorrect classifications in the other. Irrespective of the cause of the deviation between between decision accuracy and speedup, it can be concluded that while the examination of decision accuracy can provide insight into the behaviour of an algorithm, only raw speedup provides a reliable measure of its effectiveness.

A further experiment was conducted in order to test the hypothesis advanced above that the *Threshold* algorithm, despite its good performance in the experiment above, does not represent an effective general-purpose approach. To this end, the experiment above was repeated, but with execution taking place across four cluster nodes instead of two. *Least Loaded* was used as the remote processor selection algorithm. The results obtained are shown in Figure 5.5. *Always Overloaded* and *Above Average* again performed poorly, exhibiting poor decision accuracies, load balancing and speedups in comparison to their peers. It would appear that, as in the previous experiment, the use of these algorithms results in over-aggressive task delegation. The *Threshold* algorithms again displayed an increasing tendency to delegate instructions as the threshold value was incremented. In this instance, the best-performing value for t was 1 rather than 2. This indicates that although good performance can be obtained using *Threshold*, the value of t must be tuned to the application and hardware characteristics at hand, confirming the hypothesis presented above. The best results in terms of speedup were obtained using *Not Least Loaded*, demonstrating its effectiveness as a general-purpose load categorization algorithm. The divergence between decision accuracy and speedup was more pronounced in these results; although *Threshold* $(t = 1)$ exhibited by far the best decision accuracy, it was outperformed by *Not Least Loaded* which had the third lowest.

An additional experiment was conducted in order to investigate the importance of the choice of load information broadcast interval. For both experiments above, the value used was 500 ms, i.e., half the mean instruction cost. In order to investigate the effects of lengthening the broadcast interval, the experiments above were repeated using a broadcast interval ten times greater. The results are shown in Figures 5.6 and 5.7 below. When

Figure 5.6: The decision accuracies and speedups observed during experimental evaluation of the candidate load categorization algorithms across two cluster nodes when the load information broadcast interval was increased to 5 s.
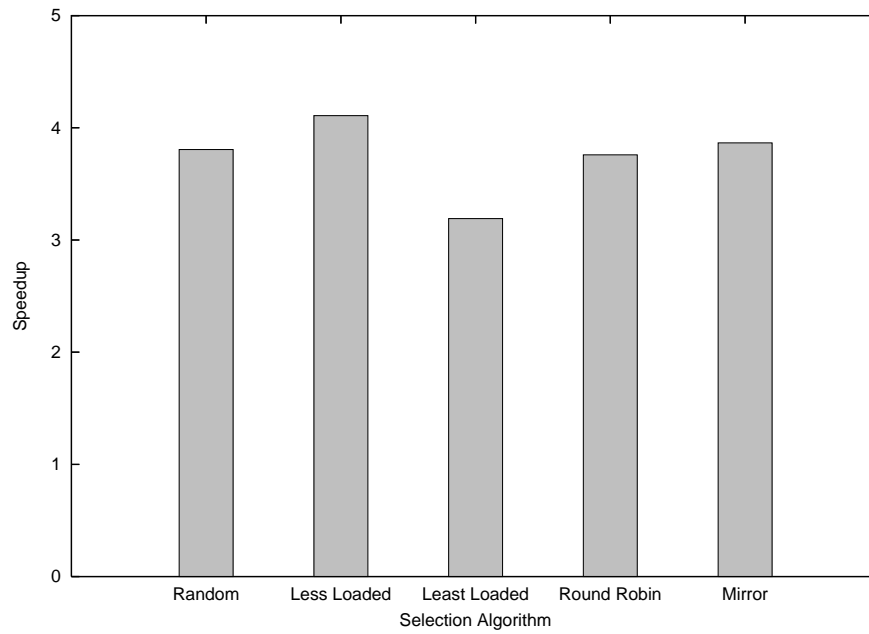

Figure 5.7: The decision accuracies and speedups observed during experimental evaluation of the candidate load categorization algorithms across four cluster nodes when the load information broadcast interval was increased to 5 s.

executing across two nodes, very little difference in execution characteristics was observed: the characteristics of *Above Average* were unchanged and only a slight drop was observed in the speedup associated with *Not Least Loaded*. This lack of variation can be explained by the updates performed by the Scheduler to the local load information table whenever an instruction is delegated. As new instructions tend to be created in groups as multiple nodes receive inputs from a newly-fired node, it would appear that the updates to the load information table whenever delegations occur results in reasonably accurate load information across two nodes, even when updates to the load information table are rare. The impact of lengthening the broadcast interval when executing across four nodes was more pronounced; although *Above Average* was again unchanged, *Not Least Loaded* recorded a drop of more than 15% in speedup.

5.2.5   Evaluation of Remote Processor Selection Algorithms

The evaluation of the candidate remote processor selection algorithms was performed in a similar fashion to the evaluation of the candidate load categorization algorithms. Initially, the same application set used to evaluate the load categorization algorithms were executed across four cluster nodes using the various remote processor selection algorithms. *Not Least Loaded* was used as the load categorization algorithm during all evaluations as it was found to be the best performing candidate in the previous section. Examination of the speedup values obtained revealed that there was little to choose between the effectiveness of the various algorithms. It was therefore decided that the experiment should be repeated across eight cluster nodes in order to allow the algorithms to differentiate to a greater extent. Due to the extra processing resources available when eight nodes are used, the application generation program was re-run to create a new application set with similar characteristics to the previous set but with additional graph definitions and more nodes per graph.

The new application set was re-run using each candidate remote processor selection algorithm, and the resulting speedup values were recorded. The decision made by the currently configured algorithm was also recorded upon every invocation. The load state logs were analyzed post-execution to arrive at a decision accuracy in a similar fashion to the method used to evaluate the load categorization algorithms. In this case, the decisions produced by the algorithm being evaluated were found to be correct if the completion cost of the instruction in question on the selected peer was equal to the lowest possible completion cost across all peers. The results obtained are shown as a pair of bar charts (one depicting the speedup values and the other the decision accuracy values) in Figure 5.8 below.

Examination of the speedup values in Figure 5.8(a) reveals that *Random Less Loaded* was the best performer, and the only algorithm to display a mean speedup greater than four.

(a) Speedup
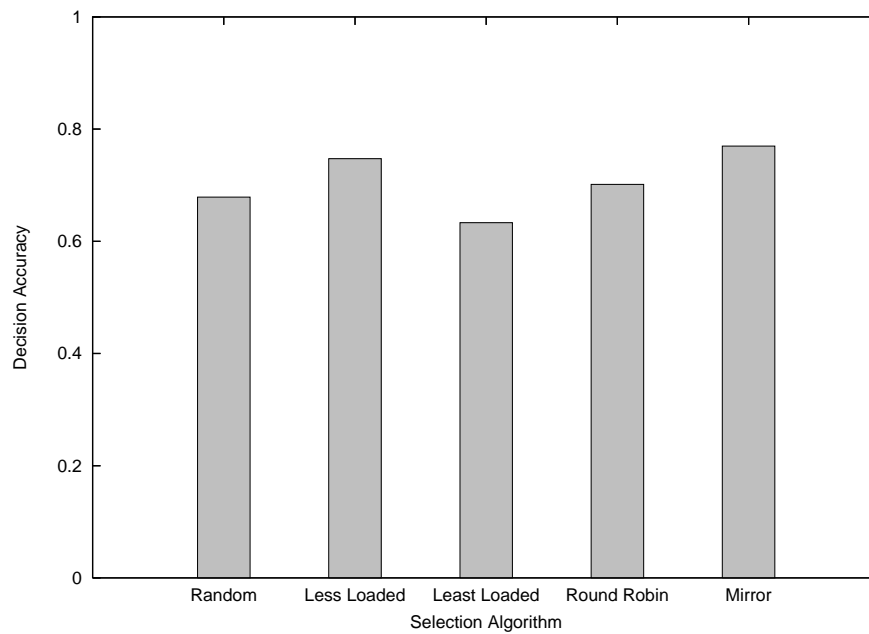
(b) Decision Accuracy

Figure 5.8: The speedup values and decision accuracies obtained when the various remote processor selection algorithms were evaluated across eight cluster nodes.

Least Loaded performed particularly poorly with a mean speedup of only 3.2, possibly due to the "hotspot" problem described above. The remaining algorithms performed similarly, with speedups of approximately 3.8. The appearance of *Random* in this group was somewhat surprising given that it does not take advantage of load information. The failure of any of the algorithms to achieve significantly greater than 50% of the theoretical maxmimum speedup (eight) was investigated through examination of the queue state logs. It was determined that the generated applications' execution profiles alternated between "quiet" periods, where not enough work existed to occupy all resources, and "busy" periods, where increases in parallelism resulted in more than enough work to keep all eight nodes busy. Full utilization of all the processing resources for the duration of a computation, and hence speedup values approaching the maximum, was therefore impossible.

Two conclusions can be drawn from examination of the decision accuracy values in Figure 5.8(b). The first is that the accuracy values obtained were significantly better than expected – even unsophisticated algorithms such as *Random* and *Localized Round Robin* exhibited decision accuracies of greater than 50%. These relatively high figures can also be explained by the observation above that for significant periods during the execution of the applications, not enough instructions were present in the system to occupy all of the resources available. Under these circumstances, any of the idle nodes represents an equally good choice of destination, skewing the resulting decision accuracies to values that are higher than might be expected if the appearance of new instructions occurred more uniformly. The second conclusion is that a greater correlation was present between speedup and decision accuracy than had been the case when evaluating the load categorization algorithms. Although the difference in speedups between *Random Less Loaded* and *Least Loaded* was borne out by the decision accuracy values, *Mirror*, despite being the best performer in terms of decision accuracy, was outperformed in terms of speedup by *Random Less Loaded*. This observation reinforces the conclusion drawn above that decision accuracy, despite being a useful indicator, cannot be relied upon to rank the relative performance of load balancing algorithms in terms of speedup.

5.3   Framework Optimizations

Once the basic ARC load balancing framework was in place, the focus turned to the evaluation of the effectiveness of a number of potential optimizations. These were: ordering instructions in the FPGA instruction execution queue by operator to reduce execution time lost to reconfigurations, the incorporation of operator triviality information into the load metric and load categorization process and the reassignment of instructions with dual

implementations to available CPUs if the available FPGAs are overloaded. The work involved in the implementation and evaluation each optimization is considered in subsequent subsections.

5.3.1   FPGA Instruction Queue Ordering

The first optimization to be considered was the ordering of instructions in the FPGA instruction execution queue. As described in Section 2.8.3, the default queuing behaviour is a simple first in, first out (FIFO) scheme. The rationale behind this optimization is that the FIFO scheme can result in unnecessary reconfigurations if the same operator appears multiple times in the queue in non-consecutive instructions. As noted in Section 4.3.2, the cost of FPGA reconfigurations is significant – approximately 170 ms on the equipment to hand. By grouping together instructions with the same operator, the number of these unnecessary reconfigurations would be reduced (see Figure 5.9).

One potential drawback of this mechanism is that it could in theory lead to certain instructions being delayed indefinitely and this in turn could adversely affect overall completion time. For instance, in a highly speculative computation, newly uncovered speculative instructions could be continually placed ahead of other instructions on the application's critical path. In theory, this would not affect the ultimate result but could affect the time to that result. The simple nature of the graphs created by the application generation program preclude this behaviour, but developers of applications making heavy use of speculation should consider whether or not enabling the optimization is in their best interest.

The ARC runtime environment was modified to incorporate instruction grouping in the FPGA instruction execution queue. The existing FIFO enqueuing scheme was replaced with the following: when a new instruction arrives, if one or more instructions with the same operator as the new instruction are present in the queue, then the new instruction is enqueued immediately behind the last matching instruction. If the queue does not contain an instruction with an identical operator but the currently executing instruction's operator is identical, then the new instruction is placed at the head of the queue. If no matching instruction is found, then the new instruction is placed at the back of the queue.

An experiment was conducted in order to evaluate the extent, if any, to which FPGA instruction queue ordering improved execution performance. Ten application sets, each composed of five applications, were generated using the application generation program. The parameters passed to the application generation program specified that every primitive operator should be nontrivial and have implementations in both software and hardware. The mean operator cost was specified as 10^4 ms, with a mean FPGA speedup of 10. A standard deviation of 10% was used in both cases. The only paramater that varied across application

(a) FIFO
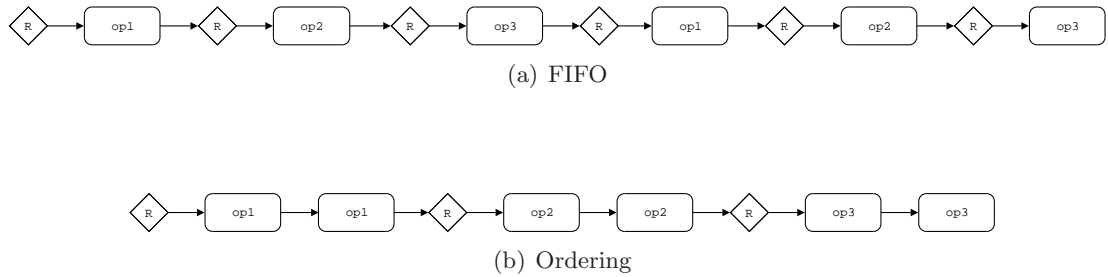
(b) Ordering

Figure 5.9: Graphical representation of an FPGA instruction queue showing how ordering by operator can significantly reduce the number of reconfigurations required. Implicit reconfigurations are denoted by diamonds.

sets was the number of primitive operators, specified explicitly (through the use of zero standard deviation values) in each case and varying from 1 to 10 across application sets. The result was a collection of applications that could be used to evaluate the effectiveness of the optimization across varying degrees of operator heterogeneity.

Before the experiment could commence, a suitable metric for measuring the effectiveness of the optimization had to be found. The ratio of reconfigurations performed to FPGA instructions executed (*reconfiguration ratio*), was chosen as it provides a value that is independent of the specific characteristics, such as instruction granularity, of the application sets and hardware used for evaluation. The ARC runtime environment was modified to record the required information in the form of a text file that is written on application termination.

The experiment was conducted by executing each of the applications twice on a single cluster node, once with the standard FIFO queuing scheme and once with the queue ordering optimization enabled. A single cluster node was used so that the conditions under which both executions of an application took place varied as little as possible. If multiple nodes had been used, then the non-deterministic nature of task delegation could have resulted in differing reconfiguration counts across identical application runs. The results were obtained by averaging the reconfiguration ratios observed for each application set using both queuing schemes and are shown in the form of a graph in Figure 5.10.

Examination of the results indicates that the optimization had a beneficial effect on every application set tested that contained more than one configuration. The reconfiguration ratio for both queuing schemes was close to zero when the number of configurations per application is one as these applications require only a single reconfiguration before the execution of the first instruction. As the need for subsequent reconfigurations does not arise, the choice of queuing scheme is moot for these applications. The ordering scheme performed

best compared to the FIFO scheme when the number of configurations per application was low, as the likelihood of instructions forming groups decreases with increasing operator heterogeneity. To illustrate this point, the ordering optimization led to reductions of 45% and 44% in the number of reconfigurations required when the number of configurations per application was two and three, respectively. In contrast, applications where the number of configurations per application was nine and ten recorded a drop of only 12% and 16% respectively. Nevertheless, the optimization was of benefit in every recorded case.

5.3.2   Incorporation of Triviality Information

Although instruction queue length provides an approximation of the workload of a processing resource in the absence of detailed cost information, the resulting estimates do not take account of the presence of trivial instructions. As noted in Section 4.1, trivial operations (such as integer additions) account for a significant proportion of the primitive operations found in real-world applications. Simple operations such as these are cheaper to execute than to delegate, and as such should always be executed locally if possible.

Although cost information in general is too difficult to incorporate into applications, the runtime environment can be notified of the presence of trivial operations with little effort on the part of the application developer – all that is required is the presence of an additional attribute in the `operator` elements of XML definition files. This information can
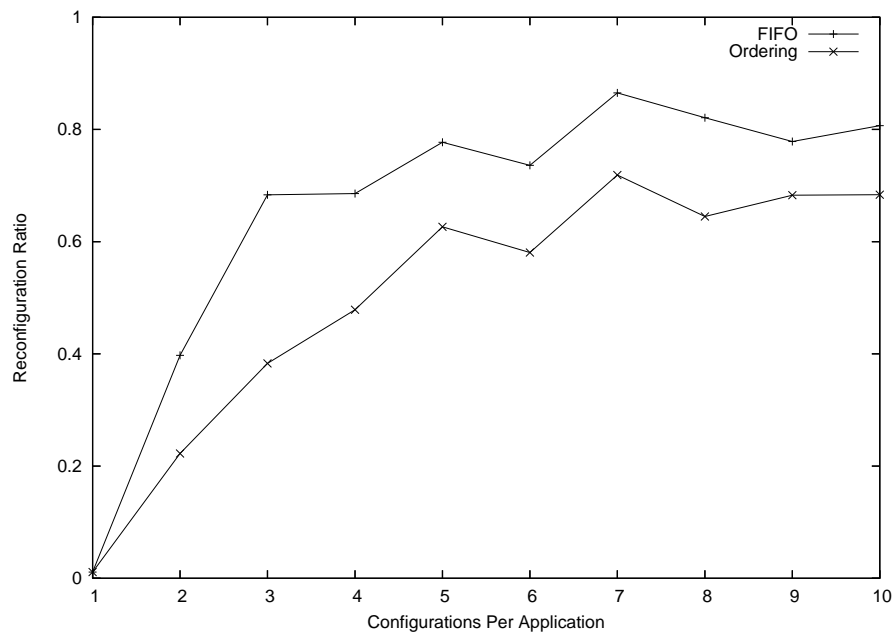
Figure 5.10: A graph illustrating the differences in reconfiguration ratios observed when the same application set was executed with both FIFO and Ordering FPGA queuing schemes.

then be included in the operator information table maintained by the runtime environment and used to improve the accuracy of load balancing decisions.

Two optimizations based on triviality information were considered: preventing the delegation of trivial instructions and the omission of trivial instructions when calculating CPU load. The reasoning behind keeping trivial instructions local was that although the ARC load balancing scheme is based upon the premise that instructions should execute where they can complete soonest, the act of submitting a trivial instruction for remote execution is more expensive than simply executing the instruction immediately. Furthermore, executing trivial instructions out of order in this fashion may allow dependent nodes to become fireable, creating new instructions that can potentially be assigned to idle resources. Trivial instructions should therefore be executed as soon as they enter the scheduler. The second optimization follows from the first; as trivial instructions are no longer enqueued on the native queue, they are not considered when calculating the CPU load. As trivial instructions by definition represent a very small amount of work, their absence from the instruction count should result in a more accurate picture of the true workload of CPUs.

Modifications were made to the application generation program, the ARC compile-time environment and the ARC runtime environment in order to evaluate the impact of incorporating triviality information. The application generation program was modified so that trivial operators are labelled as such through the use of a `triviality` attribute in `operator` elements. An example of a resulting operator declaration is presented in Figure 5.11 below. The runtime environment was modified so that the registration functions used by applications to impart operator information included triviality information. Suitable functions were added to make this information available to other ARC modules.

```
<operator category="primitive" name="op2"  cost="1" datasize="118"
         triviality="trivial">
    <typesig>Block,Block</typesig>
    <nativeimpl name="op2"/>
</operator>
```

Figure 5.11: An example of an operator generated by the application generation program that incorporates triviality information. Although the trivial nature of the operator could be determined in this case through examination of the `cost` attribute, the `cost` and `datasize` attributes are only available in generated, rather than real-world, applications.

An experiment was conducted in order to evaluate the effectiveness of incorporating triviality information into operator declarations. As the aim of identifying trivial instructions is to avoid unnecessary delegations and improve the accuracy of the load metric, it

follows that the optimization, if effective, should improve application performance. Decision accuracy was not considered to be a suitable metric for determining the impact of the optimization on performance. As noted above, the optimization is not intended to assign trivial instructions to where their execution will complete soonest, so the optimization could be expected to have an adverse effect on decision accuracy while improving performance. In any case, decision accuracy had earlier been found not to be directly related to application performance (see Section 5.2.4). In light of this, speedup was chosen as the most suitable performance metric.

Eleven application sets similar to those used to evaluate the load categorization algorithms in Section 5.2.4 were created for experimentation purposes. The distinct sets were created with differing values for the `trivialityProbability` parameter, with ten applications generated at intervals of 0.1 between zero and one inclusive. This allowed the impact of the optimization to be examined across applications composed of varying proportions of trivial instructions. Each application was executed three times; on a single machine so that the speedup could be calculated, across two machines with the optimization enabled and again across two machines with the optimization enabled. The average speedup attained for the applications at each interval value was then calculated for the runs across two machines. The result is shown in the form of a graph in Figure 5.12.

Examination of the graph reveals that the optimization had little impact when the
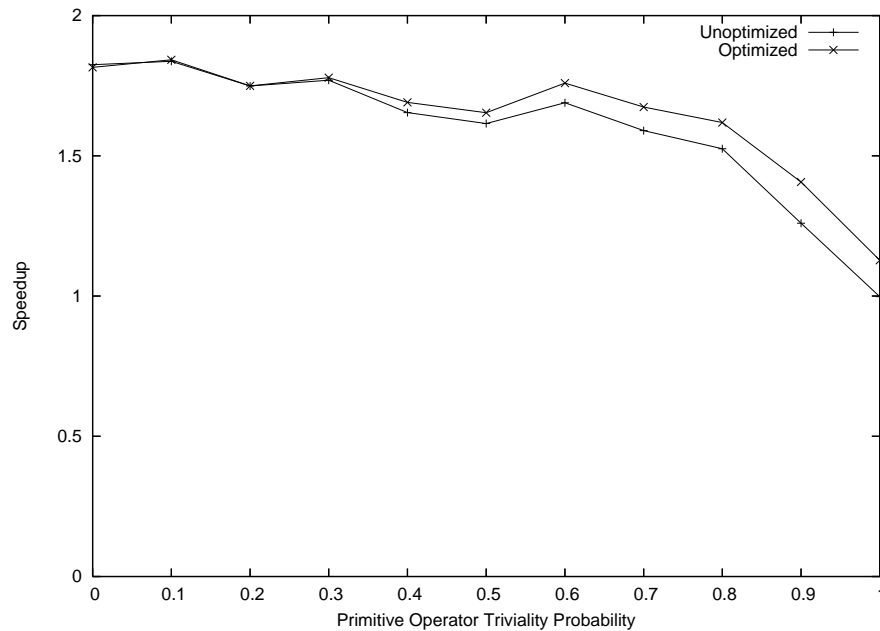
Figure 5.12: A graph illustrating the speedup observed when applications composed of varying proportions of trivial instructions were executed across two nodes both with the triviality optimization enabled and disabled.

proportion of trivial instructions was small, i.e., one third or less of the total number of instructions. However, a beneficial effect was observed with larger values of the `triviality-Probability` parameter. The extent to which the lines in the graph diverge increases with larger probability values. Although the impact on overall execution time was not particularly significant for any recorded value, it should be noted that the amount of operand data passed to and returned from the operations was deliberately kept small in this instance (the mean operand data size was 10 Bytes) to mimic the presence of the very simple operations like integer addition commonly found in real-world applications. Performing trivial operations on larger (in terms of memory) operand values would result in an increased transmission overhead and hence the impact of unnecessarily delegating trivial instructions would be more pronounced. Examples of such operations include determining the width and height of an image in an image-processing application.

5.3.3   Reassignment of Instructions with Dual Implementations

As noted above, the basic load balancing framework always assigns instructions with dual operator implementations to FPGAs. The reasoning behind this decision was that the effort required to create an FPGA implementation would suggest that the programmer would only do so in order to achieve a significant speedup. It can therefore be concluded that the assignment of an instruction with dual implementations to an FPGA is the correct decision in the majority of situations. However, in applications that make extensive use of FPGA operators, conditions may arise where a significant amount of work accumulates at the FPGA queues, with the CPU queues being comparatively underused. In situations such as this, performance benefits may be attained by assigning new instructions with dual implementations to CPUs rather than FPGAs.

Other scenarios can be envisaged where reassignment could lead to improved instruction completion times. For example, if the cost of performing a reconfiguration plus the FPGA execution cost of an instruction is greater than the CPU execution cost, then it may be worthwhile assigning the instruction in question to a CPU rather than an FPGA. Also, in an environment where communication times are non-uniform, a lower delegation overhead, and hence completion time, could be obtained by sending an instruction to a CPU available via a fast connection to an FPGA available via a slower one. However, these scenarios are not considered here; the first because of the absence of cost information in ARC, the second because a homogeneous networking environment is currently assumed by ARC.

In the absence of cost information, creating a suitable decision procedure for assigning dual-implementation instructions to processing resources represents a difficult undertaking. Simple threshold schemes would have to be fine-tuned on a per-application basis, and as

such would not represent a general solution. Instead, a scheme based on average FPGA speedup was considered.

Although FPGA speedup information is available in applications created by the application generation program, the operation of the scheme requires that the programmer supplies speedup information for every operator with a dual implementation. However, this requirement was not considered overly demanding as the hardware development process typically begins with a software implementation that is used for reference and validation purposes throughout the development process. The developer should therefore have a reasonable approximation of the speedup attained at the end of the development process, or at worst should be able to determine the speedup with little effort. The main drawback of this approach is the tacit assumption that FPGA speedups are similar across all possible inputs, which is unlikely to be the case for every application. If a significant variation in speedup across inputs is present, it then falls to the application developer to either disable the optimization (this can be achieved simply by omitting the native implementations of operators) or choosing the speedup value that is most representative of the expected inputs.

The development of the optimization began by considering the case where only a single primitive operator is used, this operator has implementations both in software and in hardware, all invocations of the operator share the same execution cost, the speedup attained through implementing the operator in hardware has been made available to the runtime environment, and the application is being executed on a single machine. Figure 5.13 illustrates the situation where the CPU queue is empty, the number of instructions awaiting execution on the FPGA queue is equal to the speedup, 5, associated with the lone operator (op0) and neither the CPU nor the FPGA is currently executing an instruction. At this point, assigning a newly created instruction (indicated by a dashed box) to the CPU rather than the FPGA results in a lower completion cost. Furthermore, this remains the case even if the instruction at the head of the FPGA queue is considered to be partially
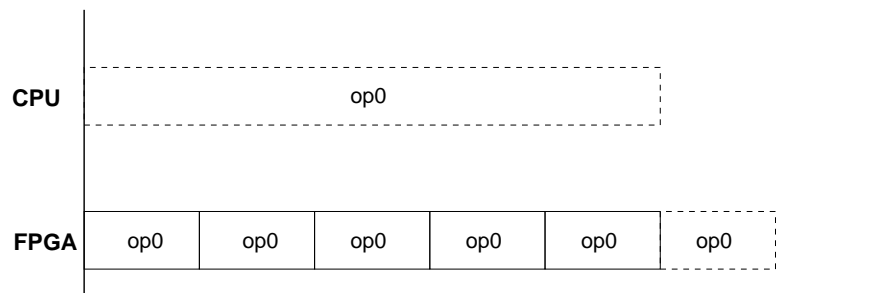
Figure 5.13: The speedup value associated a dual-implementation operator represents the threshold value beyond which a newly created instruction (indicated by a dashed box) should be assigned to the CPU in order to minimize the new instruction's completion cost.

executed rather than a member of the queue of unevaluated instructions, provided that no instruction is currently executing on the CPU. If the FPGA queue were shorter, then the lowest completion cost would be attained by assigning the new instruction to the FPGA. Similarly, if the FPGA queue were longer, then assignment to the CPU remains the best option, again assuming that the CPU is idle. The number of instructions equivalent to the FPGA speedup therefore represents the threshold value beyond which new instructions should be assigned to an idle CPU.

In the scenario presented above, an idle CPU can be considered alongside the FPGA when making load balancing decisions based on instruction queue lengths provided that the CPU is suitably penalized due to the greater execution cost it incurs by executing instructions with dual implementations. When the CPU is idle, this penalty is equivalent to the speedup of the sole operator. This scheme can be extended to consider the situation where instructions are present in the CPU queue, by adding an additional penalty for every instruction present. In general, when comparing the CPU queue to that of the FPGA, the adjusted length of the CPU queue (L'_{CPU}) is given by

$$L'_{CPU} = S(1 + L_{CPU})$$

where S is the FPGA speedup of the sole primitive operator and L_{CPU} is the CPU queue length, calculated by taking the length of the CPU instruction queue and incrementing it by one if an instruction is currently being executed.

Although this method of adjusting CPU load provides an accurate method of comparing the loads of both types of processing resource within the restricted conditions described above, several issues arise when attempting to apply the same principle to create a general-purpose optimization. The most obvious is that primitive instructions with differing operators may be present in either queue, so the notion of using the speedup of a single operator for CPU load adjustment is no longer applicable. However, this issue can be worked around by calculating the average speedup and using the resulting value when adjusting the CPU load. Unfortunately, this modification results in an unavoidable loss of accuracy, the extent of which depends on the variance in FPGA speedups. A more serious issue is the presence of instructions with single implementations in both instruction queues, as the cost of these instructions is unknown and leads to a greater divergence between the load values being used for comparison and the true load for each resource. For example, the presence of an instruction with a condensed graph operator, and hence a comparatively low execution cost, in the CPU instruction queue would be considered to be of equal importance to a more expensive primitive operation when calculating the adjusted CPU load, resulting in an overestimation of the true load of the CPU. This situation is further exacerbated when

operators with dual implementations but variable execution cost are considered.

The issues described above are unavoidable in the absence of cost information, and are applicable to load balancing in general as well as the specific optimization considered here. However, their presence calls in to question the extent to which the optimization can be successfully applied to real-world applications. To begin with, the optimization only modifies the existing behaviour when an instruction with dual implementations happens to arrive when a significant divergence has arisen between the lengths of the CPU and FPGA instruction queues. Given the relative rarity with which the optimization is invoked, it is crucial that the optimization is effective in the majority of invocations if it is to have a significant positive impact on the running times of applications. This was a question that could only be resolved through experimentation.

The ARC runtime environment was modified to facilitate the experiment by altering the scheduler to assign instructions with dual implementations to the CPU when the threshold described above was exceeded. It was decided that the optimization should initially be evaluated under the most amenable conditions possible. To this end, application sets similar to the application depicted in Figure 5.13 were generated by restricting the number of primitive operators per application to one, with the lone primitive operator having dual implementations. However, the queues created by these applications differ from the original scenario in that condensed graph instructions may be present in the native queue. Relatively
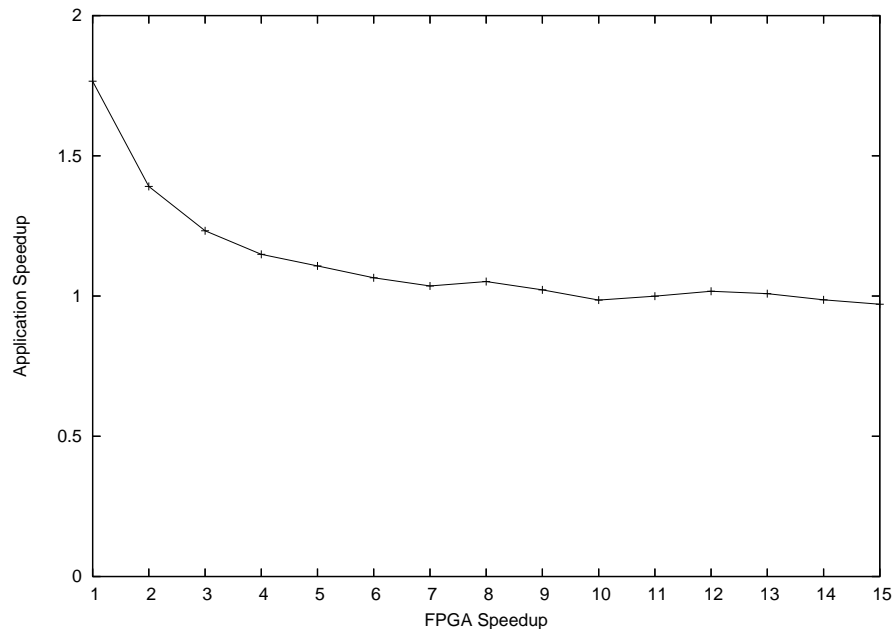
Figure 5.14: A graph illustrating the speedup observed when applications containing a single primitive operator with dual implementations were executed with the reassignment optimization enabled.

high values for the `nodesPerGraphMean` and `nodeParallelProbability` parameters were used to allow for the accumulation of significant numbers of instructions in the instruction queues. Fifteen application sets were generated, with FPGA speedup values varying between one ten across the sets. Each set was composed of ten applications. These applications were executed on a single machine, and the speedups observed when the optimization was enabled compared to when the optimization was disabled were recorded. The results are shown as a plot in Figure 5.14.

Examination of the results reveals significant speed increases for low FPGA speedup values, with a mean speedup of over 1.75 observed when the FPGA speedup was one, i.e., instructions took equally long to execute on the CPU as on the FPGA. However, the speedup attained tails off sharply, with very little difference in execution times observed for FPGA speedup values greater than five. A slight reduction in execution time was observed in some cases. The sharp dropoff in speedup can be explained by the increasing rarity with which the optimization is invoked with increasing FPGA speedup, as well as the diminishing amount of execution time saved per invocation. The slight slowdown observed in some cases can be explained by the presence of condensed graph instructions in the native queue, which would distort the true CPU workload. Also, reassigning work to the native queue may delay the evaluation of condensed graph instructions that subsequently arrive, delaying the exposition of further parallelism and potentially leaving resources idle.
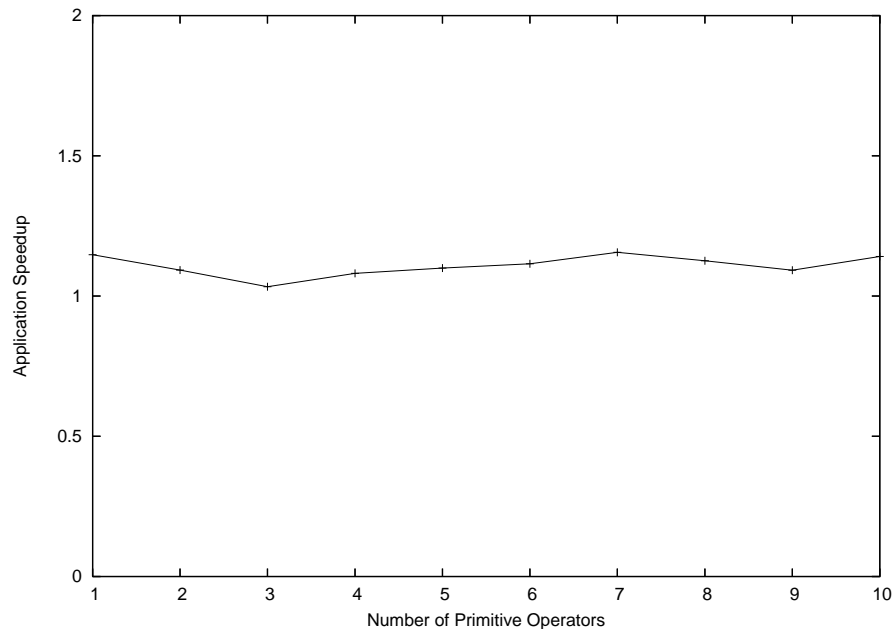
Figure 5.15: A graph illustrating the speedup observed when applications containing a varying numbers of primitive operators with dual implementations were executed with the reassignment optimization enabled compared to when the optimization was disabled.

Next, it was decided that the performance of the optimization should be examined under more adverse circumstances closer to those found in real-world applications. For this experiment, ten application sets with varying numbers of primitive operators were generated. All primitive operators were created with dual implementations. The `operationCostMean` and `operationCostSigma` parameters were specified as 1,000 and 500, respectively, resulting in a broad range of operation costs. Similarly, the values used for `fpgaSpeedupMean` and `fpgaSpeedupSigma` were 10 and 5, respectively, resulting in a broad range of FPGA speedups. Therefore, as the number of primitive operators per application set increased, more variety was introduced into the characteristics of the instructions created at runtime. These applications were executed with the optimization both enabled and disabled, and the resulting speedups are shown as a plot in Figure 5.15.

Examination of the results reveals that a slight speedup was observed in every instance, and that the extent of the speedup varies little with increasing numbers of primitive operators. These observations are somewhat surprising in light of the occasional slowdowns observed in the previous experiment. However, the results speak for themselves, and it can be concluded that the optimization performs better in real-world situations compared to the contrived scenario presented originally. It was therefore decided to include the optimization in ARC and have it enabled by default.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

Chapter 1 illustrated that Cluster Computing and Reconfigurable Computing are complementary technologies, as Cluster Computing is most suited to the execution of coarse-grained tasks, while Reconfigurable Computing is able to take advantage of localized, fine-grained parallelism. Applications developed for clusters augmented with reconfigurable hardware can therefore play to the strengths of both techniques by assigning coarse-grained tasks to cluster nodes where the fine-grained parallelism present within the tasks can then be exploited using FPGAs.

On the other hand, both techniques also suffer from inherent limitations. The classes of applications that can be accelerated effectively on clusters is limited by the latency and bandwidth of commodity networking equipment. Similarly, the performance gains achievable using FPGAs are constrained by limited throughput and poor floating-point performance. Furthermore, accelerating applications using either technique represents a more difficult and complex undertaking than developing the equivalent sequential program, although the amount of effort required to produce FPGA configurations has declined in recent years with the advent of higher-level languages such as Handel-C.

Applications developed for clusters augmented with reconfigurable hardware to date have typically used a combination of the most popular methodologies in each field. Sequential languages are used in conjunction with message passing libraries to expose coarse-grained parallel tasks for execution on cluster nodes. Hardware description languages such as VHDL are used to accelerate these tasks using reconfigurable hardware, with communications between the host and FPGA performed using calls to the FPGA driver. This technique requires that all communications be specified explicitly. Accounting for task or hardware heterogeneity results in complex applications that are difficult to develop and

maintain.

The ARC metacomputing system, described in Chapter 2, offers an alternative approach. When developing applications using ARC, the specification of the high-level parallelism present in the application, the implementation of the application logic and the acceleration of portions of the application using reconfigurable hardware, are performed separately. Communications are specified implicitly, and performed by the ARC runtime environment without any intervention on the part of the developer. This arrangement allows the developer to focus on each aspect of the application in isolation, avoiding the clutter in application logic that results from the current approach.

Furthermore, the ARC runtime environment implements complex load balancing behaviour, as described in Chapter 5. The load balancing functionality can be exploited effectively with little or no effort on the part of the application developer. This eliminates the need for developers to implement load balancing on a per-application basis, significantly reducing the effort required to efficiently utilize clusters augmented with reconfigurable hardware.

The work presented in this thesis represents a concrete contribution to the state of the art in developing for clusters augmented with reconfigurable hardware. It is hoped that the availability of ARC and other high-level computing environments will increase the acceptability, and hence popularity, of the technique. Some potential future trends in the field are discussed below.

6.1.1   Future Prospects for the Field

Given the ever-increasing popularity of cluster computing the probable continuation of the factors underpinning its growth (primarily the price/performance ratio), its future as the primary High Performance Computing technique seems to be assured. The future popularity of Reconfigurable Computing, and hence the future popularity of clusters augmented with reconfigurable hardware, is more difficult to predict.

The RC5 application presented in Chapter 3 demonstrates that the results obtained through the use of reconfigurable hardware when an application contains suitable fine-grained parallelism can be impressive. The increasing density of new models of FPGAs means that Reconfigurable Computing will allow more complex applications to be implemented in hardware in the future, while existing applications will be able to incorporate more parallel branches and pipeline stages into their implementations. The possible incorporation of floating point units, wider data paths and more tightly-coupled connections into future commodity reconfigurable hardware would address many of the shortcomings currently affecting the technique. The recent appearance of FPGA boards that plug directly

into unused processor slots on AMD Opteron motherboards (see [48]) demonstrates that progress is being made on the problem of insufficiently tight coupling.

A significant factor working against the future popularity of Reconfigurable Computing is the introduction of SIMD vector processing extensions [143], such as SSE (Streaming SIMD Extensions) and AltiVec, to commodity CPUs. These extensions are designed to allow very fine-grained parallelism to be exploited, eroding the niche currently occupied by Reconfigurable Computing. Crucially, these extensions are tightly coupled with the CPU and can operate effectively on floating point data, exposing some of the weaknesses currently associated with Reconfigurable Computing. Furthermore, commodity FPGAs are developed primarily for ASIC prototyping, and the improvements necessary to increase performance for High Performance Computing purposes into the future might not coincide with the requirements of this market.

6.2   Future Work

A number of improvements, principally related to usability, could be made to the ARC System. At present, the declaration of every user-defined datatype requires the implementation of functions to perform serialization, deserialization and string conversion on instances of the type. The addition of a polymorphic type system [144] would allow all new types to be specified as aliases or aggregations of existing types. Serialization, deserialization and string conversion could then be performed automatically using predefined behaviour for each of the constituent basic types, obviating the need for custom functions to be specified.

A related issue is that of memory management. At present, the programmer is responsible for freeing any memory allocated by operator implementations, adding complexity to the graph definition process. Modifications to ARC would allow the runtime environment to take responsibility for deallocating heap memory referenced by operands. This functionality could be implemented either by using reference counting or the integration of a garbage collector such as BoehmGC [145], and would be simplified by the addition of the type system described above.

The high-level behaviour of ARC applications is specified using manually created XML definition documents. A number of alternative methods of producing definition documents could be examined. For example, a dedicated language would serve as a more concise means of specifying the required behaviour. Suitable language constructs would also increase the accessibility of the more advanced, but often underused, aspects of the Condensed Graphs model such as speculative execution and lazy evaluation. An alternative approach would be the development of an integrated development environment (IDE), similar to that developed for WebCom, that would allow ARC applications to be specified in an interactive, graphical

fashion.

The performance model described in Chapter 4 could be extended to consider more aspects of the behaviour of the ARC runtime environment. The process of transferring data to and from FPGAs is not considered at present; modelling this behaviour would allow the performance of various FPGA connection topologies to be evaluated, as well as providing a means of determining the potential speedups achievable for applications before implementation. The model could also be extended to consider the amount of memory used at each cluster node based on the amount of data associated with the instructions present. This extension would facilitate the development of load balancing algorithms, described below, that take account of the memory utilization of cluster nodes.

The load balancing framework and algorithms described in Chapter 5 considered the movement of instructions with the sole aim of minimizing completion cost. Future work in this area could examine the possibility of delegating instructions with a view to evenly distributing memory usage as well as workload. For example, in cases where the remote processor selection algorithm has a number of peers to choose from, the instruction could be sent to the machine with the most free memory. This approach would be particularly effective for instructions with condensed graph operators, as these tend to be memory intensive rather than CPU intensive.

6.3   Afterword

The chameleon-like ability of FPGAs to alter the form that they present to the outside world inspires fascination beyond their interest-worthiness as mere functional devices. The nature of their operation forces us to rethink our mental characterizations of various aspects of computing. Consider the intuitive notion that most people would associate with the word "software": machine code for a von Neumann architecture that is loaded into RAM before execution. Similarly, most people would have a preconceived notion of what is meant by the term "hardware design": a form of blueprint for use in the manufacture of ASICs. However, the use of hardware designs as FPGA configurations blurs the distinction between the two. Clearly, the hardware design could be said to "execute", software-like, when loaded onto an FPGA. However, the same configuration could serve, without modification, as a physical design etched into silicon. The only way to resolve the resulting dilemma is to broaden our perception of what constitutes software and what is represented by a hardware design.

The nature of the operation of FPGAs can therefore lead to a paradigm shift in the way we think about hardware and software. By blurring the distinction between hardware and software, and thinking of each as a form of the other, questions worthy of further investigation can be posed and opportunities worthy of pursuit can be discovered. An

immediate result is that we gain flexibility and the ability to accelerate software applications by converting parts of them to hardware designs and executing these designs on FPGAs. The work presented in this thesis takes this idea one step further by providing a means of exploiting parallelism at the cluster level as well as the FPGA level. However, it is clear that there is plenty of scope for further innovation in the field when one considers concepts such as partial reconfiguration, self-reconfiguration, the evolution of hardware designs and FPGAs with direct network connections, and ways in which these concepts could be developed or combined.

Bibliography

[1] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[2] Rajkumar Buyya, editor. *High Performance Cluster Computing*, volume 1. Prentice Hall, 1999.

[3] T. Sterling, D. Savarese, D. J. Becker, J. E. Dorband, U. A. Ranawake, and C. V. Packer. BEOWULF: A parallel workstation for scientific computation. In *Proceedings of the 24th International Conference on Parallel Processing*, pages I:11–14, Oconomowoc, WI, 1995.

[4] Thomas E. Anderson, David E. Culler, David A. Patterson, and The NOW Team. The case for NOW (Networks of Workstations). *IEEE Micro*, 15(1):54–64, February 1995.

[5] Andreas Boklund, Christian Jiresjo, and Stefon Mankefors. The story behind Midnight, a part time high performance cluster. In *Proceedings of The 2003 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'03)*, Las Vegas, Nevada, June 2003.

[6] Rajkumar Buyya, editor. *High Performance Cluster Computing: Programming and Application Issues*, volume 2. Prentice Hall, 1999.

[7] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29–36, February 1995.

[8] Simon K. Warfield, Ferenc A. Jolesz, and Ron Kikinis. Real-time image segmentation for image-guided surgery. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, pages 1–7, San Jose, California, 1998.

[9] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Mancheck, and V. Sunderam. *PVM: Parallel Virtual Machine*. MIT Press, 1994.

[10] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface.* MIT Press, 1994.

[11] Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, 1996.

[12] Lei Huang, Barbara Chapman, and Ricky Kendall. OpenMP for clusters. In *Proceedings of the Fifth European Workshop on OpenMP (EWOMP'03)*, Aachen, Germany, September 2003.

[13] Nicholas Carriero and David Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.

[14] William W. Carlson, Jesse M. Draper, David E. Culler, Kathy Yellick, Eugene Brooks, and Karen Warren. Introduction to UPC and language specification. Technical report, IDA Center for Computing, 1999.

[15] High Performance Fortran Forum. High Performance Fortran language specification, version 1.0. Technical report, Houston, Texas, 1993.

[16] Laxmikant V. Kale and Sanjeev Krishnan. CHARM++: A portable concurrent object oriented system based on C++. In *Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 91–108, September 1993.

[17] Jon Kerridge. *Occam Programming: A Practical Approach.* Alfred Waller, August 1987.

[18] Jean-Luc Gaudiot, Tom DeBoni, John Feo, Wim Böhm, Walid Najjar, and Patrick Miller. The Sisal model of functional programming and its implementation. In *Proceedings of the 2nd International Symposium on Parallel Algorithms/Architectural Synthesis (pAs '97)*, pages 112–123, Aizu-Wakamatsu, Fukushima, Japan, March 1997.

[19] M.W. Hall, J.M. Anderson, S.P. Amarasinghe, B.R. Murphy, and S.-W. Liao. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, 29(12):84–89, 1996.

[20] Simon Thompson. *Haskell: The Craft of Functional Programming.* Addison Wesley, July 2003.

[21] Mark P. Jones and Paul Hudak. Implicit and explicit parallel programming in Haskell. Technical report, Yale University, August 1993.

[22] Amnon Barak and Oren La'adan. The MOSIX multicomputer operating system for high performance cluster computing. *Future Generation Computing Systems*, 13(4–5):361–372, 1998.

[23] Christine Morin, Pascal Gallard, Renaud Lottiaux, and Geoffroy Vallée. Towards an efficient single system image cluster operating system. *Future Generation Computing Systems*, 20(4):505–521, May 2004.

[24] A. Acharya, G. Edjlali, and J. Saltz. The utility of exploiting idle workstations for parallel computation. In *SIGMETRICS'97*, pages 225–236, Seattle, WA, May 1997.

[25] M. Litzkow, M. Livny, and M. Mutka. Condor - a hunter of idle workstations. In *Proceedings of the International Conference on Distributed Computing Systems*, pages 104–111, June 1998.

[26] John P. Morrison, Keith Power, and Neil Cafferkey. Cyclone: A cycle brokering system to harvest wasted processor cycles. In *Proceedings of The 2000 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, Las Vegas, Nevada, June 2000.

[27] Larry Smarr and Charles E. Catlett. Metacomputing. *Communications of the ACM*, 35(6):44–52, 1992.

[28] I. Foster and C. Kesselman. The Globus Project, a status report. In *Proceedings of the IPPS/SPDP '98 Heterogeneous Computing Workshop*, pages 4–18, 1998.

[29] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1998.

[30] A. Grimshaw and W. Wulf *et al*. The Legion vision of a worldwide virtual computer. *Communications of the ACM*, 40(1), January 1997.

[31] John Morrison, James Kennedy, and David Power. WebCom: A web based volunteer computer. *Journal of Supercomputing*, 18:47–61, 2001.

[32] John Morrison, Brian Clayton, David Power, and Adarsh Patil. WebCom-G: Grid enabled metacomputing. *The Journal of Neural, Parallel and Scientific Computation, Special Issue on Grid Computing*, 2004.

[33] Reconfigurable computing glossary. *EE Times*, 12 September 2002.

[34] Richard Clayton and Mike Bond. Experience using a low-cost FPGA design to crack DES keys. In *Proceedings of the Workshop on Cryptographic Hardware and Embedded Systems 2002 (CHES 2002)*, Redwood City, California, August 2002.

[35] Steffen Klupsch, Markus Ernst, Sorin A. Huss, Martin Rumpf, and Robert Strzodka. Real time image processing based on reconfigurable hardware acceleration. In *Proceedings of the IEEE Workshop on Heterogeneous Reconfigurable Systems-on-Chip*, Hamburg, Germany, 2002.

[36] W. Huang, N. Saxena, and E. McCluskey. A reliable LZ data compressor on reconfigurable coprocessors. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 249–258, 2000.

[37] Max van Daalen, Peter Jeavons, and John Shawe-Taylor. A stochastic neural architecture that exploits dynamically reconfigurable FPGAs. In *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 202–211, Los Alamitos, California, 1993.

[38] Thomas L. Floyd. *Digital Fundamentals*. Prentice Hall, 2002.

[39] Michael Barr. Programmable Logic: What's it to ya? *Embedded Systems Programming*, pages 75–84, June 1999.

[40] George Milne. Reconfigurable custom computing as a supercomputer replacement. In *4th International Conference on High Performance Computing*, Bangalore, India, December 1997.

[41] Katherine Compton and Scott Hauck. Reconfigurable computing: A survey of systems and software. *ACM Computing Surveys*, 34(2):171–210, June 2002.

[42] P. Kung. *Obtaining Performance and Programmability Using Reconfigurable Hardware for Media Processing*. PhD thesis, Massachussets Institute of Technology, 2002.

[43] Robert J. Drost, Robert David Hopkins, and Ivan E. Sutherland. Proximity communication. In *Proceedings of the Custom Integrated Circuits Conference (CICC)*, pages 469–472, San Jose, California, September 2003.

[44] ITRS. International Technology Roadmap for Semiconductors Executive Summary, 2003.

[45] John P. Morrison, Padraig O'Dowd, and **Philip D. Healy**. An investigation of the applicability of distributed FPGAs to high-performance computing. In *High Performance Computing: Paradigm and Infrastructure*, pages 277–294. John Wiley & Sons, 2005.

[46] S. Hauck, T. W. Fry, M. M. Hosler, and J.P. Kao. The Chimaera reconfigurable functional unit. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 87–96, 1997.

[47] Takashi Miyamori and Kunle Olukotun. REMARC: Reconfigurable multimedia array coprocessor. In *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*, page 261, Monterey, Claifornia, 1998.

[48] DRC Computer Corporation. DRC coprocessor module datasheet, 2006.

[49] P. Leong, M. Leong, O. Cheung, T. Tung, C. Kwok, M. Wong, and K. Lee. Pilchard - a reconfigurable computing platform with memory slot interface. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April 2001.

[50] D. Landis, L. Roth, P. Hulina, L. Coraor, and S. Deno. Evaluation of computing in memory architectures for digital image processing applications. In *Proceedings of the 1999 Intl. Conference on Computer Design*, pages 146–151, October 1999.

[51] K. Underwood, R. Sass, and W. Ligon. A reconfigurable extension to the network interface of beowulf clusters. In *Proceedings of IEEE Cluster Computing*, Newport Beach, CA, October 2001.

[52] QinetiQ Ltd. Quixilica packet-switched communications library for heterogeneous systems, December 2002.

[53] Reiner Hartenstein. A decade of reconfigurable computing: A visionary retrospective. In *Proceedings of the conference on Design, automation and test in Europe*, pages 642–649, Munich, Germany, 2001.

[54] Bozidar Radunovic and Veljko M. Milutinovic. A survey of reconfigurable computing architectures. In *Proceedings of the 8th International Workshop on Field-Programmable Logic and Applications (FPL '98)*, pages 376–385, 1998.

[55] *EDIF Electronic Design Interchange Format Version 200*. Electronic Industries Association, June 1989.

[56] Peter J. Ashenden. *The Designer's Guide to VHDL, 2nd Edition*. Morgan Kaufmann, May 2001.

[57] Philip R. Moorby and Donald E. Thomas. *The Verilog Hardware Description Language*. Kluwer Academic Publishers, May 1998.

[58] David Pellerin and Michael Holley. *Digital Design using ABEL*. Prentice Hall, 1994.

[59] Steven A. Guccione, Delon Levi, and Prasanna Sundararajan. JBits: A Java-based interface for reconfigurable computing. In *2nd Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference (MAPLD)*, September 1999.

[60] Oskar Mencer, Martin Morf, and Michael J. Flynn. PAM-Blox: High performance FPGA design for adaptive computing. In *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*, Napa Valley, California, 1998.

[61] Per Haglund, Oskar Mencer, Wayne Luk, and Benjamin Tai. PyHDL:hardware scripting with Python. In *Engineering of Reconfigurable Systems and Architectures*, Las Vegas, Nevada, June 2003.

[62] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: Hardware design in Haskell. In *International Conference on Functional Programming*, pages 174–184, 1998.

[63] Xilinx Inc. Xilinx system generator v6.2 user guide, 2004.

[64] Star Bridge Systems Inc. Viva 2.4 user guide, 2004.

[65] Celoxica Ltd. *Handel-C Language Reference Manual Version 3.1*, 2002.

[66] Thorsten Grotker, Stan Liao, Grant Martin, and Stuart Swan. *System Design with SystemC*. Kluwer Academic Publishers, May 2002.

[67] D. Galloway. The Transmogrifier C hardware description language and compiler for FPGAs. In *Proceedings of the IEEE Symosium on Field-Programmable Custom Computing Machines (FCCM)*, pages 136–144, Napa Valley, California, 1995.

[68] Roger M.A. Peel and Barry M. Cook. Occam on field-programmable gate arrays: Fast prototyping of parallel embedded systems. In *Proceedings of The 2000 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'00)*, pages 2523–2529, Las Vegas, Nevada, June 2000.

[69] José-Luis Llopis and Bernard Pottier. Smalltalk blocks revisited, a logic generator for FPGAs. In *Proceedings of the IEEE Symosium on Field-Programmable Custom Computing Machines (FCCM)*, Napa Valley, California, 1996.

[70] João M. P. Cardoso and Horácio C. Neto. Macro-based hardware compilation of Java^{TM}bytecodes into a dynamic reconfigurable computing system. In *Proceedings of the 7th IEEE Symosium on Field-Programmable Custom Computing Machines (FCCM '99)*, pages 2–11, Napa Valley, California, April 1999.

[71] M. B. Gokhale and J. M. Stone. NAPA C: Compiling for hybrid RISC/FPGA architecture. In *Proceedings of the 6th IEEE Symosium on Field-Programmable Custom Computing Machines (FCCM '98)*, pages 63–69, Napa Valley, California, 1998.

[72] Adrian Thompson. *Hardware Evolution: Automatic Design of Electronic Circuits in Reconfigurable Hardware by Artificial Evolution*. Distinguished Dissertation Series. Springer-Verlag, 1998.

[73] Melissa C. Smith, Steven L. Drager, Lt. Louis Pochet, and Gregory D. Peterson. High performance reconfigurable computing systems. In *Proceedings of 2001 IEEE Midwest Symposium on Circuits and Systems*, Fairborn, Ohio, August 2001.

[74] Ron Sass, Keith Underwood, and Walter Ligon. Design of an adaptable computing cluster. Technical report, Department of Electrical and Computer Engineering, Clemson University, South Carolina.

[75] Thomas Lehmann and Andreas Schreckenberg. *Case Study of Integration of Reconfigurable Logic as a Coprocessor into a SCI-Cluster under RT-Linux*, volume 2147 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.

[76] Jason P. Clifford and Steven J.E. Wilton. Architecture of cluster-based FPGAs with memory. In *Proceedings of the Custom Integrated Circuits Conference (CICC) 2000*, pages 131–134, Orlando, Florida, 2000.

[77] Chrilly Donninger, Alex Kure, and Ulf Lorenz. Parallel Brutus: The first distributed, FPGA accelerated chess program. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS'04)*, Santa Fe, New Mexico, 2004.

[78] Ron Sass, Keith Underwood, and Walter Ligon. Acceleration of 2D-FFT on an adaptable computing cluster. In *Proceedings of the IEEE Symosium on Field-Programmable Custom Computing Machines (FCCM)*, Napa Valley, California, 2001.

[79] G. Grastveit, H. Helstrup, V. Lindenstruth, C. Loizides, D. Roehrich, B. Skaali, T. Steinbeck, R. Stock, H. Tilsner, K. Ullaland, A. Vestbo, and T. Vik. FPGA coprocessor for the ALICE high level trigger. In *Proceedings of CHEP03*, La Jolla, California, March 2003.

[80] V. Aggarwal, I. Troxel, and A. George. Design and analysis of parallel N-Queens on reconfigurable hardware with Handel-C and MPI. In *Proceedings of the 7th International Conference on Military and Aerospace Programmable Logic Devices (MAPLD)*, Washington, DC, September 2004.

[81] Mark Jones, Luke Scharf, Jonathan Scott, Chris Twaddle, Matthew Yaconis, Kuan Yao, Peter Athanas, and Brian Schott. Implementing an API for distributed adaptive computing systems. In *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Napa, CA, April 1999.

[82] Graig Ulmer and Sudhakar Yalamanchili. An extensible message layer for high-performance clusters. In *Proceedings of the 2000 International Conference on Parallel and Distributed Processing Techniques (PDPTA'00)*, Las Vegas, Nevada, June 2000.

[83] Kris Gaj, Tarek El-Ghazawi, Nikitas Alexandridis, Jacek R. Radzikowski, Mohamed Taher, and Frederic Vroman. Effective utilization and reconfiguration of distributed hardware resources using job management systems. In *Proceedings of the Reconfigurable Architecture Workshop 2003*, Nice, France, April 2003.

[84] P. Banerjee, N. Shenoy, A. Choudhary, S. Hauck, C. Bachmann, M. Haldar, P. Joisha, A. Jones, A. Kanhare, A. Nayak, S. Periyacheri, M. Walkden, and D. Zaretsky. A MATLAB compiler for distributed, heterogeneous, reconfigurable computing systems. In *Proceedings of the 8th International Symposium on FPGA Custom Computing Machines (FCCM'00)*, April 2000.

[85] Mark T. Jones, Michael A. Langston, and Padma Raghavan. Tools for mapping applications to CCMs. In *Photonics East'98*, Boston, Massachusetts, November 1998.

[86] James B. Peterson and Peter M. Athanas. Resource pools: an abstraction for configurable computing co-design. In *High-Speed Computing, Digital Signal Processing, and Filtering Using Reconfigurable Logic, Proc. SPIE 2914*, pages 218–224, Bellingham, Washington, 1996.

[87] Ian Troxel, Aju Jacob, Alan George, Raj Subramaniyan, and Matthew Radlinski. CARMA: A comprehensive management framework for high-performance reconfigurable computing. In *Proceedings of the 7th International Conference on Military and Aerospace Programmable Logic (MAPLD)*, Washington, D.C., September 2004.

[88] Aju Jacob, Ian Troxel, and Alan George. Distributed configuration management for reconfigurable cluster computing. In *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Architectures (ERSA)*, Las Vegas, Nevada, June 2004.

[89] Ryan DeVille, Ian Troxel, and Alan George. Performance monitoring for run-time management of reconfigurable devices. In *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Architectures (ERSA)*, Las Vegas, Nevada, June 2005.

[90] Melissa C. Smith and Gregory D. Peterson. Analytical modeling for high performance reconfigurable computers. In *Proceedings of the SCS International Symposium on Performance Evaluation of Computer and Telecommunications Systems*, July 2002.

[91] Ron Sass, Keith Underwood, and Walter Ligon. Cost effectiveness of an adaptable computing cluster. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing*, pages 54–66, Denver, Colorado, 2001.

[92] Shinichi Yamagiwa, Masaaki Ono, Takeshi Yamazaki, Pusit Kulkasem, Masayuki Hirota, and Koichi Wada. *Maestro-Link: A High Performance Interconnect for PC Cluster*, volume 1482 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[93] Marek Tudruj and Lukasz Masko. Dynamic SMP clusters with communication on the fly in NoC technology for very fine grain computations. In *Proceedings of the 3rd International Symposium on Parallel and Distributed Computing (ISPDC 2004)*, pages 97–104, Cork, Ireland, July 2004.

[94] Padraig J. O'Dowd, **Philip D. Healy**, and John P. Morrison. A condensed graphs microprocessor to drive parallel computing. In *Proceedings of the 2005 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'05)*, Las Vegas, Nevada, June 2005.

[95] Matthew Smith, Bernd Klose, Thomas Friese, Ralf Ewerth, and Bernd Freisleben. *Towards High Performance Grid Services Using Reconfigurable Hardware*. Lecture Notes in Computer Science - High Performance Computing (Submitted). Springer-Verlag, 2005.

[96] Gene M. Amdahl. Validity of the single processor approach to achieving large-scale computing capabilities. In *Proceedings of the 1967 AFIPS Conference*, volume 30, pages 483–485, 1967.

[97] John L. Gustafson. Reevaluating Amdahl's Law. *Communications of the ACM*, 31(5):532–533, 1988.

[98] John P. Morrison, **Philip D. Healy**, and Padraig J. O'Dowd. Architecture and implementation of a distributed reconfigurable metacomputer. In *Proceedings of the*

2nd *International Symposium on Parallel and Distributed Computing (ISPDC 2003)*, pages 153–158, Ljubljana, Slovenia, October 2003.

[99] P.J. Moylan. The case against C. Technical report, Department of Electrical and Computer Engineering, University of Newcastle, Australia, July 1992.

[100] Micah Beck and Keshav Pingali. From control flow to dataflow. *Journal of Parallel and Distributed Computing*, 12, 1991.

[101] Kurt Wall and Willian Von Hagen. *The Definitive Guide to GCC*. APress, February 2004.

[102] Intel Corporation. *Intel® C++ Compiler for Linux Reference*, 2005.

[103] John P. Morrison. *Condensed Graphs: Unifying Availability-Driven, Coercion-Driven and Control-Driven Computing*. PhD thesis, Technische Universiteit Eindhoven, 1996.

[104] J.R. Gurd, C. C. Kirkham, and I. Watson. The Manchester prototype dataflow computer. *Communications of the ACM*, 28(1):34–52, January 1985.

[105] Tim Bray, Jean Paoli, C.M. Sperberg-McQueen, and Eve Maler. *Extensible Markup Language (XML) 1.0 (Second Edition)*. W3C Recommendation, October 2000.

[106] Scott Means and Michael Bodie. *The Book of SAX: The Simple API for XML*. No Starch Press, June 2002.

[107] Joe Marini. *The Document Object Model: Processing Structured Documents*. Osborne/McGraw-Hill, July 2002.

[108] Hartmut Liefke and Dan Suciu. XMill: An efficient compressor for XML data. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 153–164, Dallas, Texas, May 2000.

[109] James Clark, editor. *XSL Transformations (XSLT) Version 1.0*. W3C Recommendation, November 1999.

[110] John P. Morrison and **Philip D. Healy**. Implementing the WebCom 2 distributed computing platform with XML. In *Proceedings of the 1st International Symposium on Parallel and Distributed Computing (ISPDC 2002)*, pages 171–179, Iaşi, Romania, July 2002.

[111] John P. Morrison, **Philip D. Healy**, David A. Power, and Keith J. Power. The role of XML within the WebCom metacomputing platform. *Scalable Computing: Practice and Experience*, 6(1):33–43, March 2005.

[112] Mark Lutz. *Programming Python, Second Edition*. O'Reilly, March 2001.

[113] W. Richard Stevens. *Advanced Programming in the UNIX Environment*. Addison Wesley, 1993.

[114] John P. Morrison, Padraig J. O'Dowd, and **Philip D. Healy**. LinuxNOW: A peer-to-peer metacomputer for the Linux Operating System. In *Proceedings of the 2003 International Conference on Parallel and Distributed Processing Techniques (PDPTA'04)*, Las Vegas, Nevada, June 2004.

[115] John P. Morrison, Padraig J. O'Dowd, and **Philip D. Healy**. Searching RC5 keyspaces with distributed reconfigurable hardware. In *Proceedings of The 2003 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'03)*, pages 269–272, Las Vegas, Nevada, 2003.

[116] Bruce Schneier. *Applied Cryptography*. Wiley, October 1995.

[117] Gregory V. Wilson. *Practical Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, January 1996.

[118] Jens-Peter Kaps and Christof Paar. Fast DES implementation for FPGAs and its application to a universal key-search machine. *Selected Areas in Cryptography*, pages 234–247, 1998.

[119] Paul D. Kundarewich, Steven J.E. Wilton, and Alan J. Hu. A CPLD-based RC-4 cracking system. In *Proceeding of the 1999 Canadian Conference on Electrical and Computer Engineering*, May 1999.

[120] Ronald L. Rivest. The RC5 encryption algorithm. In William Stallings, editor, *Practical Cryptography for Data Internetworks*. IEEE Computer Society Press, January 1996.

[121] Willian Stallings. *Cryptography and Network Security*. Prentice Hall, August 2002.

[122] B. Kaliski and Y. Yin. On the security of the RC5 encryption algorithm. *CryptoBytes*, 1(2):13–14, 1995.

[123] R. Baldwin and R. Rivest. RFC 2040: The RC5, RC5-CBC, RC5-CBC-pad, and RC5-CTS algorithms, October 1996.

[124] Xilinx, Inc. *Virtex^{TM}-E 1.8V Field Programmable Gate Arrays Production Product Specification*, July 2002.

[125] Celoxica Ltd. *RC1000 Hardware Reference Manual*, 2001.

[126] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software – Practice and Experience*, 30(11):1203–1233, September 2000.

[127] Leslie Lamport. *LaTeX – A Document Preparation System – User's Guide and Reference Manual*. Addison Wesley, December 1985.

[128] Jane Hillston. *A Compositional Approach to Performance Modelling*. Cambridge University Press, January 1996.

[129] Peter Harrison and Naresh Patel. *Performance Modelling of Communication Networks and Computer Architectures*. Addison Wesley, 1992.

[130] Andrei V. Gurtov. Technical issues of real-time network simulation in Linux. In *Proceedings of Finnish Data Processing Week (FDPW'99)*, Petrozavodsk, Russia, June 1999.

[131] Barry Wilkinson and Michael Allen. *Parallel Programming: Techniques and Applications using Networked Workstations and Parallel Computers*. Prentice Hall, 1999.

[132] David S. Johnson. Fast algorithms for bin packing. *Journal of Computer and System Sciences*, 8:272–314, 1974.

[133] E. G. Coffman, M. R. Garey Jr., and D. S. Johnson. Approximation algorithms for bin packing: a survey. *Approximation Algorithms for NP-hard Problems*, pages 46–93, 1997.

[134] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. W. H. Freeman, 1979.

[135] Tracy D. Braun, H. J. Siegel, Noah Beck, Ladislau L. Boloni, Muthucumaru Maheswaran, Albert I. Reuther, James P. Robertson, Mitchell D. Theys, Bin Yao, Debra Hensgen, and Richard F. Freund. A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. *Journal of Parallel and Distributed Computing*, 61(6):810–837, June 2001.

[136] A. Osman and H. Ammar. Dynamic load balancing strategies for parallel computers. In *Proceedings of the 1st International Symposium on Parallel and Distributed Computing (ISPDC 2002)*, pages 110–120, Iaşi, Romania, July 2002.

[137] Muthucumaru Maheswaran, Shoukat Ali, H. J. Siegel, Debra Hensgen, and Richard F. Freund. Dynamic mapping of a class of independent tasks onto heterogeneous computing systems. *Journal of Parallel and Distributed Computing*, 59(2):107–131, November 1999.

[138] Erik M. Nystrom, Ronald D. Barnes, Matthew C. Merten, and Wen-mei W. Hwu. Code reordering and speculation support for dynamic optimization systems. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'01)*, Barcelona, Spain, September 2001.

[139] Cyril Fonlupt, Philippe Marquet, and Jean-Luc Dekeyser. Data-parallel load balancing stategies. *Parallel Computing*, 24(11):1665–1684, 1998.

[140] David Mills, Poul-Henning Kamp, and John Barnes. *Network Time Protocol*. O'Reilly, January 2005.

[141] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[142] Michel Raynal. About logical clocks for distributed systems. *ACM SIGOPS Operating Systems Review*, 26(1):41–48, January 1992.

[143] Krste Asanovic, John Hennessy, and David Patterson. Vector Processors. In *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, May 2002.

[144] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–523, December 1985.

[145] Hans Boehm. Dynamic memory allocation and garbage collection. *Computers in Physics*, 9(3):297–303, May 1995.

Index